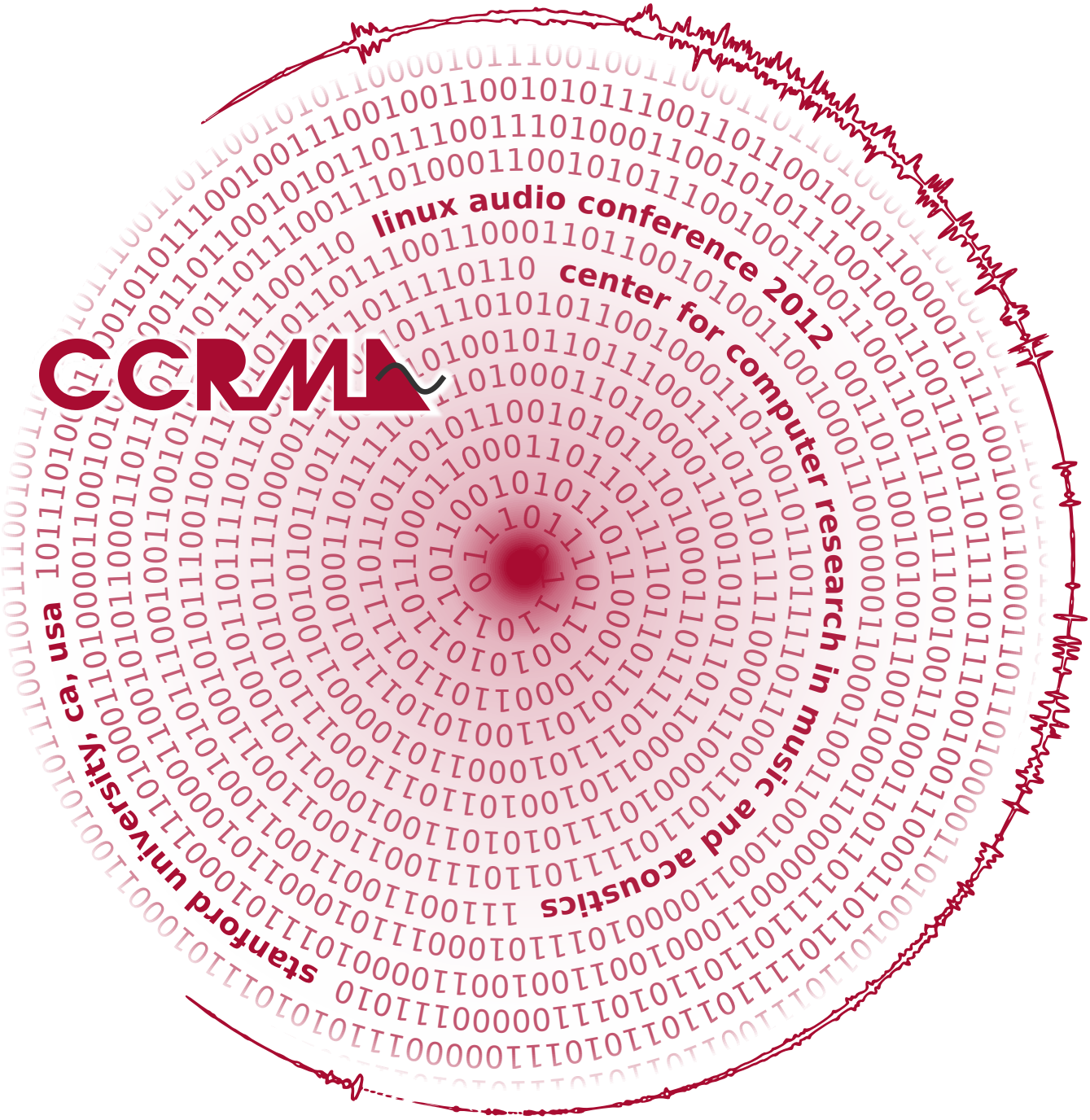
Proceedings of the

Linux Audio Conference 2012

April 12th - 15th, 2012

Center for Computer Research in Music and Acoustics (CCRMA),
Stanford University, California



Published by
CCRMA, Stanford University, California, US
April 2012

http://lac.linuxaudio.org/2012
ISBN 978-1-105-62546-6

Credits
Cover design: Fernando Lopez-Lezcano
Layout: Frank Neumann
Typesetting: LaTeX and pdfLaTeX
Logo Design: The Linuxaudio.org logo and its variations copyright Thorsten Wilms ©2006, imported into the "LAC 2012" logo by Robin Gareus

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

Printed in the US

Partners and Sponsors

CCRMA

Linux Weekly News

The Fedora Project

SICa

Stanford University

LinuxAudio.org

CiTU

Foreword

Stanford, California, March 2012

A long time ago in a country far far away, a group of Linux Audio enthusiasts got together at LinuxTag 2002 in Karlsruhe/Germany, organized a Linux Audio Developers Booth there, and plotted future plans. Little they knew (or perhaps they did?) that their initiative would have staying power and rally new people from all over the world every year. Today, the Linux Audio Conference celebrates its 10th anniversary being hosted in the United States for the first time. Read the whole story and look at the pictures at http://lac.linuxaudio.org, and you will see that most of those early enthusiasts are still around, as well as new ones that have been joining us every year.

We at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA, pronounced "karma") are very happy to be the hosts of LAC 2012. Bear with me as I tell you a bit of our history. CCRMA has been using and sharing Free Software as part of a collaborative environment for research and the arts since its beginning in the early 1970's. Mainframes were the only game in town and computer time for making music and creating new sound algorithms was so valuable that, at the beginning, it could only be borrowed at night in the Artificial Intelligence Lab at Stanford. Eventually CCRMA had its own shared computing monster and similarly-sized digital synthesizer (we called it the Samsom Box), and much music and research came out of them. But in time, mainframes, minicomputers, and mammoth synths withered away and gave way around 1990 to a network of nimble NeXT computers and pure software sound making (with Unix under the hood!, alas, not a free and open hood...). The demise of that world around 1997 was followed by a gradual and determined transition to a fully free and open GNU/Linux computing environment. All thanks to a lot of hard work by hundreds of dedicated programmers, hackers, enthusiasts, and visionaries from all over the world that pieced together a complete, working and reliable computing environment seemingly out of nothing.

By now made up of millions and millions of lines of code, all completely free and open for the world to see, use, share and improve. Amazing, if you "think abyte it!"

By 2001, the packages I had created and maintained to populate our GNU/Linux workstations with free sound and music software became available through the Internet for anyone to download. You could work at home in an environment similar to the one CCRMA users took for granted every day. The collection of packages was named Planet CCRMA and was offered to the world in the spirit of sharing. Planet CCRMA took me to the LAD conference for the first time in 2003 (LAC was LAD at the time). I found it simply wonderful that I could match faces to email addresses, and get to know a little about many of the people that had made possible the freedom we were enjoying at CCRMA. After that first time, I've had the pleasure of attending LAC pretty much every year, save for once, when the party for the inauguration of the renovated CCRMA building landed on the same weekend as LAC 2006.

For some years many LAC'ers had asked the same question: when is CCRMA going to host LAC? It was the enthusiasm of Bruno Ruviaro and his promise to help that convinced me that it was possible. The timing was good, a fantastically energetic friend committed to free software

was offering to help, and we had encouragement from Chris Chafe (CCRMA's Director, power Linux user) and the community. This LAC is going to be, for the first time, far away from its usual geographic "center of gravity," so we will miss some friends for whom the trip is too long and expensive. But we are sure we will meet new faces and match them with email addresses, exchanging new ideas with lots of people that could not go to LAC before because it was too far away.

The geekiest amongst us may see no reason to give special attention to the number 10 (after all, it is not a power of two); still, let us celebrate the 10th anniversary of LAC with four days of talks and music, science and art, free software and free and open hardware, new reflections and, of course, a lot of food, beer and coffee. 25 technical and musical talks, 3 full concerts plus the Linux Sound Night (20+ pieces total) and a continuous 3D listening session for 17 additional pieces and sound installations, 5 workshops, etc, etc. Quite a program!

I'm glad to have an old friend and long-time Linux musician, Dave Phillips, to be the keynote speaker this year. I'm sure he will captivate us with memories and prognostications, anecdotes and numbers. His book and his patient encouragement helped raise awareness of the possibilities of this evolving platform and provides a lot of the glue that holds the community together.

Many thanks to all those who made this conference a reality. Bruno Ruviaro specially, without him this would not have happened, and of course Chris Chafe and Jonathan Berger for their support and help in so many ways. Robin Gareus, amazing, fast, impossible to stop, for his help with the web site and organization. Frank Neuman for his (again!) amazing work in massaging tex, pdf and figures, and creating a great book of proceedings. Julius Smith, Carr Wilkerson, Sasha Leitman, Jaroslaw Kapuscinski and many others at CCRMA who volunteered their efforts to make this a memorable event. Support for making this conference possible came from CCRMA itself, the Music Department at Stanford, SiCa (the Stanford Institute for Creativity and the Arts), the Fedora Project and other institutions and friends. Needless to say, thanks to Jörn Nettingsmeier, Robin, and the streaming team helpers, if you are halfway around the world, you will be able to tune in and interact with most events remotely.

Here's my toast for many many more years of better and better software. Hey developers!, surprise us yet again with cool unpredictable programs! Change the world for the better! Hopefully by next year's LAC we will all be wearing cheap, fast, 16-core powered, GNU/Linux free and completely open "communicator" devices on our wrists (Dick Tracy style, that was 1946!!), of course with full bandwidth sound and running a super low latency Mini-Jack sound server that will connect us to the world and to each other.

Fernando Lopez-Lezcano
LAC 2012 Conference Chair
http://lac.linuxaudio.org/2012/
https://ccrma.stanford.edu/

PS: and last but not least come and meet Ping in California: he will welcome you to his native habitat. He's been traveling with me to many events and conferences all over the world since 2004, and he loves GNU/Linux! See what I mean here: `http://ccrma.stanford.edu/~nando/ping/`

Conference Organization

Fernando Lopez-Lezcano
Bruno Ruviaro
Robin Gareus

Live Streaming

Jörn Nettingsmeier
Robin Gareus

Conference Design and Website

Robin Gareus

Concert Sound

Carr Wilkerson
Sasha Leitman
Bruno Ruviaro
Fernando Lopez-Lezcano

Paper Administration and Proceedings

Frank Neumann

Special Thanks to

Chris Chafe
Jonathan Berger
Julius Smith
Carr Wilkerson
Sasha Leitman
Jaroslaw Kapuscinski
Tricia Schroeter
Jay Kadis
Scott Kepley
Laura Dahl

...and to everyone else who helped in many ways after the editorial deadline of this publication!

Review Committee

Fons Adriaensen	Casa della Musica, Parma, Italy
Tim Blechmann	Vienna, Austria
Ico Bukvic	Virginia Tech, VA, United States
Götz Dipper	ZKM|Institute for Music and Acoustics, Karlsruhe, Germany
John ffitch	retired/U of Bath, United Kingdom
Robin Gareus	Linuxaudio.org, France
Dr. Albert Gräf	Johannes Gutenberg University Mainz, Germany
Fernando Lopez-Lezcano	CCRMA, Stanford, United States
Frank Neumann	PenguinNoises, Germany
Yann Orlarey	Grame, France
Jussi Pekonen	Finland
Dave Phillips	linux-sound.org, United States
Martin Rumori	Institute of Electronic Music and Acoustics, Germany
Bruno Ruviaro	CCRMA, Stanford, United States
Julius Smith	CCRMA, Stanford, United States
Michael Wilson	CCRMA, Stanford, United States
IOhannes Zmölnig	Institute of Electronic Music and Acoustics (IEM@KUG), Austria

Music Jury

Bruno Ruviaro
Fernando Lopez-Lezcano

Table of Contents

The IEM Demosuite, a large-scale jukebox for the MUMUTH concert venue

Peter PLESSAS and **IOhannes m ZMÖLNIG**
Institute of Electronic Music and Acoustics (IEM), University of Music and Performing Arts (KUG)
Inffeldgasse 10/III
8010 Graz,
Austria
{plessas, zmoelnig}@iem.at

Abstract

In order to present the manifold possibilities of surround sound design in the new MUMUTH concert hall to a broad audience we developed an easy to use application running and interacting with audio demonstration scenes from a mobile computing device. An extensible number of such demonstrations using the large-scale and variable loudspeaker hemisphere in the hall are controlled with a user interface optimized for touch-sensitive input. Strategies for improving operating safety, as well as the introduction of a client/server model using the programming language Pure Data, are discussed in connection with a 3D Ambisonics rendering server, both implemented using the Linux operating system.

Keywords

demonstration, interaction, client-server systems, Ambisonics, Pure Data

1 The Mumuth

The University of Music and Performing Arts Graz (KUG) opened a new building, the *Music and Music Theatre House* (MUMUTH), in 2009. It holds a 600 sqm. concert hall, named in honor of the Austrian composer György Ligeti. This new venue is used for productions by the university's different departments. Therefore, the hall has to suit music performances of different styles as well as theatre and opera works. This multi-faceted usage pattern demands a flexible performance space in terms of stage machinery, room acoustics, sound reinforcement, lighting and seating. As consequence MUMUTH has no fixed stage and audience position but offers an array of pedestals making up the hall's floor and which can be individually raised to provide an elevated audience platform or stages of variable size and height. While this flexible layout allows for creative staging of different performances it also imposes a serious challenge for the installation of a sound reinforcement equipment, since any part of the room can now become the stage or audience area and as such be of varying size.

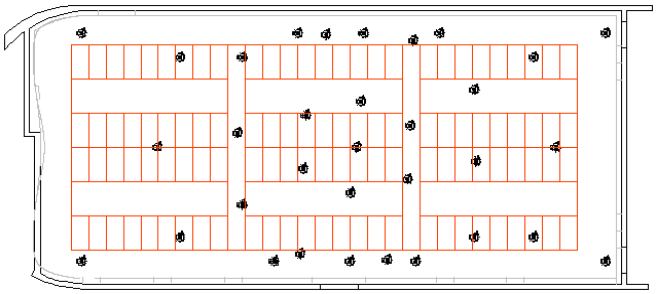


Figure 1: MUMUTH top view: Floor pedestals and loudspeaker hang points

1.1 Variable position loudspeaker hemisphere

The university's Institute of Electronic Music and Acoustics (IEM) designed a unique sound reinforcement system in order to cope with the many different demands. It consists of 33 loudspeakers, which can be lowered from the ceiling on telescopic lifts and be positioned at arbitrary heights as well as rotation and elevation angles. Figure 2 gives an impression of the room. The speaker layouts, which can be easily refined within seconds, are stored and recalled, if desired also dynamically, during a show. Electroacoustic measurements and listening tests with different loudspeaker positions permit instantaneously perceivable comparisons. It is this very setup that is used in 3D sound field rendering using higher order ambisonics from IEM's own software. It is also employed in the reproduction of the different audio scenes in the IEM Demosuite.

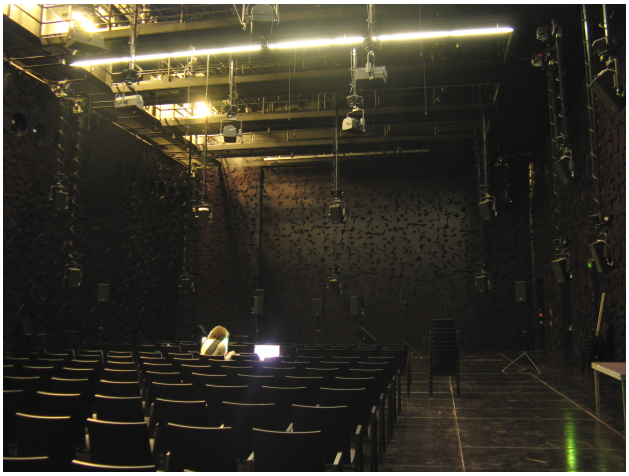


Figure 2: Loudspeaker hemisphere

1.2 The integration of higher-order Ambisonics

With many years of experience in the design and development of Ambisonics capture and playback systems, IEM developed the scalable CUBEmixer ambisonics authoring and rendering system [Musil et al., 2008], which is distributed under a GPL license[1]. This system consists of a mixing application that can run on general purpose hardware and operating systems, and which is open to user extention and external control being implemented in the Pure Data (Pd) programming language. CUBEmixer is employed in the discrete or Ambisonic spatialization of an arbitrary number of sources on the loudspeaker hemisphere. The helical positions of the mount points on the ceiling as shown in Figure 1 allow to arrange the loudspeakers in a hemisphere encircling the venue's available space with a minimum angular difference between the speaker positions. CUBEmixer integrates very well with the MUMUTH audio infrastructure, consisting of a *Lawo HD Core* mixing engine and signal matrix, controlled through the *Lawo mc*^{2}66 fanless mixing console. Inputs and outputs are via dedicated interface racks interconnected via MADI optical cables, providing a total of 4096 physical input and output channels. Interfacing to the main rendering computer is done via two *RME HDSP MADI* PCIe cards using Alsa drivers with 64 bit *Debian/GNU Linux*.

[1] http://ambisonics.iem.at/xchange/products/cubemixer

1.3 Enhanced room acoustics

In addition to the sound reinforcement system described above, MUMUTH's concert hall is equipped with a variable room-acoustics system[2], allowing to switch between different reverberation scenarios enhancing reverb envelopment and speech intelligibility. With this system the venue's acoustics can be tailored to different performance styles such as theatre, classical music, jazz and to IEM's concerts of electronic music. As the variable room-acoustics system is optimized for sound sources located on the floor level, it is in general not used together with the loudspeaker hemisphere.

2 Presenting MUMUTH's sonic abilities to a broad audience

Having build a venue the size of MUMUTH, it is important to convey the idea of a concert hall being "a musical instrument on its own" to the general public. In order to demonstrate how sound in space can be creatively composed as well as interactively explored, we decided to create a suite of music demos that would immediately catch the listener's attention and present the abilities of this new location in an entertaining and convincing way. An easy-to-use and interactive demo application for visitors, playable even while the room may be set up for a different production, had to be developed. While this suite of demos would provide ready-to-run sound scenes, it should at the same time invite pre-rendered or interactive extensions to its repertoire playlist from other artists and researchers alike.

3 Software and Hardware

While looking for a suitable software platform in which to implement the Demosuite, we decided it would be preferable to use an environment that future contributors would most likely be familiar with. In our research we and our colleagues often use Pure Data, so taking that environment as the starting point for implementing the Demosuite seemed a good choice: in theory, software taken out of our development department could be transferred directly to the suite. An additional bonus was the graphical nature of Pd, as there was no need to switch between environments

[2] *Meyersound Constellation*

when creating the visual user-interface and the DSP side of a demo. Finally, the cross-platform nature of Pd would make it possible to develop demos on one's own preferred platform, before deploying it on a stable and low-latency operating system in the venue.

The MUMUTH had already been equipped with a PC for realtime signal processing running *Debian GNU/Linux*, so we decided to use that machine as a stable and well-tested base platform that was already well integrated into the existing audio infrastructure.

This computer is usually controlled from the MUMUTH's separate sound recording studio, though it is possible to extend keyboard/monitor/mouse into the concert hall.

However, for the needs of the Demosuite, this approach was discarded as it would involve the need to set up a workstation in the concert hall every time the demo should be presented, which might in fact happen spontaneously between two rehearsals of an opera without advance warning.

Instead we decided to control this computer with a convenient and portable remote control the presenter could carry around while running the demos. While the environment of our choice (Pd) is known to run well on both iOS and Android based smartphone and tablet devices [Brinkmann et al., 2011], this is unfortunately only true for the DSP-part of Pd and not for the user interface requiring Tcl/Tk. The only available touch device we found that could run the Pd-GUI was the *Neophonie WePad*. This tablet device runs a stock i386 based Linux system[3], making it very easy to develop new and deploy already existing software on it.

In order to make Pd suitable for tablet use, its visual appearance was slightly adapted with a "kiosk mode" gui plug-in [zmoelnig, 2011] displaying a single patch window in full-screen mode without any menus.

4 Demosuite Software Design

The Demosuite software consists of two semantically different parts:

- a static *framework* that takes care of selecting and activating a demo and that is controlling some global parameters such as master volume, all optimized for touch sensitive user interfaces.
- a growing number of *demos*, which are interactive audio applications implemented as Pd patches.

[3] *Linux Foundation's MeeGo*

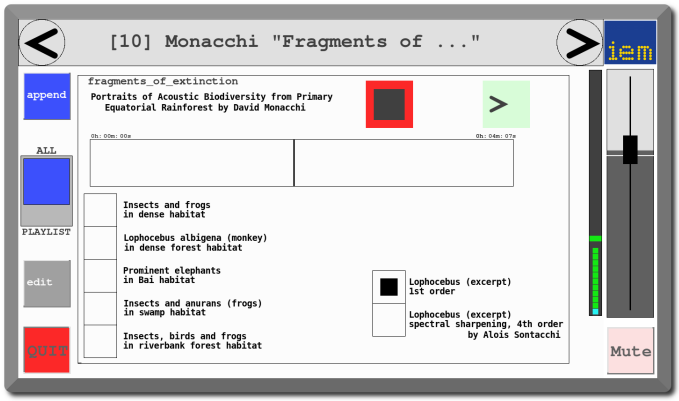


Figure 3: Example demo GUI with global volume fader right and demo selector strip on top

Demos are mutually exclusive, meaning that if a given demo is currently loaded/running, no other demo can be loaded/running at the same time. An example of a user interface for a selected demo, controlling the playback of pre-rendered sound files allowing directly switchable different Ambisonics orders is shown in Figure 3. Here, playback of pre-rendered sound files is controlled along with switchable Ambisonics orders for direct comparison, with additional written comments about the demo's contents.

To complicate things a bit, the Demosuite software is to run on two hosts in parallel:

- a powerful *DSP* server, capable of processing and interfacing multichannel sound fields in realtime
- a lightweight *GUI* client that controls this server over a network connection

The Demosuite software has to provide the communication between the DSP and the GUI parts and ensure that the internal states of the two parts stay synchronized even during network disruptions. In order to add a new demo to the playlist, corresponding GUI and DSP patches have to be copied to the two machines respectively.

4.1 Slot Management

We call the container mechanism, that ensures that the matching parts (GUI and DSP patches) of a single demo are active and synchronized across the two computers, the "slot".
Basically two different approaches to managing a variable number of demos via this slot mechanism exist:

- either all demos are preloaded into a matching number of slots and, upon selection of a given demo its containing slot gets activated while all other slots are deactivated
- or there is only a single slot that manages the synchronous loading and activation of an arbitrary demo across the two hosts.

Both methods have their advantages and drawbacks. In the case of the MUMUTH Demosuite we identified the following relevant characteristics of the "pre-load" technique:

- *Pros*
  - fast (since initialization can be done at startup time)
- *Cons*
  - more resources needed (scales badly)
  - "deactivated" demos can remain half-active (e.g. by continuously generating messages)
  - co-existing demos can influence each other (if not carefully implemented)

Whereas with the technique of "dynamic loading into a single slot" can be characterized as follows:

- *Pros*
  - resources allocated on demand (scales well)
  - "deactivated" demos don't exist and can therefore not take any resources
  - demos never co-exist
- *Cons*
  - resources allocated on the fly (slow, since no pre-loading is possible)
  - instantiation locks the Pd process
  - requires dynamic patching which makes code less readable (bad for long term maintenance)

Since the demos are to be implemented by multiple collaborators and will presumably be of diverse code quality, the perfect isolation between demos in the "dynamic loading" approach was the winning argument. Assuming that a single demo will consume moderate resources in return, the elongated load time due to dynamic resource allocation was considered acceptable. Furthermore, this approach scales well even for a very large number of demos.

4.2 Slot Interface

A slot consists of two separate programs (the DSP and the GUI part) that communicate with each other as well as produce perceptible output (the DSP will usually render sound, whereas the GUI will visualize an interface to control the former). The communication channels for the two parts are standardized, in order to be able to globally replace them with different networking techniques.

4.2.1 control communication

Usually GUI and DSP will run on different computers that are linked via a network connection. To keep the implementation simple, this communication is abstracted as unidirectional and asynchronous busses. The network message interface inside PD is implemented as single `[inlet]` resp. `[outlet]` objects in the GUI and DSP patch of each demo. Each Pd message that is to be sent from one peer to the other, has to go through that single `[outlet]` and will then appear on the other side's sole `[inlet]` object. No assumption must be made on the speed of the transmission.

4.2.2 audible and visible output

In addition to choosing and interacting with a demo, the user should be able to control the global volume with a single fader, in a unified way for each and every demo. It is therefore important that the DSP patch does not access the soundcard's DACs (or rather Pd's representation thereof) directly, but through a global audio bus. The Demosuite applies a few post-processing steps on this multi channel audio bus, namely master volume, multichannels limiting and a global mute. This is achieved by using

a special [throw~ chn-<N>] object rather than Pd's own [dac~ <N>].
The GUI part is trying to make best use of the allocated screen estate by being implemented as a Pd "graph on parent" interface of a fixed size, defining the available area for the UI part of a demo.[4]

4.3 Loading a Demo

Since the GUI and DSP parts are asynchronous processes, it is necessary to pay special attention to the sequence of actions in which to load a demo in order to guarantee consistent behavior and defined states of both systems at each instant in time.
Loading is initiated by the user selecting a scene (e.g. "foobar") on the interface. The GUI will then delete the previously loaded demo, so it can no longer send control messages to the DSP part of the demo, and enter a transitional state, (showing a "loading..." splash screen). Once this is accomplished, a request "/playlist_entry foobar" is sent to the DSP part.
Once DSP receives such a request, it will first delete its part of the the previous scene and subsequently load the "foobar/dsp.pd" patch, which receives an initialization trigger. Once the DSP part of the demo is initialized, it must then send a "/dsp_init_done" message, which will cause the DSP to send a "/create_gui_slot foobar" request back to the GUI. Now that the GUI knows that the DSP is properly loaded, it can load and display the "foobar/gui.pd" patch. The GUI part of demos having non-trivial internal states can now query the DSP to transmit the current state of internal parameters.

5 Communication between GUI and DSP

Since the DSP and the GUI are running on two different machines, their communication is done via a network connection. In MUMUTH, the tablet is connected via a wireless LAN to ensure maximum mobility of the presenter. This required the following considerations.

[4]This is probably the most tedious part when writing new demos. Unfortunately, at the time of this writing, Pd does not provide a usable auto-scaling function of "graph on parent" interfaces, which means that whenever the screen resolution of the remote control changes, the GUI patches have to be adapted...

5.1 TCP/IP vs. UDP

Traditionally, network connections involving realtime audio environments use the simple UDP rather than the more reliable TCP/IP protocol, mainly because of the following two reasons:

- *overhead*: due to it's simpler nature UDP has less traffic overhead than TCP/IP, and will therefore consume less resources
- *robustness*: since UDP does not track active connections, it handles loss of connectivity more gracefully than TCP/IP; esp. it does not cause the remote point to hang if the local point experiences a network outage. This is especially important considering the volatile connection with a wireless mobile device and the non-threaded network implementation in Pd (that would cause the main audio thread to hang in case the TCP/IP remote control left the coverage of the wireless LAN access point!)

5.2 to OSC or not to OSC...

State of the art communication between networked audio workstations is usually implemented using Open Sound Control (OSC). While Pd still has no built-in support for OSC and instead provides objects that speak PD's own FUDI protocol, it can be extended using Martin Peach's "osc" library. Unfortunately we found the various available network transport implementations needed for this "osc" library to be less stable than expected, especially in cases where the connection can easily drop, which is expected to happen once the tablet leaves the W-LAN coverage. Rather than fixing these implementations, we decided to write a set of abstractions that hide the actual implementation of the OSC transport.
On the application layer directly accessible within a demo (and within most parts of the framework), "OSC-style" messages are used. On the layer below, these messages are currently transmitted within FUDI containers. Once the issues of the network transport libraries are addressed with the ongoing development of Pd's externals, the wrapper objects can easily be switched back to use OSC containers.

5.3 Network Security

In the current implementation, absolutely no network security is built into the system. An intruder

who has knowledge of the protocol, the network address and port of the DSP machine can send arbitrary commands. Since the wireless network is only available within the concert hall and is encrypted, we have not experienced any security related problems so far. In any case, it should be easy enough to protect the two hosts via a clever set of firewall rules and network tunnelling technology such as ssh, SSL or VPN.

5.4 Operating Safety

Whenever the connection between the main DSP computer machine and the GUI remote control is interrupted, it is necessary to fallback into a safe state. In order to monitor the status of the UDP connection, a special "heartbeat" message is constantly exchanged between the two computers. Whenever the user interface assumes to be connected to the DSP server, it sends out a message "/client_alive 1" in regular intervals (currently about 3 times per second), which is answered by a "/server_alive 1" message in return. If either side does not receive its peer's "alive" message for a certain amount of time, it considers the remote side to have disconnected. Once the DSP server detects a disconnected GUI, it will immediately mute its audio outputs in order to prevent damage to the listener's ears as well as to the equipment by uncontrollable audio signals. Whenever the GUI detects a disconnected DSP server, it displays a warning message of the lost connection and locks the interface that otherwise would control the DSP part of a demo. This is last measure is important in order to ensure that the state on both DSP and GUI sides divert as little as possible, making the reconnection and resuming of a demo an easy task. The GUI will try to reestablish the connection by periodically attempting to reconnect, send a "/conn_request 1" message. This message will eventually trigger an reconnection attempt of the DSP server.
As soon as the connection is re-established, the GUI queries the DSP server about it's current state and adjusts its internal state and hence the state of the user controls accordingly.

6 Repertoire

Current examples of demos available in the suite are:

- Audio scenes of pre-rendered Ambisonic sources on different spatialisation paths on the hemisphere with dynamically changing early reflections and late reverberation, with Ambisonics-encoded reverb return signals.
- Ambisonic sources in comparison to discrete loudspeaker playback with additional depth perception cues.
- Extracts from IEM concert productions.
- Dedicated pieces from IEM staff and featured guest composers.
- A 3D panning demo using allowing interactive user control.
- Ambisonics field recording examples.

7 Summary

In this contribution we described how an easy-to-use system of audio demo scenes for the interactive exploration of the new MUMUTH concert space has been realised. We presented an approach towards networked Pd patches using connection state monitoring together with dynamic patch creation on different host computers. A solution allowing for easy extensibility of the demo repertoire while preserving system stability was discussed along with an alternative network transport layer for Pd messages. The adaption of Pd to touch-based tablet interfaces and considerations regarding operational reliability with regard to high-volume PA systems were shown.

8 Acknowledgements

While the design and realisation of the audio infrastructure in MUMUTH has been the result of a huge team of collaborators, the authors would especially like to express their gratitude to IEM/KUG members Gerhard Eckel for initiating the Demosuite project, and Thomas Musil, Stefan Warum and Ulrich Gladisch for their exceptional support.

References

Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. 2011. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention 2011*, Weimar, Germany. `http://www.uni-weimar.de/medien/wiki/images/Embedding_Pure_Data_with_libpd.pdf`.

Thomas Musil, Winfried Ritsch, and Johannes Zmölnig. 2008. The cubemixer a performance-, mixing- and masteringtool. In *Proceedings of the Linux Audio Conference 2008*, Cologne, Germany. `http://old.iem.at/projekte/publications/paper/cm/cm.pdf`.

IOhannes zmoelnig. 2011. New clothes for pure data. In *Proceedings of the Linux Audio Conference 2011*, Maynooth, Ireland. `http://lac.linuxaudio.org/2011/download/lac2011_proceedings.pdf`.

Network distribution in music applications with Medusa

Flávio Luiz SCHIAVONI and **Marcelo QUEIROZ**
Computer Science Department
University of São Paulo
São Paulo
Brazil,
{fls, mqz}@ime.usp.br

Abstract

This paper introduces an extension of Medusa, a distributed music environment, that allows an easy use of network music communication in common music applications. Medusa was firstly developed as a Jack application and now it is being ported to other audio APIs as an attempt to make network music experimentation more widely accessible. The APIs chosen were LADSPA, LV2 and Pure Data (external). This new project approach required a complete review of the original Medusa architecture, which consisted of a monolithic implementation. As a result of the new modular development, some possibilities of using networked audio streaming via Medusa plugins in well-known audio processing environments will be presented and commented on.

Keywords

Network Music, Pure Data, LADSPA, LV2, Medusa.

1 Introduction

Ever since the availability of network and Internet connections became an undisputed fact, collaborative and cooperative music tools have been developed. The desire of using realtime audio streaming in live musical performances inspired the creation of various tools focused on distributed performance, where synchronous music communication is a priority. Some related network music tools which address the problem of synchronous music communication between networked computers are NetJack [Carôt et al., 2009], SoundJack [Carôt et al., 2006], JackTrip [Cáceres and Chafe, 2009b; Cáceres and Chafe, 2009a], llcon [Fischer, 2006] and LDAS [Sæbø and Svensson, 2006].

The success cases of network music performance concerts could give the wrong impression that the only use for network music is distributed performance, but several other problems in computer music can take advantage of or benefit from network music distribution. For instance, recordings can be made using a pool of networked computers, a resource which might be seen as a scalable distributed sound card; music spatialization could be done using a mesh network topology; digital signal processing power can be vastly improved using clusters of computers; and musical composition may explore fresh new grounds by viewing computer networks as nonstandard acoustical environments for performance and listening.

There are at least two requirements for all these scenarios to be fully explorable by potentially interested users, which are often musicians or aficionados and usually nonprogrammers; first, flexible audio network tools must be made available, and second, easy integration with popular music/sound processing applications must be guaranteed. Despite the fact that most high-level linux music applications nowadays run over Jack and ALSA, we see that time and again end users can't deal with this audio infrastructure lying below the application they are running. Also, the lack of automated setup and graphical user interfaces shun common users from many existing tools, like the network music tools mentioned above.

The work presented in this paper is part of the investigation behind the development of Medusa [Schiavoni et al., 2011], a distributed audio environment. Medusa is a FLOSS project which is primarily focused on usability and transparency, as means to making network music connections easier for end users. The first implementation of Medusa, presented in LAC 2011, was developed in C++ with Qt GUI and Jack as sound

API. The Jack API was extended with a series of network functionalities, such as add/remove remote ports and remote control of the Jack Transport functionality.

Recently, the development of Medusa has been strongly guided by the attempt to deal with the two aforementioned requirements, namely flexibility and integrability. These goals may be reached by extending regular sound processing applications, such as Pure Data, Rosegarden or Ardour, allowing them to function as network music tools. The very basic idea is trying to reach end users wherever they already are.

Most music applications can be extended by plugins: Pure Data can be extended by the creation of C externals (which might be viewed as a plugin), whereas digital audio workspaces such as Ardour, Rosegarden, Qtractor and Traverso can be extended by LADSPA, VST and LV2 plugins. In this paper we will explore the possibility of developing Medusa network music plugins using three popular audio APIs: LADSPA, LV2 and Pure Data API (via C externals). It is worth mentioning that these APIs are all open-source, developed in C and widely used in Linux music environments.

A reimplementation of Medusa has been required in order to grant code reuse in the implementation of these plugins, and also for easy maintenance of the source code. All source code of the Medusa project, including Medusa plugins, are freely available in the project site[1]. This reimplementation of Medusa and the proposed architecture are presented in section 2; section 3 discusses the chosen audio APIs and presents the developed plugins, and section 4 brings some conclusions and a discussion of future works.

2 Medusa

Although some promising preliminary results had been achieved with the first version of Medusa [Schiavoni et al., 2011], the original monolithic implementation raised several difficulties in the implementation of the proposed Medusa plugins, which eventually triggered a fundamental architectural change in the project. First, the implementation language was changed from C++ to ANSI C, which is more compatible with the chosen sound APIs. Second, the monolithic structure has been changed to the development of a core Medusa library (libmedusa.so), comprising control and network functions, which could be re-used by each new plugin implementation. Third, graphical user interfaces, text-based interfaces and plugins now occupy a separate layer, that uses the core Medusa library as an API on its own.

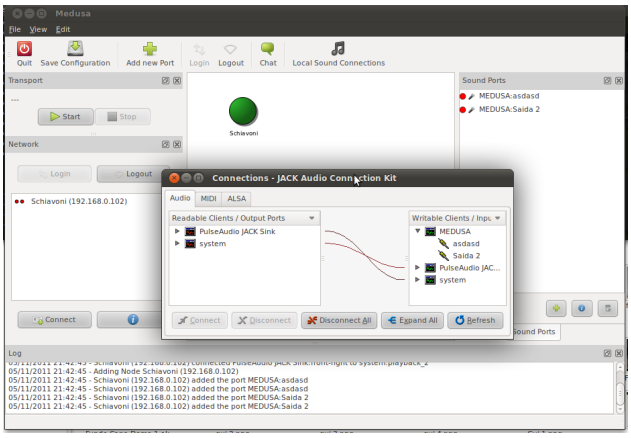


Figure 1: Medusa Jack implementation with Qt GUI

The new Medusa architecture is divided in three layers: Sound, Control and Network. The Control and Network layers are implemented as a unique library and are used by all Medusa implementations. The Sound layer corresponds to specific applications, like the proposed plugins, that are built using each plugin API and the Medusa library. This architecture facilitates code maintenance and bug correction.

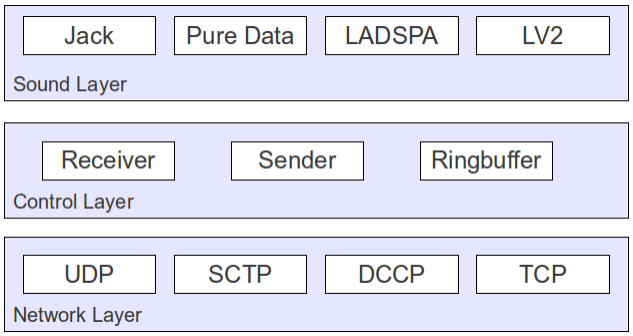


Figure 2: Architectural view of the implementation

[1] http://sourceforge.net/projects/medusa-audionet/

2.1 Network layer

The network layer is responsible for managing connections between network clients and servers. This layer's implementation was made based on a fixed set of network transport protocols, which are normally provided as part of the operating system kernel, since its development and deployment requires superuser privileges. Medusa currently allows the user to choose among 4 transport protocols: UDP, TCP, SCTP e DCCP:

UDP [Postel, 1980]: User Datagram Protocol is the classical unreliable (but faster) transport protocol.

TCP [Padlipsky, 1982]: Transmission Control Protocol is a reliable transport protocol, which ensures absence of packet losses.

SCTP [Ong and Yoakum, 2002]: Stream Control Transmission Protocol is a connection-oriented transport protocol that provides a reliable full-duplex association. This protocol was not originally meant as a replacement for TCP, but was developed for carrying voice over IP (VoIP).

DCCP [Kohler et al., 2006; Floyd et al., 2006]: Datagram Congestion Control Protocol is a transport protocol that combines TCP-friendly congestion control with unreliable datagram semantics for applications that transfer fairly large amounts of data [Lai and Kohler, 2005].

This multi-protocol network communication layer is intended to offer alternatives for users that may need different specific features in data transfer according to application context. For instance, an interactive musical performance with strong rhythmic interactions may require the smallest possible latency while tolerating audio glitches due to packet losses, and a remote recording session may tolerate high latency and jitter, but still require that every packet be delivered. With a few alternatives available, the user may choose which protocol is more appropriate to its own musical use.

2.2 Control layer

The next layer in the Medusa API is the Control layer. The Control layer essentially creates senders and receivers and provides them to the Sound layer. Instead of requiring that all protocols in the Network layer to have full-duplex communication, the Control layer always separates the roles of sending and receiving network data.

Figure 3: Sender / Receiver communication

In order to create a sender it is necessary to inform the network protocol, the network port and the number of channels that will be sent. The sender creates the network server and one ring buffer for each channel, and provides functions for the sound API to have easy access to these buffers. Each ring buffer receives sound content from applications (usually in DSP chunks), out of which the sender prepares network chunks (usually of a different size) for the Network Layer.

The receiver is created in a way similar to the sender, but besides the basic parameters the corresponding server IP address is also required. The receiver creates a network client and the required number of ring buffers (one per channel). Unlike the sender, the ring buffer of the receiver will be fed by a network client and consumed by the sound API.

2.3 Sound layer

The outermost Medusa layer is the Sound layer, which deals not only with audio streams but also with MIDI streams. The Sound layer lies between the Medusa API and several sound application APIs, and represents a collection of Medusa front-ends or interfaces, since each integration of Medusa with a particular sound application may have its own user interface, defined by each sound application API.

The integration of Sound and Network layers uses the Control layer through its sender / receiver plugins. Sender and receiver roles defined by the Control layer are converted into Sound layer plugins that simply exchange audio streams between machines.

Each plugin has a host application and a particular way to communicate with it. The API defines how a plugin is initialized, how it processes data and how it is presented. Each plugin implemen-

tation generates an independent software package that can be individually used and distributed. The separation of these implementations avoids the mixing of different plugin libraries, which is better for code maintenance, chasing bugs and so on.

3 Implementations

3.1 LADSPA

LADSPA [Furse, 2000] stands for "Linux Audio Developer Simple Plugin API", and it is the most common audio plugin API for Linux applications. Many Linux programs, such as Audacity, Ardour, Rosegarden, QTractor and Jack Rack, support LADSPA plugins. The LADSPA API is captured within a header file and is very easy to use. Some examples are provided with an SDK and there is a lot of open source code with great documentation available.

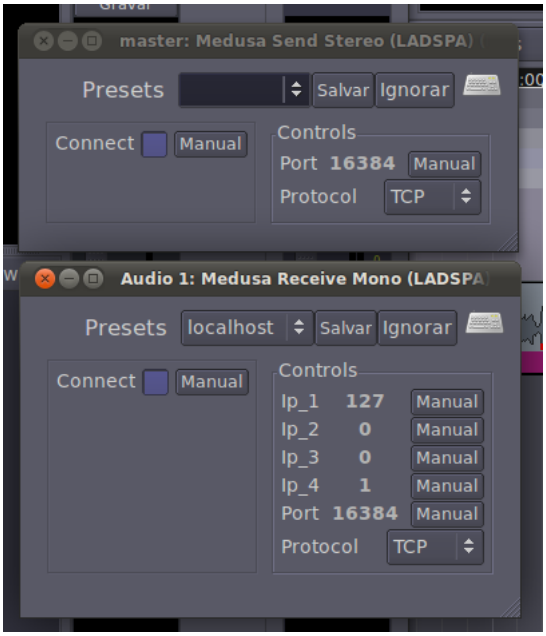


Figure 4: Medusa LADSPA implementation

Figure 4 presents Medusa LADSPA sender and receiver interfaces. The controls of a LADSPA plugin are represented exclusively by 32 bit floating point numbers, and it was awkward to use these to represent IP addresses. LADSPA GUIs are defined in RDF files that cannot be changed on-the-fly, which makes them very cumbersome as interfaces for network audio connections. LADSPA doesn't support MIDI and the documentation suggests the use of Rosegarden DSSI to deal with it.

3.2 LV2

LV2 (LADSPA version 2) [Steve Harris, 2008] is acknowledged as the official LADSPA successor. An LV2 plugin is a bundle that includes the plugin itself, an RDF descriptor in Turtle and any other resource needed. The LV2 bundle may include a GTK GUI, thus offering developers a way to create better user interfaces. As LADSPA, LV2 is an easy-to-use library, and plenty of documentation and examples are provided by the developers.

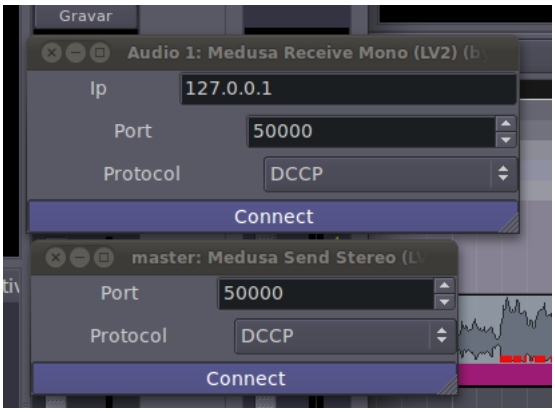


Figure 5: Medusa LV2 implementation

The possibility of creating GTK interfaces allowed a more natural integration of Medusa as an LV2 plugin. The IP mask entry allows easy input for the user, and on-the-fly changes to the interface may be used by Medusa in the future to allow for finding users and sound resources and keeping this information up-to-date on the interface.

3.3 Pure Data external

Pure Data [Puckette, 1996] (aka Pd) is a largely used realtime programming environment for audio, video, and graphical processing, which can be extended by the use of externals written in C/C++ language [Zmölnig, 2001]. Pure Data does have some network externals available, but none offers all transport protocols provided by Medusa. A Pd external may also be implemented with a tcl/tk GUI, which can be useful to improve usability and also to implement Medusa service control in the future. In addition, Pd offers a good environment to measure latency, packet loss and jitter in Medusa Network layer implementation.

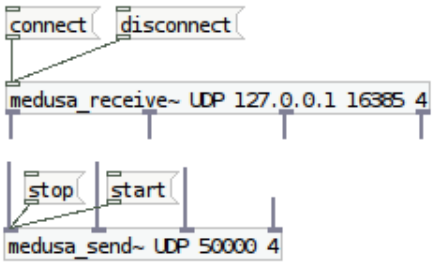


Figure 6: Medusa Pure Data external implementation

The Medusa Pd external implementation is actually a collection of externals (a library) that includes the sender and receiver objects (see figure 6) and a network meter to measure latency and packet loss. Some examples of use were also developed and are available on the project website.

4 Conclusions and Future Works

The possibility of exchanging sound data between applications using the Jack sound server, for instance, has brought new perspectives in audio software development and usage. Extending this possibility to allow for network data exchange from within popular user applications may facilitate the emergence of new ways to compose, record or play music. It may be early to conclude that network plugins for audio software will really expand the musical use of computer networks, but it is not early to observe in users and musicians new expectations about what might be done in network music, and also the desire to try out new ways to do old things, maybe more easily and more transparently than before.

The effort that went into changing Medusa architecture and reimplementeing it is totally justified, in the sense that now the effort of implementing other network music plugins (i.e. Medusa plugins for other sound applications) involve a lot of code reuse and thus have been made much lighter.

On the other hand, the first version of Medusa had an additional structure called Control Service, which was responsible for helping users to connect to network music resources in a transparent way, in other words, without having to specify IP, network ports or audio configuration, using broadcast messages to publish networked audio resources.

With that in mind, the level of usability originally pretended by this project will only be reached when Medusa's control service is implemented in the presented Medusa plugins, which is the very next step in our work. This control service is going to be responsible for improving transparency and usability, by including a discovery service, and a name server to publish networked music resources, thus allowing users to refer to other users and audio streams, instead of IP addresses and network ports. Probably the Medusa LADSPA plugin will have to be abandoned at this point because the control service is impossible to implement without on-the-fly GUI modification (although the current version of the plugin, without control service, should be kept).

Up to this point the plugin development has focused on audio streaming, but some other features must soon be addressed, such as treating MIDI streams and designing better GUIs. There are also other sound APIs that are been investigated and maybe soon will be incorporated in Medusa Sound layer.

Acknowledgements

Thanks to uncountable developers of LADSPA, LV2 and Pure Data externals and their amazing open source code. Without their anonymous help this project would not have been possible.

Thanks also go to André Jucovsky Bianchi, Beraldo Leal, Santiago Davila, Antônio Goulart, Giuliano Obici, Danilo Leite and the Computer Music Group of IME/USP for their interest, feedback and support.

This work has been supported by the funding agencies CNPq (grant 141730/2010-2) and FAPESP - São Paulo Research Foundation (grant 2008/08623-8).

References

Juan-Pablo Cáceres and Chris Chafe. 2009a. Jacktrip: Under the hood of an engine for network audio. In *Proceedings of International Computer Music Conference*, page 509–512, San Francisco, California: International Computer Music Association.

Juan-Pablo Cáceres and Chris Chafe. 2009b. Jacktrip/Soundwire meets server farm. In *In Proceedings of the SMC 2009 - 6th Sound*

and Music Computing Conference, pages 95–98, Porto, Portugal.

A. Carôt, U. Kramer, and G. Schuller. 2006. Network music performance (NMP) in narrow band networks. In *Proceedings of the 120th AES Convention*, Paris, France.

A. Carôt, T. Hohn, and C. Werner. 2009. Netjack–remote music collaboration with electronic sequencers on the internet. In *In Proceedings of the Linux Audio Conference*, page 118, Parma, Italy.

Volker Fischer. 2006. Internet jam session software. `http://llcon.sourceforge.net/`.

S. Floyd, E. Kohler, and J. Padhye. 2006. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), March. Updated by RFC 5348.

Richard Furse. 2000. Linux audio developer's simple plugin api (ladspa). `http://www.ladspa.org/`.

E. Kohler, M. Handley, and S. Floyd. 2006. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March. Updated by RFCs 5595, 5596.

Junwen Lai and Eddie Kohler. 2005. A congestion-controlled unreliable datagram api. http://www.icir.org/kohler/dccp/ nsdiabstract.pdf.

L. Ong and J. Yoakum. 2002. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May.

M.A. Padlipsky. 1982. TCP-on-a-LAN. RFC 872, September.

J. Postel. 1980. User Datagram Protocol. RFC 768 (Standard), August.

Miller Puckette. 1996. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41.

Asbjørn Sæbø and U. Peter Svensson. 2006. A low-latency full-duplex audio over IP streamer. In *Proceedings of the Linux Audio Conference*, pages 25–31, Karlsruhe, Germany.

Flávio Luiz Schiavoni, Marcelo Queiroz, and Fernando Iazzetta. 2011. Medusa - a distributed sound environment. In *Proceedings of the Linux Audio Conference*, pages 149–156, Maynooth, Ireland.

David Robillard Steve Harris. 2008. Lv2 track. `http://lv2plug.in/trac/`.

Johannes M Zmölnig. 2001. Howto write an external for puredata. `http://pdstatic.iem.at/externals-HOWTO/`.

Luppp - A real-time audio looping program

Harry VAN HAAREN,
University of Limerick,
Ireland

harryhaaren@gmail.com
http://harryhaaren.blogspot.com
https://github.com/harryhaaren/Luppp

Abstract

Luppp is a live performance tool that allows the playback of audio while applying effects to it in real time. It aims to make creating and modifying audio as easy as possible for an artist in a live situation.

At its core lies the JACK Audio Connection Kit for access to audio and MIDI, usable effects include LADSPA, LV2 and custom Luppp effects. There is a GUI built using Gtkmm, however it is encouraged to use a hardware controller for faster and more intuitive interaction.

Keywords

Live performance, real-time looping, audio production

1 Introduction

Luppp is the "Luppp Untuitive Personal Performing Program", it's a tool to be used in a live scenario to creatively produce music. The fact that it is to be used live imposes certain requirements like low latency input-output and real-time effect rendering.

This paper gives an overview of the structure of the audio processing engine in Luppp and how it delegates tasks to different threads, as well as how communication between the threads is achieved. The loading of effects is discussed, as is the user interface and its design.

2 Engine

2.1 Dependencies

The core engine depends on a number of different libraries:

- JACK [1]
- Gtkmm [2]
- libsndfile [3]
- Fluidsynth [4]
- libconfig [5]
- SUIL [6]
- LILV [7]

2.2 Overview of components

The core of the program consists of a processing engine. Its function is to process all input events and produce its output within a certain time frame. Its primary inputs are MIDI messages and events from the GUI, while its primary output is audio data that is written to the JACK ports. It also facilitates the display of data on screen in the user interface. This involves transporting data from the real-time engine thread to the user interface thread to be displayed.

2.3 Engine Events

The `EngineEvent` class is a class that represents any action that the engine can perform. For each action there is a function to set the right parameters in the class, while reading data from the class is done through the use of public data members.

2.4 Events and ring-buffers

To communicate between the various threads in the engine in a lock-free manner ring-buffers are used. Pointers to instances of the `EngineEvent` class are passed through the ring-buffers, which allows the use of these events from a real-time thread. The GUI thread creates a pool of `EngineEvent` instances while the real-time JACK thread can pull event pointers from this queue, sending the messages without having to allocate them. This causes minimal overhead in the real-time thread while a background thread occasionally polls the ring-buffer for write space, and fills it with new instances.

2.5 State store

The `StateStore` class holds the state of all other components of the engine. It is a centralized location for all data like `AudioBuffer`s, `AudioSink`s, `BufferAudioSourceState`s etc. The real-time thread takes `EngineEvent`s, and writes the data contained within it to the `StateStore`. This state data is later requested by each engine component when it is required to do its processing.

The requesting of state data is done using unique identification numbers for each state instance. When an instance of a class that needs a state is created, it automatically increments its own ID number, while also adding a new state with its own ID number to the `StateStore`. It can now request the state with its own ID number from `StateStore` and it will receive its own state data.

2.6 Audio processing

The `AudioTrack` is the main structure in audio processing, with the track's `AudioSource` providing a channel of mono audio, a list of `Effect`s being applied to this signal, and finally the `AudioSink` passes the produced data to the `Mixer` class. The `Mixer` then takes each track's output, applies master effects and writes the data to the master output ports.

Meta data is available to each element inside an `AudioTrack` so `AudioSource`, `Effect` and `AudioSink` class instances can make informed decisions how to process the next block of audio based on the engines current state.

2.6.1 Audio sources

The `AudioSource` base class is defined as the start of the processing chain. It is subclassed to provide varying functionality: The `BufferAudioSource` reads samples from an `AudioBuffer`, and then writes them to the `AudioTrack`'s buffer. In the case of instrument sources like Fluidsynth or a LV2 synth, it gathers incoming MIDI data from JACK or a playing MIDI clip and generates samples and writes them to the `AudioTrack`'s buffer.

2.6.2 Effects and plugins

The `Effect` base class is defined as a class that requests its state from the StateStore using its own ID, sets its new parameters, and performs some processing on an `AudioTrack`'s buffer. Currently there is no latency compensation done for any of the effects, however the effects that are available have been extensively tested to comply both with the real time requirements of the program, as well as not inducing unacceptable latency to the track.

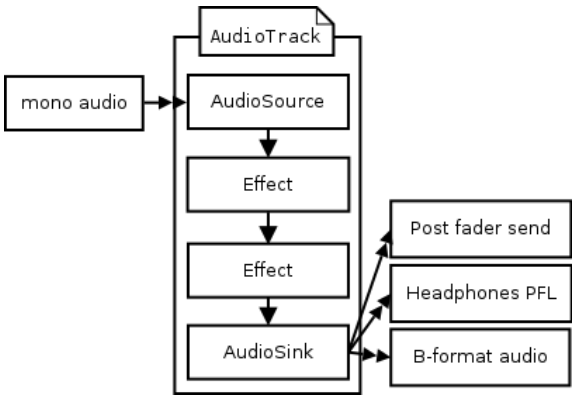


Figure 1: `AudioTrack` block diagram

This base class can be subclassed to host any kind of audio effect, the only constraint is that the number of input samples must match the number of output samples. LADSPA[8] and LV2[9] effects are implemented, however theoretically any type of plugin format could be supported.

2.6.3 Audio sinks

The `AudioSink` class is the base class that represents the end of the processing chain. Every track has an `OutputAudioSink`, whose purpose it is to mix the mono input samples into ambisonic B-format, while also copying the samples to the post-fade send buffer and headphones pre-fade listen buffers.

2.6.4 Headphones PFL

Each track has headphones pre-fade listen functionality, which allows the user to preview tracks before actually mixing them into the master output. This functionality is achieved by writing the mono signal that is passed to the `AudioSink` to the headphones buffer in `Mixer`. After each track has been processed, `Mixer` will write the headphones buffer to the JACK port.

2.6.5 Post-fade Sends

Each track has a post-fade send. All tracks are summed together into a single buffer, which is then written to a JACK audio port. This JACK port can be connected to arbitrary JACK clients to add effects to the signal. Although a mono signal is sent, the return ports are B-format to sup-

port scenarios where an ambisonic effect is used. The returned signal is then mixed into the master bus.

2.7 Real-time recording

In order to record an arbitrary length of audio an arbitrary length array is needed to store it, and due to real-time constraints we cannot create one such array in the real-time thread. This problem was solved by writing all incoming audio to a ring-buffer, and allowing the GUI thread to create the actual buffers that are later used in the engine. The buffer is passed to the real-time thread using an event and ring-buffer as described earlier. This allows the user to record any length of audio in real-time without any non-real-time operations occurring.

3 User Interface

The user interface for Luppp is geared towards live use, and hence it makes as little use of popup dialogs and extra windows as possible. The main view is laid out with tracks oriented vertically, and song-parts horizontally.

3.1 Clip view

The main part of the GUI is dedicated to the clip view, which selects the currently playing `AudioBuffer`. To load a buffer into a clip we right click on the desired clip, and we will be presented with a file-chooser dialog. It is also possible to drag-and-drop samples onto the clip using the side-pane file browser. Recording audio from a JACK audio input port is also possible, just record enable the track and click on the clip.

3.2 Track view

The lower part of the window shows the state of the currently selected track. Its `AudioSource` is on the left while any effects are listed to the right. The main parameters of effect can be modified using the mouse, by click-and-drag in the graph area. The X and Y axis of the graph are mapped to the two most used parameters of that effect type.

3.3 Master track

The master track provides feedback as to what scene is currently playing, the rotation and elevation values of the B-format ambisonic output and headphones volume levels.

3.4 Side-pane browser

On the left of the UI there is a browser that allows the previewing of samples, instruments and effects. Drag-and-drop operations from the samples to the clips is supported, as is dropping effects and instruments onto the track view. The browser filters its contents based on file extension, samples must have a .wav extension. The Instrument and Effect lists are hard coded.

4 Meta data files

To inform the engine how long an audio loop is in musical beats, meta data files are used. The files are read by libconfig, and can provide information like the length of the loop in beats, or the key of a loop. This gives the engine a chance to process the audio in a musically meaningful way, like stretching drum loops to match the tempo of the currently playing material.

4.1 Sample meta data

When loading a sample, Luppp attempts to open a file called `lupppSamplePack.cfg` from the same directory as the sample is located. If this file exists, we iterate trough all information in the file, and check if any of the metadata is relevant to the sample that we want to load. If meta data exists about the sample, we set properties on the `AudioBuffer` instance that the sample will get loaded into. These properties can be its number of beats (in the musical sense), or what musical key its in, etc.

Example `lupppSamplePack.cfg` file:

```
luppp :
{
  samplePack:
  {
    name = "SamplePackName";
    numSamples = 1;

    s0:
    {
      name = "sampleName.wav"
      numBeats = 16;
    }
  }
}
```

Figure 2: Luppp user interface showing the clip selector, track effects and file browser.

5 Future work

In future I would like to implement more features for Luppp, particularly around easier access to core Luppp functionality from controllers. Also on the todo list is MIDI sequencing functionality which will allow a faster workflow while creating new musical ideas.

Some improvements can be made in performance, and multi-threading the actual audio processing path in the engine is a possibility.

5.1 MIDI sequencing and editing

It is intended to support the looping of MIDI clips in a similar fashion to Seq24. MIDI clips will be shown in the user interface just like audio clips, and clicking on one will select it to be shown in the MIDI editor.

The MIDI editor will be as streamlined as possible, with minimal features and as fast a workflow as is possible. Some initial sketches of possible workflows have been made, however this functionality is still in its planning stages.

5.2 Immutable Engine

For performance reasons it is planned to make `AudioBuffer` instances immutable, so that they can be used in the real-time thread as well as shown in the GUI without expensive copying. Plotting the audio clips in the GUI will become an easier task, as we can safely access the `AudioBuffer`s from multiple threads due to its immutability.

To modify an `AudioBuffer` a new one will be created in the GUI thread, and that new instance will be swapped into engine. This modified buffer will keep the same unique ID number, so that all other parts of the engine will automatically use to the updated buffer.

5.3 Sidechain compression

In order to fully support sidechain compression between tracks, it will be necessary to change the direction of sample requesting in an `AudioTrack`. This is due to the fact that for sidechain compression audio data from another track is needed, and

hence a method to request the processing of another `AudioTrack` than the current one is needed.

If each `SidechainCompressor` instance has a pointer to a `SidechainSource`, it can request the source to produce the needed samples, and then continue to process its own `AudioTrack`'s audio based on the sidechain buffer of samples. This also involves implementing a system whereby each `Effect` or `AudioSource` can be marked as finished processing, so that we avoid re-calculating parts of a track that has already been processed because of the call to the `SidechainSource`.

Essentially this makes the `SidechainSource` a cache for audio samples, which can be requested to process the next block of audio either from the `AudioTrack` that is it part of, or another `AudioTrack` which needs this `SidechainSource`'s output to complete its own processing.

5.4 Additional effect classes

Initial work to host Pure Data patches or CSound instruments as `AudioSource`s or `Effect`s has went well, although some design is needed with regards to real-time constraints of the effects and loading files. Another issue that needs to be addressed is that non real-time safe operations might be carried out by user defined patches.

VAMP plugins [10] will hopefully be supported in the future, to allow the user analyze existing material, and work with a higher level of control over the music. If more meta data can be provided to the Luppp engine, more high level features can be implemented as more data exists for use during the processing cycle. I feel this is quite an important aspect of Luppp to develop, as it gives the user more scope for creativity and inspiration instead of hindering the workflow with mathematical details of the processing that is going on behind the GUI.

6 Conclusions

While Luppp is currently in a usable state, there are some known issues that must be dealt with. I would like to rethink the method of communication between the `Engine` and the `StateStore` for the user interface. The way that the user interface widgets are stored in the gui class can also be much improved. Some design issues like the concept of tracks and how ID number related to tracks could also be upgraded, as currently some functionality is hindered by how the ID's are set.

7 Acknowledgments

I would like to thank the entire LAD community for all that I have learnt through reading their code, asking questions on the LAD mailing list as well as on IRC.

Thanks also to all those who have helped me in any way while I have been working on Luppp for providing suggestions, testing buggy code and just chatting about the project in general.

8 References

[1] JACK Audio Connection Kit
`http://www.jackaudio.org`

[2] Gtkmm, official C++ interface for the popular GUI library GTK+
`http://www.gtkmm.org`

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

[4] Fluidsynth, a real-time software synthesizer based on the SoundFont 2 specifications
`www.fluidsynth.org/`

[5] libconfig, a simple library for processing structured configuration files
`http://www.hyperrealm.com/libconfig`

[6] SUIL, a lightweight C library for loading and wrapping LV2 plugin UIs
`http://drobilla.net/software/suil`

[7] LILV, a library to make the use of LV2 plugins as simple as possible
`http://drobilla.net/software/lilv`

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

[9] LV2, a plugin standard for audio systems
`http://lv2plug.in/trac/`

[10] VAMP, an audio processing plugin system for plugins that extract descriptive information from audio data
`http://vamp-plugins.org/`

Csound as a Real-time Application

Typical Problems and Solutions for the Use of Csound in a Live Context

Joachim HEINTZ
Incontri - Institute for new music
HMTM Hannover
Emmichplatz 1, 30175 Hannover, Germany
joachim.heintz@hmtm-hannover.de

Abstract

This article discusses the usage of Csound for live electronics. Embedding Csound as an audio engine in Pd is shown as well as working in CsoundQt for live performance. Typical problems and possible solutions are demonstrated as examples. The pros and contras of both environments are discussed in comparison.

Keywords

Csound, Pd, Live Electronics.

1 Introduction

Csound [1] is well known as one of the most powerful and approved audio libraries, but its main programming model stems from a non-real-time approach: 'instruments' are called for specified durations by a 'score'. This is perfectly suited for tape compositions, and there is still a predominant opinion that Csound is good for fixed media compositions but cannot be used in live situations. Or at least, that it is a pain to do so.

Indeed, if the user does not want to hit a dead end when using Csound in a real-time situation, the architecture and event structure of Csound needs to be carefully considered. It is the aim of this contribution to exemplify how Csound can be used successfully for live performance.

As Csound has no native user interface, it can be embedded in different hosts. Each host application offers different advantages and its own manner of interfacing with Csound. In the field of free software, the use of Csound in Pd [2], and the use of Csound in CsoundQt [3] are probably the most interesting choices.[1] The examples given here utilize these two hosts. They will describe the real-time transposition of a live input, which would represent a typical real-time transformation.

2 Csound in Pd: Building a polyphonic real-time transposer

Combining Csound's DSP power with Pd's flexibility and interfacing should be a really nice idea, both for Csound and Pd users. Victor Lazzarini's *csoundapi~* external for Pd[2] offers this connection. Strange enough, it is much fewer used than it would be expected to – perhaps due to a certain lack of documentation. The first example uses Csound as a polyphonic transposition machine inside Pd and discusses some typical issues.

2.1 Running the *csoundapi~* object

It is beyond the scope of this article to describe the installation of the *csoundapi~* external in Linux, MS Windows and Apple OSX. Descriptions can be found, for instance, in the Csound Floss Manual.[3]

Once the *csoundapi~* object has been installed, it is available in Pd and refers to a Csound file (*.csd).

[1] For commercial software, the use of Csound in MaxMsp is well known and perfectly documentated by Davis Pyon [4]. As any Max object itself can be embedded in Ableton Live via "Max4Live", it is possible to call Csound in Ableton Live, too. This has been popularized by Richard Boulanger and partners as "Csound4Live" [5].

[2] The sources can be found in the "frontends" directory of the csound code [1] or in the git repository at http://csound.git.sourceforge.net/git/gitweb-index.cgi

[3] www.flossmanuals.net/csound/index

Now audio and control data can be sent from Pd to Csound and back.

2.2 Building a four voice transposition instrument

In this example, audio from a microphone is received by Pd and passed to Csound which then creates the four voice transposition which is then sent back to Pd as stereo mix. Csound implements these transpositions by first converting the received audio into a frequencydomain signal and then creating the four pitch shifted signals. These four signals are converted back into the timedomain and panned and mixed to create a stereo output.[4] This stereo signal is finally passed back into Pd which in turn sends it to the speakers.

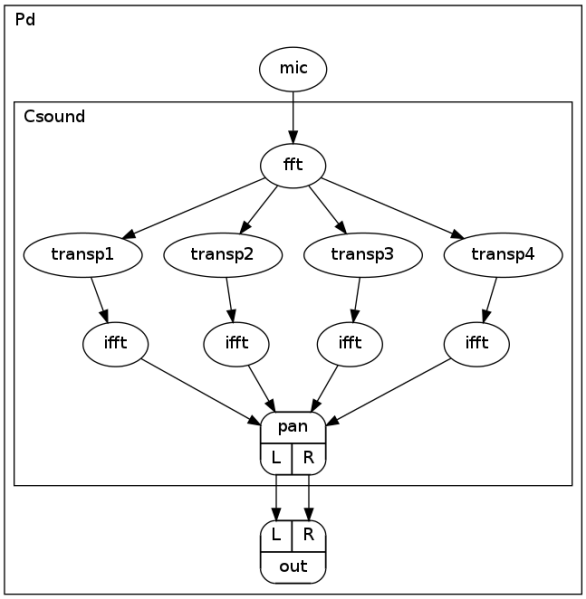


Figure 1: Scheme of embedding Csound in Pd for a four-voice live transposition

The related Csound file is very simple:

```
  <CsoundSynthesizer>
  <CsInstruments>
  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

    instr transp
;input from pd
  aIn       inch      1
;transform to frequency domain
  fIn       pvsanal   aIn, 1024, 256, 1024, 1
;transposition
  fTransp1  pvscale   fIn, cent(-600)
  fTransp2  pvscale   fIn, cent(-100)
  fTransp3  pvscale   fIn, cent(200)
  fTransp4  pvscale   fIn, cent(500)
;back to time domain
  aTransp1  pvsynth   fTransp1
  aTransp2  pvsynth   fTransp2
  aTransp3  pvsynth   fTransp3
  aTransp4  pvsynth   fTransp4
;panning
  aL1, aR1  pan2      aTransp1, .1
  aL2, aR2  pan2      aTransp2, .33
  aL3, aR3  pan2      aTransp3, .67
  aL4, aR4  pan2      aTransp4, .9
  aL        =         (aL1+aL2+aL3+aL4)/3
  aR        =         (aR1+aR2+aR3+aR4)/3
;output to pd
            outs      aL, aR
    endin

  </CsInstruments>
  <CsScore>
  i "transp" 0 99999
  </CsScore>
  </CsoundSynthesizer>
```

The whole Pd patch looks like this:

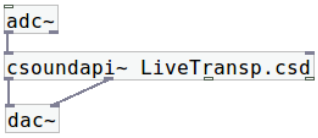


Figure 2: Pd patch for four-voice live transposition embedding Csound via csoundapi~

2.3 Sending and receiving control data

For a fuller interaction between Pd and the embedded Csound it is normally desirable to also pass control signals between Pd and Csound. The transposition values in the given example might have to be controlled from Pd instead of being fixed:

[4] A quadrophonic output redirection into Pd would also be possible.

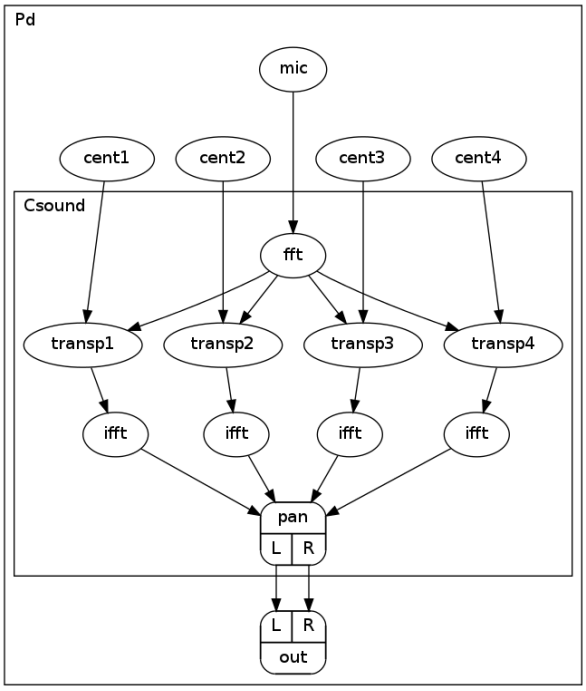


Figure 3: Sending cent values from Pd to Csound

From Pd's point of view, the cent values are a "message" to be sent to Csound. From Csound's point of view, the cent values are "control signals" which are received on certain "control channels".

The *csoundapi~* object understands the message "chnset ...". A Pd message box containing "chnset cent1 -600" means: send the value -600 on the software channel named "cent1".

In Csound, the related message needs to be received by a *chnget* opcode.[5] The instrument code has to be changed slightly now, as follows:

```
  ...
    instr transp
;audio input from pd
  aIn       inch      1
;control input from pd
  kCent1    chnget    "cent1"
  kCent2    chnget    "cent2"
  kCent3    chnget    "cent3"
  kCent4    chnget    "cent4"
;transform to frequency domain
  fIn       pvsanal   aIn, 1024, 256, 1024, 1
;transposition
  fTranspl  pvscale   fIn, cent(kCent1)
  fTranspl  pvscale   fIn, cent(kCent2)
  fTranspl  pvscale   fIn, cent(kCent3)
  fTranspl  pvscale   fIn, cent(kCent4)
  ...
```

And this is the Pd patch, with some choices to send values:

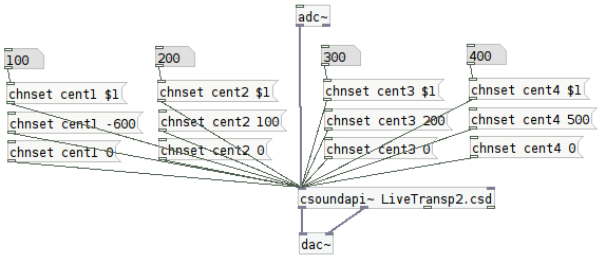


Figure 4: Pd sending cent values to Csound

2.4 The "just once" problem

It is worth to have a closer look at what is actually happening in the example above. It looks easy, and it works, but actually two languages are communicating with each other in a strange way. In this case, they understand each other, but this is pure chance if the differences have not been understood by the user.

Pd differentiates between messages and audio streams.[6] Messages are sent just once, while audio streams are sending data continuously, in the sample rate. Csound has no type for sending something in realtime "just once". If something is sent as a control value to Csound, like the cent values in the example above, Csound considers this as a control signal, i.e. a continuous stream of control data.[7]

In this case, it works. Pd sends a new cent message just once, and Csound interpretes this as a repeated control message. It works, because the *pvscale* opcode needs a control-rate input. But if the user wants to trigger something "just once", they

[5] The alternative choice is to send messages as "control ..." from Pd to Csound, and receive them with the *invalue* opcode in Csound.

[6] This is the same in MaxMsp, and is the well known difference between an object with a tilde and an object without a tilde. For instance, the object '*' multiplies two numbers: just once, when the left inlet has received a number as message. The object '*~' multiplies two audio streams (or an audio stream and a number) continuously.

[7] This stream of control data has a lower rate than the audio rate, depending on the *ksmps* value. If ksmps=32, one control sample is used for 32 audio samples.

have to ensure that Csound does not repeat the message from Pd all the time.

As a simple example, the user may want to trigger a short beep each time a message box with a '1' has been pressed:

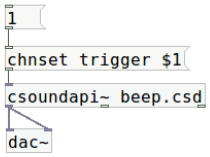


Figure 5: Sending a '1' message from Pd to Csound

It looks straightforward to code the beep.csd in this way:

```
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr master ;wrong!
kTrigger  chnget    "trigger"
 if kTrigger == 1 then
          event     "i", "beep", 0, 1
 endif
  endin

  instr beep
aBeep     oscils    .2, 400, 0
aEnv      transeg   1, p3, -6, 0
          out       aBeep * aEnv
  endin

</CsInstruments>
<CsScore>
i "master" 0 99999
</CsScore>
</CsoundSynthesizer>
```

A master instrument would receive the data on the channel called 'trigger' from Pd, and call a subinstrument "beep" each time a '1' has been received. That's the wish of the user. But what really happens: the *kTrigger* variable will always be set to '1', because there is no other message coming from Pd, and so Csound will trigger a beep not "just once", but once at each control cycle, in this case 44100/32 times per second ...

Two things have to be done here: Pd must send a '0', if the message box has not been pressed. The following patch sends, when hitting the 'bang' first a '1' and then after 10 milliseconds a '0':[8]

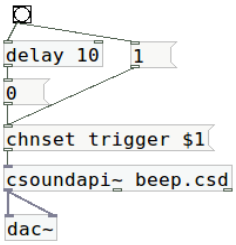


Figure 6: Pd sending '1' followed by '0' to Csound

On the Csound site, just the first '1' received after any zeros should trigger a beep. So Csound has to ignore the repetitions. This is done by the following code in the master instrument:

```
  instr master ;now correct
kTrigger  chnget    "trigger"
kNewVal   changed   kTrigger
 if kTrigger == 1 && kNewVal == 1 then
          event     "i", "beep", 0, 1
 endif
  endin
```

This opcode returns '1' exactly in the control cycle when a new value of *kTrigger* has been received. So the if-clause has to ask whether both, *kNewVal* and *kTrigger* equal 1. Only in this case the subinstrument is to be triggered.

2.5 Results

The use of Csound for realtime applications in Pd can be considered as easy and straightforward in general. The major difficulties come from the the different signal paradigms in both languages. Special care has to be taken by the user for "just once" cases, which are common in Pd but need cautious coding in Csound.

[8] Using a trigger in Pd which sends a '1' from the right output, followed by a '0' from the left output will not work, because Csound will just receive the '0'.

3 Presets, software channels and more: An extended live instrument in CsoundQt

CsoundQt has been written and developed by Andrés Cabrera since 2008.[9] It is now the most widely used frontend for Csound. It uses the Qt toolkit[10] for building a graphical user interface.

CsoundQt offers all necessary tools to work with live electronics in Csound, without using another host application. But how can Csound be trimmed to support presets which are necessary for each extended live electronic context? An example is given which uses a midi keyboard as instrument for controlling both some real-time processings of a microphone input, and electronic sounds generated in real time.

3.1 Presets

A preset is a general configuration in a live electronic application which defines the meaning of certain input signals in the context of this preset environment. For instance, if the midi key number 60 has been pressed in the context of preset 1, it may mean "open the live input with a fade in". If the same key has been pressed in preset 2, it may mean something totally different, for instance "play a percussive sound".

As CsoundQt has native widgets, these widgets can be used to represent a preset.[11] Csound will then look at the value of this widget, and will decide what to do at a certain midi input. This is the general scheme:

As "triggering an event" in Csound usually means "calling an instrument", the Csound code looks like this:

```
   instr midi_receive
;getting the midi note number
  iNotNum    notnum
;getting the actual preset number
```

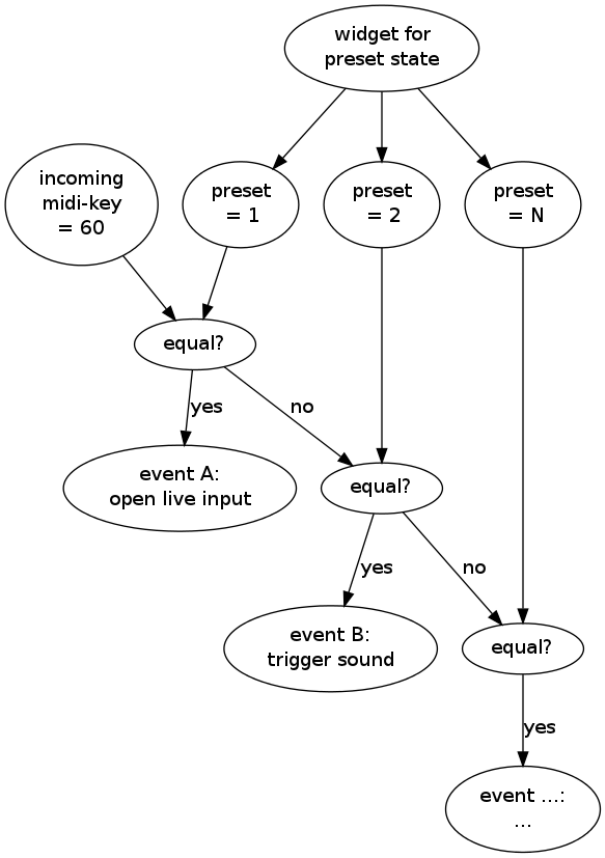


Figure 7: General preset scheme

```
  iPreset    =          i(gkPreset)
;bindings for preset 1
   if iPreset == 1 then
    if iNotNum == 60 then
             event_i "i", "live_fade_in", 0, 1
    elseif iNotNum == ... then
             event_i    ...
    endif
;bindings for preset 2
   elseif iPreset == 2 then
    if iNotNum == 60 then
            event_i "i", "trigger_sound", 0, 1
    elseif iNotNum == ... then
             event_i    ...
    endif
;bindings for preset ...
   elseif ...
   ...
   endif
    endin
```

The *gkPreset* variable which holds the status of the preset given by the preset widget, is created in an "always on" instrument. This instrument should be the first of all the instruments,[12] and its definition should contain a line like this ("preset" is a string which defines the name of the widget's channel):

[9] The name has been changed from QuteCsound to CsoundQt after discussions at the Csound Conference in Hannover in october 2011.

[10] http://qt.nokia.com

[11] CsoundQt also offers a possibility to store widget states as presets. This is very useful in many cases, but not sufficient here, because here the presets do not affect just widgets, but also determine the effect of a Midi key pressed, a parameter being set, and more.

[12] Technically speaking: the instrument with the smallest number.

```
gkPreset  invalue   "preset"
```

By this, the current preset state is received from the spin box widget, and sent as the global variable *gkPreset* to each instrument. The widget itself can be changed by the user either via the GUI or via any input event, for example a reserved midi note for increasing, and another note number for decreasing the preset number. Assuming the reserved midi note for increasing is 72, and for decreasing 48, the code looks like this:[13]

```
if iNotNum == 72 then
          outvalue   "preset", iPreset + 1
elseif iNotNum == 48 then
          outvalue   "preset", iPreset - 1
endif
```

The method proposed here for handling presets offers the user all flexibility. All events are embedded in instruments which are triggered if a midi key is received in the context of a particular preset. General conditions for a new preset can be set in a similar way as shown in chapter 2.4 with the *changed* method. For instance, if the live microphone is to be open at the beginning of preset 1, but closed at the beginning of preset 2, the code in the master instrument could be:

```
gkPreset  invalue   "preset"
kNewPrest changed   gKPreset
 if kNewPrest == 1
  if gkPreset == 1 then
kLiveVol  =         1 ;mic open
  elseif gkPreset == 2 then
kLiveVol  =         0 ;mic closed
  endif
 endif
```

3.2 Software busses for control and audio data

Many situations need a flexible interchange of control data. Suppose the user wants to transpose the live input first in four voices with the cent values -449, -315, 71 and 688, and then in the next bar with the cent values -568, -386, 428 and 498. Both events – activating the four voice transposition and changing the values – are to be triggered by two different midi keys, for instance 60 and 62.

This is a typical case for the use of internal software busses. Four audio signals are set. For each of them a software control bus is created, holding the transposition value. The first instrument sets the initial values. When the second instrument is triggered, the control busses are set to the new values (figure 8).

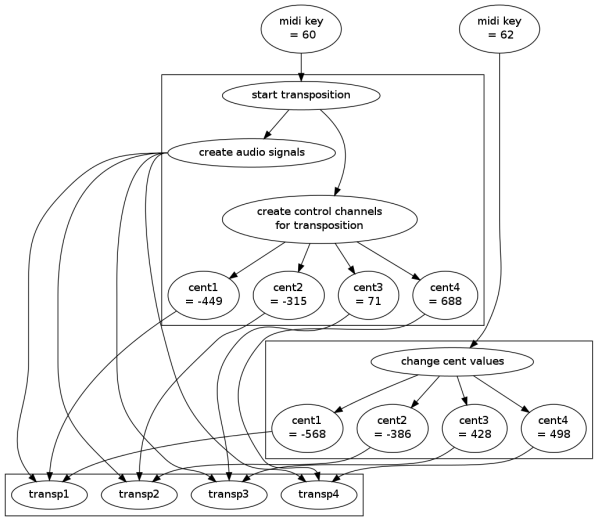


Figure 8: Working with software channels to change transposition values

This is the related Csound code:

```
    instr set_transp
;create software busses
             chn_k      "transp1", 3
             chn_k      "transp2", 3
             chn_k      "transp3", 3
             chn_k      "transp4", 3
;set initial values
             chnset     -449, "transp1"
             chnset     -315 "transp2"
             chnset     71, "transp3"
             chnset     688, "transp4"
;receive the values from the software busses
  kCent1     chnget     "transp1"
  kCent2     chnget     "transp2"
  kCent3     chnget     "transp3"
  kCent4     chnget     "transp4"
  ;receive the live audio input
  aLvIn      chnget     "live_in"
;perform fourier transform
  fLvIn      pvsanal    aLvIn,1024,256,1024,1
;perform four voice transposition
  fTp1       pvscale    fLvIn, cent(kCent1)
  fTp2       pvscale    fLvIn, cent(kCent2)
  fTp3       pvscale    fLvIn, cent(kCent3)
  fTp4       pvscale    fLvIn, cent(kCent4)
;resynthesize
  aTp1       pvsynth    fTp1
```

[13] The instrument is the same as the one above called "midi_receive". This instrument is triggered directly by a midi note on message. It works during Csound's initialization pass, which is in some cases another option for the "just once" problem.

```
  aTp2        pvsynth     fTp2
  aTp3        pvsynth     fTp3
  aTp4        pvsynth     fTp4
;add and apply envelope
  aTp         =           aTp1+aTp2+aTp3+aTp4
  kHul        linsegr     0,.3,1,p3-0.3,1,.5,0
  aOut        =           aTp * kHul
;mix to global audio bus for live out
              chnmix      aOut, "live_out"
    endin

    instr change_transp
              chnset      -568, "transp1"
              chnset      -386, "transp2"
              chnset      428, "transp3"
              chnset      498, "transp4"
    endin
```

As can be seen from this example, the software busses are not just used for control but also for audio signals. The live audio input is sent to a channel called "live_in". The instrument set_transp gets the live input via the line

```
  aLvIn       chnget      "live_in"
```

After processing, the live transposed signals are sent to an audio bus called "live_out". As other instruments may also send audio to this bus, *chnmix* is used instead of *chnset*:

```
              chnmix      aOut, "live_out"
```

Software channels are the solution to many different situations in programming live electronics: mixing, routing, interchanging of values. The *chn* opcodes offer a flexible and reliable system to work with them in Csound.

3.3 Performance tweakings

Csound's performance for live applications depends mainly on its vector and buffer sizes.[14] As mentioned above, the *ksmps* constant defines the internal vector size. A value of ksmps=32 should be adequate for most live situations. The software and hardware buffer sizes must not be kept at Csound's defaults, but should be set to lower values to avoid an audible latency.[15] CsoundQt offers an easy way to adjust them in the configure panel. These are fair values:

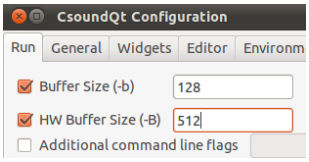


Figure 9: Adjusting the buffer sizes in CsoundQt's configuration panel

In future versions of Csound, the use of multiple cores and threads, will further improve the performance.[16] This would also be adjusted in CsoundQt's configure panel.

In addition to these general Csound adjustments, there are some performance tweaks particularly related to CsoundQt. The python callback, especially, must be disabled for the best live performance:

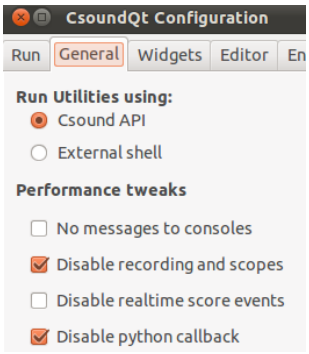


Figure 10: Performance tweaks in CsoundQt

3.4 Results

CsoundQt offers a nice graphical user interface for the use of "pure" Csound with standard widgets for live electronics. Instead of changing between different platforms, the users have an easier

[14] There are some opcodes which are not suited for live in Csound. Usually they have a fast and modern alternative. Especially the old pv (phase vocoder) opcodes should in pratical use be substituted by the excellent pvs (phase vocoder streaming) opcodes: www.csounds.com/manual/html/SpectralRealTime.html

[15] More information can be found at www.csounds.com/manual/html/UsingOptimizing.html

[16] Thanks to the work of John ffitch and others, the new parser for Csound is at the time of writing this article (end of the year 2011) in the process of becoming the default. Amongst other improvements, it will offer multithreading.

workflow than using Csound in Pd. (When making changes to a Csound file running within Pd it is necessary to save the Csound file and to reset the *csoundapi~* object that uses it for those changes to take effect.) Users can add their own functions ("user defined opcodes") or integrate any from the repository,[17] for instance for simplifying the use of the ascii keyboard or for easily recording and playing buffers.

On the user's wish list for using CsoundQt live is certainly an increased array of widgets, for example an audio meter, a midi keyboard widget and an interactive table editor. The possibility of creating more than one widget panel would certainly make CsoundQt much more flexible for use live.[18]

4 Conclusion

Csound can be used perfectly for any live application. The usage of Csound in Pd is very easy. It offers many new possibilities for Pd users, and an easy connection with any external devices (game pads, arduino boards and more) for Csound users.

The usage of CsoundQt for live applications offers a clean and pleasant internal graphical interface and a comfortable workflow to the user. Complex features like presets and software busses can be programmed easily. However, some additional widgets and GUI options might be desireable.

Ultimately, whether a user ought to use Csound live within Pd or purely with CsoundQt as the front-end, will depend on their personal preferences and the situation.

So it depends on the users, their preferences and situations, whether they prefer working with Csound for live in Pd or in CsoundQt.

5 Acknowledgements

Thanks to Anna, Alex, Andrés and Iain for reading the manuscript.

[17] www.csounds.com/udo

[18] Although not covered in this article, it should be
http://www.youtube.com/watch?v=O9WU7DzdUmE
http://www.youtube.com/watch?v=Hs3eO7o349k
http://www.youtube.com/watch?v=yUMzp6556Kw

References

[1] Csound: http://csound.sourceforge.net (with many links to related sites)

[2] Pd: http://puredata.info

[3] CsoundQt: http://qutecsound.sourceforge.net

[4] csound~ (a Max external to run Csound): www.davixology.com/csound~.html

[5] Csound4Live (Csound for use in Ableton Live via csound~ [4] and the Max4Live bridge): www.csoundforlive.com

Csound for Android

Steven YI and **Victor LAZZARINI**
National University of Ireland, Maynooth
{steven.yi.2012, victor.lazzarini}@nuim.ie

Abstract

The Csound computer music synthesis system has grown from its roots in 1986 on desktop Unix systems to today's many different desktop and embedded operating systems. With the growing popularity of the Linux-based Android operating system, Csound has been ported to this vibrant mobile platform. This paper will discuss using the Csound for Android platform, use cases, and possible future explorations.

Keywords

Csound, Android, Cross-Platform, Linux

1 Introduction

Csound is a computer music language of the MUSIC N type, developed originally at MIT for UNIX-like operating systems[Boulanger, 2000]. It is Free Software, released under the LGPL. In 2006, a major new version, Csound 5, was released, offering a completely re-engineered software, which is now used as a programming library with its own application programming interface (API). It can now be embedded and integrated into several systems, and it can be used from a variety of programming languages and environments (C/C++, Objective-C, Python, Java, Lua, Pure Data, Lisp, etc.). The API provides full control of Csound compilation and performance, software bus access to its control and audio signals, as well as hooks into various aspects of its internal data representation. Several frontends and composition systems have been developed to take advantage of these features. The Csound API has been described in a number of articles [Lazzarini, 2006], [Lazzarini and Piche, 2006] [Lazzarini and Walsh, 2007].

The increasing popularity of mobile devices for computing (in the form of mobile phones, tablets and netbooks), has brought to the fore new platforms for Computer Music. Csound has already been featured as the sound engine for one of the pioneer systems, the XO-based computer used in the One Laptop per Child (OLPC) project [Lazzarini, 2008]. This system, based on a Linux kernel with the Sugar user interface, was an excellent example of the possibilities allowed by the re-engineered Csound. It sparked the ideas for a Ubiquitous Csound, which is steadily coming to fruition with a number of parallel projects, collectively called the *Mobile Csound Platform* (MCP). One such project is the development of a software development kit (SDK) for Android platforms, which is embodied by the CsoundObj API, an extension to the underlying Csound 5 API.

Android [1] is a Linux-kernel-based, open-source operating system, which has been deployed on a number of mobile devices (phones and tablets). Although not providing a full GNU/Linux environment, Android nevertheless allows the development of Free software for various uses, one of which is audio and music. It is a platform with some good potential for musical applications, although at the moment, it has a severe problem for realtime use that is brought by a lack of support for low-latency audio.

In this article we will discuss Csound usage on Android. We will explore the CsoundObj API that has been created to ease developing Android applications with Csound, as well as demonstrate some use cases. Finally, we will look at what Csound uniquely brings to Android, with a look at the global Csound ecosystem and how mobile apps can be integrated into it.

[1] `http://www.android.com`

2 Csound for Android

The Csound for Android platform is made up of a native shared library (libCsoundandroid.so) built using the Android Native Development Kit (NDK)[2], as well as Java classes that are compilable with the more commonly used Android Dalvik compiler. The native library is linked using the the object files that are normally used to make up the libcsound, libcsnd, and libsndfile[3] libraries that are found part of the desktop version of Csound. The Java classes include those commonly found in the csnd.jar library used for desktop Java-based Csound development, as well as unique classes created for easing Csound development on Android.

The SWIG[4] wrapping used for Android contains all of the same classes as those used in the Java wrapping that is used for desktop Java development with Csound. Consequently, those users who are familiar with Csound and Java can transfer their knowledge when working on Android, and users who learn Csound development on Android can take their experience and work on desktop Java applications. However, the two platforms do differ in some areas such as classes for accessing hardware and different user interface libraries. To help ease development, a CsoundObj class was developed to provide out-of-the-box solutions for common tasks (such as routing audio from Csound to hardware output). Also, applications using CsoundObj can be more easily ported to other platforms where CsoundObj is implemented (i.e. iOS).[5]

One of the first issues arising in the development of Csound for Android was the question of plugin modules. Since the first release of Csound 5, the bulk of its unit generators (opcodes) were provided as dynamically-loaded libraries, which resided in a special location (the OPCODEDIR or OPCODEDIR64 directories) and were loaded by Csound at the orchestra compilation stage. However, due to the uncertain situation regarding dynamic libraries (not only in Android but also in other mobile platforms), it was decided that all modules without any dependencies or licensing issues could be moved to the main Csound library code. This was a major change (in Csound 5.15), which made the majority of opcodes part of the base system, about 1,500 of them, with the remaining 400 or so being left in plugin modules. The present release of Csound for Android includes only the internal unit generators. Another major internal change to Csound, which was needed to facilitate development for Android, was the move to use core (memory) files instead of temporary disk files in orchestra and score parsing.

Audio IO has been developed in two fronts: using pure Java code through the AudioTrack API provided by the Android SDK and, using C code, as a Csound IO module that uses the OpenSL API that is offered by the Android NDK. The latter was developed as a possible window into a future lower-latency mode, which is not available at the moment. It is built as a replacement for the usual Csound IO modules (PortAudio, ALSA, JACK, etc.), using the provided API hooks. The Csound input and output functions, called synchronously in its performance loop, pass a buffer of audio samples to the DAC/ADC using the OpenSL enqueue mechanism. This includes a callback that is used to notify when a new buffer needs to be enqueued. A double buffer is used, so that while one half is being written or read by Csound, the other is enqueued to be consumed or filled by the device. The code fragment below in listing 1 shows the output function and its associated callback. The OpenSL module is the default mode of IO in Csound for Android. Although it does not currently offer low-latency, it is a more efficient means of passing data to the audio device and it operates outside the influence of the Dalvik virtual machine garbage collector (which executes the Java application code).

The AudioTrack code offers an alternative means accessing the device. It pushes/retrieves input/output frames into/from the main processing buffers (spin/spout) of Csound synchronously at control cycle intervals. It is offered as an option to developers, which can be used for instance, in older versions of Android without OpenSL support.

[2] http://developer.android.com/sdk/ndk/index.html

[3] http://www.mega-nerd.com/libsndfile/

[4] http://www.swig.org

[5] There are plans to create CsoundObj implementations for other object-oriented desktop development languages/-platforms such as C++, Objective-C, Java, and Python, but at the time of this writing, CsoundObj is only available in Objective-C for iOS.

3 Application Development using CsoundObj

Developers using the CsoundObj API will essentially partition their codebase into three parts: application code, audio code, and glue code. The application code contains the standard Android code for creating applications, including such things as view controllers, views, database handling, and application logic. The audio code is a standard Csound CSD project that contains code written in Csound and will be run using a CsoundObj object. Finally, the glue code is what will bridge the user interface with Csound.

```
/* this callback handler is called every time a buffer finishes playing */
void bqPlayerCallback(SLAndroidSimpleBufferQueueItf bq, void *context)
{
   open_sl_params *params = (open_sl_params *) context;
   params->csound->NotifyThreadLock(params->clientLockOut);
}

/* put samples to DAC */
void androidrtplay_(CSOUND *csound, const MYFLT *buffer, int nbytes)
{
    open_sl_params *params;
    int     i = 0, samples = nbytes / (int) sizeof(MYFLT);
    short* openslBuffer;

    params = (open_sl_params *) *(csound->GetRtPlayUserData(csound));
    openslBuffer = params->outputBuffer[params->currentOutputBuffer];
    if (params == NULL)
      return;
    do {
      /* fill one of the double buffer halves */
      openslBuffer[params->currentOutputIndex++] = (short) (buffer[i]*CONV16BIT);
      if (params->currentOutputIndex >= params->outBufSamples) {
         /* wait for notification */
          csound->WaitThreadLock(params->clientLockOut, (size_t) 1000);
         /* enqueue audio data */
         (*params->bqPlayerBufferQueue)->Enqueue(params->bqPlayerBufferQueue,
                          openslBuffer,params->outBufSamples*sizeof(short));
         /* switch double buffer half */
         params->currentOutputBuffer = (params->currentOutputBuffer ?  0 : 1);
         params->currentOutputIndex = 0;
         openslBuffer = params->outputBuffer[params->currentOutputBuffer];
      }
    } while (++i < samples);
}
```

Listing 1: OpenSL module output C function and associated callback

```
public interface CsoundValueCacheable {
  public void setup(CsoundObj csoundObj);
  public void updateValuesToCsound();
  public void updateValuesFromCsound();
  public void cleanup();
}
```

Listing 2: CsoundValueCacheable Interface

```
String csd = getResourceFileAsString(R.raw.test);
File f = createTempFile(csd);
csoundObj.addSlider(fSlider, "slider", 0.0, 1.0);
csoundObj.startCsound(f);
```

Listing 3: Example CsoundObj usage

CsoundObj uses objects that implement the CsoundValueCacheable interface for reading value from and writing values to Csound (listing 2). Any number of cacheables can be used with CsoundObj. The design is flexible enough such that you can design your application to use one cacheable per user interface or hardware sensor element, or one can make a cacheable that reads and writes along many channels.

CsoundObj contains utility methods for binding Android Buttons and SeekBars to a Csound channel, as well as for a method for binding the hardware Accelerometer to preset Csound channels. These methods wrap the View or sensor objects with pre-made CsoundValueCacheables that come with the CsoundObj API. Since these are commonly used items that would be bound, the utility methods were added to CsoundObj as a built-in convenience to those using the API. Note that CsoundValueCacheables are run within the context of the audio processing thread; this was done intentionally so that the cacheable could copy any values it needed to from Csound, then continue to do processing in another thread and eventually post back to the main UI thread via a Handler.

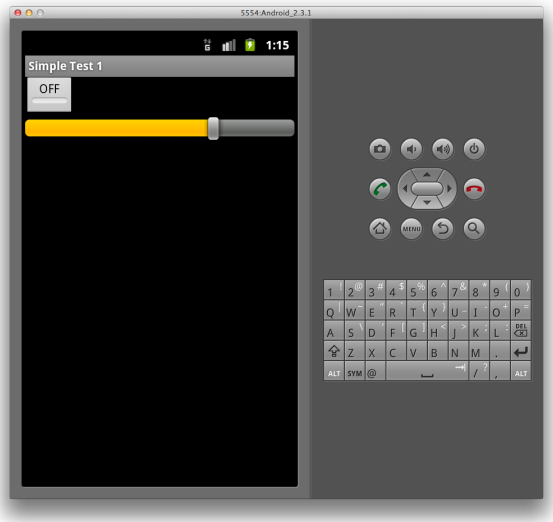


Figure 1: Android Emulator showing Simple Test 1 Activity

Listing 3 shows example code of using CsoundObj with a single slider, from the Simple Test 1 Activity, shown in Figure 1.The code above shows how a CSD file is read from the projects resources using the getResourceFileAsString utility method, saved as a temporary file, then used as an argument to CsoundObj's startCsound method. The 2nd to last line shows the addSlider method being used to bind fslider, an instance of a SeekBar, to Csound with a channel name of "slider" and a range from 0.0 to 1.0. When Csound is started, the values from that SeekBar will be read by the Csound project using the chnget opcode, which will be reading from the "slider" channel.

Figure 2 shows the relationships between different parts of the platform and different usage scenarios. An application may work with CsoundObj alone if they are only going to be starting and stopping a CSD. The application may also use CsoundValueCacheables for reading and writing values from either CsoundObj or the CsoundObject. Finally, an application may do additional interaction with the Csound object that the CsoundObj has as its member, taking advantage of the standard Csound API.

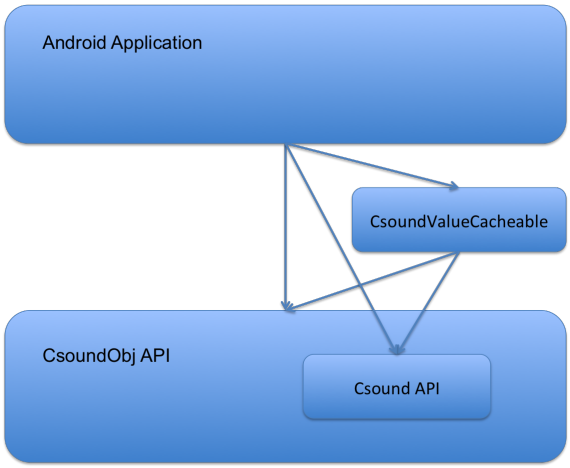


Figure 2: CsoundObj Usage Diagram

A Csound for Android examples project has been created the contains a number of different Csound example applications. These examples demonstrate different ways of using the CsoundObj API as well as different approaches to applications, such as realtime synthesis instruments and generative music. The examples were ported over from the Csound for iOS examples project and users can study the code to better understand both the CsoundObj API on Android as well as what is required to do cross-platform

development with Csound as an audio platform.

4 Benefits of using Csound on Android

Using Csound on Android provides many benefits. First, Csound contains one of the largest libraries of synthesis and signal processing routines. By leveraging what is available in Csound, the developer can spend more time working on the user interface and application code and rely on the Csound library for audio-related programming. The Csound code library is also tested and supported by a open-source community, meaning less testing work required for your project.

In addition to the productivity gain of using a library for audio, Csound projects–developed in text files with .csd extensions–can be developed on the desktop, and later moved to the Android application. Developing and testing on the desktop allows for a faster development process than testing in the Android emulator or on a device, as it removes the application compilation and deployment stage, which can be slow at times.

Having the audio-related code in a CSD file for a project also brings with it two benefits. First, development of an application can be split amongst multiple people; one can work on the audio code while the other focuses on developing other areas of the application. Second, developing an application based around Csound allows for moving that CSD to other platforms, such as iOS or desktop operating systems. The developer would then only have to develop the user-interface and glue code to work with that CSD on each platform.

Additionally, cleanly separating out the audio system of an application and enforcing a strict API (Application Programmer Interface) to that system is a good practice for application development. This helps to prevent tangled, hard to maintain code. This is of benefit to the beginning and advanced programmer alike.

5 Conclusions

From its roots in the Music N family of programs, Csound has grown over the years, continually expanding it features as a synthesis library as well as its usefulness as a music platform. With its availability on multiple operating systems, Csound offers a multi-platform option for developing musical applications. Current Csound 6 developments to enable realtime modification of the processing graph as well as other features will expand the types of applications that can be built with Csound. As Android is now supported within the core Csound repository, it will continue to be developed as a primary platform for deployment as part of the MCP distribution.

6 Availability

The Csound for Android platform and examples project are included in the main Csound GIT repository. Build files are included for those interested in building Csound with the Android Native Development Kit. Archives including a pre-compiled Csound as well as examples are available at `http://sourceforge.net/projects/csound/files/csound5/Android/`.

7 Acknowledgements

This research was partly funded by the Program of Research in Third Level Institutions (PRTLI 5) of the Higher Education Authority (HEA) of Ireland, through the Digital Arts and Humanities programme.

References

R. Boulanger, editor. 2000. *The Csound Book*. MIT Press, Cambridge, Mass.

V Lazzarini and J. Piche. 2006. Cecilia and tclcsound. In *Proc. of the 9th Int. Conf. on Digital Audio Effects (DAFX)*, pages 315–318, Montreal, Canada.

V Lazzarini and R. Walsh. 2007. Developing ladspa plugins with csound. In *Proceedings of 5th Linux Audio Developers Conference*, pages 30–36, Berlin, Germany.

V Lazzarini. 2006. Scripting csound 5. In *Proceedings of 4th Linux Audio Developers Conference*, pages 73–78, Karlsruhe, Germany.

V Lazzarini. 2008. A toolkit for audio and music applications in the xo computer. In *Proc. of the International Computer Music Conference 2008*, pages 62–65, Belfast, Northern Ireland.

Ardour3 - Video Integration

Robin Gareus
gareus.org, linuxaudio.org, CiTu.fr
Paris, France
robin@gareus.org

Abstract

This article describes video-integration in the Ardour3 Digital Audio Workstation to facilitate sound-track creation and film post-production.

It aims to lay a foundation for users and developers, towards establishing a maintainable tool-set for using free-software in A/V soundtrack production.

To that end a client-server interface for communication between non-linear editing systems is specified. The paper describes the reference implementation and documents the current state of video integration into Ardour3 and future planned development. In the final sections a user-manual and setup/install information is presented.

Keywords

Ardour, Video, A/V, post-production, sound-track, film

1 Introduction

The idea of combining motion pictures with sound is nearly as old as the concept of cinema itself.

Ever since the invention of the moving picture, films have entailed sound and music. Be it mechanical music-boxes accompanying candle-light projections, the Kinetoscope in the late 19th century; live-music supporting silent-films in the early 20th century to completely digital special effects in the 2000s.

Charlie Chaplin composed his own music for City Lights (1931) and many of his later movies. He was not clear whose job was to score the sound-tracks [Scaruffi, 2007]. That was the exception, and few film-makers would imitate him. In the 1930s, after a few years of experimentation, scoring film soundtracks became an art in earnest. By the mid-1940s, cinema's composers had become a well-established category.

Apart from the film-score (music), a film's soundtrack usually includes dialogue and sound effects.

One goal is to synchronize dramatic events happening on screen with musical events in the score - but there are many different (artistic) methods for syncing music to picture. With only a few exceptions - namely song or dance scenes - music composition and sound-design usually takes place after recording and editing the video [Wikipedia, 2011].

Major problems persisted, leading to motion pictures and sound recording largely taking separate paths for a generation. The primary issue was - and is - synchronization: pictures and sound are recorded and played back by separate devices, which were difficult to start and maintain in tandem. This stigma still prevails for most professional audio/video recordings for both creative and technical: ie. there is too much gear on camera already, more (budgetary) choices are available, unions define different job functions. . . .

Post-production requires to synchronize the original audio recorded on the set (on-camera voice) with the edited video. In the *analog days* visual cues (slate) have been used towards that end. With the advent of digital technology time-code (most commonly SMPTE produced by the camera) is commonly recorded along with the sound to allow reconstructing the audio-tracks after the video has been cut.

As you can imagine, the technical skills and details involved can become quite complex and neither composers nor sound-designers do want to concern themselves with that task. Creative technicians have come up with various tools to aid the process.

These days many Digital Audio Workstations

provide features to import EDL (edit decision lists) or variants thereof (AAF: Advanced Authoring Format, MXF: Material eXchange Format, BWF: Broadcast Wave Format, OMF+OMFI: Open Media Framework Interchange,...) and display a film-clip in sync with the audio in order to facilitate soundtrack creation.

There are very few to none free-software solutions for this process. However, Ardour [Davis and others, 1999 2012] provides nearly all relevant features for composition, recording as well as mastering. The underlying JACK audio connection kit allows for inter-application synchronization [Davis and others, 2001 2012] and suggests itself to be used in the context of film post-production.

Figure 1: Ardour 3.0-beta1 with video-timeline patch and video-monitor

2 Design

Other FLOSS projects are tackling the process of video production (NLE: non-linear editing), most notably Blender [Foundation, 1999 2012], lives [Finch, 2004 2012] and cinelerra [Ltd, 2002 2011]; however these emphasize on the visuals and often significantly lack on the audio side.

Other free-software digital audio workstations (DAWs) capable of the job include musescore [Schweer and others, 1999 2012] and qtractor [Capela, 2005 2011] amongst others, however with MIDI support arriving in Ardour3, continued cross-platform support and available high-end professional derivatives (Mixbus, Harrison) it is currently the most suitable and promising DAW from a developer's perspective.

2.1 Design-goals and use-cases

The aim is to provide an easy-to-use, professional workflow for film-sound production using free-software. More specifically: Integration of video-elements into the Ardour Digital Audio Workstation.

The resulting interface must not be limited to the software at hand (Ardour, Xjadeo, icsd) but allow for further adaption or interoperability.

2.1.1 General

An important aspect of the envisaged project is modular design. This is a prerequisite to allow the project to be developed gradually and maintained long-term. With Interface definitions in place, the building blocks can be implemented individually. It is also crucial to cope with the current situation of video-codec licensing: Depending on the availability of licenses and user preferences, it should be possible to en/disable certain parts of the system without breaking overall functionality.

The utter minimum for both composing as well as producing sound-tracks is a video-player that synchronizes to an external time-source provided by the DAW.

To navigate around in larger projects a timeline becomes very valuable.

Session-management as well as archiving is important as soon as one works on more than one project. Project-revisions or snapshots come in handy.

Import, export and inter-operability with other software: Those are likely going to be the hardest part, yet also the most important.

Transcoding the video on import is necessary to provide smooth response times for forward/backward seeking. It may also be required for inter-operability (codecs). Demuxing is needed to import edited video-tracks along with the original set of audio tracks, aligned to timecode.

Exporting the soundtrack means aligning the materials and multiplexing the original video with the audio or to provide exact information on how to do that. Various formats - both proprietary and free - are in use which complicates the process.

Sometimes you (or the director) notices during sound-design, that some video edit decisions were not optimal; or that there is just a video frame missing for proper alignment. With digital

technology incremental updates of the video are not only feasible but relatively simple. Empowering the audio-engineer to quickly do minor video editing is a very useful but also dangerous feature.

2.1.2 Short-Term

Increase the usability of existing tools with focus on soundtrack creation (video-timeline, video-monitor).

In particular the learnability and efficiency for the workflow should be streamlined: video-transcoding, video-monitor and A/V-session management functionality should be accessible from a single application. The complexity of the whole process should be abstracted and focus on the use-case at hand while not limiting the actual system.

Practically this is mapped to adding support into Ardour3 to communicate with and control the Xjadeo [Gareus and Garrido, 2006 2012b] video-monitor. Furthermore a video-timeline axis is aligned to the Ardour-canvas. Lastly, the possibility to invoke ffmpeg from within Ardour's GUI to import and export audio/video is implemented. The user-interaction is kept to a minimum: Import Video, Show/Hide Timeline, Show/Hide video-monitor.

2.1.3 Mid-Term

Improve system integration. Configuration presets for multi-host setups. Streamline video-export, MUX/codec/format presets.

Stabilize video-session and edit API. Add interoperability with external video-editors. Import various EDL dialects.

2.1.4 Long-Term

Top Notch client-server distributed A/V non-linear editor, using Ardour3 for the sound, dedicated GUI for the video. Seamless integration.

2.2 Architecture

Ardour itself is separated in two main parts: the back-end (audio-engine, libardour) which manages audio-tracks, audio-routes and session-state and the front-end (gtk2-ardour) which provides a stateless graphical user interface to the back-end.

The idea is to construct the system such than intrusion in Ardour itself is minimal: Only Ardour's front-end GUI should be *aware* of video-elements.

The video-decoding and video-session management is done in a separate application. A client-server model is used to partition tasks. Ardour as client does not share any of its resources, but requests video-content from the server. The overall outline is depicted in figure 2.

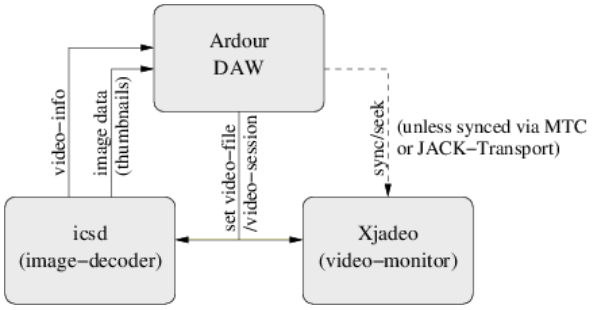


Figure 2: Overview of the client-server architecture

The video-server itself is a modular system. The essential and minimum system comprises a video-decoder and video-frame cache. Video-frames are referenced by frame-number. Time-code mapping is optional and can be done on the client and server-side (using server-side sessions). However, the server must provide information about the file's (or session's) frame-rate, aspect-ratio and start-time-offset. Furthermore, the server must be able to present server-side sessions as single file to the client.

Minimal configuration that needs to be shared by the server and client is the server's access URL and document-root.

Even though tight integration is planned, the prototype architecture leaves room for side-chains. In particular this concerns extraction of audio-tracks from video files as well as final multiplexing and mastering the final video. While interfaces are specified to handle these on the server-side, tight user-interface integration and rapid prototyping motivates a client-side implementation. This is accomplished by sharing the underlying file-system storage.

3 Interface

For read-only (no video-editing) access to video information, all communication between the client (here: Ardour) and the video-server is via HTTP.

This is motivated by:

- HTTP is a well supported protocol with existing infrastructure.
- with client-side caching: throughput is more important than low latency. HTTP overhead is reasonably small[1]
- web-interface possibility
- persistent HTTP connections are possible
- possibility to make use of established proxy and load-balancing systems

3.1 Protocol: Request Parameters

HTTP verbs can be taken into account for specific requests e.g. `HEAD`: query file-information, `DELETE`: remove assets, regions, files. The vast majority will bet `GET` and `POST` requests.

The server URL must conform to RFC3986 specifications. It must be possible to have the server in a sub-path on the server's root (e.g. http://example.org/my/server/). The URI must allow path and file-names to be added below that path (e.g. http://example.org/my/server/status.html). A default endpoint handler needs to catch access to the doc-root and provide service for the following requests URLs, which are described in further detail below:

/status print server-status (user-readable, version, etc) must return *200 OK* if the server is running.

/info return file or session info

/frame query image-frame data

/ generic handler to above according to request-parameters

/admin/flush_cache (optional, `POST`) reset cache

/admin/shutdown (optional, `POST`) request clean shutdown of server

/stream (optional) server-side rendering of video chunks - export and preview

/index/* (optional) file and directory index

/session/* (optional) server-side project and session management

The file extension defines the format of the returned data. For textual (non-binary) data, valid requests include `.html`, `.htm` , `.json`, `.xml`, `.raw`, `.edl`, `.null`. Valid image file extensions are `.jpg`, `.jpeg`, `.png`, `.ppm`, `.yuv`, `.rgb`, `.rgba`. For video streaming, the extension defines the container format e.g. `.avi`, `dv`, `.rm`, `.ogg`, `.ogv`, `.mpg`, `.mpeg`, `.flv`, `.mov`, `.mp4`, `.webm`, `.mkv`, `.vob`, ... The extension is case-insensitive.

Alternatively the format can be specified using the `format=`*FMT* query parameter which overrides the extension, if any.

3.1.1 File and Session information

The `/info` handler returns basic information about the file or session.

Request parameters:

file file-name relative to the server's document root.

Reply:

version reply-format version. Currently `1`, may change in the future

framerate a double floating point value of video frames-per-second

duration long integer count of video-frames

start-offset double precision - time of first video-frame; specified in (fractional) seconds

aspect-ratio floating point value of the width/height geometry ratio including pixel and display-aspect ratio multipliers

width (optional) integer video width in pixels

height (optional) integer video height in pixels

length (optional) video-duration in seconds

size (optional) video-size in bytes

The *raw* (un-formatted) reply concatenates the values above in order, separated by unix-newlines ('\n', ASCII 0x0a). Optional values require all previous values to be given: i.e. length requires width and height.

json or *xml* formatted replies must return an associative array with the key-name as specified

[1] RGB24 frame sizes, thumbnail 160x90: 337kiB, 768x576/PAL:10.1MiB, 1920x1080/full-HD: 47.5MiB. HTTP request/response headers are typically between 200 and 800 bytes.

```
#curl -d file=tmp/test.avi \
      http://localhost:1554/info
1
25.000
15262
0.0
1.833333
```

Figure 3: Example request of file information

in the reply format list. *html* or *txt* replies are intended for user readability and may differ in formatting, but should include the required information.

3.1.2 Image (preview, thumbnails)

Request parameters:

file file-name relative to the server's document root.

frame the frame-number (starting at zero for the first frame).

w (optional) width in pixels - default -1: auto

h (optional) height in pixels - default -1: auto

format (optional) image format and/or encoding

If neither width or height are specified, the original size (in pixels, not scaled with pixel or display-aspect ratio) must be used. If both width and height are given, the image must be scaled disregarding the aspect-ratio. If either width or height are specified, the returned image must be scaled according to the movie's real aspect ratio.

The default format is `raw` RGB, 24 bits per pixels.

3.1.3 Request video export, rendering

The `/stream` handler is used to encode a video on the server.

Request parameters:

file file-name relative to the server's document root or session-id.

frame (optional) the frame-number (starting at zero for the first frame) where to start encoding. the default is 0, start at the beginning.

duration (optional) number of frames to encode (minus one), negative values indicate no limit. A value of 0 (zero) must encode exactly one video frame. the default value is -1: until the end.

w (optional) width in pixels; default -1: auto

h (optional) height in pixels; default -1: auto

container (optional) video format; default : avi

nosound (optional) if set (1) only video is encoded - default: unset, 0

3.1.4 Server administration

Since the only way to communicate with the server is HTTP, specific interfaces are needed for administrative requests. The current standard defines only two; namely `re-start` (or cache-flush) and `shutdown`.

These commands are mostly for debugging and development purposes: The *clean shutdown* was motivated to check for memory-leaks; but comes in handy to launch temporary servers with each Ardour-session.

Cache-flush is useful if the video-file changes. Cached images must be cleaned and the decoders must release open file-descriptor and re-read the updated video-file. Future implementations should actually monitor the file for modification and do this automatically.

3.1.5 The Session API

Session API replicates `/info`, `/frame` and `/stream` request-URLs below the `/session/` namespace. Each request to those handlers must specify a `session` query parameter. The `file` parameter - if given - can be used to identify chunks inside a session.

Additional request handlers are needed to provide access to editing functionality, in particular asset-management and clip arrangement.

The detailed description of the session API is beyond the scope of this paper and will be published separately [2].

3.2 Extensions

3.2.1 Web GUI

The `/gui/` namespace is reserved for an end-user web-interface which is out of the scope of this paper. Currently a prototype using XSLT and icsd2's XML function is prototyped in XHTML

[2] A prototype of the session API for research is implemented in `libs/libsodan/jvsession.c` and `src/ics_sessionhandler.c`

and JavaScript and available with the source-code (see figures 4, 7).

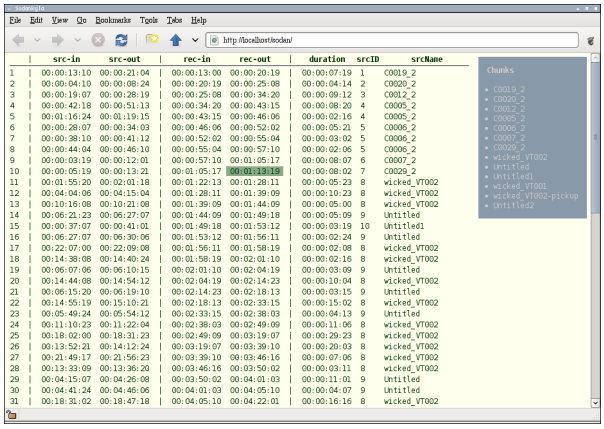


Figure 4: icsd2: JavaScript+XHTML EDL editor

3.2.2 Out-of-band notifications

Since it is not possible to use HTTP to send out-of-band notifications, it is not a suitable protocol for concurrent editing.

Both XMPP (Extensible Messaging and Presence Protocol) and OSC (OpenSoundControl) are options.

OSC has less overhead, yet only OSC over TCP provides reliable transport. XMPP has the advantage that by spec processing between any two entities at the server must be in order. Furthermore XMPP is a protocol intended for publish/subscribe, and capable to notify multiple clients. Authentication methods have also been defined for XMPP, whereas both authentication and Pub/Sub would require custom development on top of OSC.

3.2.3 Authentication and Access Control

Both HTTP as well as XML offer means of authentication below the application layer.

While some requests motivate access-control as part of the protocol - e.g. user-specific read-only access to certain frame-ranges or chunks - these can in general be handled by URI filtering, much as a .htaccess files does in apache.

3.3 Server Internal API

This section covers the planned interface for server-side sessions and non-linear video-editing. The back-end is partially implemented in `icsd2` and there is a basic web-interface to it.

3.3.1 Database and persistent session information storage

The server keeps a database for each session. A session consists of one or more chunks (audio or video sources) and a map how to combine them. Furthermore the session-database includes configuration and session-properties in a generic key-value store.

An extended EDL table is used to for mapping the chunks. It provides for (track) layering of chunk-excerpts (in-point, out-point). The data-model does not yet provide for per edit or per chunk attributes (zoom, effect automation) but transitions between chunks can be specified in EDL style. The current database schema can be found in source-code.

3.3.2 Server-side video monitoring

An important feature is to be able to to physically separate audio (Ardour) and video (server and monitor). This is motivated by various factors: All video-outputs of the client computer may be needed for audio, leaving none available for a full-screen video projector. It also preempts resource conflicts between audio processing and video-decoding.

To provide server-side video monitoring (video-display), two approaches are currently prototyped:

- monolithic approach: share the decoder and frame-cache back-end between the HTTP-server and the video-monitor using shared-memory.
- modular approach: share the source-code/libraries: HTTP-server and video-monitor are independent applications.

The former obviously requires less resource and puts less strain on the server (memory and CPU usage) at the cost of potential instability (resource conflicts) since maintaining cache-coherence complicates the code.

A middle-ground - building a video-monitor on top of the existing HTTP `frame/` interface - is unsuitable for low-latency seeks. Even though it performs nicely in linear playback it effectively purges the cache and introduces a penalty for time-line use.

With computing resources being easily available (and cheap compared to video-production in

general), a modular approach - using independent video-monitor software - is the way to follow.

3.3.3 Server to Server communication

A cache-coherence protocol MOESI (Modified, Owned, Exclusive, Shared, Invalid) is envisaged to distribute load among servers. An algorithm based on file-id and frame-number can be used to dispatch requests to servers in a pool. This can be done directly in the client implementation or by a proxy-server (http load balancer).

Server-to-server communication will use publish/subscribe to synchronize and distribute updates among servers in the pool.

4 Implementation

Historically things went a bit backwards. After the groundwork (Xjadeo, Ardour1) had been laid, different endeavours (gjvidtimline, xj-five) culminated in the video-editing-server software (vcsd) for Ardour3. The recurring need to focus on movie-soundtracks motivated the short-term goals of the current implementation: Ardour3 [Davis and others, 1999 2012] provides the DAW as well as the GUI. The video-server (icsd) project is named Sodankylä [Gareus, 2008 2012] after its place of birth.

4.1 client – Ardour3

The patch[3] can be broken out into small parts.

- Video-timeline display
- HTTP interaction with the video-server
- Xjadeo remote control
- ffmpeg interaction
- Dialogs (the largest part)
- Support and helper functions

The implementation is completely additive[4], and all video related code is separated by `#ifdef`s.

4.1.1 Video timeline

`gtk2_ardour/video_timeline.cc` defines the timing and alignment of the video-frames depending on the current scroll-position and zoom-level of the Ardour canvas.

The video-timeline is a unique "ruler" in Ardour (similar to bar/beat or markers) and globally accessible via `ARDOUR_UI::instance()->video_timeline`.
The video-timeline contains video-frame images: `gtk2_ardour/video_image_frame.cc`.

The timeline is populated by aligning the first video-frame. The zoom-level (audio-frames per pixel) and timeline-height * aspect-ratio (video-frame width in pixels) define the spacing of video-frames after the first frame. Images on the current page of the canvas as well as the next and previous page are requested and cached locally to allow smooth scrolling.

`VideoTimeLine` and `VideoImageFrame` are the only two classes that communicate with the video-server. The former is used to request video-session information (frame-rate, duration, aspect-ratio), the latter handles (raw RGB) image-data. The actual `curl_http_get()` function is defined in the file `video_image_frame.cc` which also includes a threaded handler for async replies of image-data.

The GET request-parameters to query information about the video-file are:
SERVER-URL`/info?file=`*fileURI*`&format=plain`
The GET request-parameters for image data are as follows:
SERVER-URL`/?frame=`*frame-number*
`&w=`*width*`&h=`*height*
`&file=`*fileURI*`&format=rgb`

The width and height are calculated by using the video's aspect-ratio and the height of the timeline-ruler. Current options are defined in `editor_rulers.cc` and include:

- "Small": $3H_t$
- "Normal": $4H_t$
- "Large": $6H_t$

with H_t = `Editor::timebar_height = 15.0;` pixels defined in `editor.cc`.

4.1.2 system-exec

Starting an external application with bi-directional communication over standard-IO, requires dedicated code.
`gtk2_ardour/system_exec.cc` implements a cross-platform compatible (POSIX, windows) API to launch, terminate and pipe data to/from child processes.

[3] `http://gareus.org/gitweb/?p=ardour3.git;a=commitdiff_plain;hp=master;h=videotl`

[4] No existing code is removed or changed, however the build-script and some of the documentation is modified.

It is used to communicate with Xjadeo [5] as well as to launch ffmpeg and the video-server (on localhost).

4.1.3 transcoding

`gtk2_ardour/transcode_ffmpeg.cc` includes low-level interaction with ffmpeg. It is concerned with locating and starting `ffmpeg` and `ffprobe` executables as well as parsing output and progress information. The command-parameters (presets, options) as well as progress-bar display is part of the `transcode_video_dialog.cc`.

4.1.4 Dialogs

The vast majority of the code contributed to Ardour consists of dialogs to interact with the user.

`gtk2_ardour/video_server_dialog.cc` asks to start the video-server on localhost. It is shown on video-import if the configured video-server can not be reached.

`gtk2_ardour/add_video_dialog.cc` Called from Session-menu → Import → Video, asks which file to import. The file-selection dialog assumes that the video-server runs on localhost or the server's document-root is shared via NFS[6]. The dialog offers options to transcode the file on import, or copy/hardlink it to the session folder. Furthermore there are options to launch the video-monitor and set the Ardour session's timecode-rate to match the video-file's FPS (see figure 5). `gtk2_ardour/video_copy_dialog.cc` is a potential follow-up dialog with progress-bar and overwrite confirmation.

If the video-server is running on localhost, the dialog also checks if the (imported) file is under the configured document-root of the video-server.

`gtk2_ardour/transcode_video_dialog.cc` allows to transcode video-files on import. It is recommended to do so in order to decrease the CPU load and optimize I/O resource usage of the video-decoder. The dialog also allows to select an audio-track to be extracted from the video-file and imported as audio-track into Ardour.

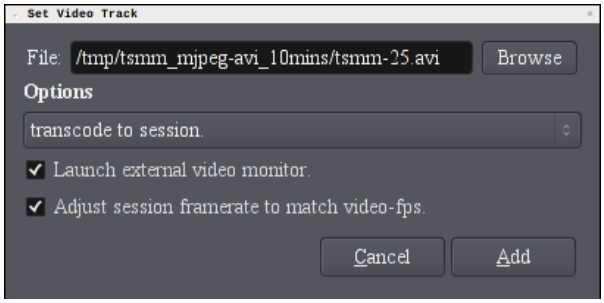


Figure 5: A3: the video-open dialog

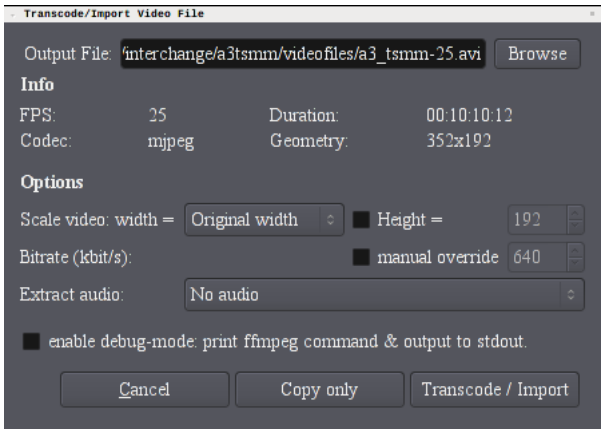


Figure 6: A3: the import-video, transcode dialog

`gtk2_ardour/open_video_monitor_dialog.cc` optional settings for Xjadeo. This dialog can be bypassed (configuration option) and allows to customize settings of Xjadeo that should be retained between sessions: e.g. window-size, position and state, On-Screen-Display settings, etc.

`gtk2_ardour/export_video_dialog.cc` is the most complex dialog. It includes various (hardcoded) presets for video-encoding and ffmpeg options. It also includes a small wrapper around Ardour's audio-export function.

4.1.5 video-monitor / Xjadeo

`gtk2_ardour/video_monitor.cc` implements interaction with Xjadeo. In particular sending video-seek messages if Ardour is using internal

[5] actually xj-remote which in turn communicates with Xjadeo, platform dependent either via message-queues or RPC calls

[6] A prefix can be configured to be removed from the selected path for making requests to the video-server.

transport and time-offset parameters if the video is re-aligned.

Switching Ardour's sync mechanism between JACK and internal-transport automatically toggles the way Ardour controls Xjadeo. Also Xjadeo's window state (on-top), on-screen-display mode (SMPTE, frame-number display) as well as the window position and size are remembered across sessions. Ardour sets override-flags in Xjadeo's GUI that disable some of the direct user-interaction with Xjadeo; in particular: you can not close the monitor-window nor load different files by drag-drop on the Xjadeo window or neither modify the time-offset using keyboard shortcuts. These interactions need to be done through Ardour.

4.1.6 misc

There are various small patches - mostly glue - to Ardour's `editor_*.cc`, `ardour_ui*.cc` as well as preference (video-server URL) and session-option (sync, pull up/down) dialogs. Dedicated video functions have been collected in `gtk2_ardour/editor_videotimeline.cc` and `gtk2_ardour/utils_videotl.cc`

4.1.7 libardour

While the video-timeline patch itself only concerns the Ardour GUI, a few helper functions semantically belong in *libardour*. They are almost exclusively related to saving preferences or resolving directories.

4.2 prototype server

The source-code includes a simple PHP script `tools/videotimeline/vseq.php` that emulates the behaviour of the video-server (no frame-caching) and implements the minimal interface protocol specifications. It is built on top of the command-line applications: ffmpeg, ffprobe and ImageMagick's convert.

4.3 video-server – Sodankylä – icsd

The video-server was born in Sodankylä June 2008. It's a motley collection of tools for handling video files. The 'image compositor socket daemon' (`icsd` - the names goes back to Ardour-1) implements a video-frame-caching HTTP server according to specs defined in section 3.1. `icsd2` in the same source-tree is a (work-in-progress) version adding the video-session and NLE capabilities.

By default `icsd` listens on all network interfaces' TCP port 1554 and runs in foreground, serving files below a given *document-root*.

```
usage: icsd [OPTION] <document-root>
```

The number of cached video-frames can be specified with the `-C <num>` option. It defaults to 128. The IP address to listen on can be set with `-P <IPv4 address>` (default: 0.0.0.0). There are various options regarding daemonizing, chroot, set-uid, set-gid and logging that allow icsd be started as system-service. They are documented in the manual-page.

Note that, icsd does not perform any access control. As with a web-server, all documents below the document-root are publicly readable.

The internal video frame-cache uses LRU (least-recently requested video-frame) cache-line expiry strategy. icsd is multi-threaded, it keeps a pool of decoders and tries to map consecutive frames from one keyframe to the same-decoder; however a new decoder is only spawned if the current one is busy in another thread.

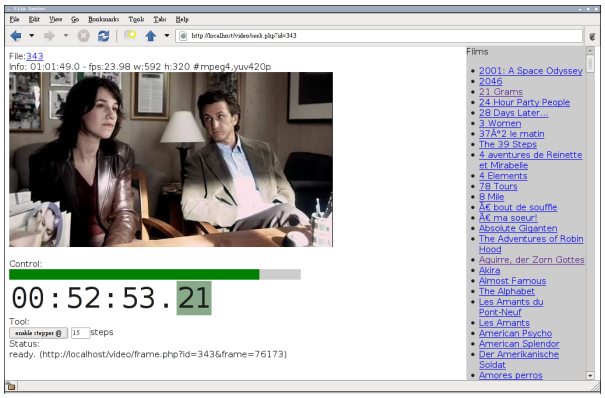


Figure 7: icsd2: JavaScript+XHTML video-monitor

`icsd2` extends the simple API towards video-session and non-linear-editing.

To allow interoperability with various client-side tools, a REST-API is being defined. XLST is used to export information in XML, as well as generate XHTML pages. For prototyping and debugging a simple web-player and JavaScript EDL editor came to be (screenshot in figure 7).

5 QA and Performance Tests

The most important part is to assure fitness for the purpose. Usability and integration is useless if fundamental quality is not sufficient. Criteria that require verification include accuracy, time-code mapping, as well as latency and system-load.

Depending on the zoom-level video-frames in the audio-timeline must be properly aligned. During playback the video-monitor must not lag behind. Audio latency must be compensated so that the video-frame on screen corresponds to the audio-signal currently reaching the speakers. The system load must remain within reasonable bounds and the applications should run smoothly without spiking.

5.1 Quality Assurance and Testing

Audio/Video frame alignment has been tested using the "time-stamp-movie-maker" [Gareus and Garrido, 2006 2012a]. It was used to create movies with time-code and frame-number rendered on the image at most common frame-rates (23.976, 24, 24.976, 25, 29.97df, 30, 59.94 and 60 fps) using common codecs + format combinations (mjpeg/avi, h264/avi, h264/mov, mpeg4/mov, theora/ogg). All 40 videos have been loaded into Ardour and tested to align correctly.

Figure 8: testing time-code mapping using a time-stamped movie

Latency compensation is verified by playing a movie of a unique series of black and white frames full-screen synchronously to an audio-track that sends an impulse on every white frame. The audio-impulse signal must occur aligned to the (luminance) signal of the video within the bounds of display frequency. Since there are 3 different clocks (display-refresh frequency, video frame-rate, audio sample-rate) involved the problem is not trivial. However audio sample-rate is an order of magnitude smaller than the usual video frame-rates and can be neglected. Systematic latency-compensation tests have successfully been performed at 25fps file of alternating black and white frames on a 75Hz display frequency. Other rates have been verified but not analyzed in depth.

5.2 Performance tests

Performance is evaluated by benchmarking unit-tests of the individual components as well as monitoring overall system performance.

The former includes measuring request latency to query video-frames which is further broken up into distributing workload on multiple video-decoders. An average session will spawn 16-20 video-decoders on a dual-core. The memory footprint of a decoder is negligible compared to the cached image data, yet it greatly improves response time when seeking in files to use a decoder positioned close to the key-frame.

Request and decode latencies have been measured with `ab` - The Apache HTTP server benchmarking tool as well as by timing the requests with Ardour. Cache hits are dominated by transfer time which is on the order of a few (1-3) milliseconds depending on image-size and network. Decoder latency is highly dependent on the geometry of the movie-file, codec, scaling and CPU or I/O. On slower CPUs (< 1.6 GHz Intel) a full HD video can be decoded and scaled at 25 fps using mjpeg codec width only intra-frames at the cost of high I/O. Faster CPUs can shift the load towards the CPU. Parallelizing requests increases latency to up to a few hundred milliseconds. This is intended behavior: image for a whole view-point page will arrive simultaneously.

6 User Manual

6.1 Setup

- get Ardour3 with video-timeline patch.
- install icsd (≥alpha-11) and optionally ffmpeg, ffprobe and Xjadeo (≥ 0.4.12) to $PATH.
- make sure that there is no firewall on localhost TCP-port 1554 (required for communication between Ardour and the video-server).

If you run the video-server on the same host as Ardour, no further configuration is required.

When opening the first video, Ardour3 will ask for the path of the video-server binary (unless

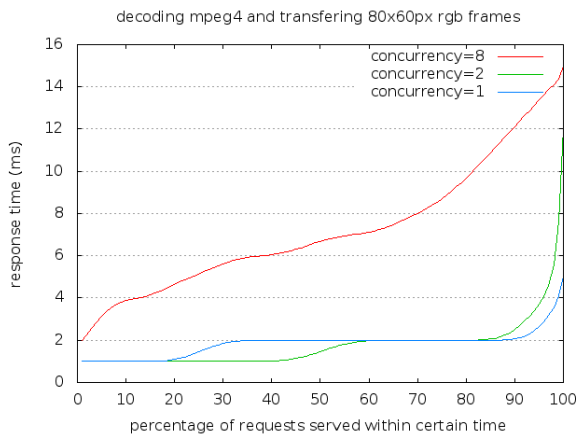


Figure 9: latency for decoding thumbnail frames for 1,2 and 8 parallel requests

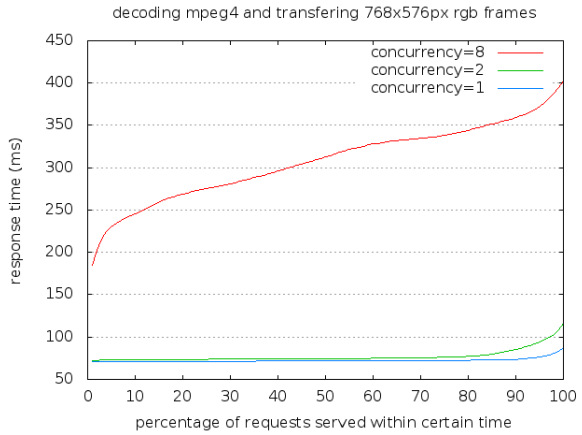


Figure 10: decoder and request latency for PAL video.

it is found in $PATH) and also allows you to change the TCP port number as well as the set cache size. Transcoding is only available if ffmpeg and ffprobe are found in $PATH (fallback on OSX: `/usr/local/bin/ffmpeg_sodankyla` - provided with icsd.pkg).

The video-monitor likewise requires `xjremote` to be present in $PATH or on OSX under `/Applications/Jadeo.app/Contents/MacOS/` and windows `$PROGRAMFILES\xjadeo\`. Xjremote comes with Xjadeo and Jadeo respectively.

6.2 Getting Started

It is pretty much self-explanatory and intended to be intuitive. All functions are available from the Ardour Menu. Select actions are accessible from the context-menu on the video-timeline.

- Menu → Session → Open Video
- Menu → Session → Export → Video
- Menu → View → Rulers → Video (or right-click the ruler/marker bar)
- Menu → View → Video Monitor (Xjadeo)
- Menu → Edit → Preferences → Video
- Menu → Session → Video maintenance → . . . (manual video server interaction)

A quick walk-through goes something like this: Launch Ardour3, Menu → Open Video. If the video-server is not running, Ardour asks to start it. You choose a video-file and optionally extract the original sound of the video to an Ardour track. By default Ardour also sets the session FPS and opens a video-monitor window.

From then on, everything else is plain Ardour. Add audio-tracks, import samples, mix and master,. . .

Eventually you'll want to export your work. Session → Export → Video covers the common cases from file-creation for online-services over producing a snapshot for the client at the end of the day to optimized high-quality two-pass MPEG encoding. However it is no substitute for dedicated software (such as dvd-creator) or a video engineer with a black-belt in ffmpeg or expertise in similar transcoding software, especially when it comes to interlacing and scan-ordering..

The export dialog defaults to use the imported video as source, note however that using the original file usually produces better results; every transcoding step potentially decreases quality.

6.3 Advanced Setups

The video-server can be run on a remote-machine with Ardour connecting to it. In this case you need to launch icsd on the server-machine and configure Ardour to access it. The server's URL and docroot configuration can be accessed via Ardour's menu → Edit → Preferences.

You should have a fast network connection (≥100Mbit/s, low latency (a switch performs better than router) and preferably a dedicated network) between the machine running Ardour and the video-server.

While not required for operation, it is handy to share the video-server's document-root file-system with the machine running Ardour. It is, however, required to

- browse to a video-file to open[7]
- display local video-monitor window
- import/transcode a video-file to the Ardour session folder
- extract audio from a video-file [7]
- export to video-file[7]

File-system sharing can be done using any network-file-system (video-files reside on the server, not the Ardour workstation) or using NAS. Alternatively a remote replica of the Ardour-project-tree that only contains the video-files is an option. The local project folder only needs a copy of the video-file for displaying a video-monitor.

The document-root configured in Ardour is removed from the local absolute-path to the selected file when making a request to the video-server.

Xjadeo itself includes a remote-control API that allows to control the video-monitor over a network connection. Please refer to the Xjadeo manual for details.

6.4 Known Issues

Video-monitor settings (window state, position, size) are sent to Ardour when the video-monitor is terminated. Ardour saves and restores these settings. The video-monitor is closed when a session is closed. If any of the video-settings have changed since the last save, session-close will ask again to save the session, even if it was saved just before closing. Every session-save should query and save the current state of the video-monitor to prevent this.

Operating Ardour with very close-up zoom (only one or two video-frames visible in the timeline), caching images only for the previous and next viewpoint is not sufficient. On playback video-frames are missing (black-cross images) in the timeline. The Ardour internal frame-cache should require a minimum of frames regardless of canvas pages to compensate for the request and redraw latency.

[7] later versions of the video-server and Ardour may not require this anymore.

7 Acknowledgements

Thanks go to Paul Davis, Luis Garrido, Rui Nuno Capela, Dave Phillips and Natanael Olaiz.

This project would not have been possible without various free-software projects, most notably ffmpeg/libav. Kudos go to the Ardour and JACK development-teams and to linux-audio-users for feedback.

Last but not least to Carolina Feix for motivating this project in the first place.

References

Rui Nuno Capela. 2005–2011. Qtractor. `http://qtractor.sourceforge.net/`.

Paul Davis et al. 1999–2012. Ardour. `http://ardour.org`.

Paul Davis et al. 2001–2012. Jack transport. `http://jackaudio.org/files/docs/html/transport-design.html`.

Gabriel Finch. 2004–2012. Lives. `http://lives.sourceforge.net/`.

Blender Foundation. 1999–2012. Blender. `http://blender.org`.

Robin Gareus and Luis Garrido. 2006–2012a. Time-stamp-movie-maker. `http://xjadeo.sf.net`.

Robin Gareus and Luis Garrido. 2006–2012b. Xjadeo - the x-jack-video-monitor. `http://xjadeo.sf.net`.

Robin Gareus. 2008–2012. Sodankylä, icsd. `http://gareus.org/wiki/a3vtl`.

Heroine Virtual Ltd. 2002–2011. Cinelerra. `http://cinelerra.org/`.

Piero Scaruffi. 2007. *A brief history of Popular Music before Rock Music*. Omniware.

Werner Schweer et al. 1999–2012. Musescore: Music score editor. .

Wikipedia. 2011. Film-score. `http://en.wikipedia.org/wiki/Film_score`.

INScore
An Environment for the Design of Live Music Scores

D. Fober and **Y. Orlarey** and **S. Letz**
Grame - Centre national de création musicale
9, rue du Garet BP 1185
69202 Lyon Cedex 01,
France,
{fober, orlarey, letz}@grame.fr

Abstract

INScore is an open source framework for the design of interactive, augmented, live music scores. Augmented music scores are graphic spaces providing representation, composition and manipulation of heterogeneous and arbitrary music objects (music scores but also images, text, signals...), both in the graphic and time domains. INScore includes also a dynamic system for the representation of the music performance, considered as a specific sound or gesture instance of the score, and viewed as *signals*. It integrates an *event based* interaction mechanism that opens the door to original uses and designs, transforming a score as a user interface or allowing a score self-modification based on temporal events. This paper presents the system features, the underlying formalisms, and introduces the OSC based scripting language.

Keywords

score, interaction, signal, synchronization

1 Introduction

Music notation has a long history and evolved through ages. From the ancient neumes to the contemporary music notation, the western culture is rich of the many ways explored to represent the music. From symbolic or prescriptive notations to pure graphic representation, the music score has always been in constant interaction with the creative and artistic process.

However, although the music representations have exploded with the advent of computer music [Dannenberg, 1993; Hewlett and Selfridge-Field, 2001], the music score, intended to the performer, didn't evolved in proportion to the new music forms. In particular, there is a significant gap between interactive music and the static way it is usually notated: a performer has generally a traditional paper score, plus a computer screen displaying a number or a letter to indicate the state of the interaction system. New needs in terms of music representation emerge of this context.

In the domain of electro-acoustic music, analytic scores - music scores made *a postériori*, become common tools for the musicologists but have little support from the existing computer music software, apart the approach proposed for years by the Acousmograph [Geslin and Lefevre, 2004].

In the music pedagogy domain and based on a mirror metaphor, experiments have been made to extend the music score in order to provide feedback to students learning and practicing a traditional music instruments [Fober et al., 2007]. This approach was based on an extended music score, supporting various annotations, including performance representations based on the audio signal, but the system was limited by a monophonic score centered approach and a static design of the performance representation.

Today, new technologies allow for real-time interaction and processing of musical, sound and gestural information. But the symbolic dimension of the music is generally excluded from the interaction scheme.

INScore has been designed in answer to these observations. It is an open source framework[1] for the design of interactive, augmented, live music scores. It extends the traditional music score to arbitrary heterogeneous graphic objects: it supports symbolic music notation (Guido [Hoos et al., 1998] or MusicXML [Good, 2001] format), text (utf8 encoded or html format), images (jpeg, tiff, gif, png, bmp), vectorial graphics (custom basic shapes or SVG), video files, as well as an original performance representation system [Fober et al., 2010b].

Each component of an augmented score has a

[1] http://inscore.sf.net

graphic and temporal dimension and can be addressed in both the graphic and temporal space. A simple formalism, is used to draw relations between the graphic and time space and to represent the time relations of any score components in the graphic space. Based on the graphic and time space segmentation, the formalism relies on simple mathematical relations between segments and on relations compositions. We talk of *time synchronization in the graphic domain* [Fober et al., 2010a] to refer to this specific feature.

INScore is a message driven system that makes use of the Open Sound Control [OSC] [2] format. This design opens the door to remote control and to interaction using any OSC capable application or device. In addition, it includes interaction features provided at score component level by the way of *watchable events*.

All these characteristics make INScore a unique system in the landscape of music notation or of multi-media presentation. The next section gives details about the system features, including the underlying formalisms. Next the OSC API is introduced with a special emphasis on how it is turned into a scripting language. The last section gives an overview of the system architecture and dependencies before some directions are presented for future work.

2 Features

Table 1 gives the typology of the graphic resources supported by the system. All the score elements have graphic properties (position, scale, rotation, color, etc.) and time properties as well (date and duration).

INScore provides a message based API to create and control the score elements, both in the graphic and time spaces. The Open Sound Control [OSC] protocol is used as basis format for these messages, described in section 3.

2.1 Time to graphic relations

All the elements of a score have a time dimension and the system is able to graphically represent the time relationships of these elements. INScore is using segmentations and *mappings* to achieve this *time synchronization in the graphic domain*.

The segmentation of a score element is a set of segments included in this element. A segment may be viewed as a portion of an element. It could be a 2D graphic segment (a rectangle), a time interval, a text section, or any segment describing a part of the considered element in its *local* coordinates space.

Mappings are mathematical relations between segmentations. A composition operation is used to relate the graphic space of two arbitrary score elements, using their relation to the time space. Table 2 lists the segmentations and mappings used by the different component types. Mappings are indicated by arrows (↔). Note that the arrows link segments of different types. Segmentations and mappings in *italic* are automatically computed by the system, those in **bold** are user defined.

Note that for music scores, an intermediate time segmentation, the *wrapped time*, is necessary to catch repeated sections and jumps (to sign, to coda, etc.).

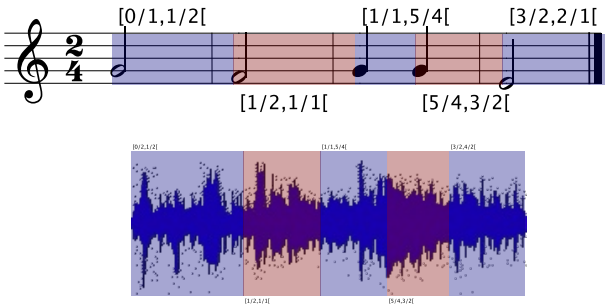


Figure 1: Graphic segments of a score and its performance displayed using colors and annotated with the corresponding time segments.

Figure 1 shows segments defined on a score and an image, annotated with the corresponding time segments. Synchronizing the image to the score will stretch and align the graphic segments corresponding to similar time segments as illustrated by figure 2.

Description of the image graphic to time relation is given in table 3 as a set of pairs including a graphic segment defined as 2 intervals on the x and y axis, and a time segment defined as 2 rationals expressing musical time (where 1 represents a whole note).

[2] http://opensoundcontrol.org/

Symbolic music description	Guido Music Notation and MusicXML formats
Text	plain text or html (utf8)
Images	jpg, gif, tiff, png, bmp
Vectorial graphics	line, rectangle, ellipse, polygon, curve, SVG code
Video files	using the phonon plugin
Performance representations	see section 2.2

Table 1: Graphic resources supported by the system.

type	segmentations and mappings required
Symbolic music description	*graphic* ↔ *wrapped time* ↔ *time*
Text	*graphic* ↔ **text** ↔ **time**
Images	*graphic* ↔ **pixel** ↔ **time**
Vectorial graphics	*graphic* ↔ **vectorial** ↔ **time**
Performance representations	*graphic* ↔ **frame** ↔ **time**

Table 2: Segmentations and mappings for each component type

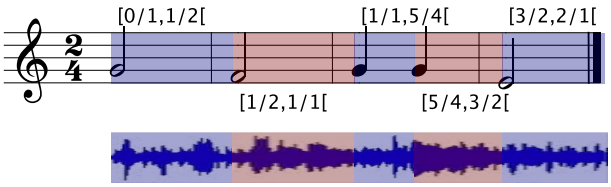


Figure 2: Time synchronization of the segments displayed in figure 1.

([0, 67[[0, 86[ )	([0/2, 1/2[ )
([67, 113[[0, 86[ )	([1/2, 1/1[ )
([113, 153[[0, 86[ )	([1/1, 5/4[ )
([153, 190[[0, 86[ )	([5/4, 3/2[ )
([190, 235[[0, 86[ )	([3/2, 4/2[ )

Table 3: A mapping described as a set of relations between graphic and time segments.

2.2 Performance representation

Music performance representation is based on signals, whether audio or gestural signals. To provide a flexible and extensible system, the graphic representation of a signal is viewed as a *graphic signal*, i.e. as a composite signal made of:

- a y coordinate signal
- a thickness signal h
- a color signal c

Such a composite signal (see figure 3) includes all the information required to be drawn without additional computation.

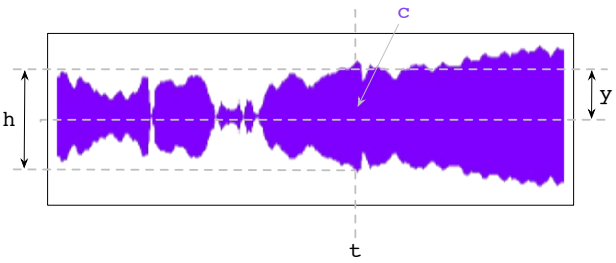


Figure 3: A composite *graphic signal* at time t.

The following examples show some simple representations defined using this model.

2.2.1 Pitch representation

Represents notes pitches on the y-axis using the fundamental frequency (figure 4).

Figure 4: Pitch representation.

The corresponding graphic signal is expressed as:

$$g = S_{f0} \ / \ k_t \ / \ k_c$$

where S_{f0} : fundamental frequency

k_t : a constant thickness signal
k_c : a constant color signal

2.2.2 Articulations

Makes use of the signal RMS values to control the graphic thickness (figure 5). The corresponding

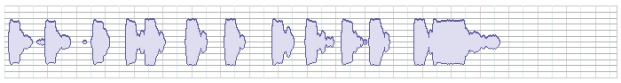


Figure 5: Articulations.

graphic signal is expressed as:

$$g = k_y \ / \ S_{rms} \ / \ k_c$$

where k_y : signal y constant
S_{rms} : RMS signal
k_c : a constant color signal

2.2.3 Pitch and articulation combined

Makes use of the fundamental frequency and RMS values to draw articulations shifted by the pitches (figure 6).

Figure 6: Pitch and articulation combined.

The corresponding graphic signal is expressed as:

$$g = S_{f0} \ / \ S_{rms} \ / \ k_c$$

where S_{f0} : fundamental frequency
S_{rms} : RMS signal
k_c : a constant color signal

2.2.4 Pitch and harmonics combined

Combines the fundamental frequency to the first harmonics RMS values (figure 7). Each harmonic has a different color.

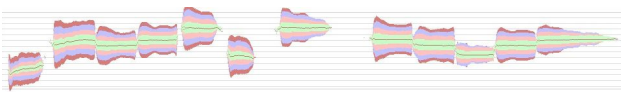


Figure 7: Pitch and harmonics combined.

The graphic signal is described in several steps. First, we build the fundamental frequency graphic as above (see section 2.2.3) :

$$g0 = S_{f0} \ / \ S_{rms0} \ / \ k_c0$$

where S_{f0} : fundamental frequency
S_{rms0} : f0 RMS values
k_c0 : a constant color signal

Next we build the graphic for the harmonic 1:

$$g1 = S_{f0} \ / \ S_{rms1} + S_{rms0} \ / \ k_c1$$

S_{rms1} : harmonic 1 RMS values
k_c1 : a constant color signal

Next, the graphic for the harmonic 2:

$$g2 = S_{f0} / \ S_{rms2} + S_{rms1} + S_{rms0} \ / \ k_c2$$

S_{rms2} : harmonic 2 RMS values
k_c2 : a constant color signal

etc.

Finally, the graphic signals are combined into a parallel graphic signal:

$$g = g2 \ / \ g1 \ / \ g0$$

2.3 Interaction

INScore is a message driven system that makes use of Open Sound Control [OSC] format. It includes interaction features provided at individual score component level by the way of *watchable events*, which are typical events available in a graphic environment, extended in the temporal domain. The list of supported events is given in table 4.

mouse events	time events	misc.
mouse enter	time enter	new element
mouse leave	time leave	(at scene level)
mouse down		
mouse up		
mouse move		
double click		

Table 4: Typology of *watchable* events.

Events are associated to user defined messages that are triggered by the event occurrence. The message includes a destination address (INScore by default) that supports a url-like specification, allowing to target any IP host on any UDP port. The message associated to mouse events may use predefined variables, instantiated at event time with the current mouse position or the current position time.

This simple event based mechanism makes easy to describe for example an *intelligent cursor* i.e. an arbitrary object that is synchronized to the score and that turns the page to the next or previous one, depending on the time zone it enters.

3 INScore messages

The INScore API is an OSC messages API. The general format is illustrated in figure 8: it consists in an address followed by a message string, followed by parameters.

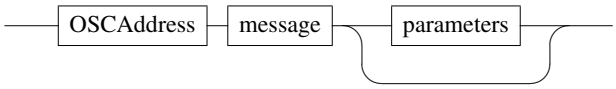


Figure 8: General format of a message.

The address may be viewed as an object pointer, the message string as a method name and the parameters as the method parameters. For example, the message:

```
/ITL/scene/score color 255 128 40 150
```

may be viewed as the following method call:

```
score->color(255 128 40 150)
```

3.1 Address space

The OSC address space is strictly organized like the internal score representation, which is a tree with 4 depths levels (figure 9). Messages can be sent to any level of the hierarchy.

The first level is the application level: a static node with a fixed address that is also used to discriminate incoming messages. An application contains several scenes, which corresponds to different windows and different scores. Each scene contains components and two static nodes: a *sync* node for the components synchronization and a *signal* node, that may be viewed as a special folder containing signals. The name in blue are user defined, those in black are static reserved names.

3.2 Message strings

This section gives some of the main messages supported by all the score components. The list is far from being exhaustive, it is intended to show examples of the system API.

Score components are created and/or modified using a `set` message (figure 10) that takes the

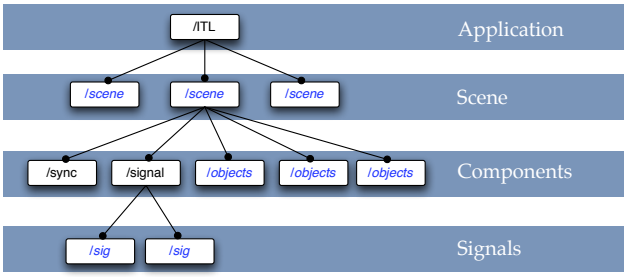


Figure 9: INScore address space.

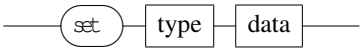


Figure 10: The `set` message.

object type as argument, followed by type specific parameters.

Messages given in figure 11 are used to set the objects graphic properties.

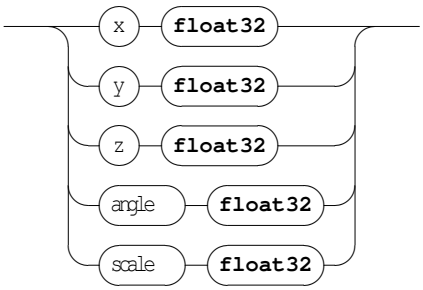


Figure 11: Graphic space management.

Messages given in figure 12 set the time properties. Time is encoded as rational values representing music time (where 1 is a whole note). Graphic space and time management messages have relative positioning forms: the message string prefixed with 'd'.

All the messages that modify the system state have a counterpart `get` form illustrated by figure 13.

Synchronization between components is based on a master / slave scheme. It is described by a message addressed to the static `sync` node as illustrated by figure 14. `syncmode` indicates how to align the slave component to its master: horizontal and/or vertical stretch, etc.

Interactive features are available by requesting to a component to watch an event using

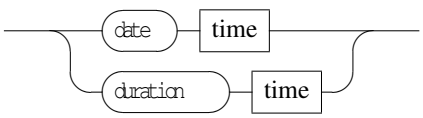


Figure 12: Time management.

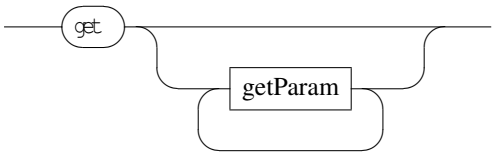


Figure 13: Querying the system state.

a message like figure 15, where `what` indicates the event (mouseUp, mouseDown, mouseEnter, mouseLeave, timeEnter, timeLeave) and `OSCMsg` represents any valid OSC message including the address part.

The example in figure 16 creates a simple score, do some scaling, creates a red cursor and synchronizes it to the score with a vertical stretch to the score height. Output is presented by figure 17.

3.3 INScore scripts

Actually, the example given in figure 16) is making use of the file format of a score, which consists in a list of *textual OSC messages*, separated by a semi-colon. This textual form is particularly suitable to be used as a scripting language and additional support is provided in the form of variables and javascript and/or lua support.

3.3.1 Variables

INScore scripts supports variables declarations in the form illustrated by figure 18. Variables may be used in place of any message parameter prefixed using a $ sign. A variable must be declared before being used.

3.3.2 Scripting support

INScore scripts may include javascript sections, delimited by `<?javascript` and `?>` The javascript code is evaluated at parse time. It is expected to produce valid INScore messages as output. These messages are then expanded in place of the javascript code.

Variables declared before the javascript section are exported to the javascript environment, which make these variables usable in both contexts.

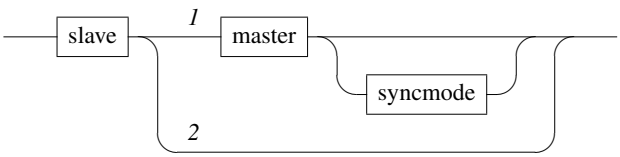


Figure 14: Synchronization message: the second form (2) removes the synchronization from the `slave` object.

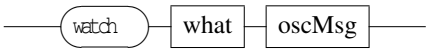


Figure 15: Watching events.

Similarly, INScore may support the lua language in sections delimited by `<?lua` and `?>`. By default, lua support is not embedded in the INScore binary distribution. The engine needs to be compiled with the appropriate options to support lua.

4 Architecture

INScore is both a shared library and a standalone score viewer application without user interface. Some insight of the system architecture is given in this section for a better understanding and an optimal use of the system. The general architecture is a Model View Controller [MVC] designed to handle OSC message streams.

4.1 The MVC Model

The MVC architecture is illustrated in figure 19. Modification of the model state is achieved by incoming OSC messages or by the library C/C++ API, that is actually also message based. An OSC message is packaged into an *internal messages* representation and stacked on a lock-free fifo stack. These operations are synchronous to the incoming OSC stream and to the library API call.

On a second step, messages are popped from the stack by a `Controler` that takes in charge

```
/ITL/scene/score set 'gmn' '[g e f d]';
/ITL/scene/score scale 5.0;
/ITL/scene/cursor set 'rect' 0.01 0.1;
/ITL/scene/cursor color 255 0 0 150;
/ITL/scene/sync 'cursor' 'score' 'v';
```

Figure 16: INScore sample script.

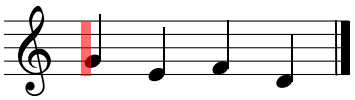


Figure 17: Sample script output.

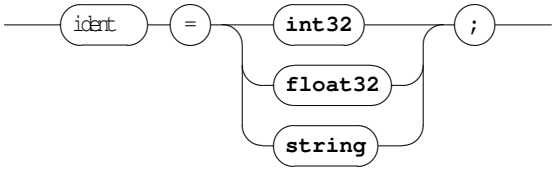


Figure 18: Variable declaration.

address decoding and message passing to the corresponding object of the model. The final operation concerns the *view* update. An `Updater` is in charge of producing a view of the model, which is currently based on the Qt Framework.

This second step of the model modification scheme is asynchronous: it is processed on a regular time base.

4.2 Dependencies

The INScore project depends on external open source libraries:

- the Qt framework[3] providing the graphic layer and platform independence.
- the GuidoEngine[4] for music score layout
- the GuidoQt static library, actually part of the Guido library project
- the oscpack library[5] for OSC support
- and optionally:
  - the MusicXML library[6] to support the MusicXML format.
  - the v8 javascript engine[7] to support javascript in INScore script files.
  - the lua engine[8] to support lua in INScore script files.

[3] http://qt.nokia.com/
[4] http://guidolib.sourceforge.net
[5] http://www.rossbencina.com/code/oscpack
[6] http://libmusicxml.sourceforge.net
[7] http://code.google.com/p/v8/
[8] http://www.lua.org/

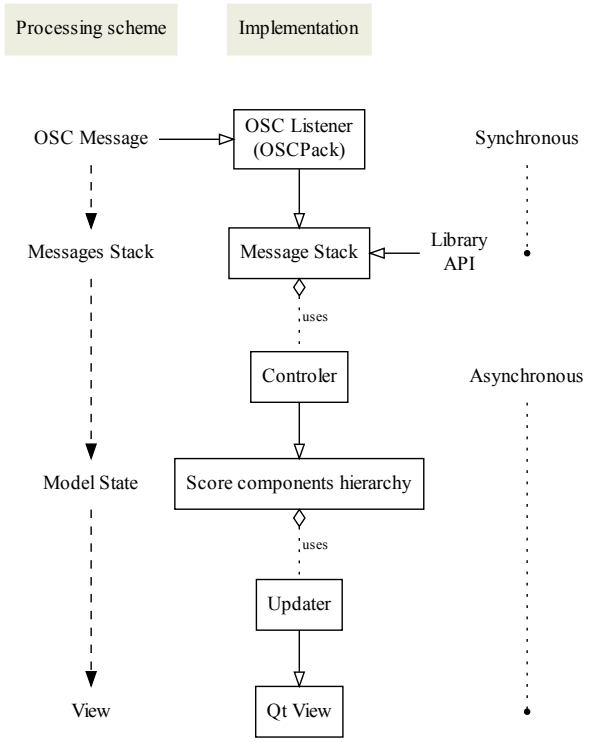


Figure 19: An overview of the MVC architecture

The GuidoEngine, the GuidoQt and oscpack libraries are required to compile INScore.

Binary packages are distributed for Ubuntu Linux, Mac OS and Windows. Detailed instructions are included in the project repository for compiling.

5 Future work

Interactive features described in section 2.3 result from an experimental approach. The formalization and extension of these features are planned.

Time synchronization features are based on a continuous music time. Supporting other time representations like the Allen relations [Allen, 1983] is part of the future work.

Due to its dynamic nature, INScore is particularly suitable to interactive music, i.e. where parts of the music score could be computed in real-time by an interaction system. It could be particularly interesting to provide a musically meaningful visualization of the interaction system state; extensions in this direction are also planned.

6 Acknowledgements

INScore was initiated in the Interlude project funded by the French National Research Agency [ANR- 08-CORD-010].

References

James F. Allen. 1983. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26:832–843, November.

R. B. Dannenberg. 1993. Music representation issues, techniques and systems. *Computer Music Journal*, 17(3):20–30.

D. Fober, S. Letz, and Y. Orlarey. 2007. Vemus - feedback and groupware technologies for music instrument learning. In *Proceedings of the 4th Sound and Music Computing Conference SMC'07 - Lefkada, Greece*, pages 117–123.

D. Fober, C. Daudin, S. Letz, and Y. Orlarey. 2010a. Time synchronization in graphic domain - a new paradigm for augmented music scores. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 458–461.

D. Fober, C. Daudin, Y. Orlarey, and S. Letz. 2010b. Interlude - a framework for augmented music scores. In *Proceedings of the Sound and Music Computing conference - SMC'10*, pages 233–240.

Yann Geslin and Adrien Lefevre. 2004. Sound and musical representation: the acousmographe software. In *ICMC'04: Proceedings of the International Computer Music Conference*, pages 285–289. ICMA.

M. Good. 2001. MusicXML for Notation and Analysis. In W. B. Hewlett and E. Selfridge-Field, editors, *The Virtual Score*, pages 113–124. MIT Press.

Walter B. Hewlett and Eleanor Selfridge-Field, editors. 2001. *The Virtual Score; representation, retrieval and restoration.* Computing in Musicology. MIT Press.

H. Hoos, K. Hamel, K. Renz, and J. Kilian. 1998. The GUIDO Music Notation Format - a Novel Approach for Adequately Representing Score-level Music. In *Proceedings of the International Computer Music Conference*, pages 451–454. ICMA.

A Behind-the-Scenes Peek at World's First Linux-Based Laptop Orchestra – The Design of L2Ork Infrastructure and Lessons Learned

Ivica Ico Bukvic
Virginia Tech, DISIS, L2Ork, ICAT
Department of Music - 0240
Blacksburg, VA, 24061
ico@vt.edu

Abstract

In recent years we've experienced a proliferation of laptop orchestras and ensembles. Linux Laptop Orchestra or L2Ork founded in the spring 2009 introduces exclusive reliance on open-source software to this novel genre. Its hardware design also provides minimal cost overhead. Most notably, L2Ork provides a homogenous software and hardware environment with focus on usability and transparency. In the following paper we present an overview of L2Ork's infrastructure and lessons learned through its design and implementation.

Keywords

Linux Laptop Orchestra, L2Ork, Infrastructure, Lessons Learned

1 Introduction

Laptop orchestras arguably need no introduction. Their number is now growing at an exponential rate with new ensembles being introduced at Universities and other independent institutions across the world ("International Association of Laptop Orchestras," n.d.). With the ensuing growth, the laptop ensemble repertoire is increasingly hampered by the lack of software and hardware standardization. The mounting cost of obtaining and/or building supporting infrastructure has resulted in compromises, further exacerbating the said fragmentation (I. Bukvic, Martin, Standley, & Matthews, 2010; Kapur et al., 2010; Trueman, Cook, Smallwood, & Wang, 2006; Wang, Bryan, Oh, & Hamilton, 2009). One notable aspect of ensembles based on the PLOrk model (Trueman et al., 2006), is the use of hemispherical speakers whose purpose is to provide co-located sound. Prior to the introduction of the hemispherical speakers, the computer music genre has primarily relied on either acousmatic spatialization or a more traditional stereoscopic means of sound reproduction. Both of the said approaches pose a challenge in that they can disassociate the music-making process with its source. In such a setting typically one or more computer musicians located on stage making (or mixing) sound are being heard across a vast array of speakers, none of which correspond with their physical location. This is where hemispherical speakers make a considerable difference in bridging the said cognitive gap (Smallwood, Cook, & Trueman, 2009) and reintroducing the critical notion of co-located sound inherent to acoustic music instruments.

2 Introducing L2Ork

Linux Laptop Orchestra or L2Ork (I. Bukvic et al., 2010) was founded in Spring 2009 as part of a research initiative with focus on providing a discipline-agnostic point of convergence for arts, technology, and engineering. The initiative has gained considerable traction, attracting twenty on-campus stakeholders and four corporate sponsors. In part due to its open design, and in part due to its ambitious goal of complementing K-12 (elementary school to high school) education by introducing this unusual integration of arts and sciences, L2Ork has also received an unprecedented amount of media attention, including the front cover feature in the Linux Journal (Phillips, 2010). By partnering with the regional Boys & Girls Club (an after-school program for inner city children) and through the help of a series of grants the project has produced a smaller complementing satellite laptop orchestra for the purpose of training 4th and 5th graders in the newfound art of making laptop ensemble music

(Ivica Bukvic, Martin, & Matthews, 2011). Since its inception, the ensemble has had four major tours, including a twenty-one-day tour of Europe with stops at STEIM and IRCAM. Last February, a work titled *Half-Life* won the first place on the first international laptop orchestra composition competition (Elliott, 2011). Apart from the fact that the ensemble is a result of academic research and a part of academic curriculum, one of its core goals is also to establish L2Ork as a professional and active ensemble outside the academic circles.

By its very design, the ensemble attracts students from various backgrounds, amounts of musical training, and/or computer literacy. For this reason, a part of the project's focus is on the development of optimal interfaces for the delivery of critical musical cues with the assumption that users have no prior musical experience. This challenge is further amplified by ensemble's focus on physical presence and choreography, most recently with the inclusion of Taiji (Tai Chi) mind-body practice and system's ability to "harvest" ensuing motion data and translate it into meaningful musical cues audience can clearly associate with.

Four unique, yet mutually dependent components that in many ways define the ensemble's approach to the laptop orchestra genre are:

1. interest in developing physical presence and choreography;
2. exclusive reliance on open-source;
3. affordable design, and
4. focus on homogenous environment and optimal usability.

In the following section we will visit some of the most notable challenges and solutions to problems related to both the genre and the Linux platform.

3 Homogenous Environment

One of the challenges in maintaining a laptop orchestra is the uncertainty caused by the use of individualized software platforms. Even for ensembles relying on fairly homogenous environments, such as Windows and OSX, hurdles remain in terms of installed software, some of which can at times collide with each other and cause compatibility problems. Such challenges can easily waste a majority of time allocated for a rehearsal on seemingly trivial things, often resulting in a frustrating experience. There is clearly a need for a turnkey solution, a system that simply does what it is meant to do. The author argues that chances of achieving such an environment, particularly within the Linux ecosystem, is to ensure that the OS and supporting hardware are truly homogenous. L2Ork achieves this by essentially loaning out fully preinstalled and preconfigured systems together with supporting hardware to students/users who are expected to treat the ensuing amalgamation of hardware and software as an integrated instrument. This is where the affordable design plays a critical role. With each seat relying on an MSI Wind Intel Atom-based notebook, the total cost per station at $750 is typically less than most mainstream mobile devices and their requisite audio hardware. Given the convenient name, each station was assigned a letter of the alphabet, a corresponding wind name (e.g. Austru, Briza, Cyclone, etc.) and a static IP address. IP addresses are distributed evenly starting with 192.168.2.10 for Austru, *.12 for Briza, *.14 for Cyclone, etc. with the remaining odd IP addresses reserved for projected future growth.

Another advantage of such low-power setup is in the way it encourages creation of new content for the ensemble. The severe computational limits of individual computers in the ensemble literally force artists to approach new compositions by thinking about each station as a small piece of a much larger ecosystem, rather than composing essentially entire work on a single laptop and then "exploding" parts to individual machines. Based on the experience accrued through L2Ork, it is author's belief that the two approaches yield distinctly different results for performers and audience alike. Other advantages related to the choice of the aforesaid netbooks is reliable open-source driver support. As a result, L2Ork's setup supports all functions provided by the hardware, including standby, hibernate, and other advanced functionality that may be of use during pre-concert tech setup. For a more detailed breakdown of L2Ork's hardware infrastructure, please consult the NIME 2011 publication (I. Bukvic et al., 2010).

While initially based on modified Ubuntu 9.04 and 9.10 releases, L2Ork currently relies on the mainstream Ubuntu 10.04 LTS distribution. In the near term the ensemble is looking to migrate to a newer hardware with the next target OS choice likely being Ubuntu 12.04. With every Ubuntu

upgrade, the author has identified fewer system modifications that were necessary to produce optimal configuration. The most notable alteration to the current iteration is the real-time kernel that enables even low-power Atom-based notebooks to deliver reliable low-latency performance. To further streamline system use in both rehearsal and performance scenarios the desktop has been enhanced with a set of operational and usability tweaks and customizations. Operational enhancements deal with use-specific needs that minimize potentially undesirable degradation in system's performance. These include real-time priorities for audio-related tasks, appropriate HD sleep and swappiness settings that abate potential xruns caused by HD usage, CPU throttling policy that automatically detects JACK server running and transitions in the "performance" mode, disabling screen savers and screen sleep timers that may interfere with the display during rehearsals and performances, associating static local IP address through the wired ethernet interface, and automatic reloading of bluetooth driver upon resuming the notebook to minimize potential problems with Wiimote pairing. In addition, usability enhancements include a series of shell script wrappers that provide turnkey initialization of various pieces stored in a form of application shortcuts (we will discuss these further below), as well as use of compiz ("Compiz Home," n.d.) for the purpose of providing a more responsive and user-friendly desktop environment. A limited number of desktop shortcuts provides access to the most common functionality, like Web browsing, as well as a shortcut for force-quitting potential runaway processes. For the purpose of minimizing initial setup overhead, every system by default has auto-login enabled and the desktop's appearance is streamlined to promote seamless performer migration among different stations, should such a need arise.

3.1 Synchronization

L2Ork's infrastructure supports up to sixteen performers and despite its homogeneous design, several challenges arose from the ongoing attempts to synchronize stations and provide a new way to affect performance structure by cross-pollinating control data. A number of works written specifically for the ensemble rely on this component—for example in composition *Rain* one performer's action audibly percolates through other systems. Consequently, increased performer activity can quickly result in a relatively high amount of traffic to the point where the ensuing aural soundscape is built from stochastic percolations of short attack-based sounds propagated through the network. To achieve the said goal in this and other similar scenarios we rely primarily upon control data. Given the ensemble relies on wired, rather than wireless network, it is also possible to exchange high-bandwidth audio data among different members of the ensemble. In such, potentially high-network-traffic environment, wireless communication has proved surprisingly unreliable. Even when experimenting with industrial grade Cisco wireless router in conjunction with TCP packets over an isolated local network, we still encountered one or more machines receiving time-critical cues up to a second later. Therefore wireless communication has proven inadequate for any kind of synchronized music production.

The ensuing local area network is shielded from the external network traffic and thus limits potential network packet collisions. For this reason, all notebooks broadcast control data using UDP packets, enabling the individual stations to simply filter streams they wish to monitor and/or interact with. This has allowed the system to have a high flexibility requiring minimal configuration (Fig.1).

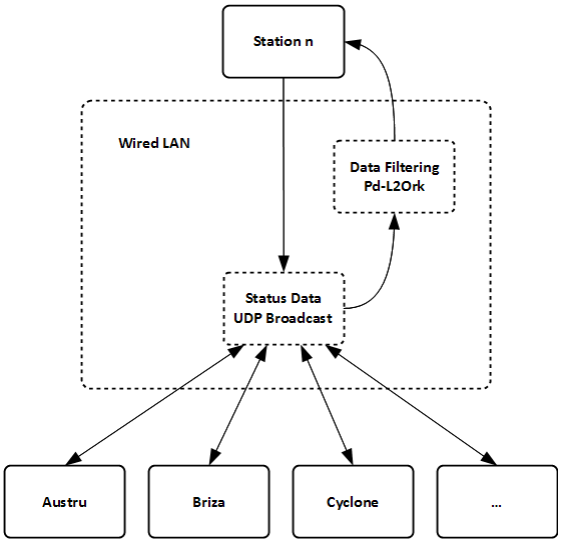


Figure 1. System synchronization.

Another challenge associated with synchronization is related to a seemingly trivial

matter of connecting concurrently up to 16 co-located Nintendo Wiimotes. Discovering devices in such an environment simply isn't an option as there is no guarantee that the Wiimote will pair with the right station. For this reason, the system uses ethernet connectivity with predetermined local IP address as the foundation for all synchronization with external devices through the use of a series of shell scripts. All shell scripts are run as part of initialization sequence for each of the pieces. *whatismyip* script is in charge of detecting local machine's IP address that may be used for matching score part with the station/performer. *whatismywiimote* uses IP address to pair it with an appropriate Wiimote MAC address. The reason we did not rely on computer's MAC address as a means of identifying individual stations is because, unlike the hard-wired MAC address, the IP configuration can be easily altered. Therefore, through a simple alteration of the IP address, any station can be re-purposed to take over a role of a different computer in the ensemble. This has proved critical in situations where a potential hardware failure has rendered a particular station inoperable and yet where a work required first n contiguous machines, as is the case with the aforesaid *Rain* composition that echoes individual aural events through the ensemble.

Finally, as the ensemble grew, there was a growing overhead associated with patching and synchronization of each system, including latest L2Ork-related software updates, as well as revisions to compositions and the supporting software. This has proven a problem of exponential nature often requiring hours to administer on all 16 machines. It has also proven prone to human error due to its repetitive nature. Therefore, the ensemble developed a series of shell scripts (*l2ork-send* and *l2ork-do*) that akin to GRENDL technology (Beck, Jha, Ullmer, Branton, & Maddineni, 2010) provides near real-time synchronization with two notable differences:

1. The L2Ork synchronization system relies on the git framework, and
2. L2Ork is a homogeneous environment, which has made synchronization considerably easier (Fig.2).

As a result, at the beginning of every rehearsal, systems are synced from the instructor/developer's master machine in a matter of seconds. Additional supporting tools were provided using ssh and sftp protocols where instructor is capable of uploading new versions of system software that needs to be applied locally using administrator permissions. Such files are accompanied by an installer that administers all changes with minimal interaction from users. All these changes have vastly minimized the amount of time required to administer the ensemble, as well as provided for a more enjoyable experience among its users, many of whom have limited computer literacy and/or knowledge of music.

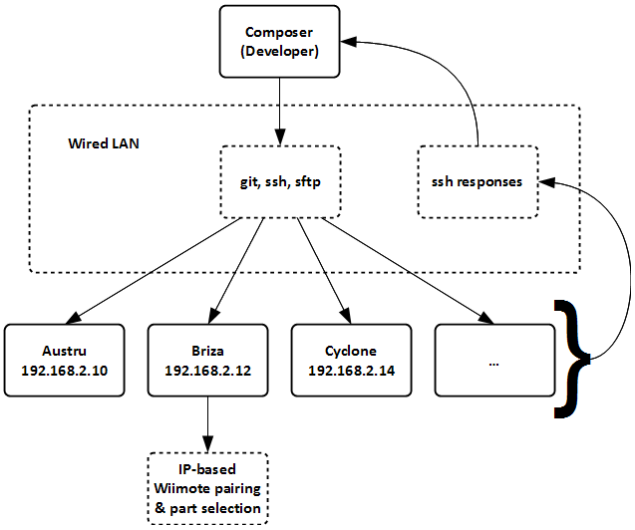


Figure 2. System synchronization.

3.2 Pd-L2Ork

For its audio-related digital signal processing and interfacing with external devices, L2Ork relies exclusively on the combination of JACK server (Davis & others, n.d.) and in-house version of Pure-Data (PD) (Puckette, 1996) titled Pd-L2Ork (Ivica Bukvic, n.d.). Former is integrated through QjackCtl JACK front-end that is also responsible for running a series of start-up scripts to ensure that the OS is running optimally (e.g. disabling wireless network to minimize potential confusion from having two active network connections, and setting CPU into performance mode).

Pd-L2Ork is an ambitious project in and of itself. Initially, the project's output was limited to upstream

patches. In the fall of 2010, however, mainly due to increasingly divergent interests and needs between L2Ork and the upstream PD distribution, the author decided to create a fork. Since, the project has significantly altered the PD's core functionality with more than two hundred bug-fixes and new features. Based on the 0.42.6 branch of Pd-Extended ("Pd-extended — PD Community Site," n.d.), Pd-L2Ork focuses on two key areas: robust and stable operation of both the audio engine and GUI, and usability improvements to the runtime environment and editor. Unlike early iterations, more recent versions of Pd-L2Ork have had a crash-free streak for over a year, including over twenty evening-long shows and performances. The system's GUI is more resilient in high-bandwidth scenarios, while a series of new objects, including a threaded implementation of Linux-specific Nintendo Wiimote external that allows bi-directional communication without xruns have further enhanced its usability. The *disis_wiimote* external is of particular importance as L2Ork makes extensive use of haptic feedback to allow performers to monitor critical operation, even sense tempo and beat, without having to look at the screen, something that has proven particularly useful in practicing Taiji choreography.

Another notable area of Pd-L2Ork are extensive improvements to the editor. Some of them include improved scrolling algorithm, fixing all known GOP redrawing bugs and instabilities, addressing all known major bugs, adding infinite undo, ability for iemgui objects' and canvas' properties to be altered via GUI handles rather than through a separate properties window, and the re-skinning the GUI to give the application a more contemporary feel. Most of the core editing actions now rely on Tcl toolkit's "tags," e.g. displacing a large number of objects. This change has resulted in a significantly lower CPU footprint. Based on preliminary tests, moving a modest graph-on-parent-enabled abstraction now yields no noticeable CPU overhead on a low-power Intel i3 processor, while the same action using continuous redraws (legacy approach) encumbers up to 15% of the CPU on the same hardware. Objects can be also easily layered on top of each other and the *magicglass* signal monitoring tool has offered a more efficient approach to debugging. All of these features have helped streamline the production of performance interfaces (e.g. input/output monitoring and visual score engine) within PD's environment. Pd-L2Ork project is certainly interested in seeing its contributions adopted upstream. However, given the lack of control over this process or its pace, it is author's belief that maintaining a separate version has allowed the ensemble to dramatically increase the pace of software's development and as such remains the preferred choice.

4 Lessons Learned

In the world of computing there is a saying that the last 20% towards fine-tuning a turnkey system take 80% of the overall project time. Undoubtedly, the same holds true for L2Ork as well. Many of the seemingly trivial bug-fixes to the core Linux system and more notably pd-l2ork code base have taken weeks if not months to identify. Now, we are finally in a position where our integrated system is observed by a newcomer as something that just works. L2Ork infrastructure is therefore regarded as one fully integrated whole and the fact that it relies on Linux, beyond the obvious potential benefits of such an approach, are becoming entirely transparent to the user. It is author's conviction that this is where Linux's future lies. It is not the advocacy, or that curious yet often incomplete feature. It is the transparency coupled with unprecedented flexibility and power of such a system that make it a truly powerful platform. It is truly those last 20% that have consumed countless hours and yet rewarded in unforeseen ways.

Ironically, as we look forward to L2Ork's next rehearsal and a tour, the excitement behind the fact that the infrastructure is robust and stable has all but worn off. And even though some of us still remember the countless hours spent on many of pd-l2ork's novel features, we've finally arrived at the point where one can simply expect nothing less than stability, thinking this is simply the way it's supposed to be.

For additional information on the project, pd-l2ork and other supporting software, and video instructables, visit L2Ork at http://l2ork.music.vt.edu.

5 Acknowledgments

The author would like to hereby thank all L2Ork's Stakeholders and Sponsors without whose vision and support this project would've never seen the light of the day. Stakeholders are (listed in alphabetic order): Virginia Tech Alumni Relations,

the Arts Initiative, College of Architecture and Urban Studies, Collaborative for Creative Technologies in the Arts and Design, Center for Human-Computer Interaction, Center for Instructional Design and Educational Research, College of Liberal Arts & Human Sciences, Creative Technologies Experiential Galleria, Digital Interactive Sound & Intermedia Studio, Institute for Creativity, Arts & Technology, Institute for Critical Technology and Applied Science, Institute for Distance and Distributed Learning, Institute for Society, Culture & Environment, Information Technology, Learning Technologies, Department of Music, Office of Academic Assessment, Outreach & International Affairs, School of Education, and the School of Performing Arts and Cinema. Sponsors are MSI Computer Inc., Renoise Inc., Roland Corporation, and Sweetwater Inc.

References

Beck, S. D., Jha, S., Ullmer, B., Branton, C., & Maddineni, S. (2010). GRENDL: grid enabled distribution and control for Laptop Orchestras. *ACM SIGGRAPH 2010 Posters* (p. 84).

Bukvic, I., Martin, T., Standley, E., & Matthews, M. (2010). Introducing L2Ork: Linux Laptop Orchestra. *Interfaces*, (Nime), 170–173.

Bukvic, Ivica. (n.d.). Virginia Tech Department of Music L2Ork - Software - Linux Laptop Orchestra. Retrieved February 8, 2012, from http://l2ork.music.vt.edu/main/?page_id=56

Bukvic, Ivica, Martin, T., & Matthews, M. (2011). Moving Beyond Academia Through Open Source. Solutions–Introducing L2Ork, Virginia Tech's Linux Laptop Orchestra. *Journal SEAMUS*.

Compiz Home. (n.d.). Retrieved February 28, 2012, from http://www.compiz.org/

Davis, P., & others. (n.d.). *JACK Audio Connection Kit (software)*.

Elliott, J. (2011, September 29). Virginia Tech's Linux Laptop Orchestra to perform at Virginia State Fair | Virginia Tech News | Virginia Tech. Retrieved February 28, 2012, from http://www.vtnews.vt.edu/articles/2011/09/092911-clahs-l2orkstatefair.html

International Association of Laptop Orchestras. (n.d.). Retrieved February 28, 2012, from http://ialo.org/doku.php/laptop_orchestras/orchestras

Kapur, A., Darling, M., Wiley, M., Vallis, O., Hochenbaum, J., Murphy, J., Diakopolus, D., et al. (2010). The Machine Orchestra. *Proceedings of the International Computer Music Conference* (pp. 554–558).

Pd-extended — PD Community Site. (n.d.). Retrieved February 8, 2012, from http://puredata.info/community/projects/software/pd-extended/?searchterm=pd-extended

Phillips, D. (2010, May). Introducing L20rk: the Linux Laptop Orchestra | Linux Journal. Retrieved February 28, 2012, from http://www.linuxjournal.com/article/10694

Puckette, M. (1996). Pure Data: another integrated computer music environment. *IN PROCEEDINGS, INTERNATIONAL COMPUTER MUSIC CONFERENCE*, 37—41.

Smallwood, S., Cook, P., & Trueman, D. (2009). Don't Forget the Laptop: A History of Hemispherical Speakers at 10. M. Kuo, D. Poxson, Y. Kim, F. Mont, J. Kim, E. Schubert, and S. Lin. *Proceedings of the New Instruments for Musical Expression*, Pittsburgh, PA, USA.

Trueman, D., Cook, P., Smallwood, S., & Wang, G. (2006). PLOrk: Princeton laptop orchestra, year 1. *Proceedings of the International Computer Music Conference* (pp. 443–450).

Wang, G., Bryan, N., Oh, J., & Hamilton, R. (2009). Stanford laptop orchestra (slork). *Proceedings of the International Computer Music Conference* (Vol. 505, p. 508).

Studio report: Linux audio for multi-speaker natural speech technology

Charles FOX, Heidi CHRISTENSEN and **Thomas HAIN**
Speech and Hearing
Department of Computer Science
University of Sheffield , UK

charles.fox@sheffield.ac.uk

Abstract

The Natural Speech Technology (NST) project is the UK's flagship research programme for speech recognition research in natural environments. NST is a collaboration between Edinburgh, Cambridge and Sheffield Universities; public sector institutions the BBC, NHS and GCHQ; and companies including Nuance, EADS, Cisco and Toshiba. In contrast to assumptions made by most current commercial speech recognisers, natural environments include situations such as multi-participant meetings, where participants may talk over one another, move around the meeting room, make non-speech vocalisations, and all in the presence of noises from office equipment and external sources such as traffic and people outside the room. To generate data for such cases, we have set up a meeting room / recording studio equipped to record 16 channels of audio from real-life meetings, as well as a large computing cluster for audio analysis. These systems run on free, Linux-based software and this paper gives details of their implementation as a case study for other users considering Linux audio for similar large projects.

Keywords

Studio report, case study, speech recognition, diarisation, multichannel

1 Introduction

The speech recognition community has evolved into a niche distinct from general computer audio and Linux audio in particular. It has its own large collection of tools, some of which have been developed continually for over 20 years such as the HTK Hidden Markov Model ToolKit [Young et al., 2006][1]. We believe there could be more crosstalk between the speech and Linux audio worlds, and to this end we present a report of our experiences in setting up a new Linux-based studio for dedicated natural speech research.

In contrast to assumptions made by current commercial speech recognisers such as Dragon Dictate, natural environments include situations such as multi-participant meetings [Hain et al., 2009], where participants may talk over one another, move around the meeting room, make non-sentence utterances, and all in the presence of noises from office equipment and external sources such as traffic and people outside the the room. The UK Natural Speech Technology project aims to explore these issues, and their applications to scenarios as diverse as automated TV programme subtitling; assistive technology for disabled and elderly health service users; automated business meeting transcription and retrieval, and homeland security.

The use of open source software is practically a prerequisite for exploratory research of this kind, as it is never known in advance which parts of existing systems will need to be opened up and edited in the course of research. The speech community generally works on offline statistical, large data-set based research. For example corpora of 1000 hours of audio are not uncommon and require the use of large compute clusters to process them. These clusters already run Linux and HTK, so it is natural to extend the use of Linux into the audio capture phase of research. As speech research progresses from clean to natural speech, and from offline to real-time processing, it is becoming more integrated with general sound processing [Wolfel and McDonough, 2009], for example developing tools to detect and classify sounds as precursors to recognition. The use of Bayesian techniques in particular emphasises the advantages of considering the sound process-

[1] Currently owned by Microsoft, source available *gratis* but not *libre*. Kaldi is a *libre* alternative currently under development, (kaldi.sourceforge.net).

ing and recognition as tightly coupled problems, and using tightly integrated computer systems. For example, it may be useful for Linux cluster machines running HTK in real-time to use high level language models to generate Bayesian prior beliefs for low-level sound processing occurring in Linux audio.

This paper provides a studio report of our initial experiences setting up a Linux based studio for NST research. Our studio is based on a typical meeting room, where participants give presentations and hold discussions. We hope that it will serve as a self-contained tutorial recipe for other speech researchers who are new to the Linux audio community (and have thus included detailed explanations of relatively simple Linux audio concepts). It also serves as an example of the audio requirements of the natural speech research community; and as a case study of a successful Linux audio deployment.

2 Research applications

The NST project aims to use a meeting room studio, networked home installations, and our analysis cluster to improve recognition rates in natural environments, with multiple, mobile speakers and noise sources. We give here some examples of algorithms relevant to natural speech, and their requirements for Linux audio.

Beamforming and ICA are microphone-array based techniques for separating sources of audio signals, such as extracting individual speakers from mixtures of multiple speakers and noise sources. ICA [Roberts and Everson, 2001] typically makes weak assumptions about the data, such as assuming that the sources are non Gaussian in order to find a mixing matrix M which minimises the Gaussian-ness over time t of the latent sources vector x_t, from the microphone array time series vectors y_t, in $y_t = Mx_t$.

ICA can be performed with as few microphones as there are sound sources, but gives improved results as the number of microphones increases. Beamforming [Trees, 2002] seeks a similar output, but can include stronger physical assumptions - for example known microphone and source locations. It then uses expected sound wave propagation and interference patterns to infer the source waves from the array data. Beamforming is a high-precision activity, requiring sample-synchronous accuracy between recorded channels, and often using up to 64 channels of simultaneous audio in microphone arrays (see for example the NIST Mark-III arrays [Brayda et al., 2005]).

Reverberation removal has been performed in various ways, using single and multi-channel data. In multi-channel settings, sample-synchronous audio is again used to find temporal correlations which can be used to separate the original sound from the echos. In the iPhone4 this is performed with two microphones but performance may increase with larger arrays [Watts, 2009].

Speaker tracking may use SLAM techniques from robotics, coupled with acoustic observation models, to infer positions of moving speakers in a room (eg. [Fox et al., 2012], [Christensen and Barker, 2010]). This can be used in conjunction with beamforming to attempt retrieval of individual speaker channels from natural meeting environments, and again relies on large microphone arrays and sample-accurate recording.

Part of the NST project called 'homeService' aims to provide a natural language interface to electrical and electronic devices, and digital services in people's homes. Users will mainly be disabled people with conditions affecting their ability to use more conventional means of access such as keyboard, computer mouse, remote control and power switches. The assistive technology (AT) domain presents many challenges; of particular consequence for NST research is the fact that users in need of AT typically have physical disabilities associated with motor control and such conditions (e.g. cerebral palsy) will also often affect the musculature surrounding the articulatory system resulting in slurred and less clear speech; known as *dysarthric* speech.

3 Meeting room studio setup

Our meeting room studio, shown in fig. 1, is used to collect natural speech and training data from real meetings. Is centred on a six-person table, with additional chairs around the walls for around a further 10 people. It has a whiteboard at the head of the table, and a presentation projector. Typical meetings involve participants speaking from their chairs but also getting up and walking around to present or to use the whiteboard. A 2×2.5m aluminium frame is suspended from the ceiling above the table and used for mounting au-

Figure 1: Meeting room recording setup. The boxes on the far wall are active-badge trackers. The frame on the ceiling and the black cylinder on the table each contain eight condenser microphones. Participants wear headsets and active badges.

dio equipment. Currently this consists of eight AKG C417/III vocal condenser microphones, arranged in an ellipse around the table perimeter. A further eight AKG C417/IIIs are embedded in a 100mm radius cylinder placed in the table centre to act similarly to eight-channel multidirectional tele-conferencing recorder. The table can also include three 7-channel DevAudio Microcones (www.dev-audio.com), which are commercial products performing a similar function. The Microcone is a 6-channel microphone array which comes with propriety drivers and an API. A further 7th audio channel contains a mix of the other 6 channels as well as voice activity detection and sound source localisation information annotation. Some noise reduction and speech enhancement capabilities are provided, although details of the exact processing are not made public.

There are four Sennheiser ew100 wireless headsets which may be mounted on selected participants to record their speech directly. The studio will soon also include a Radvision Scopia XT1000 videoconferencing system, comprised of a further source-tracking steered microphone and two 1.5m HD presentation screens. In the four upper corners of the meeting room are mounted Ubisense infrared active badge receivers, which may be used to track the 3D locations of 15 mobile badges worn by meeting participants. (The university also has a 24-channel surround sound diffusion system used in an MA Electroacoustic music course [Mooney, 2005], which may be useful for generating spatial audio test sets.)

Sixteen of the mics are currently routed through two MOTU 8Pre interfaces, which take eight XLR or line inputs each. Both currently run at 48kHz but can operate up to 96kHz. The first of these runs in A/D conversion mode and sends all 8 digitised channels via a single ADAT Lightpipe fibre optic cable to the second 8Pre. The second 8Pre receives this input, along with its own eight audio channels and outputs all 16 channels to the DAW by Firewire 400 (IEEE 1394, 400 bits/sec). (The two boxes must be configured to have (a) the same speed, (b) 1x ADAT protocol and (c) be in be in converter/interface mode respectively.) Further firewire devices will be added to the bus later to accommodate the rest of the microphones in the room.

4 Linux audio system review

Fig. 2 outlines the Linux audio system and highlights the parts of the many possible Linux audio stacks that are used for recording in the meeting room studio. The Linux audio architecture has grown quite complex in recent years, so is reviewed here in detail.

OSS (Open Sound System) was developed in the early 1990s, focused initially on Creative SoundBlaster cards then extending to others. It was a locking system allowing only one program at a time to access the sound card, and lacked support for features such as surround sound. It allowed low level access to the card, for example by `cataudio.wav > /dev/dsp0`. ALSA (Advanced Linux Sound Architecture) was designed to replace OSS, and is used on most current distributions including our Ubuntu Studio 11.10. PortAudio is an API with backends that abstract both OSS and ALSA, as well as sound systems of non-free platforms such as Win32 sound and Mac CoreAudio, created to allow portable audio programs to be written. Several software mixer systems were built to resolve the locking problem for

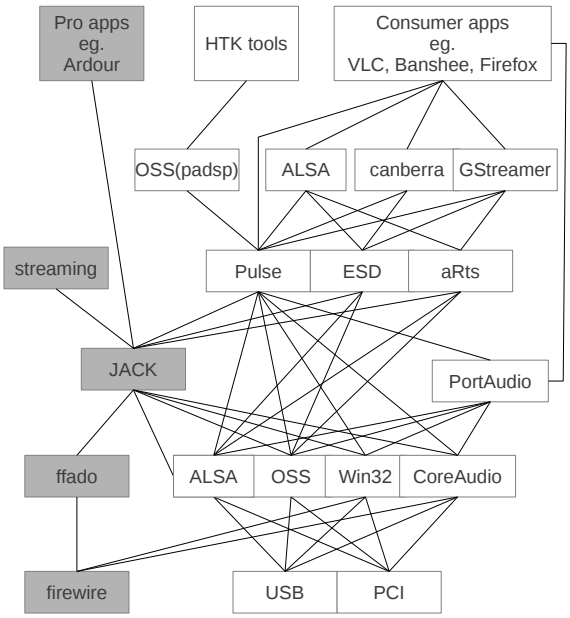


Figure 2: Audio system for recording.

consumer-audio applications, including PulseAudio, ESD and aRts. Some of these mixers grew to take advantage of and to control hardware mixing provided by sounds cards, and provided additional features such as network streaming. They provided their own APIs as well as emulation layers for older (or mixer-agnostic) OSS and ALSA applications. (To complicate matters further, recent versions of OSS4 and ALSA have now begun to provide their own software mixers, as well as emulation layers for each other.) Many current Linux distributions including Ubuntu 11.10 deploy PulseAudio running on ALSA, and also include an ALSA emulation layer on Pulse to allow multiple ALSA and Pulse applications to run together through the mixer. Media libraries such as GStreamer (which powers consumer-audio applications such as VLC, Skype and Flash) and libcanberra (the GNOME desktop sound system) have been developed closely with PulseAudio, increasing its popularity. However, Pulse is not designed for pro-audio work as such work requires very low latencies and minimal drop-outs.

The JACK system is an alternative software mixer for pro-audio work. Like the other soft mixers, JACK runs on many lower level platforms – usually ALSA on modern Linux machines. The bulk of pro-audio applications such as Ardour, zynAddSubFx and qSynth run on JACK. JACK also provides network streaming, and emulations/interfaces for other audio APIs including ALSA, OSS and PulseAudio. (Pulse-on-JACK is useful when using pro and consumer applications at the same time, such as when watching a YouTube tutorial about how to use a pro application. This re-configuration happens automatically when JACK is launched on a modern Pulse machine such as Ubuntu 11.10.)

5 Software setup

Our DAW is a relatively low-power Intel E8400 (Wolfdale) duo-core, 3GHz, 4Gb Ubuntu Studio 11.10-64-bit machine. Ubuntu studio was installed directly from CD – not added as packages to an existing Ubuntu installation – this gives a more minimalist installation than the latter approach. In particular the window manager defaults to the low-power XFCE, and resource-intensive programs such as Gnome-Network-Monitor (which periodically searches for new wifi networks in the background) are not installed. Ubuntu was chosen for compatibility and familiarity with our other desktop machines. (Several other audio distributions are available including PlanetCCRMA(Fedora), 64Studio(Debian), ArchProAudio(Arch)).

The standard ALSA and OSS provide interfaces to USB and PCI devices below, and to JACK above. However for firewire devices such as our Pre8, the `ffado` driver provides a *direct* interface to JACK from the hardware, bypassing ALSA or OSS. (Though the latest/development version provides an ALSA output layer as well.) Our DAW uses `ffado` with JACK2 (Ubuntu packages: `jack2d, jack2d-firewire, libffado, jackd, laditools`. JACK1 is the older but perhaps more stable single-processor implementation of the JACK API) and fig. 3 shows our JACK settings, in the `qjackctl` tool. The firewire backend driver (`ffado`) is selected rather than ALSA.

We found it useful to unlock memory for good JACK performance.[2] As well as ticking the un-

[2]By default, JACK locks all memory of its clients into RAM, ie. tells the kernel not to swap their pages to virtual memory on disc, see mlock(2). Unlock memory relaxes this slightly, allowing just the large GUI components of clients

lock memory option, the user must also be allowed to use it, eg. `adduser charles audio`. Also the file `/etc/security/limits.d/audio.conf` was edited (followed by a reboot) to include

```
@audio - rtprio 95
@audio - memlock unlimited
```

These settings can be checked by `ulimit -r -l`.

The JACK sample rate was set to 48kHz, matching the Pre8s. (This is a good sample rate for speech research work as it is similar to CD quality but allows simple sub-sampling to power-of-two frequencies used in analysis.)

Fig. 4 shows the JACK connections (again in `qjackctl`) for our meeting room studio setup. The eight channels from the converter-mode Pre8 appear as ADAT optical inputs, and the eight channels from the interface-mode Pre8 appear as 'Analog' inputs, all within the firewire device. Ardour was used with two tracks of eight channel audio to record as shown in fig. 5.

5.1 Results

Using this setup we were able to record simultaneously from six overhead microphones, eight table-centre microphones, and two wireless headsets, as illustrated in fig. 5. We experienced no JACK xruns in a ten minute, 48kHz, 32-bit, 16-channel recording, and the reported JACK latency was 8ms. A one hour meeting recording with the same settings experienced only 11 xruns. Total CPU usage was below 25% at all times, with `top` listing the following typical total process CPU usages: `jack` 11%, `ardour` 8%, `jack.real` 3%, `pulseaudio` 3%.

However, we were unable to play audio back through the Pre8s, hearing distorted versions of the recording. For our speech recognition this is relatively unimportant, and can be worked around by streaming the output over the network with JACK and playing back on a second machine. The `ffado` driver's support for the Pre8 hardware is currently listed as 'experimental' so work is needed here to fix this problem.

The present two Pre8 system is limited to 16 audio channels, we plan to extend it with further firewire devices to record from more audio sources around the meeting room and from tele-

to perform swapping, leaving more RAM free for the audio parts of clients.

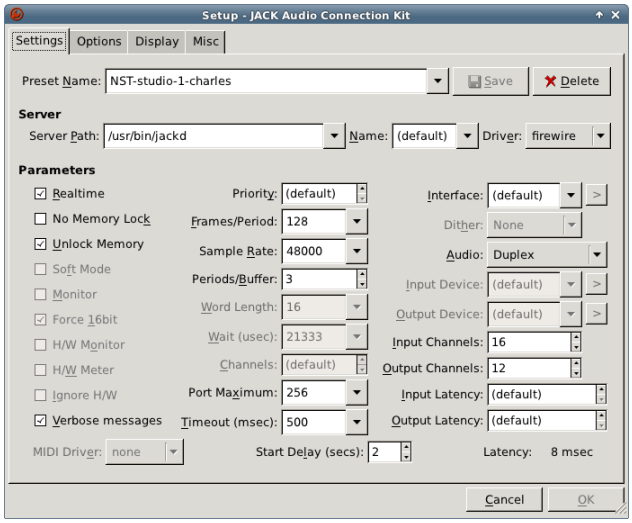


Figure 3: 16 channel recording JACK settings.

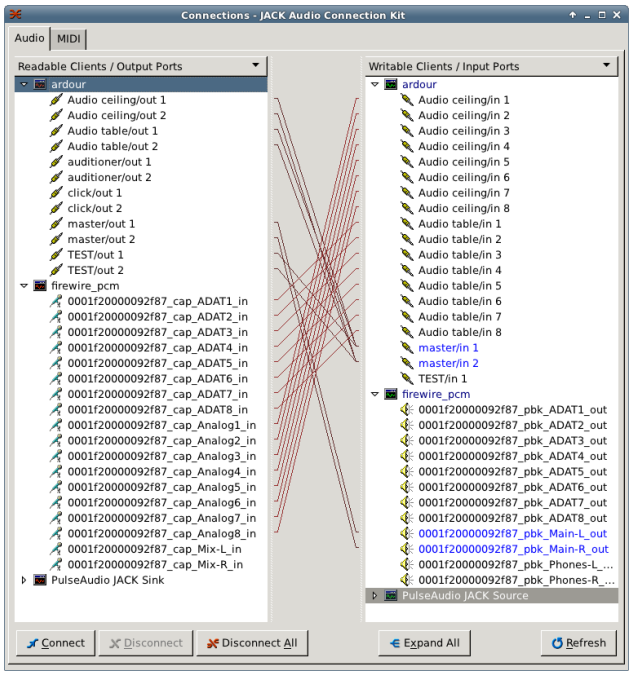


Figure 4: 16 channel recording JACK connections.

conferencing channels in future. We have not yet needed to make further speed optimisations, but we note that for future, more-channel systems, two speedups include disabling PulseAudio (adding `pulseaudio -kill` and `pulseaudio -start` to qjackctl's startup and shutdown option is a simple way to do this); and installing the real-time `rt-linux` kernel.

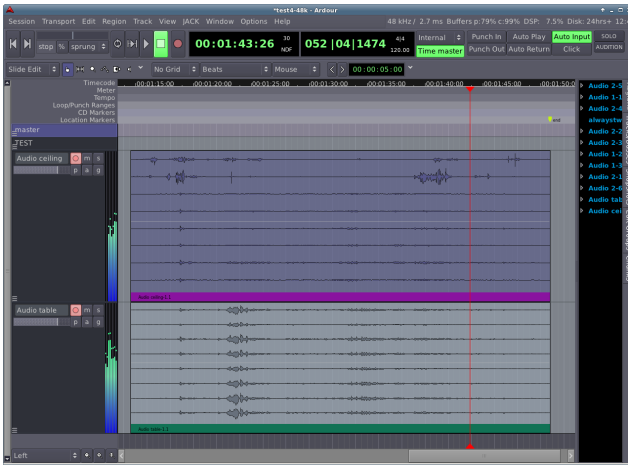


Figure 5: 16 channel meeting room recording in Ardour, using two 8-channel tracks.

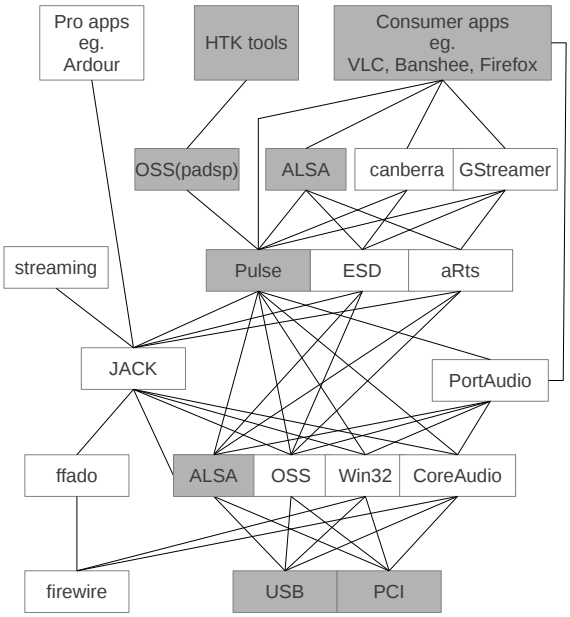


Figure 6: Audio system for data analysis.

5.1.1 Audio analysis

Analysis of our audio data is performed on a compute cluster of 20 Linux nodes with 96 processor cores in total, running the Oracle (formerly Sun) Grid Engine, the HTK Hidden Markov Model Tool Kit [Young et al., 2006] and the Juicer recognition engine [Moore et al., 2006]. During analysis, audio playback on desktop machines is useful and is done with the setup of fig. 6. For direct audio connections the HTK tools make use the OSS sound system, which may be emulated on PulseAudio on a modern machine, by installing the `padsp` tool (Ubuntu package `pulseaudio-utils_0.9.10-1ubuntu1_i386`) then prefixing all audio HTK commands with `padsp`. Similarly, Juicer allows the use of an OOS front, the implementation of a JACK plugin is in progress.

The cluster may be used both for online and offline processing. An example of online processing can be found in the description of an online transcription system for meetings [Garner et al., 2009]. Such systems distinguish between far and near-field audio sources and employ beamforming and echo cancellation procedures [Hain et al., 2009] for which either sample synchronicity or at least minimal latency between channels is of utmost importance. For both offline and online processing typically audio at 16kHz/16bit is required, to be converted into so-called feature streams (for example Perceptual Linear Predictive (PLP) features [Hermansky, 1990]) of much lower bit rate. Even higher sampling frequencies (up to 96kHz are commonplace) are often required for cross-correlation based sound source localisation algorithms to provide sufficient time resolution in order to detect changes in angles down to one degree. In offline mode the recognition of audio typically operates at 10 times real-time (i.e. 1 hour of audio in many channels takes 10 hours to process). However, the grid framework allows the latency of processing to drop to close to 1.5 times real-time using massive parallelisation.

6 Future directions

The ultimate goal of NST is to obtain transcriptions of what was said in natural environments. Traditionally, the source separation and denoising techniques of sec. 2 have been treated as a separate preprocessing step, before the cleaned audio is passed to a separate transcriber such as HTK. However for challenging environments this is suboptimal, as it reports only a single estimate of the denoised signal rather than Bayesian information about its uncertainty. Future integrated systems could fuse the predictions from transcription language models with inference and reporting of low-level audio data, for example by passing real-time probabilistic messages between HTK's transcrip-

tion inference (on a Linux computing cluster) and low-level audio processing (on desktop or embedded Linux close to the recording hardware.)

For training and testing speaker location tracking systems it is useful to build a database of known speaker position sequences, which need to be synchronised to the audio. Positions change at the order of seconds so it is wasteful to use audio channels to record them – however we note that JACK is able to record MIDI information alongside audio, and one possibility would be to encode position from our Ubisense active badges as MIDI messages, synchronous with the audio, and record them together for example in Ardour3. It could also be useful to – somehow – synchronise video of meetings to JACK audio.

The homeService system has two major parts, namely hardware components that are deployed in people's homes during trial (Android tablet, Microcone, and Linux box responsible for audio capture, network access, infrared and blue tooth communication), as well as our Linux computing cluster back at the university running the processes with particularly high demands in terms of processing power and memory usage. The main audio capturing will take place on a Linux box in the users' home, and we plan to develop a setup which will enable the Microcone and JACK to work together and provide – if needed – live streaming of audio over the network.

The main requirements of NST research for Linux audio are support for sample-synchronous, many (e.g. 128 or more) channel recording, and communication with cluster computing and speech tools such as HTK. As natural speech technology makes closer contact with signal processing research, we expect to see more speech researchers moving to Linux audio in the near future, and we hope that this paper has provided some guidance for those who wish to make this move, as well as a guide for the Linux audio community about what technologies are important to this field.

Acknowledgements

The research leading to these results was supported by EPSRC Programme Grant EP/I031022/1 (Natural Speech Technology).

References

L. Brayda, C. Bertotti, L Cristoforetti, M. , Omologo, and P. Svaizer. 2005. Modifications on NIST MarkIII array to improve coherence properties among input signals. In *Proc. of 118th Audio Engineering Society Conv.*

H. Christensen and J. Barker. 2010. Speaker turn tracking with mobile microphones: combining location and pitch information. In *Proc. of EUSIPCO*.

C. Fox, M. Evans, M. Pearson, and T. Prescott. 2012. Tactile SLAM with a biomimetic whiskered robot. In *Proc. ICRA*.

P. N. Garner, J. Dines, T. Hain, A. el Hannani, M. Karafiat, D. Korchagin, Mike L., V. Wan, and L. Zhang. 2009. Real-Time ASR from Meetings. In *Interspeech'09*, pages 2119–2122.

T. Hain, L. Burget, J. Dines, P. N. Garner, A. el Hannani, M. Huijbregts, M. Karafiat, M. Lincoln, and V. Wan. 2009. The AMIDA 2009 meeting transcription system. In *Proc. Interspeech*.

H. Hermansky. 1990. Perceptual linear predictive analysis for speech. *J. Acoustic Society of America*, pages 1738–1752.

J. Mooney. 2005. *Sound Diffusion Systems for the Live Performance of Electroacoustic Music*. Ph.D. thesis, University of Sheffield.

D. Moore, J. Dines, M. Magimai Doss, J. Vepa, O. Cheng, and T. Hain. 2006. Juicer: A weighted finite state transducer speech decoder. In *Machine Learning for Multimodal Interaction*, pages 285–296. Springer-Verlag.

S. Roberts and R. Everson, editors. 2001. *Independent Components Analysis: Principles and Practice*. Cambridge.

H. L. Van Trees. 2002. *Optimum array processing*. Wiley.

L. Watts. 2009. Reverberation removal. In *United States Patent Number 7,508,948*.

M. Wolfel and J. McDonough. 2009. *Distant Speech Recognition*. Wiley.

S. Young, G. Evermann, M. Gales, T. Hain, D. Kershaw, XA Liu, G. Moore, J. Odell, D. Ollason, D. Povey, et al. 2006. The HTK book.

A framework for dynamic spatial acoustic scene generation with Ambisonics in low delay realtime

Giso GRIMM
Medical Physics Group
Universität Oldenburg
26111 Oldenburg,
Germany
g.grimm@uni-oldenburg.de

Tobias HERZKE
HörTech gGmbH
Marie-Curie-Str. 2
26129 Oldenburg,
Germany,
t.herzke@hoertech.de

Abstract

A software toolbox developed for a concert in which acoustic instruments are amplified and spatially processed in low-latency is described in this article. The spatial image is created in Ambisonics format by a set of dynamic acoustic scene generators which can create periodic spatial trajectories. Parameterization of the trajectories can be selected by a body tracking interface optimized for seated musicians. Application of this toolbox in the field of hearing research is discussed.

Keywords

Ambisonics, digital processing, concert performance

1 Introduction

Spatially distributed presentation of music can contribute to an improved perception. In the 16th century, many composers used several choirs and groups of musicians at distributed places in church music. However, most conventional concert locations do not provide the capabilities to have the musicians placed around the audience. Spatial presentation with loudspeakers offers new possibilities.

The ensemble ORLANDO*viols*[1] developed and performed a concert program, in which the audience is surrounded by the five musicians. Additionally, the sound is processed by a low-latency Ambisonics system and presented in dynamic spatial configurations. The instruments virtually move around the audience. Trajectories matched to the music potentially provide easier access to the concepts in music and may add further levels of interpretation. A central piece in the program is an "In Nomine" fantasy from the 16th century by Mr. Picforth, in which each voice is related to one of the planets known in that time [Grimm, 2012]. For example, during this piece the musicians virtually follow the planet's trajectories.

Such a concert with simultaneous presentation of acoustic sources and their spatially processed images requires careful considerations of psychoacoustic effects, specifically the precedence effect. Appropriate amplification and placement of loudspeakers and musicians are a prerequisite for a spatial image which is dominated by the processed sound and not by the direct sound of the musicians [Grimm et al., 2011]. A set of software components is required to make such a concert possible. The Open Sound Control (OSC) protocol [Wright, 2002] facilitates communication of these components across the borders of processes and hardware machines. The several software components and their applications are described in the following sections.

2 Application in a concert setup

The tools described in this paper have been developed for a concert performance entitled "Harmony of the Spheres". The ensemble consists of five musicians playing on viola da gamba (or viols), a historic bowed string instrument. Viola da gambas come as a family - the smallest has the size of a violin, the largest that of the double bass. The instrument – except for the largest one – is held with the legs, which led to its name ('gamba' means leg in Italian).

In the concert, five musicians (including one of the authors) sit on the corners of a large pentagon. Within the pentagon a 10 channel 3rd order horizontal Ambisonics system is placed on a regular decagon with a radius of approximately 6 meters. Within this decagon sits the audience, with space for roughly 100 listeners. The instruments are picked up with cardioid microphones

[1] `http://www.orlandoviols.de/`

at a distance of approximately 40 cm, which is a compromise between close distance for minimal feedback problems and a large distance for a balanced sound of the historic acoustic instruments. The microphone signals are converted to the digital domain (Behringer ADA8000) and processed on a Linux PC (Athlon X2 250). The processed signals are then played back by the powered near field monitors (KRK RP5). Due to the large distance between the musicians – up to 15 meters – monitoring is required. The monitor signals are created on the hardware mixer of the sound card (RME HDSP9652) and played back to the musicians through in-ear monitor headphones. Two hardware controllers (Behringer BCF2000) are used to control the loudspeaker mix and the monitor. A separate PC is used for visualization of levels, monitor mixer settings and the spatial parameters. A foot switch (Behringer FBC1010) and a virtual foot switch (Microsoft Kinect connected to a netbook PC) are used for parameter control by one of the musicians. An additional netbook PC is used for projection of the trajectories of the virtual sources at the ceiling of the room.

Figure 1: Concert setup in a church, during rehearsal. The diameter of the loudspeaker ring was 13 m, fitting about 100 chairs.

3 Software components

The software components used in the toolbox for dynamic scene generation can be divided into a set of third party software, and specifically developed applications. The third party software includes the jack audio connection kit (jack) [Letz and Davis, 2011]. Jack2 is used to allow for parallel processing of independent signal graphs. The digital audio workstation 'ardour' [Davis, 2011] is used for signal routing, mixing, recording and playback. Convolution reverb is used for distance coding; the convolution is done by 'jconvolver' [Adriaensen, 2011]. The Ambisonics signals are decoded into the speaker layout with 'ambdec' [Adriaensen, 2011]. The tool 'mididings' [Sacré, 2010] triggers the execution of shell scripts upon incoming MIDI events.

3.1 Spatial processing: 'sphere' panning application

For the spatial processing a stand-alone jack panning application with an integrated scene generator is used. The scene generator is inspired by the planet movements. It can create Kepler ellipses with an additional epicyclic component and a random phase increment. The parameters are radius ρ, rotation frequency f, eccentricity ε, rotation of ellipse main axis θ, starting angle φ_0, and the epicycle parameters radius ρ_{epi}, rotation frequency f_{epi} and starting angle $\varphi_{0,epi}$. The source position on a complex plane is

$$z = \frac{\rho\sqrt{1-\varepsilon^2}}{1-\varepsilon\cos(\varphi-\theta)}e^{i\varphi} + \rho_{epi}e^{i\varphi_{epi}}. \qquad (1)$$

The azimuth $\angle z$ is the input argument of a 3rd order horizontal ambisonics panner. Distance perception in reverberant environments is dominated by the amount of reverberation [Sheeline, 1982; Bosi, 1990]. The distance $|z|$ is here coded by adding a reverberant copy of the signal. The reverberant signal is created by a convolution stereo-reverb. A virtual stereo loudspeaker pair centered around the target direction given by the virtual source azimuth is used for playback. The width of the virtual stereo pair is controlled by the distance. It is chosen to be maximal at the critical distance ρ_{crit}, and converging to zero for close and distant sources. Also the ratio between the dry signal and the reverberant stereo signal is controlled by the distance. The distance to the origin normalized by the critical distance is $\hat{\rho} = \rho/\rho_{crit}$. Then the parameters of the distance coding are:

$$w(\rho) = w_{max}\frac{\hat{\rho}}{\hat{\rho}^2+1} \qquad (2)$$

$$G_{dry} = \frac{1}{1+\hat{\rho}} \qquad (3)$$

$$G_{wet} = 1 - G_{dry} \quad (4)$$

The maximal width w_{max} and the critical distance ρ_{crit} can be controlled externally. Examples of trajectories used in the concert are shown in Figure 2.

The scene generator and panning application, named 'sphere', is a jack application with an input port for the dry signal and a stereo input port pair for the reverberated signal. The parameters can be controlled via OSC messages. The OSC receiver can be configured to listen at a multicast address, which allows to control multiple instances in a single OSC message. To achieve a dynamic behavior independent from the actual processing block size, the update rate of panning parameters is independent from the block rate. Changes of different parameters can be accumulated and applied simultaneously, either immediately or slowly within a given time.

Other features of the scene generator are sending of the current azimuth to other scene generator instances via OSC, application of parameter changes at predefined azimuths, randomization of the azimuth, and level-dependent azimuth.

The ambisonics panner uses horizontal panning only. For artistic reasons, a simulation of elevation is reached by gradually mixing the signal to the zeroth order component of the ambisonics signal. By doing so the resulting ambisonics signal does not correspond to any physical acoustic sound field anymore, however, the intention is to have the possibility of creating a virtual sound source without any distinct direction.

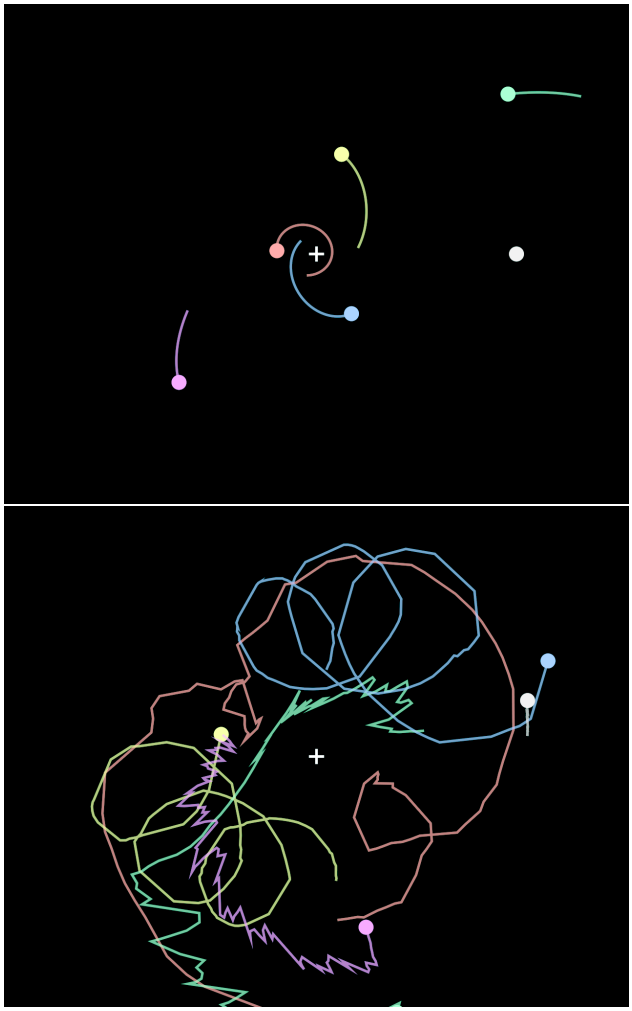


Figure 2: Example trajectories in two of the pieces: Kepler ellipses in the "In Nomine" by Mr. Picforth (top), and chaotic movements in the piece "Five" by John Cage (bottom).

3.2 Matrix mixer

Creating an individual headphone monitor mix for several musicians and many sources is a demanding task, and usually needed in situations when a quick and efficient solution is required. In this setup, the RME HDSP9652 sound card was used, which offers a hardware based matrix mixer. The existing interfaces on Linux - amixer (console) and hdspmixer (graphical) - provide full access to that mixer. However, the lack of hardware based remote control option makes these tools inefficient in time-critical situations. Therefore, a set of applications has been developed which attempts to provide a flexible solution: One application provides an interface to the audio hardware. A second application reads and stores the mixer settings from and to XML files. Mixer settings include panning – sin-cos-panning for two-channel output, vector-based amplitude panning (VBAP) for multi-channel output – and gains for a pre-defined set of inputs and outputs. Channels which are not used in a configuration will not be accessed. No logical difference is made between software and hardware inputs. A third application allows control via MIDI CC events (here coming from a Behringer BCF2000). Feedback is sent to the controller device. A fourth application is visualizing the gains. Editing of the gains is planned, but not implemented yet at time of writing. A screen-shot of the visualization tool is

shown in Fig. 3. The four applications share their settings via OSC messages. A feedback filter prevents from OSC message feedback.

Figure 3: Screen shot of a matrix mixer example session.

3.3 Level metering

For the polyphonic character of many of the pieces played in the above mentioned concert, well balanced levels of the five instruments are of major importance. Level differences between the instruments of only a few dB can easily destroy the musical structure in some pieces. Space limitations make it impossible to place the mixing desk and its operator within the Ambisonics system. Therefore, a level metering application was developed. This level meter can measure the root-mean-square (RMS) level and short time peak values of multiple inputs. To account for the loudness difference between low instruments (double bass size) and medium and high instruments, a band pass filter with a pass band from 300 Hz to 3 kHz is applied before metering. The RMS level meter operate on 10 s and 125 ms rectangular windows. The 125 ms window is overlapping by 50%. The peak levels are measured in the same windows as the short time levels. The levels are sent via OSC messages to a visualization application. The level meter application was implemented in the HörTech Master Hearing Aid (MHA) [Grimm et al., 2006].

To allow for optimal mixing even without hearing the balanced final mix at the sweet spot, an indicator is used which shows the long term target level. This level can be changed by the presets for each piece and also within pieces. Long term level deviations can then be corrected by the operator. Automatic adaption to the desired levels bares the risk of causing feedback problems.

3.4 Parameter exchange and preset selection

All applications exchange parameters and control data via OSC messages. To allow a flexible inter process communication across hardware borders, most applications subscribe themselves to UDP multicast addresses.

Settings of the scene generators 'sphere' can be loaded from setting files. Reading of preset files and sending them as OSC messages can be triggered either from the console, from the virtual footswitch controller, from the physical footswitch or by ardour markers. For the last option, an application has been developed which converts ardour markers into OSC messages: The application reads an ardour session file, subscribes to jack, and emits OSC messages whenever an ardour marker position falls into the time interval of the current jack audio block.

The several preset files can also control the transport of ardour, to control recording of inputs and trigger playback of additional sounds.

3.5 Visualization

Simple visualization classes have been implemented for display of the trajectories of the 'sphere' scene generators, for level monitoring and for visual control of the monitor mixer. With these classes applications can be built to contain any combination of visualization tools. No graphical user interaction is possible.

3.6 Implementation of virtual foot switch

In the concert application, one of the musicians is responsible for switching scene generation presets at musically appropriate times while playing. Since viola da gambas are played while sitting on a chair, and the instruments are held between the legs, it is sometimes difficult to use a real footswitch, especially elevating the heel, without interrupting the music. To circumvent these problems, a virtual footswitch is used which can be controlled by a minor movement of the tip of the foot, without the need of elevating the heel. Care

was taken that this footswitch is sufficiently robust to avoid false alarms and missed movements.

The virtual foot switch is implemented using the Prime Sense depth camera of the Microsoft Kinect game accessory [Kin, 2010]. The depth camera captures the area of the right foot of the musician in charge of the switching. The switch differentiates between 4 states: No foot present ("free"), and 3 different directions of the foot ("straight", "left" and "right"). The switch is inactive in the states "free" and "straight", and triggers actions in the states "left" and "right".

The depth camera uses an infrared laser diode to illuminate the room with an infrared dot pattern, which is then picked up by the camera sensor. From the distortion of the dot pattern, the camera is able to associate image pixels with depth information. The computation of depth information is performed inside the camera device. Because the infrared laser diode and the camera sensor are laterally separated, not the complete scene visible to the camera sensor is illuminated with the infrared dot pattern. For these shadow areas, the camera does not provide depth information. Also, for small surfaces like fingertips and reflecting surfaces like glass, the camera will often not return a valid depth information.

The camera produces depth images of 640x480 pixels at a frame rate of 30 Hz[2]. The depth information in each captured depth image consists of a matrix of 11 bit integer depth values d_{raw}. The highest bit indicates an invalid depth value, e.g. a shadow. The range of detectable depth is approximately 0.5m to 5m, with better resolution ($<$ 1cm) at the start of this range. The depth in meters is

$$d = 0.1236\text{m} \tan\left(\frac{d_{raw}}{2842.5} + 1.1863\right) \qquad (5)$$

[Magnenat, 2011]. For depth pixel values where this formula yields a depth $d < 0$m or $d > 5$m, we conclude an invalid depth value.

To capture the foot of the musician, the depth camera is mounted on a small pedestal next to the music stand. The camera is tilted downwards by 16 degrees to capture the floor in the area of the foot as shown in Figure 4.

[2]We use the libfreenect library from the OpenKinect [Blake, 2011] project to receive a stream of depth images.

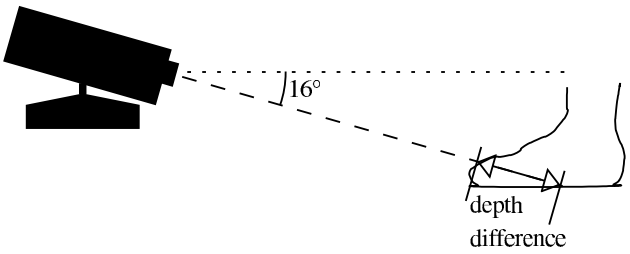


Figure 4: Kinect depth camera detects depth difference caused by foot

Our software for the virtual foot switch starts with a training phase before entering the detection phase. It first needs to capture the scene without the foot, then directs the musician to place the foot pointing naturally straight, or rotated, around the heel, to the left and right, respectively. 10 images are captured for each of the 4 situations. Mean and standard deviation for each pixel's depth is computed. Three differential depth images are computed by subtracting the mean depth information of the images containing a foot from the empty scene. Differential depth for pixels with invalid depth information or too high standard deviation in either image is set to 0, effectively excluding these pixels from further consideration. The presence of a foot where previously was only floor in the image will reduce the depth information captured by the camera as shown in Figure 4. As we are mainly interested in the position of the front part of the foot, we restrict the differential depth image to depth differences in the range of 3cm to 9cm. All depth differences outside this range are also set to zero and will not be considered for locating the foot. Figure 5 shows an example of such a differential depth image for a straight pointing foot. To identify the foot, the software searches for a cluster of depth difference pixels from bottom to top. The first such cluster of at least 100 pixels that is found is considered to be caused by the foot. The minimum rectangle that contains all three (straight, left, right) foot clusters defines the region of interest (ROI) to be used during the detection phase. Figure 6 shows the ROI from an example session, containing three foot clusters.

Also from depth images captured during detection, differential images are computed by subtracting their depth information from the empty scene training image, and considering only depth

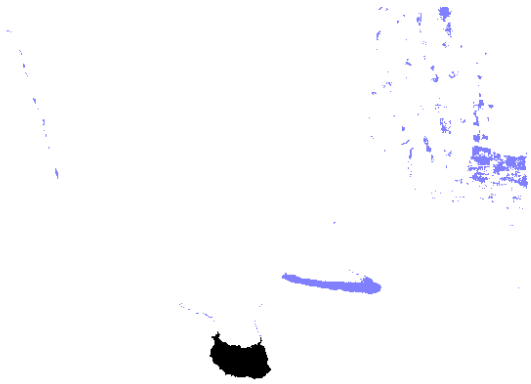


Figure 5: Differential depth image for the straight foot state from training, identified foot cluster painted black

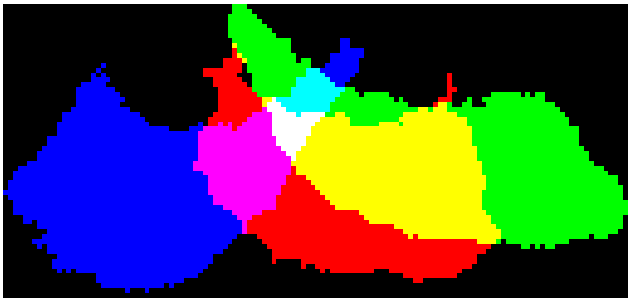


Figure 6: The region of interest (ROI) for the virtual foot switch containing the three foot clusters (displayed using additive color)

differences in the range of 3cm to 9cm. During detection, this processing is restricted to the ROI determined from the training data.

To be able to classify depth images captured during detection, an error function to compute the distance of the current state to each of the three training states containing a foot is used. Within the ROI, for each scan line, the horizontal coordinate $x_{y,i}$ with the maximum depth difference (within the range of 3cm to 9cm) is located, where y is one of the scan lines in the ROI, and $i \in \{\text{current}, \text{straight}, \text{left}, \text{right}\}$. The error function that we use is

$$e_i = \sum_y (x_{y,\text{current}} - x_{y,i})^2 \qquad (6)$$

For some combinations of y and i, $x_{y,i}$ is not defined. These combinations do not contribute to the error summation.

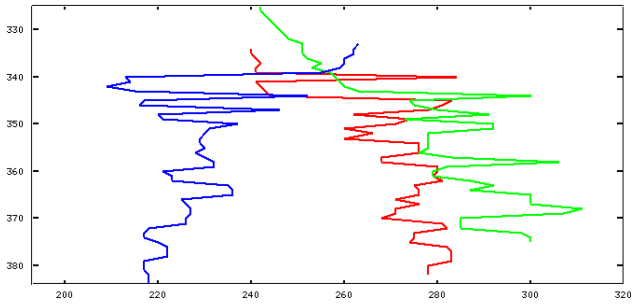


Figure 7: For each foot cluster from Figure 6 and for each scan line, the horizontal coordinate with the maximum depth difference between the training image with the foot and the empty scene is shown

Figure 7 shows these horizontal coordinates for the example foot clusters of Figure 6.

The training state with the lowest squared error sum e_i is then considered recognized. If, however, the lowest error is zero, then the empty scene is recognized.

If the left or right foot state is recognized for seven consecutive frames (i.e. for a quarter of a second), then the switch performs the configured action. The straight foot state and the empty scene do not trigger actions.

The virtual foot switch is initialized with a list of actions to perform. When the foot is recognized in the right or left position reliably, then the software triggers the next or previous action from this list, respectively. Actions are system commands and executed in a shell. For the concert, the actions influence the trajectory generator by sending appropriate OSC messages.

4 Application in hearing research

The scene generation toolbox described in the previous sections is planned to be used also in hearing research and hearing aid development, after some modifications and extensions. Here, the task is to generate dynamic acoustic scenes which are fully reproducible and controllable. Dynamic acoustic scenes can be presented to real listeners, who - within a certain area - can move freely. Hearing aid algorithms can directly be evaluated[3].

[3]For the evaluation of hearing aid algorithms in Ambisonics systems, the effect of the limitations of Ambisonics

In hearing aid research, discrete loudspeaker setups are commonly used for spatial evaluation of algorithms. However, discrete setups are inable to acoustic scenes with moving sources. Vector based amplitude panning (VBAP) as a simple extension of discrete speaker setups have the disadvantage of creating phantom sources when the virtual source is between two speakers and real sources when the virtual source direction is matched by a speaker. Phantom sources are a general problem for directional filter algorithms. Higher order Ambisonics offers the advantage that moving sources can be created with a steady image. A detailed evaluation of applicability for hearing aid algorithm testing is content of an ongoing study.

5 Discussion

Low-latency real-time spatial processing of acoustic instruments is a field with many gaps. Although much effort has been done to find solutions to most problems, many open questions still remain. One major flaw is the coding of distances: While distance coding by changing the amount of reverberation may work in low-reverberant rooms like the monitor room used during development, the effect will break down in more reverberant rooms, like most performance spaces. A potential solution might be to simulate the Doppler effect in order to at least create an image of distance changes, at the cost of additional delay. Also spectral changes which are observed in large distance differences may help in the distance perception.

Pseudo-elevation was used in the concert to create the image of sources without any distinct direction. Mixing a source to the zeroth order Ambisonics channel results in an identical signal from all loudspeakers. The effect of no distinct direction, however, can only be perceived right in the middle of the ring. At any other listening postion, the precedence effect will cause the nearest loudspeaker to be perceived as the dominant direction. The main difference to correct panning is that the perceived direction will differ for all listeners. A solution might be to use a full 3-dimensional Ambisonics system with at least first order in vertical direction, or to find alternative panning strategies.

systems on the respective algorithms (e.g., spatial aliasing, unmatched wave fronts, high frequency localization) has to be carefully considered.

6 Conclusions

A set of software tools for application in a concert performance with spatial processing of acoustic instruments has been described in this paper. The technical components of the performance were completely build on top of Linux Audio. All major parts are implemented as open source applications.

The authors believe that spatial processing can substantially contribute to a concert experience with Early and contemporary music. This believe was supported by statements of many concert listeners – "the experience of musicians moving behind your back sends you a shiver down your spine", "I felt like being in a huge bell, right in the heart of the music". However, this software toolbox leaves still space for further development, for improvement of the concert experience and for application in hearing research.

7 Acknowledgements

Our thanks go to the members of the ensemble ORLANDO*viols*, who all shared the enthusiasm in creating this concert project. Special thanks also go to Fons Adriaensen for his great linux audio tools and for helpful advice on Ambisonics. This study was partly funded by the German research funding organization Deutsche Forschungsgemeinsschaft (DFG) in the Research Unit 1732 "Individualisierte Hörakustik" and by "klangpol – Neue Musik im Nordwesten" (Contemporary Music in north-west Germany).

References

Fons Adriaensen. 2011. Linux audio projects at kokkini zita. `http://kokkinizita.linuxaudio.org/`.

Joshua Blake. 2011. Openkinect. `http://openkinect.org`.

Marina Bosi. 1990. An interactive real-time system for the control of sound localization. *Computer Music Journal*, 14(4):59–64.

Paul Davis. 2011. Ardour. `http://ardour.org/`.

Giso Grimm, Tobias Herzke, Daniel Berg, and Volker Hohmann. 2006. The Master Hearing

Aid – a PC-based platform for algorithm development and evaluation. *Acustica · acta acustica*, 92:618–628.

Giso Grimm, Volker Hohmann, and Stephan Ewert. 2011. Object fusion and localization dominance in real-time spatial processing of acoustic sources using higher order ambisonics. In György Wersény and David Worrall, editors, *Proceedings of the 17th Annual Conference on Auditory Display (ICAD)*, Budapest, Hungary. OPAKFI Egyesület. ISBN 978-963-8241-72-6.

Giso Grimm. 2012. Harmony of the spheres - cosmology and number aesthetics in 16th and 20th century music. *Musica Antiqua*, 1(1):18–22, Jan-Mar.

2010. Kinect. `http://microsoft.com/Presspass/press/2010/mar10/03-31PrimeSensePR.mspx`.

Stephane Letz and Paul Davis. 2011. Jack audio connection kit. `http://jackaudio.org/`.

Stéphane Magnenat. 2011. `http://openkinect.org/wiki/Imaging_Information`, version from June 7 2011. the wiki page attributes this formula to Stéphane Magnenat.

Dominic Sacré. 2010. Mididings. `http://das.nasophon.de/mididings/`.

Christopher W. Sheeline. 1982. *An investigation of the effects of direct and reverberant signal interactions on auditory distance perception*. Ph.D. thesis, CCRMA Department of Music, Stanford University, Stanford, California.

Matthew Wright. 2002. Open sound control 1.0 specification. `http://opensoundcontrol.org/spec-1_0`.

A Toolkit for the Design of Ambisonic Decoders

Aaron J. HELLER
AI Center, SRI International
Menlo Park, CA, US
heller@ai.sri.com

Eric M. BENJAMIN
Surround Research
Pacifica, CA, US
ebenj@pacbell.net

Richard LEE
Pandit Litoral
Cooktown, QLD, AU
ricardo@justnet.com.au

Abstract

Implementation of Ambisonic reproduction systems is limited by the number and placement of the loudspeakers. In practice, real-world systems tend to have insufficient loudspeaker coverage above and below the listening position. Because the localization experienced by the listener is a nonlinear function of the loudspeaker signals it is difficult to derive suitable decoders analytically. As an alternative, it is possible to derive decoders via a search process in which analytic estimators of the localization quality are evaluated at each search position. We discuss the issues involved and describe a set of tools for generating optimized decoder solutions for irregular loudspeaker arrays and demonstrate those tools with practical examples.

Keywords

Ambisonic decoder, HOA, periphonic, nonlinear optimization

1 Introduction

Ambisonics is a versatile surround sound recording and reproduction system. One of the attractions is that the transmission format is independent of the loudspeaker layout. However, this means that each playback system needs a custom decoder that is matched to the loudspeaker array. The decoder creates the loudspeaker signals from the transmission signals. Ambisonics theory provides simple encapsulations of high- and low-frequency auditory localization that can be used to design decoders, as well as theorems that ease the design of decoders for regular polygonal and polyhedral loudspeaker arrays.

In earlier papers, the present authors have discussed the design and testing of first-order decoders for regular horizontal loudspeaker layouts [Heller et al., 2008] as well as the use of nonlinear optimization to design decoders for ITU 5.1 arrays [Heller et al., 2010]. In this paper, we extend that work to full periphonic (3-D) arrays and higher-order Ambisonics (HOA). The techniques are implemented as a MATLAB [2011] and GNU Octave [Gnu, 2011] toolkit that makes use of the NLOpt library [Johnson, 2011] to perform the optimization.

We use the term *decoder* to mean the configuration for a decoding engine that does the actual signal processing. Examples are Ambdec [Adriaensen, 2011] that operates in real time, as well as an offline decoder we have implemented as part of this toolkit.[1]

In this paper, we use bold roman type to denote vectors, italic type to denote scalars, and sans serif type to denote signals. A scalar with the same name as a vector denotes the magnitude of the vector. A vector with a circumflex ("hat") is a unit vector, so, for example, $\hat{\mathbf{r}}_\mathbf{E} = \mathbf{r}_\mathbf{E}/r_E$.

We start with a discussion of the design process and the tradeoffs involved, then the specifics of the optimization process, and finally results of two arrays, a third-order decoder for the 22-loudspeaker CCRMA array, and a second-order decoder for the 12-loudspeaker 30° tri-rectangle array.

2 Designing Ambisonic Decoders

Ambisonics represents a sound field with a group of signals that are proportional to spherical harmonics. The original Ambisonic systems were first order only, but more recently, higher-order sys-

[1] Another important function of an Ambisonic decoder is to provide *near-field compensation*. This compensates for the curvature of the wavefronts due to the finite distance to the loudspeaker and is strictly a function of distance of the speaker from the center of the array and the order of reproduction. Ambdec and the offline decoder in this toolkit provide such filters and they will not be discussed further in this paper.

tems have been implemented. In first-order Ambisonics the zeroth-order component represents the sound pressure, and the three first-order components represent the acoustic particle velocity. If these components are reproduced exactly, then the sound will be correct at the center. However, it is not possible to get the first-order components correct except at a single point and not practical to get them correct at higher frequencies, where the wavelengths become smaller than the size of the human head. The task of the decoder is to create the best *perceptual impression* that the soundfield is being reproduced accurately given the loudspeaker array being used.

In practical terms, the following are necessary:

- Constant amplitude gain for all source directions
- Constant energy gain for all source directions
- At low frequencies, correct reproduced wavefront direction and velocity
- At high frequencies, maximum concentration of energy in the source direction
- Matching high- and low-frequency perceived directions

These criteria may, themselves, have different interpretation or importance depending on the source material and the intended use. We can identify three distinct types of program:

- Natural recordings made with a first-order soundfield microphone.
- Natural recordings made with higher order microphones. As of this writing, such microphones are just becoming available commercially, but practical constraints will mean that these are still first order at lower frequencies.
- Artificial recordings. First order as well as Higher Order Ambisonic (HOA) program material.

The first case is Ambisonic's greatest strength. Good first-order Ambisonic reproduction is probably the closest to recreating a virtual sound environment, whether the buzz of a busy Asian marketplace or the sound of a concert in a good hall in your living room. It will most likely be used to create realistic atmosphere even if more precise methods like HOA are used for special sounds.

To preserve this advantage requires the preservation of a good facsimile of the diffuse field. Energy gain that varies with direction and "bunching" of directions, particularly in the horizontal plane, are all detrimental, as is "speaker detent" where individual loudspeakers draw attention to themselves.

2.1 Auditory Localization

Due to the range of wavelengths involved, the human auditory localization mechanism utilizes different directional cues over different frequency regimes. At low frequencies, localization depends on the detection of Interaural Time Differences (ITDs), but at high frequencies there is an ambiguity because a human head is multiple wavelengths across above about 1 kHz. For this reason, localization switches abruptly, depending on Interaural Level Differences (ILDs) above that frequency. One way to predict localization would be to use Head Related Transfer Functions (HRTFs) to calculate the actual ear signals of a listener, but this turns out to be computationally difficult and would vary from listener to listener.

Gerzon developed a series of metrics for predicting localization that are simpler than using the HRTFs [Gerzon, 1992]. The simplest of these metrics are the velocity localization vector, $\mathbf{r_V}$, and the energy localization vector, $\mathbf{r_E}$. The direction of each indicates the direction of the expected localization perception, while the magnitude indicates the quality of the localization. In natural hearing from a single source, the magnitude of each vector should be exactly 1, and the direction of the vectors is the direction to the source. It should be noted that, while $\mathbf{r_V}$ is proportional to the physical quantity of the acoustic particle velocity, $\mathbf{r_E}$ is an abstract construct.[2]

Following Gerzon [1992], the pressure (amplitude gain), P, and total energy gain, E, are

$$P = \sum_{i=1}^{n} G_i \tag{1}$$

[2]Note that these metrics are not specific to Ambisonics; they can be used to predict the quality of the phantom images produced by any multispeaker reproduction system, regardless of the panning laws used, including plain old two-channel stereo. Gerzon shows this for several well-known stereo phenomena [Gerzon, 1992].

$$E = \sum_{i=1}^{n} (G_i {G_i}^*) \tag{2}$$

The magnitude and direction of the velocity vector, r_V and $\hat{\mathbf{r_V}}$, at the center of an array with n loudspeakers is

$$r_V \hat{\mathbf{r_V}} = \frac{1}{P} \mathrm{Re} \sum_{i=1}^{n} G_i \hat{\mathbf{u_i}} \tag{3}$$

whereas the magnitude and direction of the energy vector, r_E and $\hat{\mathbf{r_E}}$, are computed by

$$r_E \hat{\mathbf{r_E}} = \frac{1}{E} \sum_{i=1}^{n} (G_i {G_i}^*) \hat{\mathbf{u_i}} \tag{4}$$

where the G_i are the (possibly complex) gains from the source to the i-th loudspeaker, $\hat{\mathbf{u_i}}$ is a unit vector in the direction of the loudspeaker, and ${G_i}^*$ is the complex conjugate of G_i.

The velocity vector points in the same direction and is proportional to the acoustic particle velocity. It has been shown that the velocity vector predicts the ITDs very accurately [Benjamin et al., 2010]. The energy vector predicts the ILDs, but in practice it is not possible to get $r_E = 1$ unless the sound is coming from just one loudspeaker. This is representative of a pervasive problem in multichannel sound reproduction. The maximum average value of r_E that can be obtained for a given Ambisonic order is shown in Figure 1. The formulas to compute these are given in Appendix A.

Because different sets of gains are needed to satisfy the low- and high-frequency models, many ambisonic decoders split the audio into two bands, apply different decoder matrices, and then recombine to produce the loudspeaker signals.[3] Daniel has suggested that a three-band decoder may provide better reproduction under some listening conditions [Daniel, 2001]. This remains an open question at this time.

2.2 Computing the Low-Frequency Matrix

The low-frequency matrix provides gains from each channel of the ambisonic program material

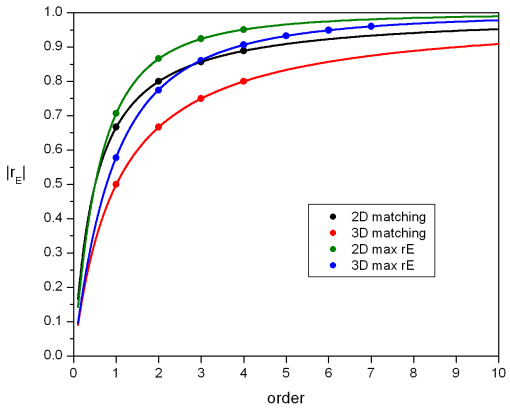


Figure 1: Maximum average r_E depending on order and type. "matching" and "max r_E" refer to the decoder matrices described in Sections 2.2 and 2.3, respectively.

to each loudspeaker that are needed to optimize localization as predicted by the velocity localization vector, $\mathbf{r_V}$. Numerous authors have provided derivations of the low-frequency solution for a given loudspeaker array, and thus a number of different terms are used to refer to it, including "velocity", "matching", "basic", "exact", "mode matching", "re-encoding" and so forth.

In practice, these reduce to projecting (or encoding) the loudspeaker directions onto the selected spherical harmonic basis set,[4] assembling these vectors into an array, and computing the Moore-Penrose pseudoinverse of the array [Weisstein, 2008]. Examples of this can be found in Appendix A of [Heller et al., 2008]. In general, there are an infinite number of solutions and this procedure provides the solution with the minimum L2-norm (i.e., the least-squares fit), which has the desirable property of requiring the minimum total radiated power.[5]

[3] This places certain constraints on the phase response of the band splitting filters. We discuss the design and implementation of suitable filters in Appendix B of [Heller et al., 2008] and note that the filters in Ambdec meet these requirements.

[4] The toolkit is neutral as to the conventions for component ordering and normalization. These conventions are encapsulated in a single function. The current implementation supports the Furse-Malham set [Malham, 2003], but others can be added easily.

[5] Recently, some authors, drawing upon compressive sending theory, have suggested that the L1-norm may be more suitable [Wabnitz et al., 2011; Zotter et al., 2012]. L1-norm minimizes the sum of absolute errors. Compared to least-squares, it allows larger maximum errors in exchange for more zero errors.

Except in the case of degenerate configurations, where all the loudspeakers lie in the null of one or more of the spherical harmonics, this procedure will result in a decoder matrix that satisfies the low-frequency localization criteria exactly; however, it may utilize a great deal of power to get them correct in directions where there is a large angular separation between the loudspeakers in the array. This will result in low r_E values in those directions. As we shall see, except in the case of regular polyhedra and polygons, it is impossible to fully satisfy all the ambisonic criteria simultaneously. This implies that while the ambisonic transmission format is independent of the loudspeaker array, not all loudspeaker array geometries perform equally well.

2.3 Computing the High-Frequency Matrix

The high-frequency matrix provides gains from each channel of the ambisonic program material to each loudspeakers that are needed to optimize localization as predicted by the energy localization vector, $\mathbf{r_E}$. Gerzon proved two theorems for first-order reproduction, the polygonal decoder theorem and the diametric decoder theorem. They state that in an array with a minimum of four loudspeakers for 2-D and six speakers for 3-D, where the loudspeakers are spaced in equal angles or in diametrically opposed pairs, $\mathbf{r_E}$ is guaranteed to point in the same direction as $\mathbf{r_V}$. The polygonal decoder theorem also holds for higher-order Ambisonic reproduction, provided there is an adequate number of loudspeakers in the array to support the desired order. This simplifies the task of designing the high-frequency matrix to that of selecting the gain for each order such that the overall magnitude of $\mathbf{r_E}$ is maximized. For first-order decoders, Gerzon provided the values of $\frac{\sqrt{2}}{2}$ for horizontal arrays and $\frac{\sqrt{3}}{3}$ for periphonic arrays. Daniel derived general formulas for these gains [Daniel, 2001], which are given in Tables 1 and 2. (See Appendix A for programs that compute the values in these tables.)

As we will see in the example in Section 2.5, once the array deviates from having equal angles, there is no longer a guarantee that $\mathbf{r_E}$ and $\mathbf{r_V}$ point in the same direction or that there is a single set of gains that maximize r_E in every direction. Because of this, we must trade off the various criteria and due to the nonlinear nature of the criteria, numerical optimization is needed to compute the solutions, which will be discussed in Section 3.

2.4 Merging the LF and HF Matrix

The existence of different optimum decoder coefficients for optimum $\mathbf{r_V}$ and $\mathbf{r_E}$ would typically mean having to make a choice or compromise between the two. In this case, however, both can be had. The decoder that optimizes $\mathbf{r_V}$ can be used at low frequencies and the decoder that maximizes $\mathbf{r_E}$ can be used at high frequencies, by the simple expedient of using filters to cross over between the two. This is typically done at around 400 Hz.

Because the higher-order components are reduced in order to maximize $\mathbf{r_E}$, this causes a reduction in the total signal level of the high-frequency decoder outputs, and thus a reduction in the high frequencies heard by the listener. The gains that maximize r_E specify the relationship among the signals of different order, but not how that gain should be apportioned between high-frequency cuts and low-frequency boosts. There are three possibilities:

1) Preservation of the amplitude. That is, simply use the gains produced by the optimizer or those given in Tables 1 and 2.
2) Preservation of the root-mean-square (RMS) level. This is what Gerzon [1980] suggests and is what is implemented in older analog decoders.
3) Conservation of the total energy. Daniel [2001] suggests this, and the configuration files included with Ambdec follow this recommendation. This method results in more high frequencies with more speakers.

The calculations involved are given in Appendix B. In listening tests, we have found that preservation of the RMS level works well for small arrays. We have also found that using the conservation of energy approach on large 3-D arrays results in overemphasizing high frequencies and near-head imaging artifacts and nulls. In practice, we the set the LF/HF balance by ear, comparing the balance of the two-band decoder to that of a single-band r_E-max decoder. More work is needed to find a procedure for this that does not involve tuning by ear.

Order	Max r_E	Gains
1	0.707107	1, 0.707107
2	0.866025	1, 0.866025, 0.5
3	0.92388	1, 0.92388, 0.707107, 0.382683
4	0.951057	1, 0.951057, 0.809017, 0.587785, 0.309017
5	0.965926	1, 0.965926, 0.866025, 0.707107, 0.5, 0.258819

Table 1: Per-order gains for max-r_E decoding with 2-D regular polygonal arrays.

Order	Max r_E	Gains
1	0.57735	1, 0.57735
2	0.774597	1, 0.774597, 0.4
3	0.861136	1, 0.861136, 0.612334, 0.304747
4	0.90618	1, 0.90618, 0.731743, 0.501031, 0.245735
5	0.93247	1, 0.93247, 0.804249, 0.62825, 0.422005, 0.205712

Table 2: Per-order gains for max-r_E decoding with periphonic regular polyhedral arrays.

2.5 Selection of a speaker array

Due to symmetry, regular loudspeaker arrays have the advantage of uniformity in the localization predictors $\mathbf{r_V}$ and $\mathbf{r_E}$. As noted above, practical difficulties usually prevent the attainment of a completely regular array. There will be a tendency for r_E to be greater in the directions where the angular density of loudspeakers is greater, and less in the directions where there are few loudspeakers and $\mathbf{r_E}$ will tend to point in the directions of concentrations of loudspeakers.

It should be noted at this point that it is impossible to get r_E to be larger in the direction between loudspeakers than the value achieved simply by driving the loudspeakers nearest to the gap equally. This means that the best that can be achieved by an Ambisonic decoder is to have a smooth transition between areas where the performance is good (large number of loudspeakers, high magnitude of r_E and $\mathbf{r_E}$ points in the intended direction) and areas where the performance is less good (fewer loudspeakers, $\mathbf{r_E}$ has small magnitude and points in an incorrect direction). As such, we must be careful in choosing the decoder parameters so that the performance in the good directions is good enough, and the performance in the poor directions is not too bad.

A simple example of a four-speaker array will illustrate these difficulties. A square horizontal array has a basic decoder solution of

$$\begin{bmatrix} \mathsf{LF} \\ \mathsf{RF} \\ \mathsf{RR} \\ \mathsf{LR} \end{bmatrix} = \frac{\sqrt{2}}{4} \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & -1 & 0 \\ 1 & -1 & -1 & 0 \\ 1 & -1 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathsf{W} \\ \mathsf{X} \\ \mathsf{Y} \\ \mathsf{Z} \end{bmatrix} \tag{5}$$

This gives exact recovery of the pressure and velocity at the center of the array: $|\mathbf{r_V}| = 1$ and points in the intended direction. But because the angular separation of the loudspeakers is 90°, $r_E = \frac{2}{3}$. However, if we investigate what happens as the ratio of pressure (W) and velocity (X, Y and Z) is varied, it develops that r_E is maximum for the case where the first-order components are reduced to $\frac{\sqrt{2}}{2}$ of their original value. This gives a magnitude of $\mathbf{r_E}$ of $\frac{\sqrt{2}}{2}$.

If the square is replaced with a rectangle with an aspect ratio of $\sqrt{3} : 1$, the front and rear loudspeakers now subtend an angle of 60° and the side loudspeakers subtend an angle of 120°. This reduces r_E at the sides but increases it in the front, relative to a square. If the same gain as derived for the square ($\frac{\sqrt{2}}{2}$ for the first-order components) is applied, then there is a substantial improvement in r_E to the sides, and a very tiny decrease in r_E in the front. This is shown in Figure 2.

But is this the "optimum"? Figure 3 shows that if we further vary the ratio of the zero- and first-order components it develops that r_E, evaluated at the sides, is a maximum at a different ratio.

It is thus possible to maximize r_E in front *or* at

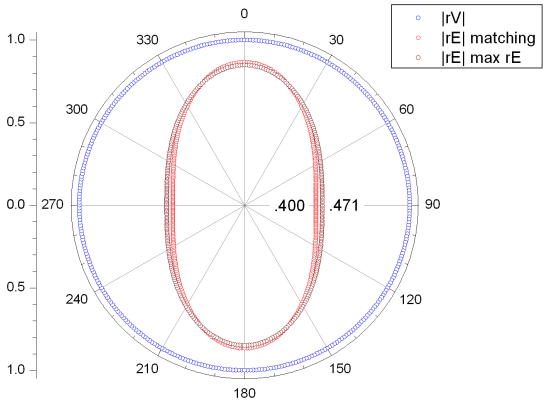


Figure 2: Locus of $\mathbf{r_E}$ for a rectangular array, matching and "max r_E" decoders.

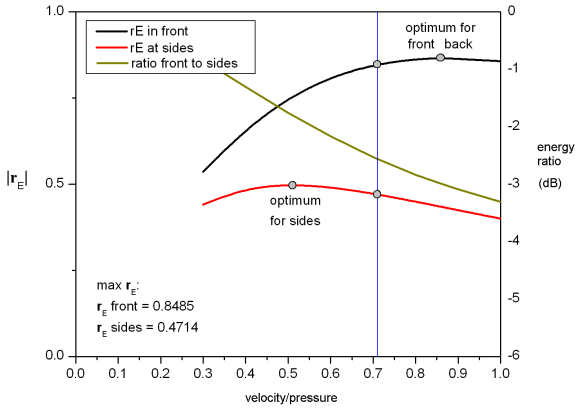


Figure 3: r_E and the energy, E, as a function of the ratio of the first-order to zero-order scaling.

the sides, but not both at once. One might wish to use a different decoder depending on whether sound images are expected at the front, or on the sides, or a decoder that gives a compromise between the two.

Thus far, only the quality and direction of the localization have been discussed. There is also an effect on the loudness of the sound, depending on direction. If the loudspeaker array is irregular, then the solution to recover pressure and velocity results in an increase in energy in the directions where the angular spacing is greatest. This results in an increase in reproduced loudness in those directions.

For the previous example of a rectangular array with a $\sqrt{3}:1$ aspect ratio, the ratio of the energy in the forward direction to the energy at the sides was calculated and is also plotted in Figure 3. r_V obtains its correct value of 1 in all directions and the pressure response is also omnidirectional. At higher frequencies, where the "max-r_E" decoder is in effect, r_E is maximized for front and back directions (where the speakers are closer together). At these frequencies, sounds from the sides (perceived as "energy") are 2.4 dB louder. This is a pervasive problem for irregular arrays and will be addressed in greater detail below. The simple $\sqrt{3}:1$ rectangle as above is used widely and is known to give good results. On the other hand, listening tests indicate that the 5.5 dB energy imbalance exhibited by some first-order decoders for ITU 5.1 arrays is too large. From this, we propose 3 dB variation in "energy" with horizontal direction as the maximum imbalance acceptable.

2.5.1 Discussion of compromises of speaker arrays

The selection of a loudspeaker array for Ambisonic reproduction is subject to a number of compromises, notably the space available to house the array and the budget for purchasing loudspeakers. It may be that the array already exists, in which case the decoder design task is one of selecting a decoder design that provides the best audible performance. In other situations, however, the design of the array has not been fixed although the number of loudspeakers may have been. In that case, there is substantial latitude to trade off between high-order performance horizontally and periphonic performance.

3 Optimizer

As noted in Section 2.5, in an irregular array, simply scaling the LF and HF matrices does not result in $\mathbf{r_V}$ and $\mathbf{r_E}$ pointing in the same direction; hence, the design procedure becomes somewhat more complex.

Because the key psychoacoustic criteria for good decoder performance are nonlinear functions of the speaker signals, we utilize numerical optimization techniques. To do this, a single objective function is formulated that takes as input the decoder matrix and produces a single figure of merit that decreases as the decoder performance improves. The nonlinear optimization algorithm

will then try different sets of matrix elements, attempting to arrive at the lowest value possible. Because there are a number of criteria, we use the weighted sum to provide an overall figure of merit. A user can adjust the weights to set the relative importance of the different criteria, say uniform energy gain (loudness) versus angular accuracy. In addition, each test direction can have its own set of weights, so that, for example, angular accuracy can be emphasized for the front, while uniform energy gain is emphasized in other directions. This might be the preferred configuration for classical music recorded in a reverberant performance hall. On the other hand, environmental recordings made outdoors have very little diffuse content, so overall angular accuracy is more important. Another application of direction weightings is in highly asymmetrical arrays, such as a dome, where few speakers are below the listener. In this case, we expect poor performance in those directions, so they are deemphasized when computing the objective function.

We have employed the NLOpt library for nonlinear optimization [Johnson, 2011]. NLOpt provides a common application programming interface (API) for a collection of nonlinear optimization techniques. In particular, it supports a number of "derivative free" optimization algorithms, which are well suited to the current application where the objective function is the result of a computation, rather than an analytic function.

An earlier version of the optimizer that was limited to first-order horizontal arrays was written in C++ [Heller et al., 2010]. To extend that to higher-order and periphonic arrays required a significant rewrite, so an initial prototype was written in MATLAB, with plans to recode in C++. Because the bulk of the computation is matrix multiplication, which is handled by highly optimized code in MATLAB, it turned out that the execution speed was almost as fast as the original C++ version, so we abandoned plans for the rewrite. To make the code widely usable, it was kept compatible with GNU Octave. The key change is that GNU Octave does not support nested functions, so a number of variables need to be declared global to make them accessible to the objective function.

3.1 Optimization Criteria

For each test direction, the following are computed: amplitude gain, P, energy gain, E, the velocity localization vector, $\mathbf{r_V}$, and the energy localization vector, $\mathbf{r_E}$. From these, we compute the pairwise angles between the test direction, $\hat{\mathbf{r_V}}$, and $\hat{\mathbf{r_E}}$. These are summarized with the following figures of merit: deviation of amplitude gain from 1 along the x-axis, minimum, maximum, and RMS values of amplitude gain, energy gain, magnitude of $\mathbf{r_V}$, magnitude of $\mathbf{r_E}$, and the pairwise angular deviations.

It is important that the criteria are "well behaved" near zero, so as not to trigger oscillating behavior in the optimizer. They should be continuous and have first derivatives. In practical terms, absolute value and thresholds should not be used; squaring can be used for the former and the exponential function for the latter cases.

Finally, directional weightings are applied to each criteria and then an overall weighted sum produces the single figure of merit for that particular configuration.

3.2 Test Directions

As mentioned in the previous section, each candidate set of parameters is evaluated from a number of directions. For 2-D speaker arrays, 180 or 360 evenly spaced directions are often used [Wiggins et al., 2003; Moore and Wakefield, 2008]. For 3-D arrays, the situation is more complex because no more than 20 points can be distributed uniformly on a sphere (a dodecahedron).

Lebedev-Laikov quadrature defines sets of points on the unit sphere and weights with the property that they provide exact results for integration of the spherical harmonics [Lebedev and Laikov, 1999]. The current implementation provides a function that returns Lebedev grids of points and corresponding weights for as many as 5810 directions. Our current experiments have used a grid with 2702 points, which corresponds roughly to 3°. The toolkit also has functions providing 2-D and 3-D grids that are sampled in uniform azimuth and elevation increments, which are useful for visualization of the results.

3.3 Optimization Behavior

As part of the optimization setup, the user supplies a set of stopping criteria. This can be specified as a threshold on the absolute and relative

changes in the parameters and/or the objective function, as well as a maximum running time and maximum number of iterations. The default values in the current implementation are 1×10^{-7} for the parameters and objective function.

For small 2-D arrays (say, 12 to 24 parameters), the optimizer typically converges in less than 1 minute, examining 40,000 to 1,500,000 configurations. For large high-order arrays (say, 200 to 400+ parameters), it typically converges in less than an hour. These timings were done with Octave version 3.2.4-atlas on a 2.66 GHz Intel Core i7 with 8 GB of memory. The bulk of the computation comprises matrix multiplications and is therefore suitable for parallel implementations. The timings in MATLAB were approximately 2x faster than Octave since it can make use of the multiple cores in the i7 processor.

With large optimization problems, using a local optimization algorithm and providing an initial solution that is near to the optimum is important for reliable convergence. The toolkit currently supports three strategies:

- Using the low-frequency solution modified with the per-order gains that would provide the max-r_E solution for a uniform array.
- "Musil Design" where additional "virtual" loudspeakers are inserted into the array to make the spacing more uniform, and hence more suited to a pseudo-inverse solution. After the optimization is complete, the signals for the virtual speakers are either ignored or distributed to the adjacent speakers [Zotter et al., 2010].
- A hierarchical approach, decomposing the optimization by establishing a solution for each order consecutively, freezing the individual coefficients for orders below the current one, but allowing an overall adjustment on the gain of the lower orders.

4 Examples

The software tools described above were applied to the derivation of decoders for several real-world systems. The examples given here are the CCRMA listening room[6] and a tri-rectangle with the upper and lower loudspeakers at ±30° with respect to horizontal.

4.1 CCRMA Listening Room

The described software was applied to deriving decoders for the Listening Room at CCRMA (Center for Computer Research in Music and Acoustics) at Stanford. This facility consists of 22 identical loudspeakers arranged in five rings. There is a horizontal ring of eight loudspeakers, two rings of six loudspeakers, one 50° below and one 40° above horizontal, and one loudspeaker directly above and one directly below the listening position. The two hexagonal rings are thus not exactly horizontally opposed. A schematic of the array is shown in Figure 4a.

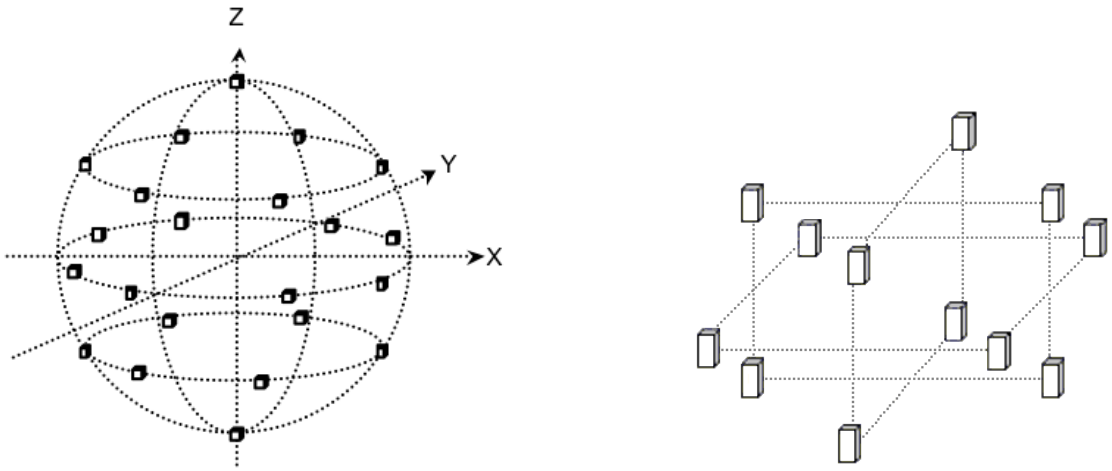
An initial solution was derived by calculating the pseudoinverse of the loudspeaker projection matrix as described above. The decoder was modified to optimize the magnitude of r_E at high frequencies by applying the weighting factors given in Table 2 to the gains of the signal components of each order. Given that the theoretical maximum average value for r_E can be no greater than 0.866 at third order, the average value of 0.850 for the third order decoder given here does not leave a great deal of margin for improvement. Nonetheless, the optimization software was applied to the problem. Figure 5 shows the performance of the initial solution and the optimized result. Average r_E was increased slightly, and maximum directional error reduced by a factor of 5.

An informal listening test comparing this decoder to the existing one was conducted using third-order test signals and studio recordings, as well as first-order acoustic recordings. The general impression was that the new decoder did a better job of keeping horizontal sources in the horizontal plane, whereas the existing decoder rendered such sources above the horizontal plane.

4.2 The 30° Tri-Rectangle

As discussed elsewhere, a dodecahedron or other large regular array is difficult to fit into normally dimensioned spaces. One large array that does fit into normal spaces is the so-called *tri-rectangle*, patterned after a suggestion by Gerzon. A schematic is shown in Figure 4b. It consists of three interlocking rectangles of loudspeakers, one in the horizontal plane, one in the XZ plane, and one in the YZ plane. The projection of the loud-

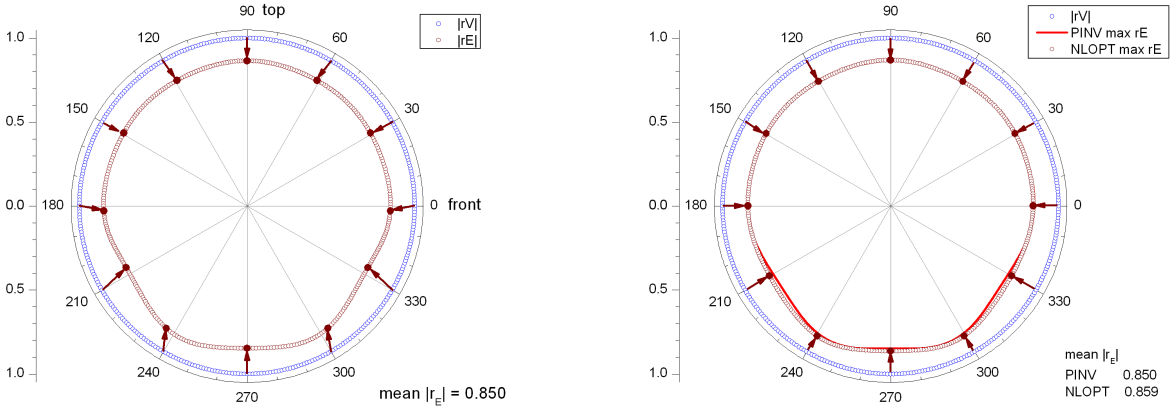
[6] See https://ccrma.stanford.edu/room-guides/listening-room/



(a) The 22-loudspeaker array at CCRMA

(b) The 12-loudspeaker tri-rectangle.

Figure 4: Schematics of the loudspeaker arrays used in the examples.



(a) The initial solution calculated by pseudoinverse and max-r_E gains.

(b) The optimized solution.

Figure 5: $\mathbf{r_E}$ in the vertical plane for the CCRMA array before and after optimization. The arrows show the directional error between the low- and high-frequency matrices. In this case, average r_E was increased slightly, from 0.85 to 0.86 and the maximum directional error reduced by a factor of 5.

speakers into any plane is an octagon, which hints at its utility for reproducing second-order program material. However, to enable it to fit into typical spaces the vertical rectangles must be squashed to an approximate ±30° vertical angle. This gives a solid angle of 120° above and below the listening position with no loudspeakers. Naturally, this has a profound effect on the localization for sources above and below the listening position.

Performance of the initial solution by inversion is shown in Figure 6. The magnitude of r_E is in red, with both the horizontal and vertical (in the XZ plane) shown. The horizontal shape is essentially circular, with perfect direction (not shown in the figure), but the magnitude of $\mathbf{r_E}$ decreases dramatically for sources above or below about ±30° of elevation. Furthermore, there is an increasing error in the direction of $\mathbf{r_E}$ indicating that high-

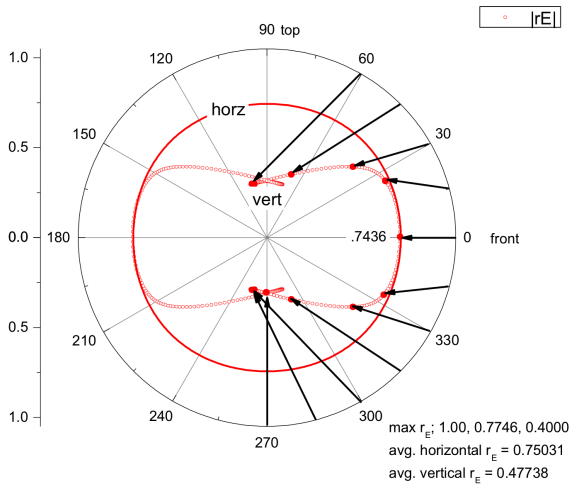


Figure 6: $\mathbf{r_E}$ in the horizontal and vertical planes for the initial second-order decoder.

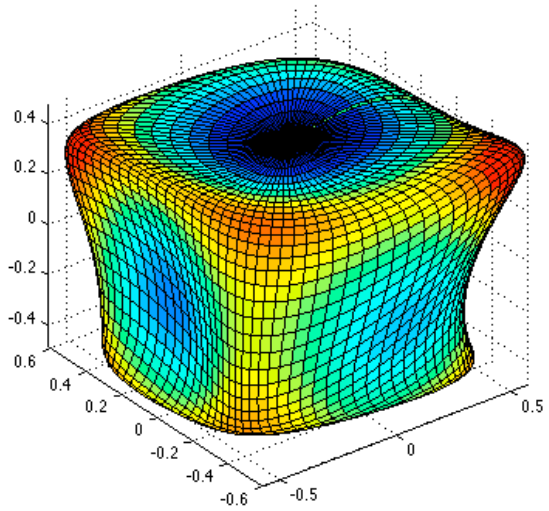


Figure 7: 3-D plot of $\mathbf{r_E}$ from an unconstrained optimization of the second-order decoder for the 30° tri-rectangle.

frequency sounds will be perceived as coming from near the poles. The extreme errors in the direction of r_E are compounded by the low values, making localization in the up and down directions vague in any case.

The large angle subtended by the loudspeaker placement with respect to the vertical axis makes it impossible to get precise localization for sources directly above or below the listening position. It may, however, be possible to improve the localization for sources near horizontal by correcting the direction of r_E.

Running the optimizer with this configuration as an initial solution resulted in a highly distorted solution where the sounds are drawn strongly to the loudspeakers. A 3-D plot of $\mathbf{r_E}$ for this solution is shown in Figure 7. As can be seen, the performance is very non-uniform (a sphere would be ideal) and the maximum angular error is over 30°.

Next, a Musil design was attempted. Virtual loudspeakers were inserted into the array directly above and below the center. This was optimized and then the signals for the virtual loudspeakers reassigned to the nearest real loudspeakers. This resulted in improved r_E in the horizontal plane, as well as elevations as high as ±30°; however, it suffers from directional errors as large as 31°. Figure 8a shows the optimized solution.

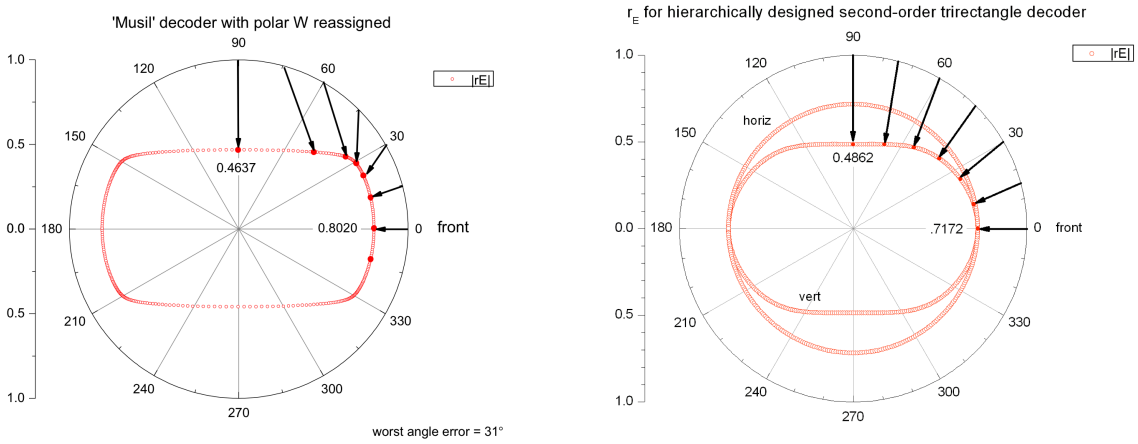
Finally, a hierarchical design was attempted, where each subsequent order is optimized separately. This resulted in a slightly lower r_E, but significantly reduced angular error in the vertical plane. Figure 8b shows the optimized solution.

5 Conclusions

An open source package for the design of ambisonic decoders has been presented. The software allows the derivation of decoders for arbitrary loudspeaker arrays, 2-D or 3-D. The software operates under Octave or MATLAB, with the nonlinear optimization performed by the open source package NLOPT. Auditory localization at middle and high frequencies is a nonlinear function of the loudspeaker signals, which necessitates the finding of solutions that work well for those frequencies via an optimization process.

Two example systems were solved. The first was a third-order decoder for the 22-loudspeaker CCRMA listening room. That system is nearly regular, and it was found that a solution obtained by inversion of the loudspeaker matrix, with per-order gains, was nearly as good as one obtained by the nonlinear optimization process. Nonetheless, the magnitude of $\mathbf{r_E}$ was improved and the angle error was reduced.

The second system was a 12-loudspeaker tri-rectangle, with the upper and lower loudspeakers at 30° above and below the horizontal plane. A decoder derived for that system via the technique



(a) Musil design.

(b) Hierarchical design.

Figure 8: $\mathbf{r_E}$ in the vertical plane for second-order decoders for the 30° tri-rectangle.

of inversion followed by per-order gains shows high magnitudes of r_E in the horizontal plane but low magnitudes in the polar regions and large errors in the direction of $\mathbf{r_E}$.

Two additional methods were tried in a search for a superior solution. The first was the Musil decoder in which the array was filled out with virtual loudspeakers at the poles and the signals for those speakers are routed to the nearest real speakers.

The second method was a hierarchical one in which a solution for each order was established consecutively, such that a higher-order decoder is also optimum for lower-order program sources. This results in a very well behaved decoder, but with slightly lower values of r_E.

6 Acknowledgements

The authors thank Fernando Lopez-Lezcano for encouraging us to write this paper and for posing the problem of deriving a third-order decoder for the CCRMA listening room; Andrew Kimpel and Elan Rosenman for stimulating discussion and access to their second-order periphonic and fifth-order horizontal loudspeaker arrays; and Isaac Heller for the idea of using the exponential function to implement soft thresholds.

References

Fons Adriaensen, 2011. *AmbDec User Manual*, 0.4.3 edition. http://kokkinizita.linuxaudio.org/ambisonics/. 1

Eric Benjamin, Richard Lee, and Aaron Heller. 2010. Why Ambisonics Does Work. In *AES 129th Convention*, San Francisco, November. 3

Jerome Daniel. 2001. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia*. Ph.D. thesis, University of Paris. 3, 4

Michael A. Gerzon. 1980. Practical Periphony: The Reproduction of Full-Sphere Sound. In *65th Audio Engineering Society Convention Preprints*, number 1571, London, February. AES E-lib http://www.aes.org/e-lib/browse.cfm?elib=3794. 4

Michael A. Gerzon. 1992. General Metatheory of Auditory Localisation. In *92nd Audio Engineering Society Convention Preprints*, number 3306, Vienna, March. AES E-lib http://www.aes.org/e-lib/browse.cfm?elib=6827. 2

2011. Octave. http://www.octave.org/ (accessed July 1, 2011). 1

Aaron J. Heller, Richard Lee, and Eric M. Benjamin. 2008. Is My Decoder Ambisonic? In *AES 125th Convention*, number 7553. 1, 3

Aaron Heller, Eric Benjamin, and Richard Lee. 2010. Design of Ambisonic Decoders for Irregular Arrays of Loudspeakers by Non-Linear Optimization. In *AES 129th Convention*, San Francisco. 1, 7

Steven G. Johnson. 2011. The nlopt nonlinear- optimization package. http://ab-initio.mit.edu/nlopt. 1, 7

V.I. Lebedev and D.N. Laikov. 1999. A Quadrature Formula for the Sphere of the 131st Algebraic Order of Accuracy. *Doklady Mathematics*, 59(3):477–481. 7

David G Malham. 2003. Space in music - music in space: Higher order ambisonic systems. Master's thesis, University of York. 3

MATLAB. 2011. *version 7.13.0 (R2011b)*. The MathWorks Inc., Natick, Massachusetts. 1

David Moore and Jonathan Wakefield. 2008. The Design of Ambisonic Decoders for the ITU 5.1 Layout with Even Performance Characteristics. In *124th Audio Engineering Society Convention Preprints*, number 7473, Amsterdam, May. 7

Andrew Wabnitz, Nicolas Epain, A van Schaik, and C Jin. 2011. Time Domain Reconstruction of Spatial Sound Fields using Compressed Sensing. In *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, pages 465–468. IEEE. 3

Eric W. Weisstein. 2008. Moore-Penrose Matrix Inverse. From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/Moore-PenroseMatrixInverse.html. 3

Bruce Wiggins, Iain Paterson-Stephens, Val Lowndes, and Stuart Berry. 2003. The Design and Optimisation of Surround Sound Decoders Using Heuristic Methods. In *Proceedings of UKSim 2003, Conference of the UK Simulation Society*, pages 106–114. UK Simulation Society. Availible from http://sparg.derby.ac.uk/SPARG/PDFs/SPARG_UKSIM_Paper.pdf (accessed May 15, 2006). 7

Franz Zotter, H. Pomberger, and M. Noisternig. 2010. Ambisonic Decoding with and without Mode-Matching: A Case Study Using the Hemisphere. *2nd Ambisonics Symposium, Paris*. 8

Franz Zotter, H. Pomberger, and M. Noisternig. 2012. Energy-Preserving Ambisonic Decoding. *Acta Acustica united with Acustica*, 98(1):37–47, January. 3

A Formulas for maximum average r_E and per-order gains

A.1 Horizontal Arrays

For regular horizontal arrays (Table 1), the maximum value of r_E and the gains for each Ambisonic order, M, are given by

$$r_E = \text{largest root of } T_{M+1}(x) \tag{6}$$

$$g_m = T_m(r_E),\ m = 0 \dots M \tag{7}$$

where T_m is the m^{th} Chebyshev polynomial of the first kind. In Mathematica, this can be written as[7]

```
Table[ChebyshevT[Range[0, M],
  x /. FindRoot[ChebyshevT[M+1,x], {x,1}]],
  {M, 1, 5}]
```

A.2 Periphonic Arrays

For regular periphonic arrays (Table 2), the maximum value of r_E and the gains for each Ambisonic order, M, are given by

$$r_E = \text{largest root of } P_{M+1}(x) \tag{8}$$

$$g_m = P_m(r_E),\ m = 0 \dots M \tag{9}$$

where P_m is the m^{th} Legendre polynomial. In Mathematica, this can be written as

```
Table[LegendreP[Range[0, M],
  x /. FindRoot[LegendreP[M+1,x], {x,1}]],
  {M, 1, 5}]
```

B LF/HF Matching

As mentioned in Section 2.4, there are three approaches to adjusting the g_m to match LF/HF loudness, $g'_m = g'_0\ g_m$. For approach 1, $g'_0 = 1$. For approaches 2 and 3, g'_0 is calculated as

$$E_{\{g_m\}} = \sum_{m=0}^{M} C_m\, g_m^2 \tag{10}$$

$$g'_0 = \sqrt{N / E_{\{g_m\}}} \tag{11}$$

where C_m is the number of signals in the m^{th} order component. In 3-D, $C_m = m^2 + 1$; in 2-D, $C_1 = 1$ and $C_{m>1} = 2$. For approach 2, N is the total number of components: in 3-D, $(M+1)^2$; in 2-D, $2M + 1$. For approach 3, N is the number of loudspeakers in the array.

[7] For those without access to Mathematica, these can also be computed interactively using the Wolfram|Alpha online service, http://alpha.wolfram.com.

JunctionBox for Android: An Interaction Toolkit for Android-based Mobile Devices

Lawrence FYFE
InnoVis Group
University of Calgary
2500 University Drive NW
Calgary, AB T2N 1N4
Canada
ljfyfe@ucalgary.ca

Adam TINDALE
Alberta College of
Art + Design
1407 14 Avenue NW
Calgary, AB T2N 4R3
Canada
adam.tindale@acad.ca

Sheelagh CARPENDALE
InnoVis Group
University of Calgary
2500 University Drive NW
Calgary, AB T2N 1N4
Canada
sheelagh@ucalgary.ca

Abstract

JunctionBox is an interaction toolkit specifically designed for building multi-touch sound control interfaces. The toolkit allows developers to build interfaces for Android mobile devices, including phones and tablets. Those devices can then be used to remotely control any sound engine via OSC messaging. While the toolkit makes many aspects of interface development easy, the toolkit is designed to offer considerable power to developers looking to build novel interfaces.

Keywords

Android, OSC, mobile, interaction, toolkit.

1 Introduction

The Android operating system [Google, 2012a], developed by Google Inc., runs on a wide variety of mobile devices, including phones and tablets. With the easy availability of these devices, it becomes desirable to develop sound and music applications for them.

One application scenario involves using interfaces to control sound in which the interface is one computer and the sound generator is another. The advantage of this scenario is that the quality of the audio on mobile devices will likely not be as good as with combinations of audio interface and speakers in which both are designed to provide the highest quality audio experience. Another related advantage is the possibility of multi-channel scenarios that go beyond the two channels available on portable devices.

For example, an Android device would handle input by a performer, with the option of projecting the interface, while another computer handles all sound processing. This is not a hypothetical scenario but is the current musical practice of the first author. With this kind of scenario, certain aspects of development, like messaging between computers, can be made easier by a toolkit that puts commonly used or repetitively coded features into a single place for reuse.

JunctionBox is an interaction toolkit for building sound control interfaces that addresses this scenario. The toolkit is written in Java and is designed as a library for the Processing development environment [Reas and Fry, 2006]. JunctionBox bridges the interaction gap between the visual design possibilities offered by Processing and the sound control possibilities afforded by the Open Sound Control (OSC) protocol [Wright, 2005]. Figure 1 shows how JunctionBox relates these two functions. It does this by handling touch interactions and by making it easy for developers to map those interactions to OSC messages (via the JavaOSC library [Ramakrishnan, 2003]). JunctionBox has mapping features that are described in detail in an earlier paper by the current authors ([Fyfe et al., 2011]).

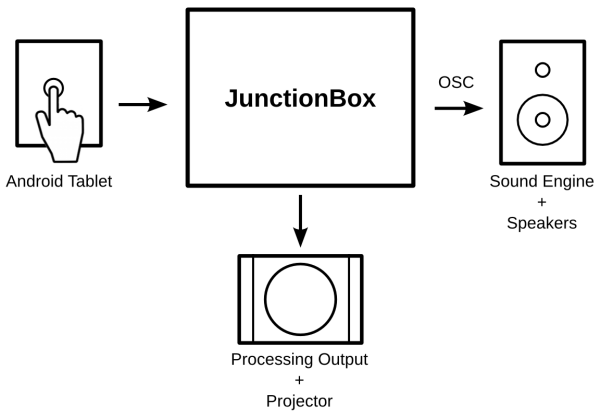


Figure 1: How JunctionBox serves as a hub for both sound and visuals.

2 Background

Other toolkits exist that have features similar to that of JunctionBox. The MT4J toolkit [Laufs et al., 2010] has many features for multi-touch interaction building and is available for use with Android. However, MT4J, lacks the OSC mapping capabilities that make JunctionBox a toolkit for building sound and music control applications. TouchOSC [hexler.net, 2012] offers functionality that is similar to what is offered by JunctionBox in which widgets can be mapped to custom OSC messages. Widgets include faders, push buttons and rotary controls and others that are are skeuomorphs of controls for physical hardware devices like mixers.

Instead of simply offering widgets, JunctionBox is a full-fledged development toolkit that is meant to be used by developers who are not necessarily interested in skeuomorphs. The JunctionBox approach to interaction design can be summed up as BYOW, for build your own widgets. The focus of JunctionBox is to provide functionality like OSC message mapping while offering the full range of visual design options available via Processing. This approach to toolkit design is meant to encourage greater creativity in the design of both interactions and interfaces.

3 JunctionBox for Android

The original version of the JunctionBox toolkit was not developed for Android. However, since it was written in Java, porting to Android did not require a complete rewrite of the existing code. The main difference between the standard Java version and the Android version of JunctionBox is that TUIO [Kaltenbrunner et al., 2005] is not needed on the Android version since Android devices have a separate system for handling touch interactions. Android-based systems get touch information via handling the data contained in the MotionEvent class [Google, 2012b].

Both versions of JunctionBox contain a Dispatcher class that is responsible for handling touch interactions that ultimately get mapped to OSC messages for controlling audio. Figure 2 shows the relationship of the MotionEvent class to JunctionBox classes including the Dispatcher. The mapping process is exactly the same for both the standard version of JunctionBox and the Android version.

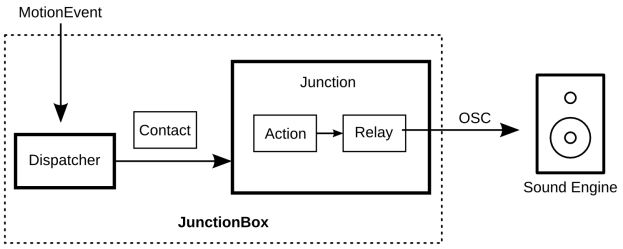


Figure 2: JunctionBox/Android internals with classes shown as boxes with solid lines.

Differences in using the Dispatcher in code have been minimized with both versions using the same code for construction. A new Dispatcher will take four arguments: the width and height of the screen/touch interface and a socket destination for OSC messages. The following code constructs a new Dispatcher object:

```
Dispatcher dispatcher = new Dispatcher(
    screenWidth,
    screenHeight,
    "192.168.1.1",
    7000);
```

All code written for the standard version of JunctionBox will work with Android with the addition of a single method made available in the Android Dispatcher class. The additional method, handleMotionEvent(), gets touch information from the MotionEvent class. The Dispatcher's handleMotionEvent() method is utilized by creating a dispatchTouchEvent() method in the code for the Processing sketch:

```
boolean dispatchTouchEvent(MotionEvent ev) {
    dispatcher.handleMotionEvent(ev);
    return true;
}
```

While only a single additional method is required for the Android version of JunctionBox, internally an important difference in touch location handling is that TUIO touch tracking data is normalized relative to the size of the interface while for Android devices, touch position is absolute. This difference is made transparent to the developer and the same code will work in both systems with only one minor change, the addition of the new handleMotionEvent() method. Making differences like this transparent is part of the JunctionBox approach of making development easier without taking away the ability to build highly customized interfaces.

4 Interfaces

The first author of this paper wrote the code for JunctionBox not only to provide the community with a toolkit for creating networked instruments but for his own performance practice. That practice currently consists of the building of touch interfaces for Android devices that run the interface to handle touch input. That interface then communicates with another computer running audio via OSC. The audio computer used to develop the following interfaces runs Ubuntu Linux, JACK and Pd.

4.1 Orrerator

The Orrerator interface has widget-like controls but with a much more stylized appearance. It is designed both as a general instrument that could be used for a variety of pieces but also specifically for the composition by the first author entitled *Sol Aur*.

The metaphor for the Orrerator is the Orrery or a model of the solar system, as shown in Figure 3. The sound engine controlled by the Orrerator is written in Pd and features four FM oscillators. The interface has four planet buttons that light up when toggled by simply touching the particular planet. Each planet button turns a single FM oscillator on or off.

In addition to on or off controls, each planet button can be rotated around its orbit by touching the orbit area and rotating it around the center. Orbit areas look increasingly lighter towards the inner planet. This change of rotation will detune that particular FM oscillator from its base frequency by a factor of between one and two with the starting vertical position being one and a full rotation back to that same position being two. Orbital rotations are limited to 360 degrees from the starting position.

On the left and right edges of the interface are two sliders: the left slider changes the index of modulation and the modulation frequency and the right slider changes the gain of all four oscillators.

4.2 Distance2

The Distance2 interface is designed specifically for a composition entitled *Distance 2 (Toshi Ichiyanagi)* by the first author. It features ten tiles numbered 1-5 with half of the tiles being white and half black. The black tiles feature reversed numbers 1-5. Each of the tiles can be moved anywhere in the interface with a single touch. When the tiles are moved into the target in the center of the interface and the touch is released, a sound file corresponding to that tile number is played. Reversed tiles play the same sound reversed. Figure 4 shows the tiles and the target. Sound files play until either they have reached the end of the file or if a touch is applied while playing. This allows the sounds to be paused and moved across the target.

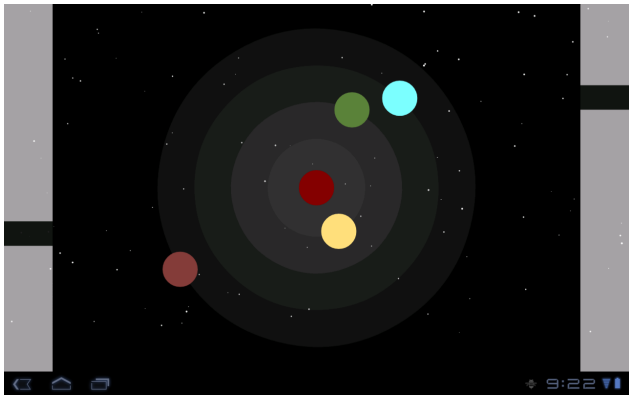


Figure 3: The Orrerator interface with the first and third planets active and playing.

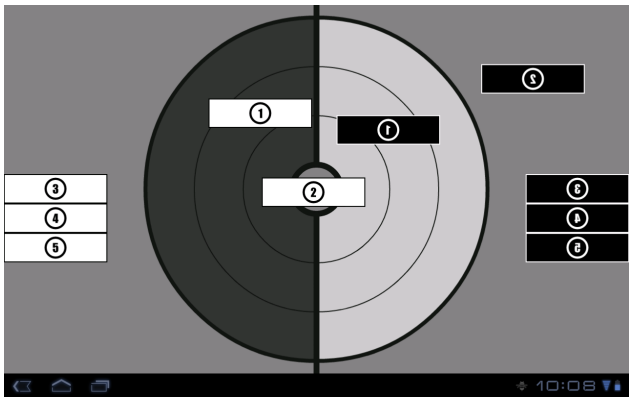


Figure 4: The Distance 2 interface with tile 2 in the center set to play at normal rate.

Tiles placed in the very center of the target are played back at normal playback rate. Tiles in the next circle out play at half that rate. The next circle plays at one-third rate and the outermost circle plays at one-quarter rate. If a tile is paused by touch and moved from one target circle to an-

other, the sound will pause and resume at the new playback rate.

Tiles placed in the circle but on the left of the vertical dividing line play in the left channel while tiles on the right side play in the right channel. Tiles placed in the center (normal rate) play in the center of the stereo field.

The sound engine for this piece is written in Pd with a custom file playback control mechanism designed by the first author. The 5 sound files are all various recorded clips from an interview with experimental composer Toshi Ichiyanagi. *Distance 2*, the piece, is based on a piece by Ichiyanagi called *Distance* [Ichiyanagi, 1961] in which the performer must be at least three meters away from his or her instrument while playing. In a time of computer music in which wireless networking makes this kind of setup trivial, the notion of distance can be explored in other ways. In the context of the piece, distance from the center of the target represents the recognizbility of the recorded clips with the clips getting further away from the original as the tiles get further away from the center.

5 Summary

Bringing the JunctionBox toolkit to the Android operating system allows developers to build sound and music control interfaces on an increasing array of touch devices. With a focus on interface coding rather than providing pre-built widgets, JunctionBox provides developers with an opportunity to create new touch-based interactions with highly customized visuals. Two example interfaces, the Orrerator and Distance2, show some of the creative interfaces that can be built using the toolkit.

6 Acknowledgements

We would like to thank the Alberta College of Art + Design, the Canada Council for the Arts, the Natural Science and Engineering Research Council of Canada, SMART Technologies, the Canadian Foundation for Innovation, and the Alberta Association of Colleges and Technical Institutes for research support. We would also like to thank the members of the Interactions Lab at the University of Calgary for feedback and support during the development of this project.

References

Lawrence Fyfe, Adam Tindale, and Sheelagh Carpendale. 2011. Junctionbox: A toolkit for creating multi-touch sound control interfaces. In *Proceedings of the Conference on New Interfaces for Musical Expression*, pages 276–279.

Google. 2012a. Android. `http://developer.android.com/index.html`.

Google. 2012b. Motionevent. `http://developer.android.com/reference/android/view/MotionEvent.html`.

hexler.net. 2012. Touchosc. `http://hexler.net/software/touchosc`.

Toshi Ichiyanagi. 1961. Distance.

Martin Kaltenbrunner, Till Bovermann, Ross Bencina, and Enrico Costanza. 2005. Tuio - a protocol for table-top tangible user interfaces. In *Proceedings of the 6th International Workshop on Gesture in Human-Computer Interaction and Simulation*.

Uwe Laufs, Christopher Ruff, and Jan Zibuschka. 2010. Mt4j - a cross-platform multi-touch development. In *Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '10, New York, NY, USA. ACM.

Chandrasekhar Ramakrishnan. 2003. Javaosc. `http://www.illposed.com/software/javaoscdoc/`.

Casey Reas and Ben Fry. 2006. Processing: programming for the media arts. *AI & Society*, 20(4):526–538.

Matthew Wright. 2005. Open sound control: an enabling technology for musical networking. *Organised Sound*, 10(3):193–200.

An Introduction to the *Synth-A-Modeler* Compiler: Modular and Open-Source Sound Synthesis using Physical Models

Edgar BERDAHL
Audio Communication Group, TU Berlin
Einsteinufer 17c
10587 Berlin
Germany
eberdahl@mail.tu-berlin.de

Julius O. SMITH III
CCRMA, Stanford University
Stanford, CA
94305
USA
jos@ccrma.stanford.edu

Abstract

The tool is not a synthesizer—it is a Synth-A-Modeler! This paper introduces the Synth-A-Modeler compiler, which enables artists to synthesize binary DSP modules according to mechanical analog model specifications. This open-source tool promotes modular design and ease of use. By leveraging the Faust DSP programming environment, an output Pd, Max/MSP, SuperCollider, VST, LADSPA, or other external module is created, allowing the artist to hear the sound of the physical model in real time using an audio host application. To show how the compiler works, the example model "touch a resonator" is presented.

Keywords

physical modeling, virtual acoustics, Faust DSP, haptic force-feedback, open source

1 Introduction

Simulating the equations of motion of acoustic musical instruments, also known as physical modeling, has been employed for decades to synthesize sound digitally [Smith, 2010; Smith III, 1982; Cadoz et al., 1981]. It is an intriguing paradigm for synthesizing sound in new media applications because it enables sound synthesis for fictional acoustic-like instruments that would be impractical or very labor-intensive to construct in real life.

Physical modeling DSP algorithms incorporate internal feedback loops, which means that an error or inaccuracy in the modeling algorithm can cause the algorithm to become unstable. This results in a loud sound, which can be very unpleasant for an artist working with a physical model.

Several tools have been packaged for implementing modular physical modeling with algorithms that are designed to be stable. The basic concept is that the artist specifies an interconnection of passive mechanical, acoustical, and/or electrical elements, rather than programming any equations. This makes it much easier for artists to employ physical modeling to make new sounds without needing to understand all of the details under the hood.

However, most of these tools have not been immediately available to artists, partly due to the cost of purchasing the tools. For example, the Modalys modal synthesis environment requires both the purchase of Max/MSP and the IRCAM Forum Recherche [Ellis et al., 2005]. GENESIS is a complete modeling package and can be purchased from the Association pour la Création et la Recherche sur les Outils d'Expression [Castagne and Cadoz, 2002]. GENESIS includes a graphical user interface, which makes it easier for artists to specify the interconnection of the mechanical elements.

In contrast, the BlockCompiler by Matti Karjalainen is open-source; however, it is a complex package that has been rewritten at least three times and requires the artist to write Lisp code to create models [Karjalainen, 2003]. Stefan Bilbao has written a modular environment for synthesizing percussion sounds, but it runs in MATLAB and so is not immediately applicable to real-time synthesis [Bilbao, 2009]. Finally, the Synthesis ToolKit (STK) has become popular due to its MIDI capability and ability to run within Max/MSP, Pd, and other sound synthesis environments [Cook and Scavone, 1999].[1] However, new models can only be created in the STK by artists who can manually write stable difference equations in C++ for physical modeling.

For the above reasons, we decided to create a new, free and open-source tool for modular sound

[1] See `http://ccrma.stanford.edu/software/stk`

synthesis using physical models.

2 Synth-A-Modeler

2.1 Requirements

The following requirements of the Synth-A-Modeler project have guided the design. The Synth-A-Modeler compiler should

- enable efficient real-time physical modeling sound synthesis for new media applications,
- be free and open-source,
- be as modular as possible,
- be easy to extend and modify,
- serve as a platform for pedagogical exploration of the physics of mechanically vibrating systems,
- be accessible to artists who may have little or no experience in programming, digital signal processing (DSP), or physics,
- be accessible from as many sound synthesis host environments as possible,
- enable the development of MIDI-based synthesizers, and
- be compatible with programming haptic force-feedback systems.

2.2 Faust

To enable efficient real-time synthesis while targeting as many host environments as possible, we decided to use the Functional AUdio STreaming (Faust) programming language [Orlarey et al., 2009; Barkati et al., 2011; Orlarey et al., 2002]. In fact, some physical models had already been written directly in Faust code [Michon and Smith III, 2011]; however, most artists would not be ready to put in the detailed effort required to program physical models in Faust. Hence, we planned to extend Faust with the development of Synth-A-Modeler.

2.3 Dataflow

Consequently, we adopted the dataflow shown in Figure 1. The Synth-A-Modeler compiler receives a netlist-like model specification in an MDL file and compiles it into Faust DSP code [Vladimirescu, 1994]. Then the Faust compiler together with `g++` can transform the code into a target binary format as suitable for Pd, SuperCollider, Max/MSP, CSound, LADSPA plug-in, VST plug-in, a generic audio application, generic C++ code, or other host target as desired by the artist.

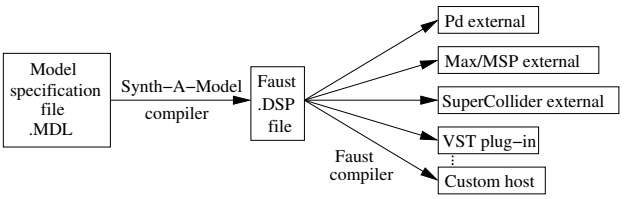


Figure 1: Dataflow for synthesizing a model with Synth-A-Modeler

2.4 Specifying MDL Files

For synthesizing sound with traditional signal flow-based approaches such as Pure Data (pd), Max/MSP, Simulink, or others, a user specifies a directed graph of sound sources, processing elements, and sound outputs. The signal flow is considered to be unidirectional: from the sources to the sinks.

However, in the case of physical modeling, the signal flow is *bidirectional* among the elements. One reason for this is Newton's third law: "For every action, there is an equal and opposite reaction." Hence, in physical modeling, a graph with bidirectional edges describes the signal flow. An engineer must be keenly aware of the bidirectional signal flow between each pair of elements; however, an artist designing a physical model using a modular approach needs only to understand which elements are connected to which. For this reason, the artist can simply specify an *un*directed graph of virtual physical elements [Castagne and Cadoz, 2002]. For the Synth-A-Modeler compiler, the artist specifies the model in an MDL file by specifying connections between elements, such as digital waveguides, masses, springs, dampers, ports, etc., using a netlist-like format with some extensions. Of course the artist may also specify the physical parameters for each element.

3 Modeling Paradigm

3.1 Example Model

Consider the model shown in Figure 2, which implements a very simple synthesizer with only a

single resonance frequency. The model describes a series of mechanical elements that connect to a user's finger, allowing the user to "touch" a virtual mechanical resonator. This model features objects as in GENESIS and CORDIS-ANIMA, except that the units are SI units and English names are employed [Castagne and Cadoz, 2002][Cadoz et al., 1993][Kontogeorgakopoulos and Cadoz, 2007]. The following text specifies the same model using an MDL model specification file:

```
link(4200.0,0.001),ll,m1,g,();
touch(1000.0,0.03,0.0),tt,m1,dev1,();

mass(0.001),m1,();
ground(0.0),g,();
port( ),dev1,();

audioout,a1,m1,1000.0;
```

The external `port` named *dev1* is connected by a `touch` link *tt* to a `mass` *m1* of 0.001 kg. The mass resonates because the linear `link` *ll* connects it to mechanical `ground` *g*, which always remains at the position 0 m. The linear `link` *ll* consists of the parallel combination of a spring with stiffness 4200 N/m and damper with parameter 0.001 N/(m/s). The touch link `tt` (analogous to the *BUT* element in GENESIS and CORDIS-ANIMA) is similar to a linear link, except it only results in a force when one of the objects is pushing "inside" the other one [Kontogeorgakopoulos and Cadoz, 2007].

3.2 Inputs And Outputs

A port models a mechanical connection from within a model to the outside. By default each port brings an external position signal into the model and sends a collocated force signal outside the model. This is why a port can easily be used to control a haptic force-feedback device [Berdahl, 2009].

In addition, the `audioout` object in Synth-A-Modeler provides a simple audio output. When applied to a mass-like object, it outputs position (see Figure 2, right), and when applied to a link-like object, it outputs force.

3.3 Resonator Abstraction

The generic `resonator` object (related to CEL in GENESIS) could have been employed instead of

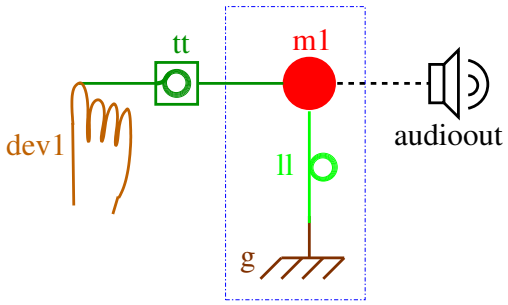


Figure 2: Model for "touch a resonator"

the content within the dash-dotted box in Figure 2. Our resonator implementation in Synth-A-Modeler allows for the frequency and damping time of the resonator to be adjusted in real-time while minimizing transients. Modalys might operate in a similar fashion as it also allows interpolation of resonance frequencies in real time. In Synth-A-Modeler, we use a state-space implementation employing a well-conditioned rotation matrix [Mathews and Smith III, 2003]. Max Mathews employed banks of this style of resonator in his piece *Angel Phasered*, which was performed at the CCRMA Transitions Concert on Sept. 16, 2010.

4 The Synth-A-Modeler Compiler

4.1 Strategy

Although it is possible to represent any explicitly computable linear block diagram in Faust [Orlarey et al., 2002], Faust's block diagram algebra is oriented toward signals that flow from the left to the right. To obtain a signal flowing back from the right to the left, it is necessary to employ Faust's recursive composition operator, which automatically incorporates a single sample of delay. In physical modeling, signals are bidirectional, which means that roughly half of the signals must flow from the right to the left. However, it is challenging to organize Faust networks in this fashion without inserting a plethora of single-sample delays, which can detract from the stability of the models.

For example, consider the Faust code for a mass of m [kg] with zero initial conditions and a linear link but with stiffness k [N/m], damping factor R [N/(m/s)], and spring centering position offset o [m], as given in Figure 3. These equations are similar to those in CORDIS-ANIMA and GENE-

```
mass(m) = (/(m*fs*fs) : ((_,_ : +) ~ _) : ((_,_ : +) ~ _));
link(k,R,o) = _ : (_-o) <: (_,_) : (*(k), (_<:(_,_):(_,mem):-:*(R*fs))) : (_,_):+:_;
```

Figure 3: Faust code for a mass and a linear link from `physicalmodeling.lib`

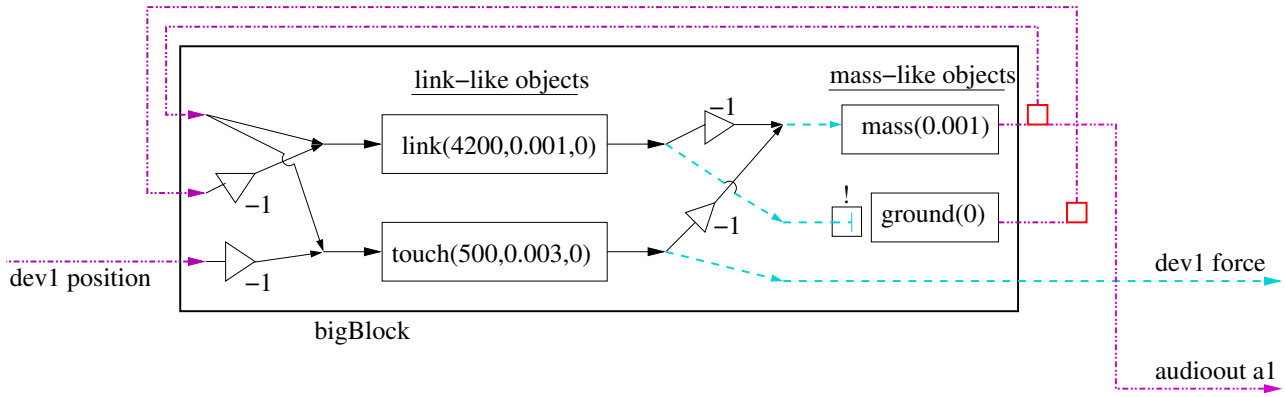


Figure 4: Representation of Faust DSP code for "touch a resonator"

SIS, except that the units are SI units.

Neither the mass nor the link incorporates a single sample of delay; however, in order to wrap these specific elements in feedback about one another, it is necessary to insert samples of delay to make the network explicitly computable. In the CORDIS-ANIMA equations, the delay is included in the masses [Kontogeorgakopoulos and Cadoz, 2007], so we follow suit by having Faust's right to left signals emanate from the masses.

For example, Figure 4 shows the representation of the Faust DSP code for the model depicted in Figure 2. The mass-like objects are placed on the right, and the link-like objects are placed further to the left. The signals representing displacements are shown in magenta, and the signals representing net forces are shown dashed in cyan (see Figure 4—for color see the online version of this paper). Linear combinations are formed in order to properly interconnect the mass-like objects and link-like objects. Finally, the single-sample of delay per feedback loop is represented on the far right using the small red boxes as in the Faust diagram notation. According to this interconnection strategy, there is no sample of delay associated with the link-like objects.

The single port in the model (ref. "dev1" in Figure 2) is responsible for the position signal input (see Figure 4, bottom left dash-dot-dotted in magenta) and the force signal output (see Figure 4, bottom dashed in cyan) could be connected to an impedance controlled haptic force-feedback device in order to control the sound synthesis. Alternatively, using a more conventional kind of new media controller, the position input could be used independently of any force output to control the sound synthesis. Finally, the additional audio output `a1`, which corresponds to the position of the virtual mass, is also provided (see Figure 4, right dash-dot-dotted in magenta) as a more sensible audio output.

4.2 Compiled Code

Applying the Synth-A-Modeler compiler to the model specification MDL file given in Section 3.1 results in the Faust code presented in Figure 5. The code begins by first importing the physical modeling library, which contains Faust code describing how each of the elements works. Then `bigBlock` is defined about which the feedback paths will be wrapped. In this case, `m1` is fed back, which represents the position of mass *m1*, and the ground position `g` is also fed back. The letter "p" is appended to each variable fed back, denoting *previous*, since the variable is delayed by a single sample (see the small red squares on the

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

bigBlock(m1p,gp,dev1p) = (m1,g,dev1,a1) with {
     // Link-like objects:
     ll = (m1p - gp) : link(4200.0,0.001,0.0);
     tt = (m1p - dev1p) : touch(1000.0,0.03,0.0);

     // Mass-like objects:
     m1 = (0.0-ll-tt) : mass(0.001);
     g = (0.0+ll) : ground(0.0);
     dev1 = (0.0+tt);

     // Additional audio output
     a1 = 0.0+m1*(1000.0);
};

process = (bigBlock)~(_,_):(!,!,_,_);
```

Figure 5: Compiled Faust DSP code for implementing "touch a resonator"

right-hand side of Figure 4).

The identifier for each element (e.g. `ll`, `tt`, `m1`, `g`, `dev1`, and `a1`) is then employed as an output variable from the element, which can only be accessed from within `bigBlock`. The inputs to the elements are formed as linear combinations of the other variables (see Figure 5).

4.3 Example Pure Data Patch

Next the Faust compiler together with `g++` can compile the Faust code into a target format as suitable for an audio host application. We briefly present an example of compiling our example into a Pure Data (pd) external object called `touch_a_resonator~`. The pd patch in Figure 6 shows how the external object can be employed for real-time sound synthesis without requiring a haptic force-feedback user input device. The leftmost inlet and outlet are automatically generated by the faust2pd script,[2] and we do not use them in this example. The remaining audio inlets and outlets (see Figure 6) are ordered in correspondence with the input and outputs in Figure 4 and in the third line of Figure 5.

The rightmost inlet corresponds to the input position `dev1`. A horizontal slider GUI object from pd is employed as a user interface. The slider's output is converted into an "audio" signal, smoothed, and fed into the `dev1` position input.

The force outlet from the `dev1` output (see Figure 6) is not needed in this case since there is no mechanism to provide force feedback. The audio output comes from the additional `a1` audio output that was specified in the model. Despite having only a single resonance frequency, we find that the quality of the user interaction with the model is intriguing, due to our process of physically modeling as many elements as possible in the interaction loop being simulated.

5 Conclusion

Due to its modular character, the Synth-A-Modeler compiler enables the creation of complex models for artistic applications using simple building blocks. In some sense, it is an extension

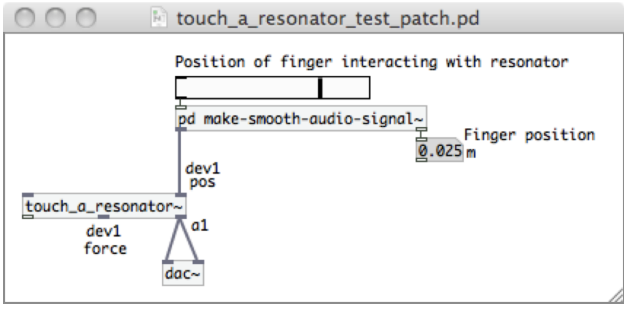


Figure 6: Example pd patch for "touch a resonator"

[2] More information is available on this inlet and outlet [Graef, 2007].

of our own prior work, rewritten to employ Faust for efficient DSP and to target multiple host applications [Berdahl et al., 2010].

We are currently working to add support for digital-waveguide objects to the Synth-A-Modeler compiler. We look forward to an open-source platform for prototyping, developing, and releasing physical modeling binaries that incorporate the popular modeling techniques of mass and link (i.e. mass-interaction) modeling, modal synthesis, and digital waveguide synthesis. The implementation is simple enough that we hope other developers can easily add support for further modeling formalisms.

Due to the open-format licensing of Synth-A-Modeler via the GPL version 2, we believe that Synth-A-Modeler could be especially attractive for commercial applications for compiling optimized and fine-tuned models into portable binary modules. We hope that this way we can create a large user base including industrial, artistic, and scientific users.

6 Acknowledgments

We graciously thank Alexandros Kontogeorgakopoulos, Claude Cadoz, Stefan Weinzierl, Annie Luciani, Andre Bartetzki, Chris Chafe, and the Alexander von Humboldt foundation.

References

K. Barkati, D. Fober, S. Letz, and Y. Orlarey. 2011. Two recent extensions to the FAUST compiler. In *Proc. of the Linux Audio Conference*, Maynooth, Ireland.

Edgar Berdahl, Alexandros Kontogeorgakopoulos, and Dan Overholt. 2010. HSP v2: Haptic signal processing with extensions for physical modeling. In *Proceedings of the Haptic Audio Interaction Design Conference*, pages 61–62, Copenhagen, Denmark.

Edgar Berdahl. 2009. *Applications of Feedback Control to Musical Instrument Design.* Ph.D. thesis, Stanford University, Stanford, CA, USA, December.

Stefan Bilbao. 2009. A modular percussion synthesis environment. In *Proc. 12th Int. Conference on Digital Audio Effects (DAFx-09)*, Como, Italy.

Claude Cadoz, Annie Luciani, and Jean-Loup Florens. 1981. Synthèse musicale par simulation des mécanismes instrumentaux. *Revue d'acouqistique*, 59:279–292.

Claude Cadoz, Annie Luciani, and Jean-Loup Florens. 1993. CORDIS-ANIMA: A modeling and simulation system for sound and image synthesis—The general formalism. *Computer Music Journal*, 17(1):19–29.

Nicolas Castagne and Claude Cadoz. 2002. Creating music by means of 'physical thinking': The musician oriented Genesis environment. In *Proc. 5th Internat'l Conference on Digital Audio Effects*, pages 169–174, Hamburg, Germany.

Perry Cook and Gary Scavone. 1999. The synthesis toolkit (STK), version 2.1. In *Proc. Internat'l Computer Music Conf.*, Beijing, China.

Nicholas Ellis, Joël Bensoam, and René Caussé. 2005. Modalys demonstration. In *Proc. Int. Comp. Music Conf. (ICMC'05)*, pages 101–102, Barcelona, Spain.

Albert Graef. 2007. Interfacing Pure Data with Faust. In *Proc. of the Linux Audio Conference*, Technical University of Berlin, Berlin, Germany.

Matti Karjalainen. 2003. BlockCompiler: Efficient simulation of acoustic and audio systems. In *Proc. 114th Convention of the Audio Engineering Society*, Preprint #5756, Amsterdam, The Netherlands.

Alexandros Kontogeorgakopoulos and Claude Cadoz. 2007. Cordis Anima physical modeling and simulation system analysis. In *Proc. 4th Sound and Music Computing Conference*, pages 275–282, Lefkada, Greece.

Max Mathews and Julius O. Smith III. 2003. Methods for synthesizing very high Q parametrically well behaved two pole filters. In *Proc. Stockholm Musical Acoustic Conference (SMAC)*, Stockholm, Sweden.

Romain Michon and Julius O. Smith III. 2011. Faust-STK: A set of linear and nonlinear physical models for the Faust programming language. In *Proc. 14th Int. Conference on Digital Audio Effects (DAFx-11)*, Paris, France.

Yann Orlarey, Dominique Fober, and Stéphane Letz. 2002. An algebra for block diagram languages. In *Proceedings of International Computer Music Conference*, Göteborg, Sweden.

Yann Orlarey, Dominique Fober, and Stéphane Letz. 2009. *Faust: an Efficient Functional Approach to DSP Programming*. Edition Delatour, France.

J.O. Smith III. 1982. Synthesis of bowed strings. In *Proceedings of the International Computer Music Conference*, Venice, Italy.

Julius O. Smith. 2010. *Physical Audio Signal Processing: For Virtual Musical Instruments and Audio Effects*. W3K Publishing, `http://ccrma.stanford.edu/~jos/pasp/`, December, ISBN 978-0-9745607-2-4.

Andrei Vladimirescu. 1994. *The SPICE Book*. Wiley, Hoboken, NJ.

pd-faust: An integrated environment for running Faust objects in Pd

Albert Gräf
Dept. of Computer Music, Institute of Musicology
Johannes Gutenberg University
55099 Mainz, Germany
Dr.Graef@t-online.de

Abstract

This paper introduces pd-faust, a library for running signal processing modules written in Grame's functional dsp programming language Faust in Miller Puckette's graphical computer music environment Pure Data a.k.a. Pd. pd-faust is based on the author's faust2pd script which generates Pd GUIs from Faust programs and also provides the necessary infrastructure for running Faust dsps in Pd. pd-faust combines this functionality with its own Faust plugin loader which makes it possible to reload Faust dsps while a patch is running. It also adds automatic configuration of MIDI and OSC controller assignments, as well as OSC-based automation features.

Keywords

Faust, Pd, Pure, functional signal processing.

1 Introduction

We assume that the reader is familiar (or has at least heard of) Grame's popular Faust programming language [5], which greatly facilitates the programming of custom audio processing plugins. Faust programs can be compiled to native code for an abundance of different signal processing environments and plugin standards. In particular, Faust has had support for Miller Puckette's Pd [6] for some time now through its puredata.cpp architecture. Pd users in the Faust community have also been using the author's faust2pd script to create Pd GUIs (graphical user interfaces) for Faust externals which seem to work quite well for operating Faust dsps inside Pd [1].

One of faust2pd's shortcomings is that it generates the Pd GUIs outside of Pd. Thus the generated GUI is static, and changing the Faust source of a dsp generally requires regeneration of the GUI and reloading of the hosting Pd patch to make the changes take effect. Another obstacle is that the necessary infrastructure for processing MIDI note input and control changes is implemented entirely in Pd, which may involve a lot of Pd objects and become a cpu hog for complex Faust programs.

pd-faust was designed to overcome these limitations. Most notably, it loads Faust dsps dynamically and allows them to be reloaded at any time, in which case the Pd GUI is regenerated automatically and instantly. Thus you can now just edit and recompile the Faust source and have pd-faust pick up the changes on the fly, while the Pd patch keeps running.

pd-faust is implemented as a library of Pd objects written in the author's Pure programming language [2], which is compiled to native code, so that Faust dsps involving a lot of different controls are handled in an efficient manner. Another advantage of using Pure is that at present it is the *only* programming language with a built-in Faust interface based on the LLVM toolkit (which Pure itself uses as its code generation backend). This enables pd-faust to directly load compiled Faust programs in LLVM bitcode format [3] if you're running a suitable version of the Faust compiler [4].

pd-faust offers a number of other enhancements facilitating the use of Faust dsps in Pd. While it is based on the GUI generation code of the faust2pd script and thus supports all of faust2pd's global GUI layout options, it also provides various options to adjust the layout of individual control items. In addition, pd-faust recognizes the `midi` and `osc` controller attributes in the Faust source and automatically provides corresponding MIDI and OSC controller mappings. OSC-based controller automation is also avail-

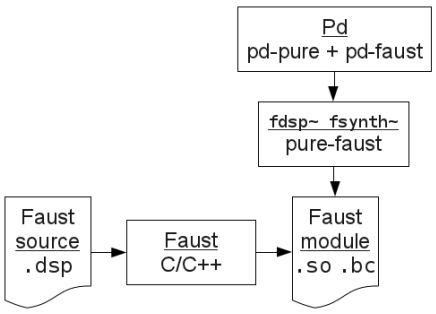


Figure 1: An overview of the pd-faust system.

able. These additional features are all described in some detail in this paper.

Section 2 starts out with a brief overview of pd-faust. In Sections 3 and 4 we then take a more in-depth look at the pd-faust objects and describe how pd-faust constructs the Pd GUI at runtime. Section 5 briefly discusses how to operate the resulting Pd patches. Sections 6 and 7 show how pd-faust uses metadata in Faust programs to define controller mappings and adjust the GUI layout. Section 8 briefly touches on the auxiliary facilities provided to ease livecoding with Faust dsps. Section 9 concludes with pointers to additional information and examples, and discusses possible directions for further work.

The paper assumes some working knowledge of Faust and Pd; please consult [5] and [6] if necessary.

2 Overview

Before we go into the technical details, let us first give a brief overview of pd-faust and the various software components which are involved in the system (cf. Fig. 1).

pd-faust is implemented as an object library for Pd. Since the pd-faust objects are written in Pure, you'll also need the pd-pure plugin loader [2] which provides the necessary infrastructure to run Pure objects in Pd. Both libraries are available in the form of shared modules which are loaded by Pd on startup, using either Pd's `-lib` option or a corresponding entry in Pd's startup preferences.

The main ingredients of pd-faust are the `fdsp~` and `fsynth~` objects which are used to load and run different kinds of Faust dsps inside Pd (the differences between these will be explained in the following section). The basic functionality to do this is actually provided by another Pure module, pure-faust, which can load Faust modules in one of two formats:

- *Native* modules are shared modules in the format supported by the host operating system (`.so` on ELF systems, `.dll` on Windows, etc.). These are created from Faust source programs by invoking the Faust compiler with the pure.cpp architecture file (`faust -a pure.cpp`) and compiling the resulting C++ source to a shared module.
- *Bitcode* modules are modules in LLVM bitcode format which can be created directly by Faust (`faust -lang llvm`). You'll need Faust2, the development version of Faust, to make this work [4]. The Pure interpreter has a built-in LLVM bitcode linker which enables it to load these modules [3]; the executable code of the module is then generated on the fly when the module is loaded.

The internal workings of pd-faust are illustrated in Fig. 2. The `fdsp~` and `fsynth~` objects use the operations provided by the Faust module to instantiate a Faust dsp and extract the needed information from the dsp in order to construct its Pd GUI. The GUI is then inserted into the hosting Pd patch by means of Pd's "FUDI" protocol.[1] All this happens automatically whenever the Pd patch is loaded or a Faust module gets reloaded by sending it a corresponding control message.

While the patch is running, an `fdsp~` or `fsynth~` object receives incoming control messages and audio data through its inlets and invokes the operations of the dsp to change the control values and compute blocks of audio data as they are requested by Pd's audio loop. The generated data is then output through the object's audio and control outlets.

3 The fdsp and fsynth objects

Working with pd-faust basically involves adding a bunch of `fsynth~` and `fdsp~` objects to a Pd patch along with the corresponding GUI subpatches, and wiring up the Faust units in some

[1] See `http://wiki.puredata.info/en/FUDI`. This protocol is also used internally by Pd to represent the contents of patches and communicate with its GUI process.

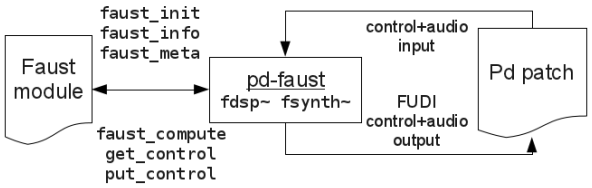


Figure 2: Internals of the pd-faust system.

variation of a synth-effects chain which typically takes input from Pd's MIDI interface (`notein`, `ctlin`, etc.) and outputs the signals produced by the Faust units to Pd's audio interface (`dac~`).

For convenience, pd-faust also includes the `midiseq` and `oscseq` objects as well as a corresponding `midiosc` abstraction which can be used to handle MIDI input and playback as well as OSC controller automation. These objects are described in more detail in Section 5.

The `fdsp~` object is invoked as follows:

```
fdsp~ dspname instname channel
```

- `dspname` denotes the name of the Faust dsp (usually this is just the name of the .dsp file with the extension stripped off). Please note that, as already mentioned, the Faust dsp must be provided in a form which can be loaded in *Pure* (not Pd!), so the pure.cpp architecture included in recent Faust versions must be used to compile the dsp to a shared library. (If you're already running Faust2, you can also compile to an LLVM bitcode file instead; Pure has built-in support for loading these.) The Makefiles included in the pd-faust distribution show how to do this.
- `instname` denotes the name of the instance of the Faust unit. Multiple instances of the same Faust dsp can be used in a Pd patch, which must all have different instance names. In addition, the instance name is also used to identify the GUI subpatch of the unit (see below) and to generate unique OSC addresses for the unit's control elements.
- `channel` is the number of the MIDI channel the unit responds to. This can be 1..16, or 0 to specify "omni" operation (listen to MIDI messages on all channels).

The `fdsp~` object requires a Faust dsp which can work as an effect unit, processing audio input and producing audio output.

The `fsynth~` object works in a similar fashion, but has an additional creation argument specifying the desired number of voices:

```
fsynth~ dspname instname channel nvoices
```

The `fsynth~` object requires a Faust dsp which can work as a monophonic synthesizer (having zero audio inputs and a nonzero number of audio outputs). To these ends, pd-faust assumes that the Faust unit provides three so-called "voice controls" which indicate which note to play:

- `freq` is the fundamental frequency of the note in Hz.
- `gain` is the velocity of the note, as a normalized value between 0 and 1. This usually controls the volume of the output signal.
- `gate` indicates whether a note is currently playing. This value is either 0 (no note to play) or 1 (play a note), and usually triggers the envelop function (ADSR or similar).

pd-faust doesn't care at which path inside the Faust dsp these controls are located, but for the synthesizer to function properly they must all be there, and the basenames of the controls must be unique throughout the entire dsp.

Like faust2pd, pd-faust implements the necessary logic to drive the given number of voices of an `fsynth~` object. That is, it will actually create a separate instance of the Faust dsp for each voice and handle polyphony by allocating voices from this pool in a round-robin fashion, performing the usual voice stealing if the number of simultaneous notes to play exceeds the number of voices.

The `fdsp~` and `fsynth~` objects respond to the following messages:

- `bang` outputs the current control settings on the control outlet in (symbolic) OSC format.[2]

[2] pd-faust represents OSC messages as ordinary Pd messages with the OSC address in the selector symbol of the message. Input and output of binary OSC messages is assumed to be handled by a separate OSC library which is not

- `write` outputs the current control settings to external MIDI and/or OSC devices. This message can also be invoked with a numeric argument to toggle the "write mode" of the unit; please see Section 6 for details.
- `reload` reloads the Faust unit. This also reloads the shared library or bitcode file if the unit was recompiled since the object was last loaded.
- `addr value` changes the control indicated by the OSC address `addr`. This is also used internally for communication with the Pd GUI and for controller automation.

In addition, the `fdsp~` and `fsynth~` objects respond to MIDI controller messages of the form `ctl val num chan`, and the `fsynth~` object also understands note-related messages of the form `note num vel chan` (note on/off) and `bend val chan` (pitch bend). In either case, pd-faust provides the necessary logic to map controller and note-related messages to the corresponding control changes in the Faust unit.

4 GUI subpatches

For each `fdsp~` and `fsynth~` object, the Pd patch should also contain an (initially empty) "one-off" graph-on-parent subpatch with the same name as the instance name of the Faust unit:

```
pd instname
```

You shouldn't insert anything into this subpatch, its contents (a bunch of Pd GUI elements corresponding to the control elements of the Faust unit) will be generated automatically by pd-faust when the corresponding `fdsp~` or `fsynth~` object is created, and whenever the unit gets reloaded at runtime. See Fig. 3 for an example.

As with faust2pd, the GUI layout follows the hierarchical structure of the controls in the Faust program which places controls in different *control groups*, please check the Faust documentation for details. The default appearance of the GUI can also be adjusted in various ways; see Section 7 for details.

part of pd-faust. E.g., one might use Martin Peach's collection of OSC objects for that purpose, see `http://puredata.info/Members/martinrp/OSCobjects`.

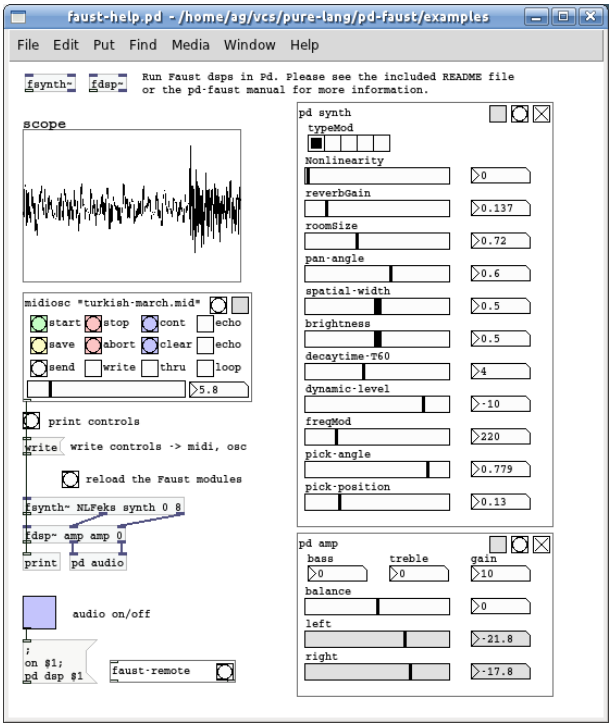


Figure 3: Sample pd-faust patch.

The relative order in which you insert an `fdsp~` or `fsynth~` object and its GUI subpatch into the main patch matters. Normally, the GUI subpatch should be inserted *first*, so that it will be updated automatically when its associated Faust unit is first created, and also when the main patch is saved and then reloaded later.

However, in some situations it may be preferable to insert the GUI subpatch *after* its associated Faust unit. If you do this, the GUI will *not* be updated automatically when the main patch is loaded, so you'll have to reload the dsp manually (sending it a `reload` message) to force an update of the GUI subpatch. This is useful, in particular, if you'd like to edit the GUI patch manually after it has been generated.

In some cases it may even be desirable to completely "lock down" the GUI subpatch. This can be done by simply *renaming* the GUI subpatch after it has been generated. When Pd saves the main patch, it saves the current status of the GUI subpatches along with it, so that the renamed subpatch will remain static and will *never* be updated, even if its associated Faust unit gets reloaded. This generally makes sense only if the

Figure 4: Close-up view of the `midiosc` abstraction.

control interface of the Faust unit isn't changed after locking down its GUI patch. To "unlock" a GUI subpatch, you just rename it back to its original name.

5 Operating the patches

The generated Pd GUI elements for the Faust dsps are pretty much the same as with faust2pd. See Fig. 3, which shows the `midiosc` abstraction and the actual Faust objects on the left, and the corresponding GUI subpatches on the right. The only obvious change is the addition of a "record" button (the gray toggle on the left of the button row located in the upper right corner of each GUI) which enables recording of OSC automation data (described below).

The `midiosc` abstraction (cf. Fig. 4) shown in the examples serves as a little sequencer applet that enables you to control MIDI playback and OSC recording. The creation arguments of `midiosc` are the names of the MIDI and OSC files. If the second argument is omitted, it defaults to the name of the MIDI file with new extension .osc. You can also omit both arguments if neither MIDI file playback nor saving recorded OSC data is required.

The abstraction has a single control outlet through which it feeds the generated MIDI data and other messages to the connected `fsynth~` and `fdsp~` objects. Live MIDI input is also accepted and forwarded to the control outlet, after being translated to the format understood by `fsynth~` and `fdsp~` objects. Moreover, you can also control `midiosc` with an external sequencer program, employing MIDI Machine Control (MMC) for synchronization.

At the bottom of the abstraction there is a little progress bar along with a time display which indicates the current song position. If playback is stopped, you can also use these to change the start position for playback and recording.

Here is a brief rundown of the available controls:

- The gray "record" toggle in the upper right corner of the abstraction enables recording of OSC controller automation data. Note that this toggle merely *arms* the OSC recorder; you still have to actually start the recording with the `start` button. However, you can also first start playback with `start` and then switch recording on and off as needed at any point in the sequence. Pushing the `stop` button then stores the recorded sequence for later playback. Also note that before you can start recording any OSC data, you first have to arm the Faust units that you want to record. This is done with the "record" toggle in the Pd GUI of each unit.
- The "bang" control next to the "record" toggle lets you record a snapshot of the current controller settings. This is also done automatically when you start recording a new sequence.
- The `start`, `stop` and `cont` controls in the first row of control elements start, stop and continue MIDI and OSC playback, respectively. The `echo` toggle in this row causes played MIDI events to be printed in the Pd main window.
- There are some additional controls related to OSC recording in the second row: `save` saves the currently recorded data in an OSC file for later use; `abort` is like `stop` in that it stops recording and playback, but also throws away the data recorded in this take (rather than keeping it for later playback); `clear` purges the entire recorded OSC sequence so that you can start a new one; and the `echo` toggle, if enabled, prints the OSC messages as they are played back.
- The controls in the third row provide some additional ways to configure the playback process. The `loop` button can be used to enable looping, which repeats the playback of the MIDI (and OSC) sequence ad infinitum. The `thru` button, when switched on, routes the MIDI data during playback through Pd's

MIDI output so that it can be used to drive an external MIDI device in addition to the Faust instruments. The `write` button does the same with MIDI and OSC controller data generated either through automation data or by manually operating the control elements in the Pd GUI, see Section 6 for details. The `send` button recalls the recorded OSC parameter settings at a given point in the sequence, and updates the GUI controls accordingly.

Once some automation data has been recorded, it will be played back along with the MIDI file. You can then just listen to it, or go on to record more automation data as needed. If you save the automation data with the `save` button, it will be reloaded from its OSC file next time the patch is opened.

OSC sequences are saved in a simple ASCII format which should be self-explanatory and can be edited with any text editor. For instance:

```
# written by oscseq Sun Dec 11 19:39:45 2011

# delta  /oscaddr value
0        /synth:Nonlinear-Filter/typeMod 0
0        /synth:Nonlinearity 0
0        /synth:Reverb/reverbGain 0.137
0        /synth:Reverb/roomSize 0.72
```

The OSC addresses are generated automatically from the hierarchical structure of the controls in the Faust program, using the instance name as well as the labels of control groups and elements to assign a unique pathname to each control of each Faust unit in the patch.

Note that `midiosc` is merely an example which should cover most common uses and can be customized for the target application as needed. You may even do without it and have your patches feed control messages directly into `fdsp~` and `fsynth~` objects instead. Internally, `midiosc` uses two utility objects `midiseq` and `oscseq` written in Pure and contained in the pd-faust object library. Together these implement most of the MIDI playback and OSC recording functionality; `midiosc` basically just adds the necessary wiring with Pd's MIDI I/O and a few control elements. The `midiseq` and `oscseq` objects can also be used directly in your patch if you prefer to forego `midiosc`, or you may replace them altogether with your own sequencer objects.

6 External MIDI and OSC controllers

The `fsynth~` object has built-in (and hard-wired) support for MIDI notes, pitch bend and MIDI controller 123 (all notes off).

Other controller data received from external MIDI and OSC devices is interpreted according to the controller mappings defined in the Faust source (this is explained below), by updating the corresponding GUI elements and the control variables of the Faust dsp. For obvious reasons, this only works with *active* Faust controls.[3]

An `fdsp~` or `fsynth~` object can also be put in *write mode* by feeding a message of the form `write 1` into its control inlet (the `write 0` message disables write mode again). For convenience, the `write` toggle in the `midiosc` abstraction allows you to do this simultaneously for all Faust units connected to `midiosc`'s control outlet.

When an object is in write mode, it also *outputs* MIDI and OSC controller data in response to both automation data and the manual operation of the Pd GUI elements according to the controller mappings defined in the Faust source, so that it can drive an external device such as a MIDI fader box or a multitouch OSC controller. Note that this works with both *active* and *passive* Faust controls.

To configure MIDI controller assignments, the labels of the Faust control elements have to be marked up with the special `midi` attribute in the Faust source. For instance, a pan control (MIDI controller 10) may be implemented in the Faust source as follows:

```
pan = hslider(
  "pan[midi:ctrl␣10]",0,-1,1,0.01);
```

pd-faust will then provide the necessary logic to handle MIDI input from controller 10 by changing the pan control in the Faust unit accordingly, mapping the controller values 0..127 to the range and step size given in the Faust source. Moreover, in write mode corresponding MIDI controller data will be generated and sent

[3] Faust distinguishes between *active* controls which take input from the user and *passive* controls displaying control data computed in the Faust program, such as RMS envelopes.

to Pd's MIDI output, on the MIDI channel specified in the creation arguments of the Faust unit (0 meaning "omni", i.e., output on all MIDI channels).

The same functionality is also available for external OSC devices, employing explicit OSC controller assignments in the Faust source by means of the `osc` attribute. E.g., the following enables input and output of OSC messages for the OSC `/pan` address:

```
pan = hslider("pan[osc:/pan]",0,-1,1,0.01);
```

Note that in contrast to some architectures included in the Faust distribution, pd-faust only allows literal OSC addresses here. That is, glob-style OSC patterns are not supported as values for the `osc` attribute. Also note that pd-faust currently does not include any facilities for actual OSC input/output, but it's easy to add this to `midiosc` if needed.[4]

7 Tweaking the GUI layout

As already mentioned, pd-faust provides the same global GUI layout options as faust2pd. There are a few minor changes in the meaning of some of the options, though. Here is a brief rundown of the available options, as they are implemented in pd-faust:

- `width=wd`, `height=ht`: Specify the maximum horizontal and/or vertical dimensions of the layout area. If one or both of these values are nonzero, pd-faust will try to make the GUI fit within this area.
- `font-size=sz`: Specify the font size (default is 10).
- `fake-buttons`: Render `button` controls as Pd toggles rather than bangs.
- `radio-sliders=max`: Render sliders with up to `max` different values as Pd radio controls rather than Pd sliders. Note that in pd-faust this option not only applies to sliders, but also to numeric entries, i.e., `nentry` in the Faust source. However, as with faust2pd's `radio-sliders` option, the option is only applicable if the control is zero-based and has a stepsize of 1.
- `slider-nums`: Add a number box to each slider control. Note that in pd-faust this is actually the default, which can be disabled with the `no-slider-nums` option.
- `exclude=pat,...`: Exclude the controls whose labels match the given glob patterns from the Pd GUI.

In pd-faust there is no way to specify the above options on the command line, so you'll have to put them as `pd` attributes on the *main group* of your Faust program, as described in the faust2pd documentation. For instance:

```
process = vgroup(
  "[pd:no-slider-nums][pd:font-size=12]",
  ...);
```

In addition, the following options can be used to change the appearance of individual control items. If present, these options override the corresponding defaults. (Each option can also be prefixed with "`no-`" to negate the option value. Thus, e.g., `no-hidden` makes items visible which would otherwise, by means of the global `exclude` option, be removed from the GUI.)

- `hidden`: Hides the corresponding control in the Pd GUI. This is the only option which can also be used for control groups, in which case *all* controls in the group will become invisible in the Pd GUI.
- `fake-button`, `radio-slider`, `slider-num`: These have the same meaning as the corresponding global options, but apply to individual control items.

These options are specified with the `pd` attribute in the label of the corresponding Faust control or control group. For instance, the following Faust code hides the controls in the `aux` group, removes the number entry from the `pan` control, and renders the `preset` item as a Pd radio control:

```
aux = vgroup("aux[pd:hidden]", aux_part);
pan = hslider(
  "pan[pd:no-slider-num]",0,-1,1,0.01);
preset = nentry(
  "preset[pd:radio-slider]",0,0,7,1);
```

[4] "Vanilla" Pd doesn't provide any objects for OSC I/O, but various suitable externals are readily available, such as Martin Peach's OSC objects already mentioned above. The pd-faust distribution includes a variation of the `midiosc` abstraction which implements this.

8 Remote control

Also included in the sources is a helper abstraction faust-remote.pd and an accompanying elisp program faust-remote.el. These work pretty much like pure-remote.pd and pure-remote.el in the pd-pure distribution, but are tailored for the remote control of Faust dsps in a Pd patch. In particular, they enable you to quickly reload the Faust dsps in Pd using a simple keyboard command (`C-C C-X` by default) from Emacs. The faust-remote.el program was designed to be used with Juan Romero's Emacs Faust mode.[5]

9 Conclusion

We hopefully convinced the reader that pd-faust is an interesting solution for developing, testing, deploying and running Faust dsps in the graphical Pd environment. It provides Pd and Faust users with a fairly complete integrated development environment for Faust, and supports a free-wheeling interactive and experimental development style which should appeal to both developers and artists.

The software described in this paper is available freely under the LGPL[6] from the Pure website.[7] As already mentioned, pd-faust is written in Pure; thus, to build and run the software you'll also need an installation of the Pure interpreter and a couple of Pure addon packages including the latest release of pd-pure [2] which is needed to run Pure externals in Pd; please check the pd-faust documentation for details.

The package also contains a few example patches and accompanying files which illustrate how this all works. Here are some of the examples that you might want to take a look at:

- test.pd: very basic example running a single Faust instrument
- synth.pd: slightly more elaborate patch featuring a synth-effects chain
- bouree.pd: full-featured example running various instruments

pd-faust development continues, so you might want to check out the latest development sources in the Pure source code repository. There are some technical issues and limitations in the current pd-faust version which will hopefully be ironed out in the future. Most notably:

- In the present implementation, pure-faust supports Faust modules either in native or in LLVM bitcode form, but not both at the same time. Therefore there are two corresponding versions of the pd-faust object library, and you'll have to decide in advance whether you'd like to work with native or bitcode modules.
- The names of the `fsynth~` voice controls (`freq`, `gain`, `gate`) are currently hardcoded, and there must be exactly one instance of each of these controls in the dsp, otherwise `fsynth~` may not function as advertized.
- Passive Faust controls are only supported in `fdsp~` objects right now.
- The OSC recording capabilities are somewhat limited in the current version. In particular, it should be possible to erase and edit already recorded controller data while recording. At present you can only change existing automation data by purging the entire sequence and starting over, or by editing the OSC file. This is sufficient for testing purposes but not adequate for real musical work.

Further research

Faust's abstract GUI model seems perfectly appropriate for the type of control processing applications that are typically written using pd-pure. pd-faust is just one of the many possible pd-pure applications which might benefit from this approach. Therefore an interesting question for further research is how to integrate the corresponding pd-faust functionality into pd-pure and make it available to *all* pd-pure applications.

Also, it might be useful to port pd-faust to other realtime environments, such as SuperCollider, and suitable plugin environments, such as LV2 and VST. In principle it's also conceivable to run a suitably modified version of pd-faust directly as a Jack client, without the extra Pd layer. Of course, the details of integrating Faust dsps with these host environments differ considerably

[5] `https://github.com/rukano/emacs-faust-mode`

[6] `http://www.gnu.org/copyleft/lgpl.html`

[7] `http://pure-lang.googlecode.com`

from the current implementation running inside Pd, so porting the interface will be a substantial amount of work.

Acknowledgements

Special thanks are due to Julius O. Smith from CCRMA for following this project from its inception, taking the time to test early alpha versions and suggesting improvements. I'd also like to thank Johannes Zmölnig from the IEM Graz for pointing me to Pd's internal FUDI protocol which made this project feasible in the first place.

References

[1] A. Gräf. Interfacing Pure Data with Faust. In *Proceedings of the 5th International Linux Audio Conference*, pages 24–32, Berlin, 2007. TU Berlin.

[2] A. Gräf. Signal processing in the Pure programming language. In *Proceedings of the 7th International Linux Audio Conference*, Parma, 2009. Casa della Musica.

[3] A. Gräf. An LLVM bitcode interface between Pure and Faust. In *Proceedings of the 9th International Linux Audio Conference*, Maynooth, 2011. NUI Maynooth, Dept. of Music.

[4] S. Letz. LLVM backend for Faust. `http://www.grame.fr/~letz/faust_llvm.html`, 2012.

[5] Y. Orlarey, D. Fober, and S. Letz. FAUST : an efficient functional approach to DSP programming. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*. Editions Delatour France, 2009.

[6] M. Puckette. *The Theory and Techniques of Electronic Music*. World Scientific Publishing Co., 2007.

The Faust Online Compiler: a Web-Based IDE for the Faust Programming Language

Romain MICHON and **Yann ORLAREY**
GRAME, Centre national de création musicale
9 rue du Garet
69202 Lyon,
France,
rmnichon@gmail.com and orlarey@grame.fr

Abstract

The FAUST ONLINE COMPILER is a PHP/JavaScript based web application that provides a cross-platform and cross-processor programming environment for the FAUST language. It allows to use most of FAUST features directly in a web browser and it integrates an editable catalog of examples making it a platform to easily share and use FAUST objects.

Keywords

Faust, Web Application, Digital Signal Processing.

1 Introduction

FAUST [Orlarey et al., 2002] is a functional programming language specifically designed for real-time signal processing and synthesis. Thanks to specific architecture files, a single FAUST program can be used to produce code for a variety of platforms and plug-in formats. These architecture files act as wrappers and describe the interactions with the host audio and GUI system.

The FAUST ONLINE COMPILER is a PHP/-JavaScript based web application that provides a cross-platform and cross-processor programming environment for the FAUST language. It makes it possible to use most of FAUST features directly in a web browser. In many ways, we think it is solving one of FAUST's weakness: the great number of dependencies that using its architecture files involved. Indeed, even though FAUST doesn't need any specific library to be installed on a system to use it, some architecture files prove to be quite complicated to use because of their dependencies. It is also solving the system architecture issues that some Windows users, as an example, might have encountered in the past.

The FAUST ONLINE COMPILER integrates an editable catalog of examples making it a platform to easily share and use FAUST objects. We also think it is a pedagogical tool where it is possible to see in the most interactive way a digital signal processing statement, its C++ equivalent, its diagram representation, its documentation and finally its results.

It has been implemented in the frame of the development of the new FAUST website[1] and it can be easily embedded in any HTML web page or installed on an Apache server.

2 Interface

The FAUST ONLINE COMPILER is a web application made of one fixed size block. Its size has been defined to fit in the new Faust website but it can be easily embedded and scaled to be adapted to any `div` of an `HTML` page using the `iframe` mechanism (see 4). A *full page* mode is also available to provide a more comfortable workspace.

The navigation between the states of the FAUST object is made through different tabs that can be selected in a navigation bar (see frame 1 in figure 1).

2.1 Editing the code

The FAUST code in the FAUST ONLINE COMPILER can be edited using two different ways. The most obvious one is by using the code editor that can be found in the *Faust Code* tab. The code editor `CodeMirror`[2] is used to carry out this task. It has been chosen because its compatibility with a great number of web browsers (Firefox, Safari, Opera, IE, Google Chrome, etc.). It uses a syntax highlighting specific to the FAUST programming language.

Thanks to an `AJAX` script the FAUST ONLINE COMPILER also allows to drop any `.dsp` file con-

[1] `http://faust.grame.fr/`
[2] `http://codemirror.net/`

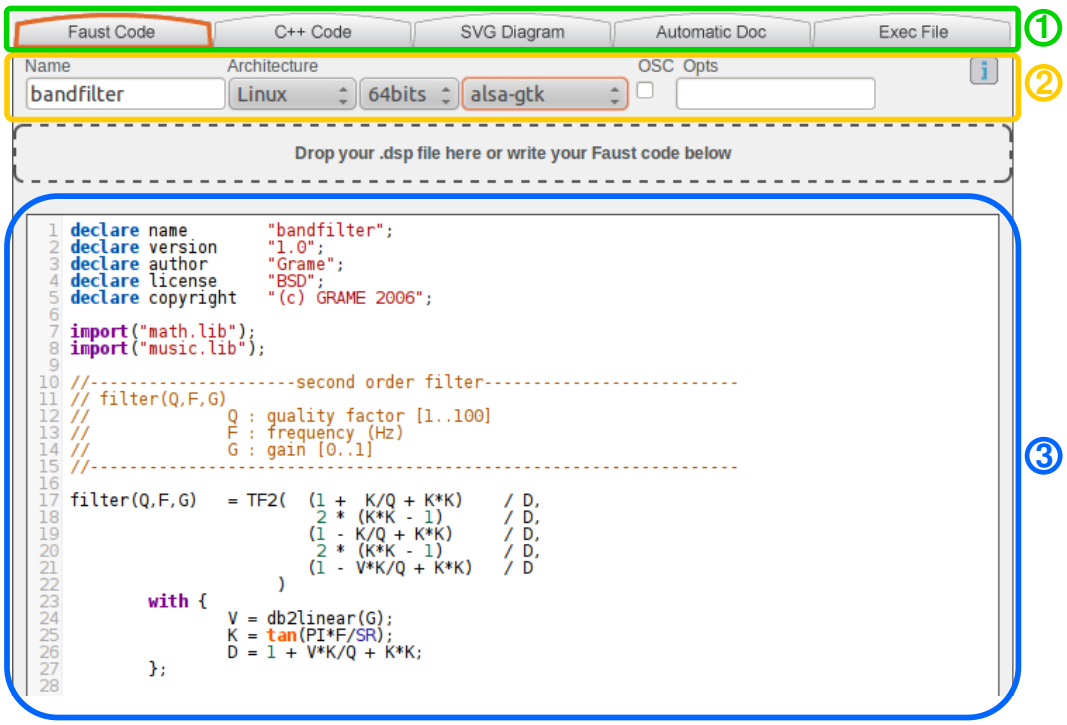


Figure 1: Overview of the compiler zone of the FAUST ONLINE COMPILER. Frame 1 contains the navigation bar, frame 2 the compilation options and frame 3 the FAUST code editor or the different kind of results of the compilation.

taining a FAUST program in the file drop area that is located just above the code editor (see figure 1). This drop area is present in every tabs. This allows to compile a FAUST code into many different elements in a very interactive way and very quickly.

2.2 Compilation

2.2.1 Compilation Options

The compilation options menu can be found just under the navigation bar (see frame 2 in figure 1). It allows to set the kind of executable file that will be generated during the `C++` compilation. Most of the FAUST architectures can be used and the `OSC` support option [Fober et al., 2011] can be selected if it is allowed by the architecture. Further options can be given to the Faust compiler by filling the `Opts` text area.

The 64 bits Linux server (Ubuntu) that hosts the FAUST ONLINE COMPILER at GRAME makes it possible to compile the `C++` code generated by the FAUST compiler in 32 or 64 bits for the following platforms (in function of the selected architecture):

- Linux
- Windows using the `MinGW`[3] cross compiler
- OSX 10.6 using the distant command execution system of `SSH`[4]

If any of these options is changed in the `C++ code` or the `Exec File` tabs, the result will be automatically updated.

2.2.2 Compilation

In order to carry out the compilation, a temporary folder containing the FAUST file and a Makefile generated in function of the options given by the user and the architecture he selected is created on the server. It will be used by the FAUST ONLINE COMPILER to generate the `C++` code, the `SVG` diagram, the documentation, the executable file and the different downloadable packages.

[3] `http://www.mingw.org/`
[4] `http://www.openssh.com/`

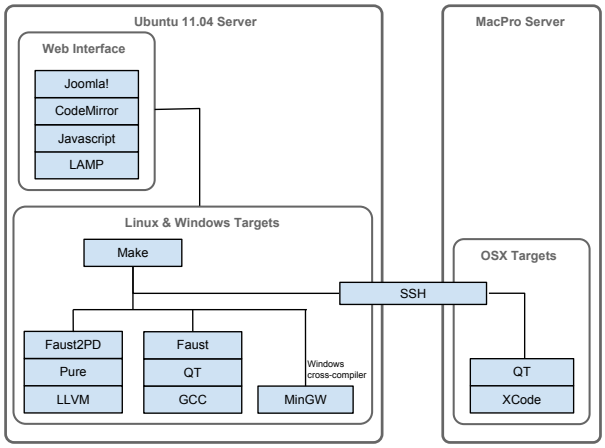


Figure 2: Diagram of the server components.

2.3 Getting the C++ Code

If the `C++` tab is selected, the FAUST compiler is called on the server by the Makefile that will also use `Highlight`[5] to color the generated `C++` code.

If the FAUST code given by the user contains some errors, the message returned by the FAUST compiler will be displayed instead of the `C++` code. The FAUST ONLINE COMPILER will react the same way in the case of the tabs `SVG diagram` and `Automatic Doc`.

2.4 Displaying the SVG diagram of the Faust Code

When the `SVG Diagram` button is clicked, in the same way than for the `C++` code, the Makefile created on the server call the FAUST compiler to generate a `SVG` block diagram of the FAUST code. The result is then displayed just under the file dropping area (see figure 3) and can be browsed by clicking on the different boxes.

2.5 Displaying the Mathematical Documentation of the Faust Object

Some recent developments made at GRAME in the frame of the ASTREE project (see *Acknowledgments*) on the FAUST compiler make it possible to generate automatically an all-comprehensive mathematical documentation of a FAUST program under the form of a complete set of LaTeX formulas and diagrams [Barkati et al., 2011]. The goals of such a self mathematical documentation are:

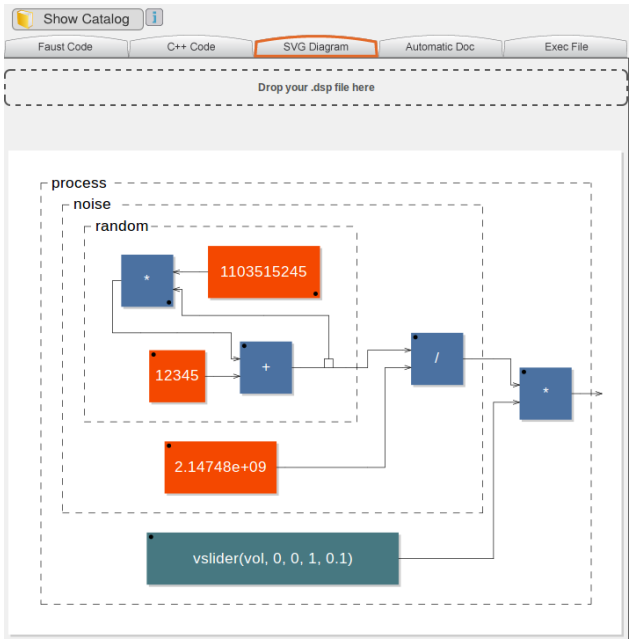


Figure 3: Screenshot of the `SVG` Diagram tab.

1. *Preservation*, i.e. to preserve signal processors, independently from any computer language but only under a mathematical form;
2. *Validation*, i.e. to bring some help for debugging tasks, by showing the formulas as they are really computed after the compilation stage;
3. *Teaching*, i.e. to give a new teaching support, as a bridge between code and formulas for signal processing;
4. *Publishing*, i.e. to output publishing material, by preparing LaTeX formulas and SVG block diagrams easy to include in a paper.

The FAUST ONLINE COMPILER is compatible with this function and can display the resulting pdf file using `Google Document Viewer`[6]. The Makefile described in 2.3 is also used here to carry out the compilation process.

A screen shot of the `Automatic Doc` tab can be seen in figure 4.

[5] `http://www.andre-simon.de/doku/highlight/en/highlight.html`

[6] `https://docs.google.com/viewer/`

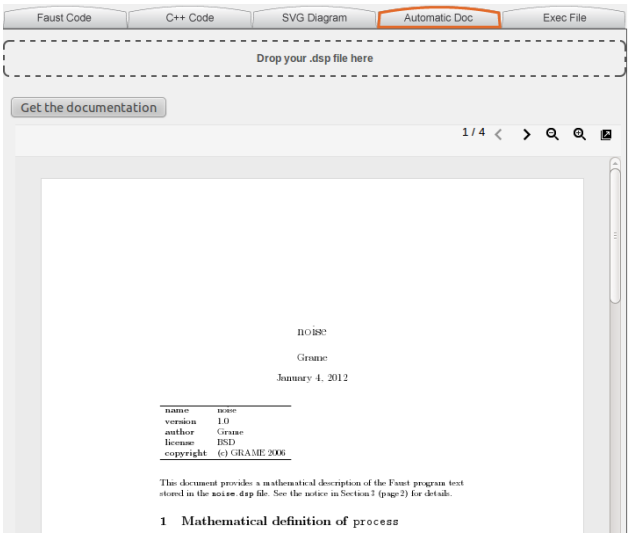


Figure 4: Screenshot of the `Automatic Doc` tab.

2.6 Executable File

When the `Exec File` button is clicked in the navigation bar, the `C++` compilation is carried out in function of the platform and the processor architecture selected. If this task is successful, a download button appear. Otherwise the error message returned by the `C++` compiler is displayed. The downloadable file will be either an executable file either a package containing several files, in function of the FAUST architecture.

3 Catalog of Examples

The FAUST ONLINE COMPILER has been upgraded with an interactive platform to easily share and use pieces of FAUST code: the catalog of examples. It is made of two elements: the catalog itself and the example saver.

3.1 The Catalog

The catalog has the form of a file browser where each FAUST code is sorted into different categories (see figure 6):

- Effects
- Faust-STK [Michon and Smith, 2011]
- Synthesizers
- Tools

A set of "user" categories editable by the users were also created.

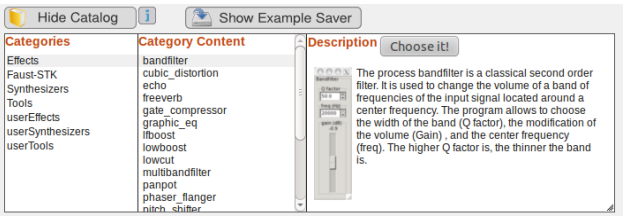


Figure 6: The catalog of examples of the FAUST ONLINE COMPILER.

When a FAUST object is selected, a short description and a screenshot of it are displayed in the `Description` column. The object can then be used in the online compiler either by double-clicking on it either by pushing the `Choose it!` button.

The catalog of examples can be displayed in every tabs of the FAUST ONLINE COMPILER. So, as an example, someone can choose just to see the block diagram of an object or its automatically generated documentation.

3.2 The Example Saver

As said before, the catalog of examples of the FAUST ONLINE COMPILER is intended to be a platform to share FAUST objects between users. This is made possible by the example saver (see figure 7) that allows someone to save the FAUST code he edited or he dropped within the catalog. No identification is required to carry out this task: anyone can freely upload his work in the catalog. When creating a new example, users can also add a quick description to their object as well as a screenshot.

If an existing example is modified by a user different than the one who created it, every users that contributed to this example will be notified by e-mail of the changes made in their FAUST code. Finally, any user can freely delete examples from the catalog. As in the case mentioned before, editors will be notified of this modification with an e-mail.

In order to keep a good order in the catalog and to avoid duplicates or SPAMs, a voting system has been implemented. This can also help to promote the best user examples from the catalog.

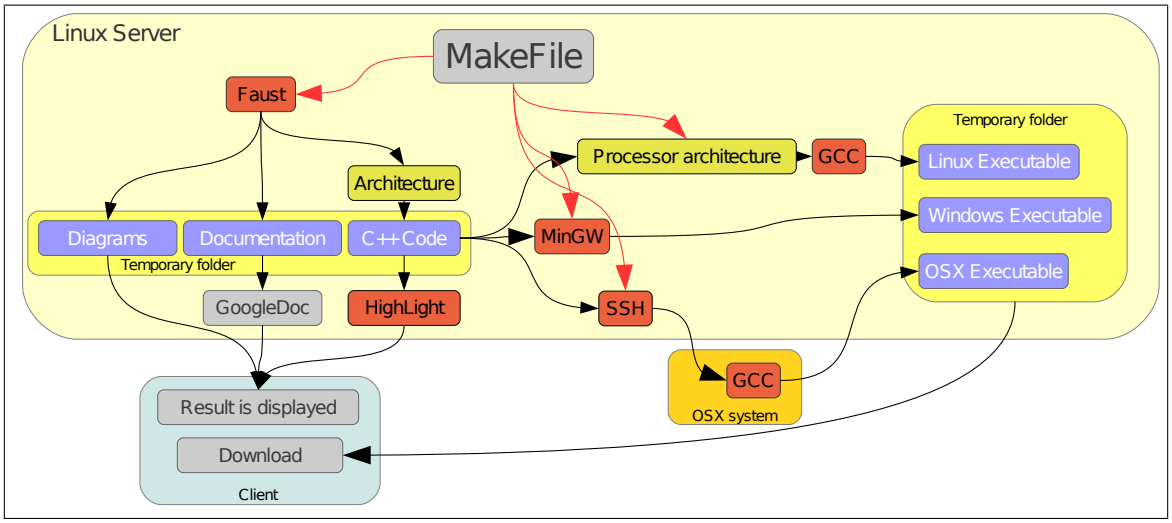


Figure 5: FAUST ONLINE COMPILER overview.

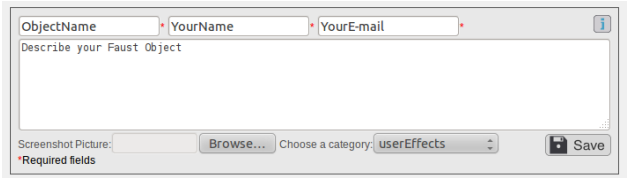


Figure 7: The example saver of the FAUST ONLINE COMPILER.

4 Exporting and embedding the Faust Online Compiler

The FAUST ONLINE COMPILER can be very easily embedded in any website using an *iframe*:

```
<iframe src="http://faust.grame.fr/
    compiler" style="border:none"
    height="1200px" width="722px"></
    iframe>
```

It can also be installed on an Apache server[7] just by copying the source code available on the FAUST repository[8] on it.

5 Current and Future Works

Recent improvements on the FAUST compiler allow it to generate `JAVA` and `LLVM` code instead of `C++` from a FAUST code. This obviously brings to the idea of implementing a FAUST architecture to generate `JAVA` applets directly usable from a web browser. Thus, the FAUST ONLINE COMPILER should soon be upgraded with a new tab running a web app generated in function of a FAUST code edited in the `Faust Code` section. Our hope is that the `JAVA` applet will be updated almost in real time without having to restart it every time a change would be made. Therefore, this feature could offer some sort of live codding possibilities.

As the FAUST ONLINE COMPILER prove to be a good solution to the platform and dependencies issues related to the use of architecture files in FAUST, it can be interesting to establish some links between it and the `FaustWorks`[9] program. In that special case, `FaustWorks` would be used to edit the FAUST code and see its resulting diagram, `C++` code, etc. but the compilation would be carried out on the server allowing it to generate FAUST applications for any platforms, processors and architectures.

6 Conclusion

The FAUST ONLINE COMPILER makes the FAUST experience easier than ever by offering a programming environment ready to use and free from any installation and compatibility issues.

It is certain that web technologies are having an increasingly important role in the domain of music technology. The arrival of new standards

[7] `http://www.apache.org/`

[8] `http://sourceforge.net/projects/faudiostream/`

[9] `FaustWorks` is an IDE for FAUST implemented with `Qt`.

like HTML5 or of new web-audio APIs is opening the way to new paths of exploration such as "Web Digital Signal Processing".

7 Acknowledgements

This work has been carried out in the frame of the ASTREE project[10] (ANR-08-CORD-003) on the preservation of real-time musical pieces.

References

Karim Barkati, Dominique Fober, Stéphane Letz, and Yann Orlarey. 2011. Two recent extensions to the faust compiler. In *Proceedings of the Linux Audio Conference (LAC-2011)*, pages 1–8, National University of Ireland, Maynooth, Ireland.

Dominique Fober, Yann Orlarey, and Stéphane Letz. 2011. Faust architecture design and osc support. In *Proceedings of the Conference on Digital Audio Effects (DAFx-11)*, pages 213–216, IRCAM, Paris, France.

Romain Michon and Julius Smith. 2011. Faust-stk: a set of linear and nonlinear physical models for the faust programming language. In *Proceedings of the Conference on Digital Audio Effects (DAFx-11)*, pages 199–204, IRCAM, Paris, France.

Yann Orlarey, Dominique Fober, and Stéphane Letz. 2002. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMA)*, pages 542–547, Gothenburg, Sweden.

[10] http://www.ircam.fr/

FaustPad : A free open-source mobile app for multi-touch interaction with Faust generated modules

Hyung-Suk Kim
Dept. of Electrical Engineering,
Stanford University
Palo Alto, CA 94305, USA
hskim08@stanford.edu

Julius O. Smith
Center for Computer Research in Music and Acoustics.
(CCRMA) Stanford University
Palo Alto, CA 94305, USA
jos@ccrma.stanford.edu

Abstract

In this paper we introduce a novel interface for mobile devices enabling multi-touch interaction with sound modules generated with Faust[1] (Functional Audio Stream) and run in SuperCollider. The interface allows a streamlined experience for experimentation and exploration of sound design.

Keywords

Faust, mobile music, SuperCollider, music app.

1 Introduction

Faust (Functional Audio Stream)[1] is a functional domain-specific language for real-time signal processing and synthesis. Programs written in Faust are translated in to highly efficient C++ code which can then be compiled into modules for various architectures and synthesis environments including SuperCollider, PureData and Chuck. Faust separates the specification from the underlying implementation and the UI, allowing the programmer to focus on the design of signal processing or synthesis modules.

Quick testing of a module is possible by compiling the program as a standalone program. However integrating a UI for a synthesis environment such as SuperCollider or PureData is not that trivial. In either case, the user interaction is limited to mouse-pointer interaction.

Mobile devices introduce a tangible, interactive experience compared to that of the traditional desktop. Recent mobile devices, i.e. smart phones, enable multi-touch interaction as well as other features, e.g. accelerometer, GPS, gyroscope, which allow novel methods of interacting with sound.

Mobile devices have become computationally powerful enough for audio synthesis directly on the device. The MoMu toolkit [2] used by the Stanford Mobile Phone Orchestra (MoPho) is a toolkit for music synthesis on iOS devices. Unfortunately most audio SDKs for mobile devices are OS dependent, requiring separate learning for each new mobile OS. The computation power of mobile devices is still limited compared to desktop systems. Computationally complex modules are hard to run on mobile devices.

Mobile devices can be used as a music controller via OSC. TouchOSC, Control OSC[2] are examples. Such apps are general purpose OSC controllers that are highly customizable. The price of customizability is that it takes time to set up a module. This can become very cumbersome if the sound module has a lot of controls. Physical modeling synthesis programs can easily have over 20 parameters which can be very time consuming to setup.

In this paper we present an OSC music controller app, FaustPad,[3] that automatically creates the interface and necessary OSC settings based on the compiled results from Faust. This approach reduces the complicated setup process, allowing a streamlined experience for experimenting with the generated sound module.

2 Overview

Setting up a module for interaction requires the following steps: 1) compiling the program with Faust, 2) setting up the synthesis environment, and 3) setting up the mobile device.

[1] http://faust.grame.fr/

[2] http://charlie-roberts.com/Control/

[3] Code and documentation for FaustPad, can be found at https://github.com/hskim08/FaustPad

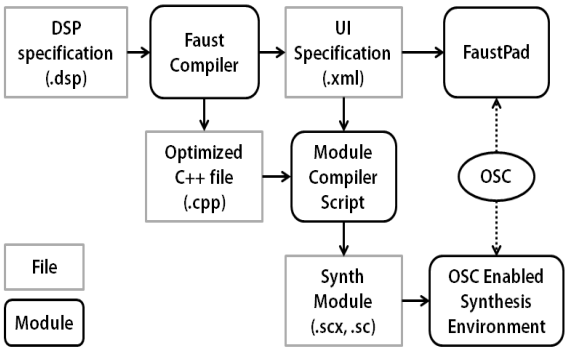


Figure 1: Overview flowchart of FaustPad setup. The file in parentheses may change depending on the environment used. Here we show the case for a SuperCollider Extension.

In this paper we cover setting up a Faust synthesis module with SuperCollider. In the last section we look at an example of installing all the modules in Faust-STK[3], the Synthesis Toolkit port for Faust.

We use Faust to create a SuperCollider extension which is loaded into `scsynth`, the SuperCollider server. FaustPad is used as an alternative `scsynth` client, replacing `sclang`, the SuperCollider interpreter.

3 Compiling with Faust

Recent Faust releases include scripts to compile programs into various environments. For SuperCollider extensions, we used `faust2supercollider` to create a SuperCollider class file (`.sc`) and a SuperCollider extension (`.scx`).

The Faust compiler can create an abstract "user interface" definition in XML format with the `--xml` option. This XML-file is used to create the UI for the mobile app.

For an in-depth description see "Audio Signal Processing in FAUST"[4].

4 SuperCollider

SuperCollider consists of two parts, the client, `sclang`, and the server, `scsynth`. The two parts communicate via OSC. FaustPad acts as an alternative client to `sclang`.

[4]https://ccrma.stanford.edu/~jos/spf/Using_FAUST_SuperCollider.html

```
SynthDef.new("sitar", {
    arg freq = 440.0, gain = 1.0, gate =
0.0, resonance = 0.7, damp = 0.72,
roomsize = 0.54, wet = 0.141, pan_angle =
0.6, spatial_width = 0.5;
    Out.ar(0, FaustSitar.ar(freq, gain,
gate, resonance, damp, roomsize, wet,
pan_angle, spatial_width));
}).load(s);
```

Figure 2: Output example of FaustScParser.

It is easy to import custom synthesis modules into SuperCollider as "extensions" by copying the `.sc` and `.scx` files into the SuperCollider extension folder.

Synth definition (SynthDef) files need to be defined and loaded to create instances of the synth modules. Control parameter names are also defined in the SynthDef files. The parameter names of the SynthDef files must match those of the app. This can be achieved by parsing the UI definition XML-file from Faust to create the SynthDef file. The Java project in the FaustPad github repository creates an executable jar-file, FaustScParser.jar, that does this.

In `sclang`, a SynthDef is compiled into byte code which is then sent to `scsynth` as an OSC message. As of the time of writing the SynthDef needs to be loaded once manually by the script created by the FaustScParser. Once loaded the SynthDef is saved on the server and will be loaded each time `scsynth` boots. We plan to port SynthDef compiling into FaustPad to further reduce the setup process.

5 FaustPad

FaustPad is an OSC controller app with auto generated UIs tailored to Faust created modules. Ease of setup comes at the price of customizability. The app will only work for Faust generated modules. Given the expandability of Faust, it is a tradeoff worth making.

One advantage of using a mobile device is multi-touch. Multiple sliders and buttons can be modified simultaneously. FaustPad currently supports iOS devices only. An iPhone or iPod Touch can handle up to 5 touches and an iPad can handle up to 11 touches at once.

5.1 Building the UI

A typical Faust user interface definition has two parts, 1) the definition of widgets, i.e. the control parameters both passive and active, and 2) the layout

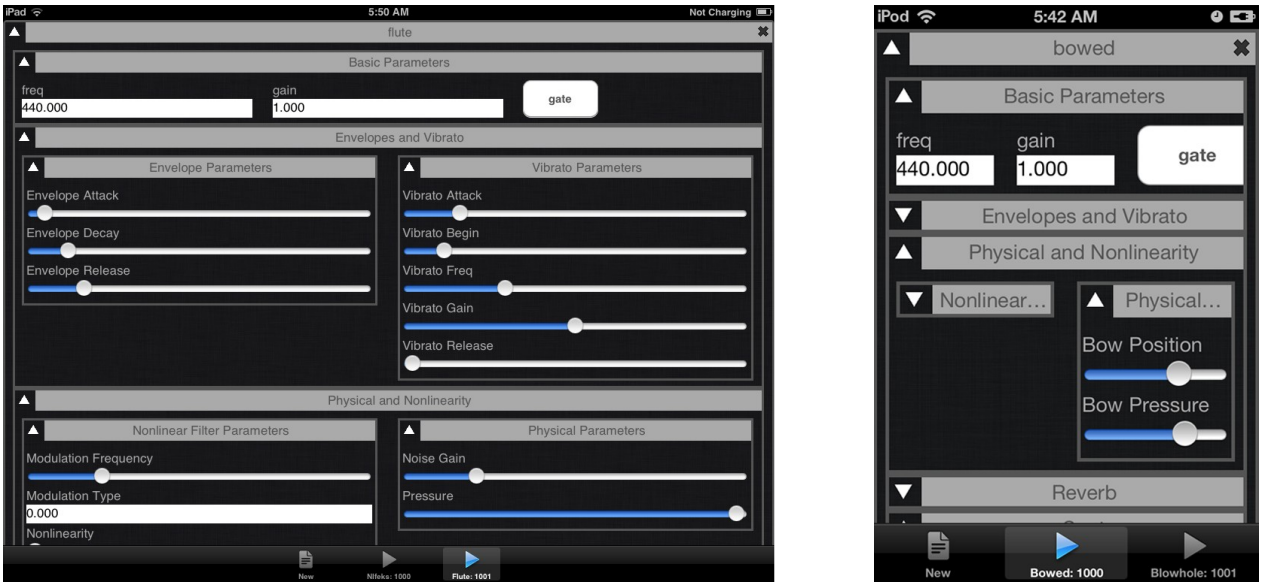


Figure 3: Screenshots of FaustPad for iPad and iPod Touch. The FaustPad UI works for any orientation. In the iPod Touch screenshot, groups are folded to show only controls of interest.

definition. The separation of components and layout frees the UI builder from a static layout.

In the current implementation of FaustPad, the app maintains the given layout, creating foldable "group views". The automated building of the UI has two phases accordingly, 1) the creation of widgets and 2) the creation of groups which form the layout.

5.2 Layout principles

Faust generated modules may have many control parameters. This is especially true for physical modeling synthesis modules such as many of the modules in Faust-STK. For many parameters, once a value is set there may be little need to change it during a performance. In FaustPad, unused groups can be folded so that the user can focus on the parameters of interest.

FaustPad also allows multiple modules to be created. Each module is created in a separate tab. Tabs are always visible in the tab bar at the bottom of the UI, allowing quick change of synth module.

5.3 OSC messaging

Each UI component holds its own data, e.g. label, min/max values, from the UI description file. When there is a user interaction event, it sends an event to the OSC messaging object which parses the event and sends a properly formatted OSC message to the server. This separation of UI component and OSC messaging allows configurations for various OSC messaging protocols for different synthesis environments. The OSC messaging object also listens for incoming OSC messages that can be used for automation and feedback from the server.

6 Example: Faust-STK

We have tested FaustPad with Faust-STK, included in the Faust release. The example files come with a `Makefile` for each environment including SuperCollider. Creating the UI description files is less trivial, because the `Makefile` removes the files at the end of the compilation. This can be avoided by modifying `faust2supercollider` and `Makefile.sccompile`. After setting up the aforementioned scripts then loading all SynthDefs, we could use all 21 Faust-STK modules in FaustPad by simply copying the files to the FaustPad documents directory.

7 Extending FaustPad

The FaustPad user interface only adheres to the user interface description file and makes no assumption of the underlying structure until the moment a OSC message is sent in the OSC messaging object. Thus it is possible to extend the FaustPad to other OSC enabled synthesis environment such as PureData or Chuck. As of the time of writing, we are in the process of adding support for PureData via `pd-faust`[5].

[5]http://docs.pure-lang.googlecode.com/hg/pd-faust.html

8 Conclusion

With FaustPad we have explored methods for enabling a streamlined experience for interacting with Faust generated synthesis modules.

The separation of specification, device implementation and UI description of Faust enables support for multiple synthesis environments with multiple UIs. With FaustPad we explored the possibilities of using an alternative UI on a mobile device.

We have tested the experience by compiling Faust-STK synthesis modules for SuperCollider. Though there were some non-trivial modifications to code that were needed to obtain the UI description files, once made, it was easy to run and install the synthesis modules.

FaustPad is still in its early stages and there are more issues to be addressed. Such issues include extension of the interface to other environments such as PureData or Chuck, easy discovery and connection of server and device via technologies such as Bonjour/Zeroconf networking.

Connecting to PureData via pd-faust is of particular interest due to the expandability of PureData itself. With pd-faust it is possible to synchronize the interface to a MIDI-OSC transport for automation allowing live performance and audio recording use of FaustPad.

Finally, we acknowledge the fact that even though FaustPad is a free open-source project under a BSD-like license, it may not be open for anyone to develop due to the limitations of the iOS SDK. Porting to other mobile OS, many of which have become open-source, is another important future work to be done.

References

[1] Orlarey, Yann; Fober, Dominique; Letz, Stéphane. 2009. "Faust: an Efficient Functional Approach to DSP Programming". *New Computanionals Paradigms for Computer Music*. Edition Delatour. ISBN 978-2-7521-0054-2.

[2] Bryan, N. J., Herrera, J., Oh, J., and Wang, G. 2010. “MoMu: A Mobile Music Toolkit.” *In Proceedings of the International Conference on New Interfaces for Musical Expression*. Sydney 2010.

[3] Michon, Romain Smith, Julius O. III. 2011. "Faust-STK: a Set of Linear and Nonlinear Physical Models for the Faust Programming Language". *Proceedings of the 11th Int. Conference on Digital Audio Effects (DAFx-11)*: 199–204

The Why and How of With-Height Surround Sound

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

Abstract

With-height reproduction is a hot marketing item in surround sound. This paper examines the (sometimes non-obvious) motivations behind it and discusses the advantages and shortcomings of different methods as to the perceptional mechanisms of height localisation.

Keywords

With-height surround sound, Ambisonics, Psychoacoustics

1 A brief history of height reproduction

With-height surround has featured in many experimental one-off installations such as Stockhausen's "Kugelauditorium" at the 1970 World's Fair, or Bayle's "Acousmonium". They have usually employed custom-tailored, ad-hoc driving techniques with little or no regard for portability. The speaker system is considered an integral part of the artwork or performance rather than an interchangeable tool.

Therefore, they are only of historical interest today, although a number of sophisticated systems in the acousmatic tradition still exist and continue to be developed. On the other hand, there have been numerous proposals to bring with-height surround to a wider market in a systematic and portable way.

As early as 1973, Ambisonics pioneer Michael Gerzon suggested a practical approach to what he called *periphonic* sound using only four channels, also known as B-format [1]. In 1992, he proposed the technology as a candidate for the then-upcoming HDTV standard [2].

In 1999, Tomlinson Holman demonstrated "10.2", the first commercial cinema sound proposal to intclude height channels, if only at the left and right front [3].

Around the same time, German tonmeister Werner Dabringhaus entered the budding DVD audio market with his 2+2+2 system [4], which trades the center and LFE channels of a 5.1-capable medium for left and right frontal height.[1]

Belgian sound engineer Wilfried van Baelen experimented with 2+2+2 in 2005, and extended the concept into what he calls Auro-3D [5]. In its simplest form, it adds four height speakers on top of the standard 5.1 layout.

Likewise in 2005, a team of NHK researchers led by Kimio Hamasaki introduced 22.2 to accompany a future ultra-high-definition TV standard. It features a complete upper ring of eight channels plus one zenith speaker, ten on the equator, and an additional three bottom channels in the front [6].

2 Classification of existing methods

With the exception of Ambisonics, the approaches mentioned so far all build upon (or drag along, depending on your point of view) previous technology. They are *channel-based*, which means that the mix has to be made specifically for the reproduction speaker layout at hand, and they

[1]Omitting the center channel might seem strange today, but in 2000, dedicated LCR microphone techniques were not in common use, and classical Tonmeisters in particular were mostly unaware of their potential. Stereo main microphone techniques on the other hand were well understood and mastered, and those who just tried to add a center without changing their L/R miking as well found only increased coloration and loss of imaging clarity.

employ stereophonic localisation techniques to create phantom sources between speakers.

Among manufacturers of Wave Field Synthesis systems, the term "3D" has been (ab)used in marketing for a long time even in the absence of height capability. More recently however, WFS manufacturer IOSONO has introduced elevated speakers, thus earning the 3D moniker, while at the same time increasing the tweeter spacing of their systems considerably.[2] [7]

Such proprietary "hybrid WFS" systems use undisclosed panning techniques to include height, very likely a combination of delay panning and VBAP. The latter, short for Vector-base Amplitude Panning, is a conceptually very simple and elegant method introduced by Pulkki in 1997 [8]. It extends the idea of level panning to triplets of speakers, allowing the positioning of a sound source anywhere on a speaker mesh surface. VBAP can be applied to arbitrary speaker layouts, but it produces timbre shifts and highly variable perceived source width depending on source location.[3] The worst-case source width is equivalent to Ambisonics (which delivers perfectly constant panning). [9]

This effect is even more pronounced with sparse arrays, making it a less-than-ideal approach for layouts such as Auro-3D, and it does not work at all if a given channel is reproduced over multiple speakers, as is common in cinema installations.

WFS, VBAP and similar techniques are *object-based*, which means that individual (usually monophonic) audio files are stored with separate positional metadata, allowing them to be modified and re-positioned easily. Another advantage is that the mix is decoupled from the layout of the speaker system used for reproduction. On the downside, it is quite cumbersome to describe natural ambient recordings (which describe a spatial continuum rather than individual "spatial samples") as monophonic objects.

Furthermore, object-based systems require an elaborate and CPU-intensive rendering process for listening, with a complexity growing linearly with the number of objects (O(N)).

Finally, Ambisonics is *soundfield-based*: the B-format carries an arbitrarily precise description of the resulting physical soundfield, where precision is determined by the order. It is not easily possible to separate single objects, but spatially continuous ambience recordings can be included perfectly. The B-format is again decoupled from the speaker layout by means of a decoder, which contains the information about the speaker positions. The decoding step is trivial compared to WFS rendering, and its complexity is constant (O(1)).

Crosstalk-cancelled binaural (or *ear-signal-based)* systems are theoretically able to deliver height cues as well. In a virtual environment at RWTH Aachen, striking effects have been demonstrated [10], but without head-tracking and individual HRTFs and in the absence of visual cues, the results will be mixed at best. In any case, the coloration is severe, and systems can accommodate at most one or two listeners.

Headphone binaural systems can deliver perfectly convincing height with excellent fidelity, but the required 3-axis head tracking systems are not yet widely available, and perfect results require individual HRTF measurements for each listener. Catering to more than one listener implies that the rendering and head tracking has to happen in the headphones, which is not feasible with current embeddable computing platforms, or that an individually pre-rendered signal is presented to each headphone, which puts a great strain on wireless bandwidth.

3 Elevation perception and stereophony

There are two totally distinct motivations for the inclusion of height speakers.

The most obvious one is the desire to position or reproduce sounds along the vertical or z-axis, not just on the horizontal plane around the listener. The

[2] This implies that textbook WFS only happens at very low frequencies, and other localisation mechanisms must be employed for the remainder of the spectrum.

[3] In its basic form, VBAP will employ one speaker if the source is directly on a speaker position, two if it is on the line between two speakers, and three anywhere else.

channel-based systems mentioned above perform poorly in this respect, because they rely on *stereophonic* localisation. Its mainstay is the clever delivery of artificial interaural level and time difference cues (ILD and ITD).

However, ILD and ITD remain constant as a source moves upwards on the median plane, which explains the comparatively poor vertical discrimination of the human hearing apparatus. The only height cue available to us is a rather subtle coloration of the sound caused by reflection, refraction, and absorption effects on pinnae, head, and torso. This cue is purely monaural.

Height perception is most acute when the subject has a clear mental reference of the natural (i.e. non-elevated) timbre of a sound source and will degrade with synthetic or otherwise unfamiliar sounds (compare [11]).

Blauert has demonstrated that height perception correlates with the spectral distribution of the sound event, provided it is sufficiently broad-band. [12]

With narrow-band signals, the location of the auditory event on the median plane depends only on the frequency, not the actual sound source location. [13]

Hence, height illusion can be created by applying equalisation which exploits these effects.

Even though sound engineers or electro-acoustic composers may sometimes employ such EQ ad-hoc, none of the with-height stereophonic methods include it as part of their standard in any systematic way.

The only remaining tool for positioning a source along the z-axis is simple amplitude panning. However, such a vertical “phantom source” will not result in any ILD or ITD information. Even worse, the coloration cue will very likely be meaningless, too, because the sum of a horizontal and an elevated pinna-colored sound is not necessarily similar to the pinna effect on a physical source in between.

Consequently, stereophonic systems exhibit a very steep localisation curve along the z-axis. They will usually produce auditory events either on the equatorial or the elevated speaker level. While vertical motion can be suggested, stationary sources between the two extremes are not stable. [14]

Producing stable auditory events above the elevated speakers is likewise impossible.

The only techniques which can deliver good localisation at arbitrary locations outside the equatorial plane are hybrid "WFS", VBAP, and higher-order Ambisonics, if and only if the speaker density is sufficient in the region of the desired auditory event.

Microphone arrays for stereophonic with-height systems will usually aim at complete decorrelation between the corresponding horizontal and elevated channels by using vertically widely spaced omnidirectional microphones, or they may try to minimize crosstalk by using highly directional ones. Both approaches clearly do not aim for a continuum of localisation along the z-axis.

Which leads to the second, dominant motivation for height speakers: timbre. Proponents of 2+2+2 and Auro-3D in particular claim that, in addition to a more convincing feeling of envelopment, the perceived tone color will be more natural in the presence of appropriate height signals. Furthermore, the listening area is believed to be larger than that of a comparable horizontal-only system.

In the author's listening experience, this is generally true, not only for Auro-3D but also for the "hybrid WFS" systems mentioned before, as well as for Ambisonics. Informal A/B tests (performed by comparing the full rig with the equatorial speakers only), suggest three advantages of with-height systems:

They appear to be more robust in the presence of room problems, perhaps because more room modes are being excited but at lower level, which might even out coloration effects.

Furthermore, they provide a more realistic and more stable sense of envelopment, which facilitates suspension-of-disbelief.

Finally, the height speakers seem to smooth the Ambisonic phasing artefacts and timbre shifts across the listening area.

4 Height reproduction in Ambisonics

In Ambisonic systems of sufficiently high order, a coherent sound field is being reconstituted in the sweet spot. While a listener may still have trouble

discerning or localizing sounds along the z-axis due to the limited resolution of the human hearing apparatus on the median plane, s/he will be able to resolve these ambiguities by moving the head.

Small subconscious movements can provide subtle differential directional cues, and intentional tilting of the head can be used to train the more acute lateral hearing mechanism on the source and gather additional information [15]. This helps the brain to fuse very stable auditory events at height, but it requires a natural soundfield, without any psychoacoustic tricks optimized to deliver artificial cues.

It appears that the source localisation remains stable even after the head is back in the rest position, as if the brain memorises the reference and uses it to align subsequent ambiguous cues.

Interestingly, despite the typical checkerboard interference pattern of Ambisonic systems, head movements seem to remain beneficial even for listeners well outside the sweet spot, an observation that needs to be validated by further study.

High-fidelity soundfield reproduction without assumptions as to listener orientation and temporal or spectral trickery will let listeners explore the spatial structure of a sound scene individually. They will find that they can rely employ their entire auditory sensorium to perceive and analyse selectively whatever sparks their interest. It is safe to assume that this helps to increase enjoyment and depth of understanding.

In contrast, height illusions created by naive pinna coloration simulation or non-tracked crosstalk cancelled binaural will fall apart the moment the head tilts away from the assumed frontal direction, failing the listener just at the time s/he displays particular interest.

5 Benefits of periphonic sound reproduction

The perceptory advantage of soundfield-reproducing techniques[4] opens up new applications which would be very hard or impossible to achieve with stereophonic with-height systems. Consider the recording and reproduction of works whose spatial organisation demands exact vertical localisation (think "spatial fidelity") rather than a vague notion of spaciousness, or of multi-layered compositions which are sonically so complex as to be downright indigestible unless the individual components are very precisely discriminated in space.

Periphonic soundfield reproduction in higher-order Ambisonics requires a substantial additional investment in both equipment and effort over conventional stereo, and it should be obvious that the justification of this investment depends on the program material.

A novice to baroque music may benefit from a little spatial separation when learning to appreciate a three-part invention or a five-part fugue from the Well-tempered clavier, but this is more a teaching aid than an artistic requirement. The musical structure itself is so resilient to spatial and timbral modifications (as aptly demonstrated by Wendy Carlos [16] and many others) that it will fare surprisingly well even as a phone ringtone.

A Gabrieli double choir from the same epoch however might become too dense if reproduced in stereo, and a historically informed reproduction would mandate at least horizontal surround.

Recordings of complex organ works can be more approachable if the original vertical separation of the divisions is retained.

Ambient nature recordings are an obvious case in point for full periphony.

Even if all direct sound emerges from a limited set of directions, full periphony is required to render the room acoustics correctly, if desired.

On the other end of the spectrum, a low-down blues song is adequately conveyed by a minimalist reproduction system; the impact of a distinctly low-fi bootleg might actually be harmed by uncalled-for technological “sophistication”. Buyer beware.[5]

[4] In theory, this includes WFS. However, "classical" WFS systems do not deal with height and "hybrid" systems are not wave field synthesis in the strict sense.

[5] In all things of technical hack value, this author considers « because we can » a perfectly adequate justification. Yet in the arts (or any other form of inter-individual communication), the very same approach can be disastrous.

6 Conclusion

With-height systems in general have the potential to be more robust than horizontal-only setups. Even if height localisation is not strictly necessary for the job at hand, the improved envelopment and timbral benefits are arguments in favour of full periphony. Stereophonic methods are of limited use for vertical localisation. Of the different approaches, those that attempt some degree of soundfield reconstruction should be more robust than ad-hoc, channel-based approaches.

[1] Michael A. Gerzon, 1973: "Periphony: With-Height Sound Reproduction", in: Journal of the Audio Engineering Society, vol. 21, pp. 2-10
[2] Michael A. Gerzon, 1992: “Hierarchical System of Surround Sound Transmission for HDTV”, in: Proceedings of the 92nd AES Convention, Vienna
[3] Barry Willis, 1999: “Holman Conducts First Public Demo of '10.2' Surround Sound”, in: Stereophile website, http://www.stereophile.com/news/10489/
[4] Dieter Steppuhn, 2001: “Begegnung anderer Art. Neue Audio-DVDs '2+2+2' bei MDG”, in: Neue Musikzeitung 02/01, http://www.nmz.de/artikel/begegnung-anderer-art
[5] Auro-3D product website, history section, http://www.auro-3d.com/about-history
[6] Kimio Hamasaki et al., 2007, “22.2 Multichannel Sound System for Ultra High-Definition TV ”, in: SMPTE Motion Imaging Journal vol. 117(3), pp *40-49*.
http://www.nhk.or.jp/digital/en/technical/pdf/IBC2007_08040907.pdf
[7] Frank Melchior, 2011: “IOSONO: 3D audio solutions ”, Workshop held at the International Conference on Spatial Audio, Detmold, Germany
[8] Ville Pulkki, 1997: “Virtual sound source positioning using vector base amplitude panning.”, in: JAES vol. 45(6), pp. 456-466.
[9] Franz Zotter et al., 2010: “Techniques and Considerations on Sound Field Recording and Reproduction Using Spherical Harmonics”, p.19.
http://iaem.at/Members/zotter/publications/IEM_SHTechniques_Zotter.pdf/view
[10]Tobias Lentz, 2006: “Dynamic Crosstalk Cancellation for Binaural Synthesis in Virtual Reality Environments”, in: JAES, vol. 54(4) pp. 283-294
[11]G. Plenge, und G.Brunschen, 1971: “Signalkenntnis und Richtungsbestimmung in der Medianebene bei Sprache”, in: Proceedings of the 7th International Congress on Acoustics, Budapest, 19 H 10, according to [12].
[12]Jens Blauert, 1997, 1983: “Spatial Hearing. The Psychophysics of Human Sound Localization”,
MIT Press, Cambridge, Mass., p. 109
[13]Jens Blauert, l.c., p. 45
[14]Günther Theile, and Helmut Wittek, 2011: “Principles in Surround Recording With Height”, in: Proceedings of the 130th AES Convention, London
[15] Jens Blauert, l.c., p. 181
[16]Wendy (then Walter) Carlos, 1968: “Switched-on Bach”, Columbia Records

Field Report II - A contemporary music recording in Higher-order Ambisonics

Capturing *Chroma XII* by Rebecca Saunders

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

Abstract

In 2010, the author had the privilege to capture a performance of Rebecca Saunders' intricately spatial composition *Chroma XII* in fully periphonic third-order Ambisonics. The production grew to considerable complexity and provides an excellent showcase for a large-scale Ambisonic production using free software. This paper discusses the artistic motivation (or even necessity) of using a with-height recording method for the work at hand.

After a short description of the composition, its instrumentation and the performance space, the microphone and mixing techniques are being discussed in detail, including hardware and software toolchains, post-production workflow, and lessons learned from subsequent replays on various systems.

This paper is a follow-up to a workshop paper presented at LAC 2010 in Utrecht [1].

Keywords

Ambisonics, live production, with-height surround, free software.

1 Introduction

In previous higher-order Ambisonic productions, the author had gained some experience in blending first-order Soundfield recordings with higher-order spot microphones in a pop context [1], and in creating entirely artificial renderings with discrete microphones only, in a church acoustic [2]. In order to hone these methods some more, and to test their limits, the author had been putting out feelers for a spatially demanding production of instrumental music.

To be suitable as a periphonic recording showcase, a piece should do away with the traditional concept of a frontal stage, with no excuse for anything but a fully isotropic recording system. It should be interesting and powerful even when recorded documentary-style, that is, naturalistic and without much interpretation during the recording process. It should include elevated sources, and it should take place in an acoustically interesting space, to tax the diffuse-field reproduction capabilities of the recording system. Lastly, it should allow the timbral quality to be judged against well-known references, which rules out electro-acoustic music in favour of instrumental or vocal works.

In the absence of any budget whatsoever, it was clear that the project would have to be piggy-backed on a live concert production.

1.1 The composition

Enter Rebecca Saunders, a highly esteemed British composer of instrumental contemporary music with a keen sense of sound texture and space. *Chroma* is one of her most successful works. Originally conceived in 2003, it has been in high demand ever since, with seventeen successful stagings in various surroundings as of March 2011, despite the considerable resources it requires.

For lack of a better category, *Chroma* is spatial chamber (or turbine hall, or castle, or museum, or philharmonic, or, in our case, baroque-gallery-with-anterooms and French garden) music. Its sounds originate from two pianos, two percussion sets, two violins, a cello, two double basses, an electric guitar, two trumpets, two clarinets, an (electric) organ, a

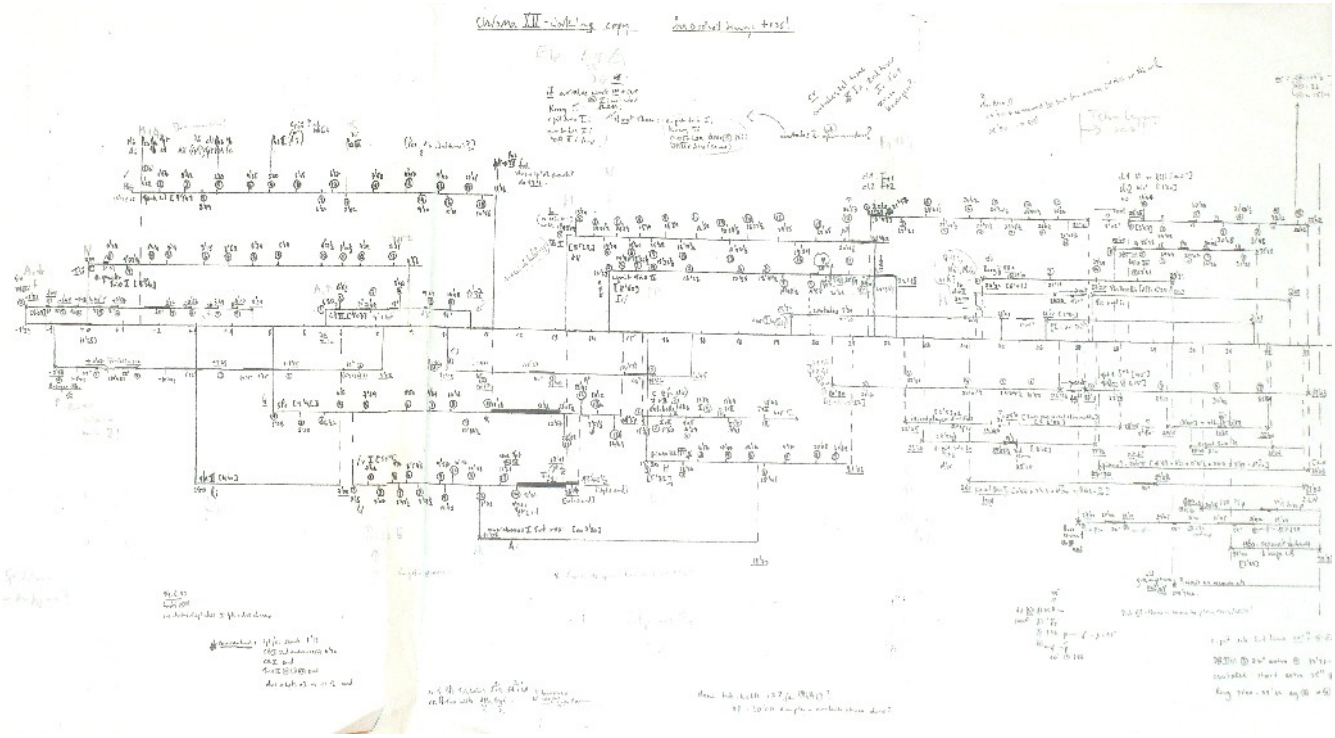


Illustration 1: The flow chart of Chroma XII used by the composer.

large number of wind-up music boxes and a portable record player. In the composer's words:

Chroma explores three different key issues: the architecture of the space, the density of the collage in the given acoustic, and the nearness or distance to the different music being performed. So firstly, it's about entering into a dialogue with an architectural space, exploring and emphasising the particular characteristics of the space. [3]

For each staging, *Chroma* is adapted to its new environment by the composer, and may grow a new musical module or two. The audience is invited to move freely around the place during the performance. Conditions permitting, the piece is performed twice in close succession, so that the listeners can explore two different paths through the soundscape. [4]

The musical language of *Chroma* consists of gestures, not melody or harmony in the traditional sense, and employs advanced non-standard playing techniques. Explosive, almost violent eruptions alternate with very soft and fragile but highly expressive sections.

The work is organized in modules, played by soloists, duos, or trios. These modules are scored individually, and within each module the voices are synchronized to a common time or rubato feel. Parts of *Chroma* can be (and are) performed individually as chamber pieces.

Each module is played in a specific place inside the larger venue, determined in advance or during rehearsals. The players of a module need not be close together - in the Hannover staging, some groups were spread out over 10 metres or more.

Sometimes, a single soloist or group will play exclusively, but usually, different modules are overlapping. There is an implied global tempo and a carefully laid out flow chart, but usually no precise rhythmic interlocking between modules[1]. Hence, the overall result is not strictly determined in time except when the composer desires a specific unison effect or close interaction. In this case, special cues will be agreed on and rehearsed as required.

The overall dramatic structure, while precisely adhered to once it has been determined, is not set in

[1]The players all carry stopwatches, which are started together a few minutes before the performace.

Illustration 2: The main hall of the Gallery building at Herrenhäuser Gärten, Hannover, looking west.

stone - it exists only in the composer's working chart, which is adapted to each new location, iteratively, during rehearsals (see illustration 1).

When a module is finished, the musicians will quietly pick up their instruments and proceed to the next location. Together with the tip-toeing audience, this imparts a constant extra-musical sense of motion to the scene.

In addition to the instrumentalists, *Chroma* incorporates some *objets trouvés* which are added to the sound collage at predetermined times: a large collection of wind-up music boxes spread out on the floor and later on music stands, and one or more portable record players with folk music recordings. In our case, the local bird life added a rather aleatoric but strikingly beautiful layer of its own.

1.2 The performers

Based in Köln, Germany, the Ensemble musikFabrik [5] is an internationally acclaimed specialist ensemble for contemporary music, with an extensive track record of Saunders performances and a history of close collaboration with the composer. One trumpet and one double bass module in *Chroma* were written specifically for members of the Ensemble musikFabrik.

The ensemble is self-governed - all repertoire decisions are made by the musicians.

The familiarity with and dedication to the material was very beneficial to the recording. Musicians, stage management and composer worked very efficiently, and there was much room for experimentation despite the time pressure.

1.3 The venue

Chroma XII[2] had been commissioned by Kunstfestspiele Herrenhausen to take place in the baroque gallery building of the Herrenhäuser Gärten in Hannover and the adjacent garden. The gallery's main hall is 67.45 m by 11.40 m, with a height of 8.0 m. Most of the modules in the first half happened here. Some musicians were seated in two adjoining hallways, and up on the main hall's two balconies. Additionally, two clarinets and a trumpet were

[2]The Roman numeral indicates the 12th staging and uniquely identifies the Hannover event. Since the premiere at Tate Modern in London, the basic score has constantly been changed, augmented and adapted to each new location.

placed on a lighting structure near the center of the main hall, directly above one another.

The building is a baroque masterpiece with floor-to-ceiling fresco paintings and one of the largest stucco ceilings in Europe.

At some point, the windows and door to the garden were to be opened, and musicians would gradually move outside and resume playing in the outdoors. The sound difference between the highly reverberant hall and the semi-anechoic garden was quite intriguing, particularly at the open windows where sounds from the inside and outside were allowed to mix. The piece concluded in the garden, gradually diminishing *al niente*, which in this case means it ducked under the natural ambience of crickets, birds and faraway chatter, and is probably still there.

The garden paths are covered in gravel, which produced clearly audible crunching footsteps in the recording. We feared this might be a problem, even though it was justified by the composer's directive that the audience move around. In the end the sound turned out to be so charming that it even received a gentle emphasis in post-production.

1.4 Recording approach

Periphonic reproduction fulfils three important roles in *Chroma.* It helps to separate in space those musical gestures which overlap in time, it provides a more complete reproduction of the room acoustics and its interaction with the music, and it has to substitute the freedom of the live audience to move around.

The author spent two days at the rehearsal studios of the musikfabrik in Köln, to listen in on as many module rehearsals as possible and to get acquainted with the musical language. A score was available for study, but while it is easy to look intellectual when sight-reading contemporary music, little insight can be gained unless one is intimately familiar with the notation, instruments and playing techniques at hand.

A preliminary microphone and cabling concept was developed after a visit to the location. The historic gallery building added its own share of complications: there is no way to suspend microphones anywhere, so stands it had to be.

In the absence of a practical higher-order microphone[3], we relied on a previously tested hybrid approach [1]: traditional soundfield microphones at the desired listening positions were combined with spot mikes for each individual instrument or group of instruments. These were panned in higher order for the purpose of augmenting and sharpening the image. In order to convey different perspectives and some sense of motion, we used two alternating main microphone positions.

At the rehearsals, we found that the most magic moment was the listener's passage from the hall into the garden - one of the many aspects of the work which could not be anticipated from score analysis. In the absence of an elaborate dolly system, the second main microphone was carefully picked up and carried outside. We made sure the cable run was neat and tried to decouple the microphone from handling noise as much as possible, but some remaining rumble had to be filtered aggressively. The operator's footsteps however were left in, because they would have been part of the live experience as well.

To increase the sense of motion during the passage, the active spot microphones were dynamically panned along.

During setup, it became clear that our gear was visually too intrusive to remain during the evening concert, so we switched from “as neat as possible” to “as easy to tear down as possible” and had to make do with a recording of the two run-throughs of the general rehearsal. It also became clear that we would not be able to spot every single instrument in the garden. In the end, the localisation performance of the first-order main mike alone was perfectly adequate, partly because the outdoor part was more sparse, and partly because there are no confusing reflections in the semi-anechoic free field. The two omnis next to the second grand piano in the garden did not contribute much spatial information, but they filled in the bottom end nicely (particularly because the main had to be rolled off due to wind noise).

[3]At the time of the recording, the commercially available Eigenmike was not yet usable as a B-format microphone. All available spherical arrays suffered from coloration, noise, and bandwidth issues.

After the general rehearsal, we recorded clicks at each instrument position into all microphones, to ensure proper time-alignment in post-production.

1.5 The hardware

Our original plan had allotted 22 channels, including two first-order main microphones.

Since it was clear that considerable distances had to be covered, we decided on a MADI ring infrastructure, to avoid the hum problems and interference of a large analog system powered by a strung-out, old, and possibly dodgy power grid. The interior MADI line alone was over 250 m long, with several remote-controlled microphone preamps distributed evenly throughout the venue, so as to keep the analog lines short.

A research team of Technicolor Hannover were bringing their Eigenmike, which was to be recorded on a separate system, synchronized to the main MADI clock.

In the “control room” (a broom cabinet), we used a splitter to distribute the stream to two independent Linux machines equipped with MADI cards, each runninng redundant disk arrays for storage.

Our modest miking plans were quickly steamrollered by the composer's restless creativity: after countless position changes and additions, we ended up with 40 channels altogether, at which point we were running 7 preamps. When the last pair of microphones was connected (and gaffer-taped to a tree because we had depleted our hardware stash), our spare cable box was down to three short XLRs, and we prayed for no more sparks of genius.

1.6 The software

For recording and post-production, we relied exclusively on free software based on Linux systems and the JACK audio server [6].

On the main recording machine, an Ardour2 instance [7] took care of writing the data and provided simple two-channel monitoring. We documented the microphone setup with Ardour's track comments feature and used track markers to take notes during the recording.

Ardour was configured to use 60 seconds of recording buffer per track, which increases data loss in the case of a crash or power outage (a situation that would have ruined the take anyways), but helped reduce the disk load, particularly as the session kept growing.

One of the tetrahedral microphones was matrixed and equalised using TetraProc [8], the other produced B-format directly. Both were converted to UHJ with jconvolver [9] and monitored in stereo.

The fallback recording machine ran jack_capture [10]. This software writes a multichannel interleaved file, which uses the disk bandwidth very efficiently. An additional load-in step is required before the audio data can be edited and mixed, but for a backup system, the issue only arises if the primary machine fails.

The preamps were remote-controlled with the custom midiremote software [11], which turned out to be invaluable, as the farthest preamp was easily a hundred-metre walk from the control room.

2 Post-production

2.1 Microphone bleed

In hybrid HOA recording, directional and very close spot microphones are desirable to minimize crosstalk from other instruments. In this particular case, the problem was even more pronounced because of the huge size of the venue: for sufficiently far-away spots, the time delay of the bleed signals exceeded the echo threshold of human hearing, and the resulting artificial echoes were very audible.

Sometimes, echoes could be attacked with heavy automation, muting or dipping unused microphones whenever possible. But sometimes, the problematic spot carried important content of its own and could not be turned off. There is no perfect solution, and a compromise between score fidelity and echo suppression must be found. In this recording, we were lucky to have many natural room echoes already, so the remaining artificial ones did not stand out too much.

Intensive spot miking can result in small and narrow-sounding sources which do not combine well, particularly if the spots are used at high levels. It gets worse the higher the order of the panners and reproduction systen is - in this sense, too much resolution can be harmful.

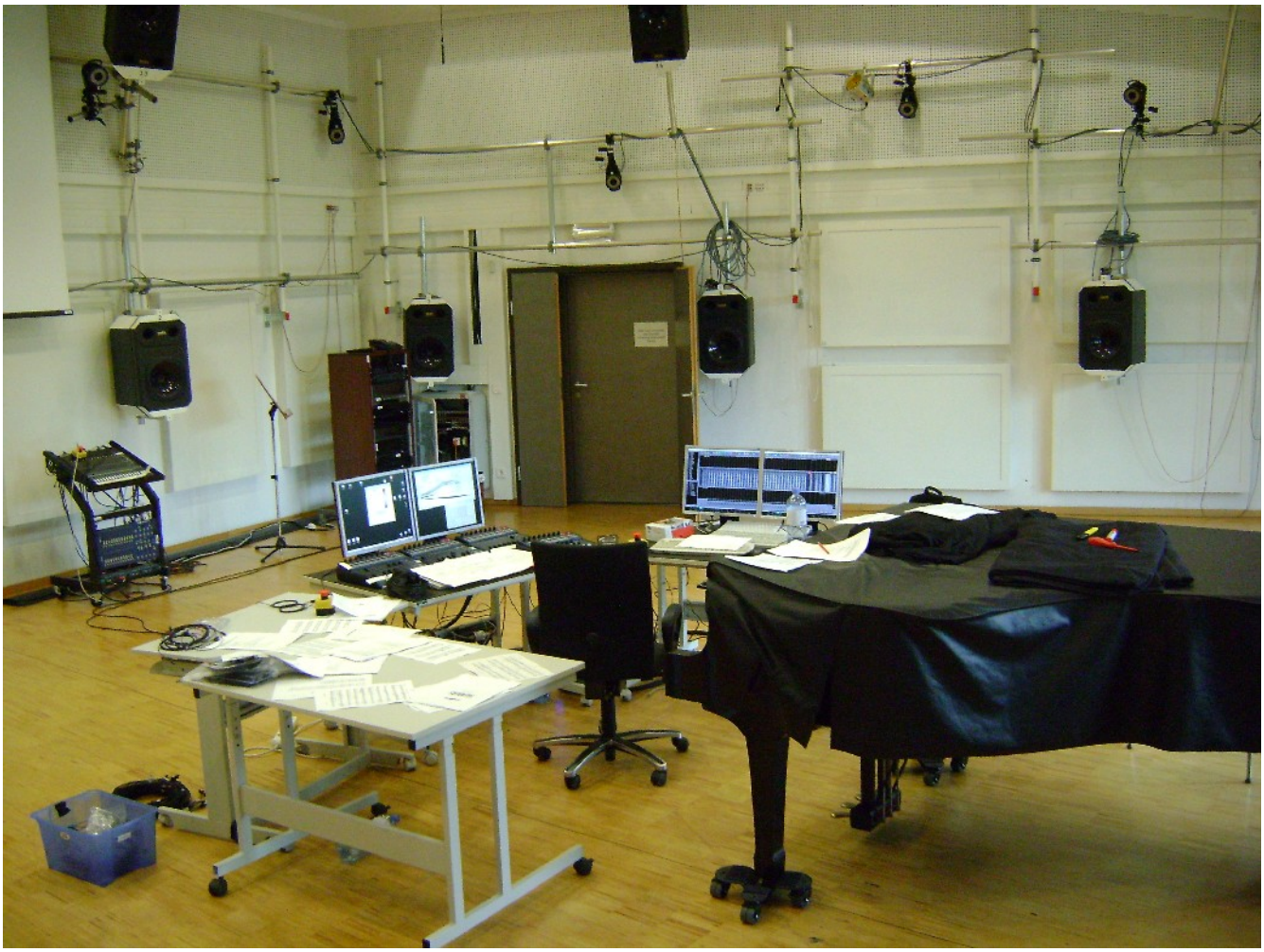


Illustration 3: Post-production at the IEM Cube in Graz, on a 24-speaker hemisphere of Tannoy 1200s. The console on the left controls the CUBEMixer. The MIDI controllers are attached to the machine on the right, which runs the Ardour session. The grand piano was extremely handy while working with the score. The two red buttons on the tables are emergency-mute switches for the speaker system.

To keep the mix from falling apart, each spot must be very carefully time-aligned with the main microphone to ensure proper blending, and it is advisable to use stereo pairs of spots whenever the sound source is large or consists of more than one instrument.

These stereo pairs should either be coincidental (X/Y or M/S) or very widely spaced, if at all possible.

Whenever strongly correlated but non-coincident signals are being used in an Ambisonic system, one must be aware that they are combined internally (i.e. with mathematical precision), which results in very pronounced comb-filtering artefacts. This sounds just as bad as a conventional A/B stereo pair panned towards the center.

While crosstalk can be low enough at higher orders to yield usable results for small A/B or ORTF-like configurations if they are panned far enough apart, the price is a strong dependency on playback order and decoder design (with severe coloration and imaging artefacts if the playback order is too low). Hence, it could be argued that the resulting signal is not proper B-format any more.

Very widely spaced pairs are sufficiently decorrelated at mid and high frequencies to be usable, but at low frequencies, combing can still occur. The main reason to choose wide pairs is that they work well with omnis, which are well regarded

for their extended LF response. Since this advantage will be lost in an Ambisonic context due to LF combing, there is really no point in using spaced omnis at all.

Furthermore, omnis will confuse Ambisonic localisation rather than improve it, by adding artificial run-time cues which should not exist in an Ambisonic system.

Coincident pairs on the other hand are inherently free of comb-filtering problems, and give you maximum control over the source width in post-production without unwanted side effects.

Despite their disadvantages, a few omnidirectional microphones did creep into the *Chroma* recording, partly for their superiour LF response, and partly for the mundane reason that we had run out of cardioids.

For each gran cassa, we used boundary microphones on the floor below. The corresponding overhead microphones of the percussion sets were high-pass filtered to mitigate LF combing.

The grand piano was covered with two cardioids spaced about 1.5 m apart, one above the tuning pegs (to accommodate non-standard playing techniques), the other at the rear end. Likewise, the upper microphone had its bass rolled off slightly.

Two small A/B setups were tried: one on a double bass duo, and the other in the garden. The bass pair was severely affected by the issues mentioned above, but a usable signal could be salvaged by switching off one of the microphones. The garden pair worked quite well, but only after it was delayed enough for the main microphone to dominate the directional perception. In the end, it was just used for some LF fill-up and a dash of "spaciousness", not for proper localisation.

It should be obvious that omnis are a lot more prone to pick up unwanted echoes. We only got away with them because either the covered instrument was extremely loud compared to everything else (the gran cassa), the particular module was effectively a solo (the double bass), or because some directional confusion was found to be tolerable (in the garden).

2.2 Post-production stages

Since full 3D mixing rooms are rare and in high demand, the workflow was split so that each stage could be completed with minimal equipment. Individual tracks were cleaned up (and equalised if necessary) on a standard stereo monitoring system. The time spent on the quite tedious cleanup runs was used to get familiar with the score, and to take notes about necessary mixing interventions.

Gradually, ideas for a "spatial interpretation" of the music began to form, to create a path through the music similar to what a moving listener would experience.

In the next step, the Ardour session was extended to third-order by adding a 16-channel master bus and the appropriate panning plugins [12], with group busses inserted as required. A basic panning scene was created blind, using a floor plan drawn by the composer and some photographs taken during the recording. All tracks were then time-aligned according to the clicks recorded earlier.

Now the production was ready to move to the author's hexagonal system, which is capable of second-order horizontal surround. With the sources roughly in place, a preliminary balance was created, and microphones were automated away when not needed.

When most of the basic homework was done, the session was taken to the IEM Cube in Graz, Austria, for a final ten-day mixing run. The Cube offers a hemispherical 24-channel monitoring system of co-axial Tannoy 1200s, in 12-8-4 configuration, using a prototype decoder by Franz Zotter et al. [13]

Slowly, the final interpretation was taking shape, and once the acoustic perspectives had been decided, all spot mikes were carefully aligned with the image of the corresponding main microphone. Finding the right balance was the most labour-intensive task. The absence of a global score did not help, as the mess in illustration 3 indicates.

In order to avoid mistakes with the automation, three MIDI controllers with 8 motor faders each were connected to Ardour to provide visual feedback, and for easy access to solo and mute functions. Panners were mapped to a set of rotary controllers with LED indicators for a quick overview of source positions.

Both run-throughs of the general rehearsal were mixed, but only the second one was selected for public playback, because it includes a number of last-minute changes made by the composer during intermission.

3 Playback

So far, the production has been played back on four periphonic and six horizontal-only systems, and it has stood up pretty well.

Some excerpts were performed during the dafx10 conference at the MUMUTH in Graz. The array was an elongated hemisphere of 29 Kling & Freitag CA 1001 arranged in layers of 12, 10, 6 and 1, driven by a max-r_E decoder designed by Thomas Musil and Peter Plessas. The rather long reverberation time disturbed the effect of the outdoor part of the recording, but otherwise the rendering was quite convincing.

During the 2010 Huddersfield Festival, where Rebecca Saunders was artist-in-residence, the author had the chance to hold a workshop at CeReNeM and to demonstrate the mix to the composer in a private session, using the SPIRAL facility [14]. The system had to be converted to Ambisonics rendering for the occasion, using a dual-band third-order decoder designed by Fons Adriaensen. It consists of 25 Genelec 8240A in three rings of eight plus a zenith speaker, and four 7270B subwoofers. With a reverb time of only 0.3s down to 150 Hz, the room was the most effective in transporting the free-field acoustics of the garden. It was surprisingly free of phasing artefacts, probably due to its ring structure.

Illustration 4: ICSA playback setup, 40 Neumann KH-120 plus four KH 810.

Illustration 5: Setting up Chroma in the SPIRAL at CeReNeM, University of Huddersfield.

The most recent replay took place at ICSA 2011 [15], on a rig of 40 Neumann KH120 monitors and four KH810 subwoofers designed by the author, again using a third-order decoder by Fons Adriaensen. This rig was vastly over-specified for third order, and even though the decoder used only a subset of speakers, the first trial run sounded phasy and irritating. It became clear that the native Soundfield signals in *Chroma* had to be kept separate from the HOA spots and needed a dedicated first-order decoder that used even fewer speakers. With this treatment, the result was as good or better than the previous renderings. The lesson is clear: mixed order content must not be lumped together into a single higher-order bus.

All these replays have sounded different, which is to be expected given the extreme variance in room acoustics, layout and speaker design. On more than one occasion, the author found himself longing for a

mastering step and clearly defined “industry best practices” for listening room acoustics. But even if the portability of Ambisonic mixes is not perfect, a fully satisfactory rendition could be achieved in each case, with only minor adjustments.

4 Conclusion and future work

Chroma is a perfect example for the artistically and aesthetically justified application of periphonic audio. The composer has been extremely happy with the listening experience and expressed her surprise at the precision and sound quality with which the live experience was reconstructed. She suggested some minor changes in balance where the mix strayed from her personal “favourite path” through the work, but was otherwise fully convinced of the technology and the execution. Workshop participants throughout were likewise convinced of the feasibility of the method, and found the music adequately represented.

To address the mixed-order problem, sharpening techniques such as the HARPEX decoder [16] could be evaluated to “upmix” the soundfield signals to third order.

Direct comparison of the hybrid HOA recordings with the IOSONO renderer and material created with the SPAT toolkit from IRCAM at the recent ICSA conference revealed other limitations. The virtual sources of *Chroma* never seemed to move more than a few meters away from the speakers, whereas the IOSONO system had no trouble vanishing the walls of the listening room altogether and suggesting a room that was vastly larger. The IRCAM SPAT workshop demonstrated the necessity of going for higher orders: it drove the rig at its maximal resolution, which resulted in greatly reduced phasing, and emphasized the need for a flexible room synthesis algorithm in the HOA recording tool chest.

Further study is necessary to determine the role of the floor reflection picked up by the spot mike, which might dominate distance perception and might account for the observed lack of depth (see Rumsey [17]). It could be eliminated by absorbers on the floor, or by designing a suitable inverse filter.

After clearing out all the equipment, we enjoyed a gorgeous live rendition of *Chroma*, free from technical anxieties, as part of the festival crowd. And when the moon rose over the double bass player while the nightingales were singing and the summer scents of the garden drifted by, it became all too obvious that even the most sophisticated recording techniques can be ridiculously inadequate.

I like to point out that we did get the nightingales, though.

Acknowledgements

Chroma XII was composed and directed by Rebecca Saunders, commissioned by Kunst-festspiele Herrenhausen 2010 and performed by musikFabrik Landesensemble NRW e.V.

Live recording by Jörn Nettingsmeier and Florian Faber, mixed by Jörn Nettingsmeier.

Thanks to:

- Florian Faber, without whom the project would have been dead in the water
- Stephan Flock and Claudio Becker-Foss for generous hardware support
- Rebecca Saunders and Mark J. Baden for their friendly cooperation despite the tight rehearsal schedule
- Lukas Hellermann, Michael Bölter, Martin Schmitz and all the musicians of musikFabrik for their curiousity and open-mindedness
- Lilly Marie Weber at Kunstfestspiele Herrenhausen for the permission to record
- Florian Trötschel and crew at Galerie Herrenhausen for competent and friendly local tech support
- Technicolor Hannover for helping hands, interesting research and useful gear
- Winfried Ritsch, Matthias Franz and Thomas Musil at IEM for the invitation to work there, and for their patience and support
- Peter Plessas for a very well-tuned concert P.A. system at dafx10
- Pierre-Alexandre Tremblay and Prof. Michael Clarke at Huddersfield University for the invitation to work at the SPIRAL
- Fons Adriaensen for a huge body of excellent free Ambisonic software, lovingly hand-crafted decoding matrices, and for answers to countless questions.

Recorded and mixed with free software, thanks to all authors and project contributors.

Financial support by musikFabrik, Technicolor Hannover, and IEM Graz.

[1] Jörn Nettingsmeier, 2010: “Field Report: A pop production in Ambisonics”, in: Proceedings of the Linux Audio Conference 2010, Hogeschool voor de Kunsten, Utrecht
http://stackingdwarves.net/public_stuff/linux_audio/lac2010/Field%20Report-A_Pop_production_in_Ambisonics.pdf
[2] Jörn Nettingsmeier, 2009: “Shake, Rattle, and Roll: An attempt to create a "spatially correct" Ambisonic mixdown of a multimiked organ concert ”, in: Proceedings of the Linux Audio Conference 2009, Casa della Musica, Parma
http://stackingdwarves.net/public_stuff/linux_audio/ambisonic_organ_remix/joern_nettingsmeier-shake_rattle_and_roll.pdf
[3] from an interview on the website of Huddersfield Contemporary Music Festival 2010,
http://www.hcmf.co.uk/Pulling-threads-of-sound-Rebecca-Saunders-interviewed
[4] previous three paragraphs taken verbatim from:
Jörn Nettingsmeier, 2010: “Reconstructing *Chroma XII* by Rebecca Saunders – a recording experiment”, program note for a concert replay at dafx 2010 in Graz, Austria.
http://dafx10.iem.at/concert/rebecca-saunders-and-joern-nettingsmeier.html
[5] http://musikfabrik.eu
[6] Paul Davis, Stephane Letz, et al.: the JACK audio connection kit, http://jackaudio.org
[7] Paul Davis et al.:Ardour digital audio workstation, http://ardour.org
[8] Fons Adriaensen: TetraProc, http://kokkinizita.linuxaudio.org
[9] Fons Adriaensen: jconvolver convolution engine, http://kokkinizita.linuxaudio.org
[10]Kjetil S. Matheussen: jack_capture, http://archive.notam02.no/arkiv/src/?C=N;O=A
[11]Florian Faber: midiremote remote control application for RME Micstasy and ADI-8, available from the author on request
[12]Fons Adriaensen, AMB plugins, http://kokkinizita.linuxaudio.org, third-order extension contributed by the author
[13]Franz Zotter, Hannes Pomberger, and Markus Noisternig, 2010: “Ambisonic Decoding with and without Mode-Matching: A Case Study Using the Hemisphere”, in: Proceedings of the 2nd International Symposium on Ambisonics and Spherical Acoustics, Paris.
http://ambisonics10.ircam.fr/drupal/files/proceedings/keynotes/K3.pdf
[14]Paul Mac, 2009: “A Multi-channel Experiment at Huddersfield University”, Audio Media Magazine, June 2009.
http://www.genelec.com/news/132/358/A-Multi-Channel-Experiment-at-Huddersfield-University/
[15]International Conference on Spatial Audio 2011, Detmold, Germany. http://icsa2011.org
[16]Svein Berge and Natasha Barrett, 2010: “High angular resolution planewave expansion”, in: Proceedings of the 2nd International Symposium on Ambisonics and Spherical Acoustics, Paris.
[17]Francis Rumsey, 2001: “Spatial Audio”, Focal Press, p.35

From Jack to UDP packets to sound, and back

Fernando Lopez-Lezcano
CCRMA, Stanford University
Stanford, CA, USA
nando@ccrma.stanford.edu

Abstract

The Mamba Digital Snakes are commercial products created by Network Sound, that are used in pairs to replace costly analog cable snakes by a single Ethernet cable. A pair of boxes can send and receive up to 64 channels at 48KHz sampling rate packed with 24 bit samples. This paper describes the evolution of jack-mamba, a small jack client that can send and receive UDP packets to/from the box through a network interface and transforms it into a high channel count soundcard.

Keywords

Jack, UDP, Networking, Soundcard, USB

1 Introduction

The jack-mamba program is the offshoot of a project that is still in progress, with the goal of creating a small trial Wave Field Synthesis system at CCRMA (a 32 channel system for experimentation and possible future expansion).

The conventional solution in most current WFS systems is to use high channel count PCI or PCIe sound cards, usually from the RME MADI family. The MADI output of the soundcard (64 channels) is then split into 8 ADAT 8 channel ports with a MADI to ADAT bridge, and each ADAT lightpipe is fed into 8 channel ADAT D/A converters that drive the speakers. This solution is proven and reliable, easy to synchronize across multiple hosts using a word clock signal, but expensive. For a base 32 channel system the price hovered around $180 per channel (64 channel systems would have a lower cost per channel, of course).

We though it was worthwhile to explore other non-conventional solutions that might lower the cost of the system.

2 A network sound card

Another possibility would be to eliminate the sound card entirely and explore delivery of audio through network packets to custom built boxes that would include a network port and D/A converters. There are already many systems that deliver high quality audio over Ethernet, but most of them use proprietary protocols (a Wikipedia article includes pointers to most[1]). Of course there is also the very complete and complex IEEE 1722 protocol[2]. This topic (an Ethernet “soundcard”) has also surfaced on the LAD, LAU and Jack mailing lists several times on recent years (see, for example, a long thread in 2009[3] started by Will Godfrey - Folderol, another one in mid-2011[4] by Dan Swain). As far as I'm aware this has not yet crystallized into working hardware.

3 The Mamba digital snake

A local Silicon Valley company (Network Sound) has created a family of products that replace high cost analog cable snakes with A/D and D/A boxes connected through a single CAT5 Ethernet cable (the Mamba Digital Snake [5][7]). A pair of boxes can interconnect two locations with up to 64 channels of 48KHz low latency 24 bit audio through a dedicated Ethernet link. Their solution is dedicated and point to point, and thus does not implement resource discovery, overall sampling rate synchronization and hence the communication protocol is very simple.

We thought it would be interesting to explore the

possibility of using one half of a 32 channel digital snake as a "soundcard". A rough cost estimate for that solution seemed to be about 1/3 of the traditional soundcard solution (this ignores for now the problem of synchronizing multiple units driven by different computers so that all the analog outputs are sample synchronous, a requirement of a WFS system).

The box is controlled by an FPGA, has 32 A/D and D/A converters and analog I/O ports, and an Ethernet connector. It is meant to be driven by an identical system, and when running as a slave it recovers its audio clock from the timing of the received UDP packets. The formatting of the UDP packets is very simple and there are no protocols to implement. We just need to send UDP packets with the proper timing and contents to get 32 analog outputs. Network Sound was kind enough to provide us with the packet format specification they receive and transmit.

3.1 The software

We started looking around for open source free software we could use as a starting point. The first candidate was Jacktrip, a system designed at CCRMA to do multichannel high quality long distance audio collaboration[6]. A we started reading the source we found that it was a complex system, difficult to tweak for our purposes, and mostly written using Qt classes, with which the author was not familiar (the original goal was to write a simple command line jack client using plain C).

In the meanwhile LAC 2011 took place and Jörn Netingsmeier pointed out the existence of *jack.tools* by Rohan Drape[7], a set of GPL jack command line tools. One of them is jack.udp and it seemed to be an almost perfect fit for the task at hand:

"jack.udp is a UDP audio transport mechanism for JACK. The send mode reads signals from a set of JACK input ports and sends UDP packets to the indicated port at the indicated host at a rate determined by the local JACK daemon. The "recv" mode reads incoming packets at the indicated port and writes the incoming data to a set of JACK output ports at a rate that is determined by the local JACK daemon. "

After looking at the source (simple C, very readable) it seemed that it would be relatively simple to modify the send and receive functions and adapt them to the packet and sample formats needed by the Mamba box, replacing the sending side of the digital snake with a general purpose Linux computer.

3.2 Initial tests

The Mamba box uses a simple packet format, in the case of the 32 channel version each UDP packet holds 8 32 channel 24 bit audio frames. Our jack client program would have to send packets at the rate of one packet every 166.67 uSecs.

The jack.udp program has (as a sender) a separate thread for sending UDP packets, which is fed samples from the Jack callback function by a lock-free ring buffer. The Jack callback pushes its samples into the lock-free ring buffer and wakes the UDP send thread through a write to a pipe. The UPD thread, which was waiting on the pipe, reads the samples in packet-sized chunks from the ring buffer and sends them out to the proper IP address and port, where normally a second instance of jack.udp in receive mode does the opposite.

It was easy to replace the packet assembly and disassembly functions with versions that packed 24 bit samples into 8 channel frames with the proper header information. We were able to test sending and receiving audio between two computers as with the original program, except that in this case the formatting of the packets corresponded to the Mamba specifications.

In May 2011 we visited Network Solutions for a first test. We used the Ethernet port of a laptop configured as a statically addressed interface to send the packets.

The first try were less than successful as the UDP packets seemed to go nowhere. Tcpdump made it was easy to see why, as nobody was answering ARP requests. The Network Sound engineers burned a different version of the FPGA code that included ARP generation, and we were able to send packets to the box. For these initial tests sending on the box was

disabled (it only received UDP packets), and internal buffering was maximized to make it as robust as possible to latency problems in the jack client program.

After that, we managed to send a sine wave to the Mamba box and got a clean output, albeit with quite frequent dropouts. But we knew that the packets were getting to the box, and that the formatting of the packets and the samples was correct.

3.3 Clock jitter issues

There was a mismatch between the modified jack.udp and the internals of the Mamba box. All UDP packets packed with samples from a Jack callback were sent together at the beginning of the callback as in the original jack.udp code. As the Mamba box recovers its audio clock from the timing of the received UDP packets the uneven timing creates jitter in the recovered audio clock.

To see if the problem could be minimized we added delays between packets to space them more evenly within a jack cycle. The timing on the receiving end was definitely better after that change.

The fact that the modified jack.udp is a normal Jack client was also going to have extra second order jitter effects. The absolute starting point of the Jack callback in time could move from period to period as Jack clients enter or leave the graph, or when connections are changed so that the order of execution of existing clients change. Those changes, while minimal, would also have an effect in the transmit timing of the UDP packets. In addition to that, the wakeup latency of the UDP send thread itself would add an additional (small) delay to the first packet being sent out in a given Jack period.

As a first approximation the code was good enough to verify that the network stack and high resolution timers in Linux could deliver packets reliably to the box every 166.6 uSecs.

3.4 More testing

Another test on June 1st was much more successful. The glitches that had plagued the previous test were the result of a coding error. The UDP send thread was actually not running in the SCHED_FIFO scheduling ring (booo!). With that fix in place the audio output was much more stable.

We received a loaner box from Network Sound with the custom FPGA code and we were able to keep testing at CCRMA, this time using a desktop machine with a second network interface dedicated to the audio link. This combination did much better than the laptop and there were almost no dropouts even when loading the computer and transferring big files through the regular eth0 interface.

3.5 Receiving audio

Receiving packets from the Mamba box did not work and it took a long time to debug the problem. We could see packets being received by the Linux machine with tcpdump, but nothing was received by the program. Eventually we found out by using netstat that the packets being sent by the Mamba box were not legal (we could see the network stack error counters being incremented). A hex disassembly of a dumped packet confirmed that the checksum was wrong. A bug in the FPGA code, which was promptly fixed by the Network Sound engineers.

Now we were at the end of June. Another flashed FPGA and a test at Network Sound and we managed to receive audio from the Mamba box with one instance of the modified jack.udp program. A jaaa process showed a perfect sine wave being received at our end (one of the channels of the Mamba box was being fed by a high quality audio analyser test signal).

3.6 Looking at the recovered audio clock jitter

We fired another instance of the modified jack.udp to *send* packets to the mamba box. And almost immediately jaaa showed visible side bands around the single spike that represented the sine wave. Puzzling, until one of the engineers (I forget her name) pointed out the obvious cause, this was the confirmation that jitter in the received packets in the Mamba box caused jitter in the recovered audio clock. See below for measurements.

The jitter in the recovered audio clock is not a problem in the real digital snake as the timing of the packets sent by the master box is hardware driven.

3.7 Thread priorities

It is necessary to optimize the priority of all SCHED_FIFO threads involved in this network based "audio stack".

As the timing of the outgoing packets is critical to minimize audio clock recovery jitter, the UDP send thread runs with a priority that is higher (by one) than the priority of the main SCHED_FIFO thread of jackd. Receiving packets is very important but the timing is not so critical, so that the UDP receiving thread runs with higher priority than the jackd client thread of jack-mamba (it has less priority than sending).

There are two additional sets of threads that can use priority tweaking, although the result is not as drastic as the main UDP send and receive threads inside the modified jack-udp program.

As in the case of Jack, the IRQ thread for the hardware Ethernet interface dedicated to audio can be raised in priority. It should be higher than the priority of jackd but probably lower than the priority of the IRQ thread that takes care of the soundcard hardware (more experimentation is needed to fine-tune this – this will probably depend on the reliability of the hardware drivers for sound and network packets).

The kernel also has network transmit and receive software interrupts that run with SCHED_FIFO priority. They can also be optimized to run at higher priority than other software interrupts.

4 Getting rid of the pipe

With the previous tests we had confirmed that it was possible to send and receive packets from a stock Linux machine running an RT patched kernel with very occasional packet delays. Almost all of the delays were short enough that the buffering inside the Mamba box could cover for them.

To get better performance and to minimize the jitter in the recovered audio clock we had to change the internal architecture of the original jack.udp program (now renamed jack-mamba). We wanted to do away with the pipe based wakeup calls that the jack callback function used to drive the UDP send thread. After all, the UDP send thread needed to send packets at a regular rate that should be locked to the sampling rate of the soundcard being used by Jack.

If the UDP send thread were to send all packets at exactly the right *time,* there should be no need for wakeup calls, the Jack callback could feed the ring buffer and the UDP send thread could empty it at exactly the same rate, and both halves would not need any additional synchronization. Enough extra buffering in the ring buffer in the form of a fixed time offset between arrival of samples to the Jack callback and delivery of those samples in UDP packets would provide for a cushion against latency variations in the Jack callbacks and uneven scheduling latencies in all processes involved, including the UDP send thread.

Based on Jack time (*jack_last_frame_time* tells us when the current Jack cycle started) we can know exactly when the next packet should be sent. The UDP thread can then use absolute time *clock_nanosleep* calls instead of relative *nanosleep* waits for delivering the packets at the right time. And we can pass the absolute time stamp for each packet in the same ring buffer that sends samples from the jack callback to the UDP send thread.

The UDP thread is now an infinite loop that checks for availability of data in the ring buffer in each iteration. If there is enough data available to fill one UDP packet, it reads it together with the absolute time when it should be sent, and waits with a call to a *clock_nanosleep* delay (with a TIMER_ABSTIME argument). The packet is then sent at the right time and the loop is iterated again.

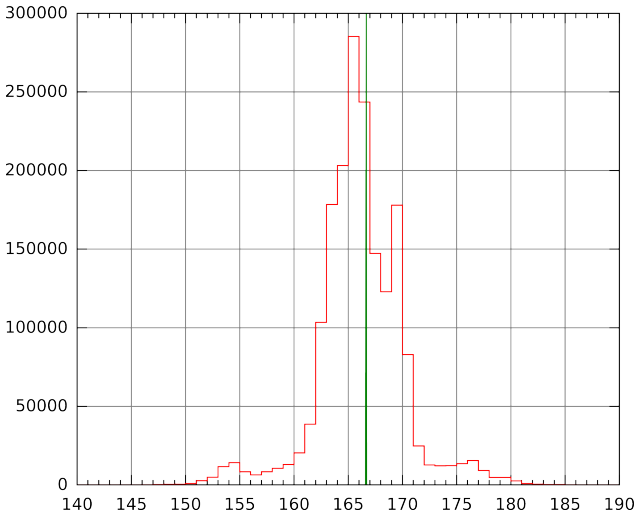
If there is not enough data available in the ring buffer the thread waits using a relative *clock_nanosleep* call that delays the thread for the duration of the pre-calculated inter-packet delay.

This can happen when there is a Jack xrun or the Jack callback thread gets delayed for a long time. After the wait the loop is iterated again.

In this way there is no additional scheduling delay for the first packet at the beginning of a jack cycle due to the pipe wakeup, all packets are scheduled at the right time (in the steady state condition), the scheduling time only depends on the start of each Jack cycle and we can add a time offset for additional buffering (and latency) in the ring buffer.

The jitter in packet delivery will of course be impacted by any scheduling delays in the UDP send thread, but with a properly configured RT patched kernel and correct IRQ priorities it works fine.

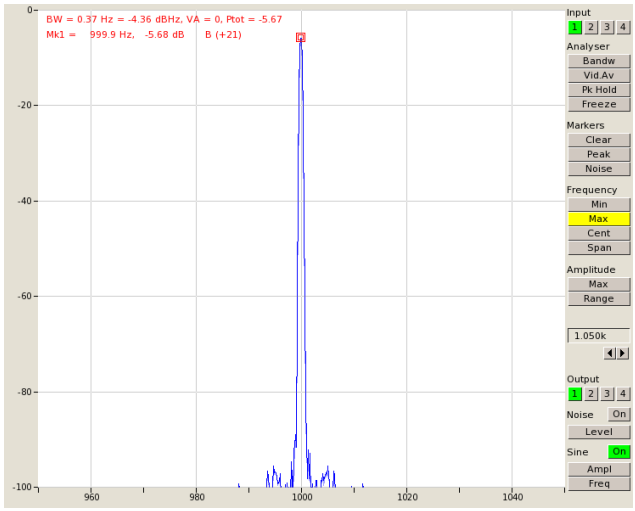
The next graph shows inter-packet scheduling delay measurements using tcpdump (with inter-packet time stamps, the "-ttt" option) on the sending machine over a period of 5 minutes, The X axis is measured packet spacing (the theoretical delay in this case should be 166.667 uSecs running Jack at 48KHz with 128 x 2) and the Y axis is number of packets that fall in a particular microsecond-wide bin:



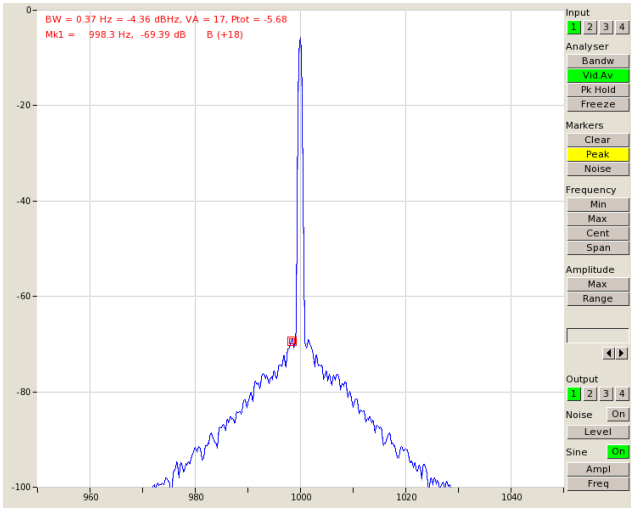
Time delay between sent packets

Even with a properly tuned system there are still occasional cases of long scheduling delays. A more complete investigation needs to be done, adding code to trigger Jack stack traces in those cases to try to see why the scheduling latencies are happening.

The figure below show a 1KHz sine wave from a Minirator MR-PRO generator being fed into channel #1 of the Mamba box and received by a single instance of jack-mamba:



The next figure shows the effect of recovered audio clock jitter due to received packet timing differences when we add another instance of jack-mamba sending silence to the Mamba box (no other changes). This is using jaaa peak hold and averaging functions to see the worst long term distortion products:



The bandwidth of the measurement is very small

(0.366Hz) and the distortion products are at or below 70dB below full scale. These measurements were taken on a workstation with quad core Q9450 Intel processors @ 2.66GHz, running Fedora 14 with a 2.6.33.19-rt31 kernel while logged into an NFS served account through the normal eth0 interface.

4.1 Managing latency

The fact that there are elastic ring buffers between the reception and transmission of the UDP packets, and the Jack callbacks means we have potentially variable input and output audio latencies.

The number of samples stored by the Jack callback in the outgoing ring buffer determines the additional output latency, and can be controlled. The number of received UDP packets stored in the incoming ring buffer determines the additional input latency. This is set at program start time by discarding packets until the latency corresponds to the desired amount. Currently there is no way to detect dropped incoming packets as there is no packet sequence information being sent by the Mamba box in the header of the packets. More work needs to be done in this regard.

4.2 Jack xruns

If an xrun occurs then the synchronization between the jack callback thread and the UDP send thread is broken. When that happens the reference timing for the two threads is recalculated so that they are synchronous again.

4.3 Error reporting

To report errors to the main program we added another lock-free ring buffer that communicates error messages back to the main program, it samples the error queue and reports statistics to the console.

To be able to properly report errors with readable timestamps we found it necessary to record the offset between the return of the *time* system call and that of *clock_gettime* at the startup of the program to compensate the printed values for an offset between the two clocks.

5 Network packet priorities

Linux provides several ways to prioritize network traffic and we tried to add those options to the program to further optimize performance and timing of the packet delivery.

The first one is setting a higher socket priority for the socket being used to transmit the UDP packets (SO_PRIORITY) using *setsockopt*. We chose the highest priority that we could use without having to run the program with special privileges (6).

The second one is to change the Type of Service (IP_TOS) of the packets. For this option it is necessary for the executable to have special permissions so it is not so easy to use. If that option is used and the function returns an error, the program suggests setting the executable to have the appropriate capability (if file based capabilities are available, of course).

6 Trial by fire

The Transitions Concert, our end of summer concert, was going to have a first night (on September 28th) curated by the author of this paper with a new outdoors sound system that used 16 main speakers in a 3D setup and 4 sub-woofers. Our goal was to have a reliable system that could drive all 20 speakers from the 32 channel Mamba D/A box.

We used a Quad Core Intel machine running Fedora 14 with a 2.6.33.14-rt31 patched kernel. A second PCI express Intel dual port Ethernet card was installed and provided a dedicated point to point connection to the Mamba box. The computer was left in the server room and just three Ethernet cables connected the system to the outdoor concert venue. Two were used to remote the monitor and USB peripherals, and the third one connected the workstation to the Mamba box. Jack-mamba worked perfectly. No xruns or glitches in the 1 1/2 hours the concert lasted.

7 A 32 channel USB soundcard

When using laptops it is normally desired to be able to use the wired Ethernet interface to have a fast network connection to the Internet. While it may be possible to share the interface with the audio packets using proper routing, it would be much better to have

a second Ethernet interface dedicated to audio.

We tried to use a USB 2.0 to Ethernet dongle and the results were very promising. We configured the Ethernet interface using the Network Managwer GUI in a Fedora 14 installation to be tied to the hardware address of the dongle, and use the proper fixed IP address that the Mamba box sends to. We also set up routing so that the default route for Internet access is not changed and the new interface does not interfere with existing network connections.

Once the interface is properly set up, it is possible to plug in the dongle, wait until it is discovered by Network Manager, and start jack-mamba right away. It is almost as easy to use as a real USB soundcard. The dongle can even be disconnected and reconnected and audio starts streaming again with no problems when the interface is again detected.

We could also create udev rules so that the priority of the USB IRQ thread in RT patched kernels is changed from the default when the dongle is connected. No tests have been made on what happens when the USB dongle shares the internal chipset USB hub with other USB ports and peripherals. Maybe this solution will not be workable in all cases.

8 Processor load

On an Intel Quad Core workstation running at 2.66GHz the jack-mamba process uses around 15.9% of a CPU (split into the three main threads that use 7.3%, 5.3% and 3.3%), the IRQ thread for the Ethernet card uses 3.6% and the network software IRQ (sirq-net-rx/1) uses 9.3% (for a total of approximately 28.8%). This is one disadvantage of this “soundcard” as it uses more processing power than a standard PCI[e] interface.

Using the USB Ethernet interface instead, the total load of jack-mamba is 17.9%, the ehci_hcd IRQ thread for the USB interface uses 9%, the uhci_hcd 2%, and assorted sirq-net* and tasklets use 8.8% for a total of around 37%.

While a USB based Ethernet dongle uses 30% more CPU than a PCI express Ethernet interface it is not unreasonable for such an interesting solution. Of course the actual numbers in both cases will change slightly with different hardware drivers.

9 Creating a networked audio hub

Another feasibility test we performed was to run in the same machine jack-mamba and instances of jack_netsource (the netone version of netjack) tied to another dedicated Ethernet port with a DHCP server running on it. With this setup a computer can use the workstation as a networked audio hub and connect to the Mamba box through a network interface. This is the scheme the author uses to implement network audio connectivity in his OpenMixer project (albeit with regular PCI based RME soundcards delivering audio to the speakers).

10 Future development

Currently the determination of the spacing between outgoing UDP packets is based just on the absolute times of the last two jack cycles, a more precise determination should be arrived at, either based on a moving average of jack period lengths, or a proper phase locked loop[8].

A very desirable next step would be to convert the jack client program into a proper jack backend. We would need a stable clock source to drive the audio packet delivery.

A simple option would be to switch the Mamba box into master mode, and use the received UDP packets to trigger the sending of the transmitted UDP packets. In this way the sampling rate would be defined by the Mamba box itself and there would be no jitter noise added due to audio clock recovery.

A Mamba Jack backend would transform the Mamba box into a real "sound card". And a USB to Ethernet dongle would make this "sound card" independent of the availability of a free Ethernet port in the machine (USB ports are plentiful).

The jack-mamba program currently has support for 16, 32, 48 and 64 channel Mamba boxes but we have only tested it with the 32 channel version. It remains

to be seen if a 64 channel box can be made to run reliably and if the CPU load of the network stack processes is low enough to make it practical.

Another task is devising a synchronization scheme so that multiple boxes can have sample synchronous output (maybe impossible). There are protocols that can synchronize the clock of several computers very accurately, perhaps with enough resolution to be able to send UDP packets from different computers to different Mamba boxes at the same absolute time. But the ring buffer induced latencies and the possibility of dropped packets could make this unreliable. Word clock I/O on the Mamba boxes could alleviate these problems.

We also need to see if Network Sound can add sequence numbers to the Mamba UDP packet headers to enable us to detect missing or out of sequence packets. It would also be neat for the box to be able to copy a word from the incoming header to the outgoing header as this would make it possible to try to detect outgoing packet loss.

The newest products from Network Sound can be programmed through special UDP packets (including the send and receive IP addresses and ports, master mode, number of channels, etc). We need to write a small program that can access that functionality.

11 Conclusion

A Jack client has been presented which sends and receives UDP packets from one half of a Mamba digital snake box and slaves its sampling rate to a n existing soundcard. It has been tested with a 32 channel box with no problems in real life situations (concert diffusion), and works even through a cheap USB to Ethernet adapter. It runs in a regular Linux workstation with no special tweaks other than an RT patched kernel and proper IRQ thread priority optimization.

While solving the original problem (a low cost network based delivery of audio for WFS systems) is not yet done, this simple program and its planned migration to a full Jack backend could be useful nevertheless for situations where a high channel count soundcard is desired with low cost.

Jack-mamba will be made available under the GPL in time for LAC2012.

12 Acknowledgements

It would have taken a much longer time to write this program from scratch if it were not for Rohan Drape, his jack.tools package and jack.udp. Of course without Jack itself and all the wonderful audio programs that can use it this would not have been possible either. Many thanks to all the members of the Linux Audio community!

It would also have been impossible to do this without the enthusiastic support of CCRMA and Chris Chafe (CCRMA's Director), and without the help of all the people at Network Sound.

References

[1] Wikipedia: *Audio over Ethernet*
http://en.wikipedia.org/wiki/Audio_over_Ethernet

[2] Wikipedia: *Audio Video Bridging*
http://en.wikipedia.org/wiki/Audio_Video_Bridging

[3] LAU/LAD: “FOSS Ethernet soundcard” thread (Will Godfrey - Folderol):
http://linuxaudio.org/mailarchive/lau/2009/11/23/162370

[4] Jack-Audio-Devel: “Ethernet-based audio interface using (net)jack idea” thread (Dan Swain):
http://permalink.gmane.org/gmane.comp.audio.jackit/24568

[5] Network Sound: Mamba Digital Snake:
http://www.networksound.com/Digsnake.html

[6] Jacktrip and SoundWire:
https://ccrma.stanford.edu/groups/soundwire/software/jacktrip/

[6] Rohan Drape, jack.tools:
http://slavepianos.org/rd/

[7] Network Sound: Mamba Audio Streamer:
http://www.networksound.com/Mamba_AS.html

[7] Using a DLL to filter time, Fons Adriansen,
http://kokkinizita.linuxaudio.org/papers/usingdll.pdf

Controlling adaptive resampling

Fons ADRIAENSEN,

Casa della Musica,
Pzle. San Francesco 1,
43000 Parma (PR),
Italy,
fons@linuxaudio.org

Abstract

Combining audio components that use incoherent sample clocks requires adaptive resampling - the exact ratio of the sample frequencies is not known a priori and may also drift slowly over time. This situation arises when using two audio cards that don't have a common word clock, or when exchanging audio signals over a network. Controlling the resampling algorithm in software can be difficult as the available information (e.g. timestamps on blocks of audio samples) is usually inexact and very noisy. This paper analyses the problem and presents a possible solution.

Keywords

Jack, ALSA, network audio, resampling

1 Introduction

Adaptive resampling is required when combining audio hardware running at incoherent sample rates into a single system. Incoherent here means that the clocks are not derived from the same source. Although the nominal sample rates are known, their ratio is not exact and may even drift over time. A fixed resampling ratio, e.g. 44100/48000, or even one that takes known errors into account, will sooner or later require samples to be inserted or dropped and this is in general not acceptable.

The problem arises when adding a second sound card to a Jack server, or when two machines, each having their own audio hardware but no common clock, need to exchange audio signals via a network connection.

In hardware this is a relatively simple problem to solve if both sample clocks are available or can be extracted from the data streams. A PLL is used to track the ratio error, and its output controls a variable ratio resampler. Some professional equipment includes such processing on selected digital inputs.

For a sofware solution the main problem is not the variable ratio resampling itself, but how to control it. Audio data is handled in blocks of typically a few milliseconds lenght, and the only information available to control the resampling algorithm are the timestamps for these blocks, and in some cases data provided by e.g. ALSA's *snd_pcm_avail*() and similar functions. All this information has considerable random and systematic errors, and it is by no means evident how to turn it into the required smoothly changing control signal for the resampling algorithm.

None of the currently available solutions such as the *alsa_in* and *alsa_out* clients that come with Jack really gets this right. The purpose of this paper is to analyse the problem and present a solution. The algorithms discussed in the following sections have been implemented in two new applications, *zita_a2j* and *zita_j2a* wich allow to add ALSA soundcards running at arbitrary sample rates as clients to a Jack server. Other implementations, e.g. for networked audio or allowing to link two Jack servers running on the same machine will follow.

2 Requirements

Resampling itself, even with a variable ratio, will not lead to any signficant loss of audio quality if implemented correctly. The main consequences when using fixed-bandwidth resampling (as e.g. in *libsamplerate* and *zita-resampler*) will be a small loss of bandwidth and some additional delay. Both depend on the length of the multiphase filter used by this algorithm, and the tradeoff between the two can be made by the user.

The effect of a non-constant resampling ratio de-

pends on the magnitude of the variation and on its spectrum. Very slow and small changes are equivalent to small movements of a listener w.r.t the speakers, or a performer w.r.t. the microphone. To put this in context, one sample at 48 kHz corresponds to about 7 millimeters in air. So provided such changes are limited to a few samples and very gradual they won't be noticed[1].

Larger variations, even when quite slow, may lead to perceptible delay and pitch changes which, while not really degrading the audio signal quality, may not be acceptable from a musical point of view.

But the really nasty effect of delay modulation occurs when the variations contain higher frequency components, even if these are quite small. The result no longer appears as a modulation of the signal (e.g. vibrato) but as parasitic signals, noise or distortion. Some types of sound are very sensitive to such phase modulation. This is really the equivalent of jitter on the sample clock of an A/D or D/A converter, and sadly, some of the existing implementations of adaptive resampling produce this at level that is some orders of magnitude higher than the worst hardware.

Apart from audio quality considerations there are some other aspects which are important. Both the resampling, and moving audio signals between domains with asynchronous periods will introduce latency. It depends on the application if this is acceptable or not. But at least the delay should be stable and repeatable. Stability means that it must not depend on e.g. whether a Jack client implementing the resampling runs at the start or near the end of a Jack cycle. Again, existing implementations fail to meet this requirement.

Applications implementing adaptive resampling can take up to a few seconds to stabilize their control loops, and need to restart if synchronisation is lost, e.g. when a misbehaving Jack client results in a timeout of the server. This is quite accecptable, but minor incidents such as Jack1 skipping one or a few cycles should not result in a change of the latency nor require a restart.

[1] Unless you are e.g. sending one channel of a stereo pair via the resampler and the other not, but that is really asking for trouble.

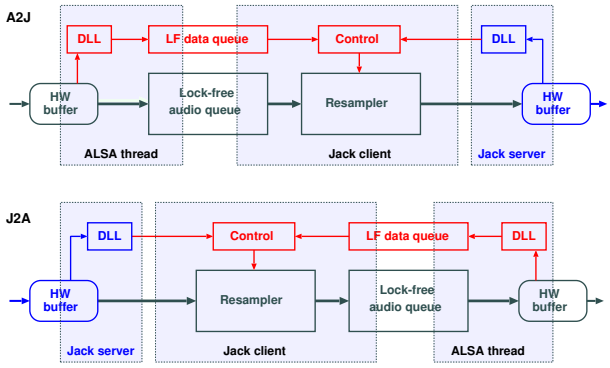


Figure 1: The structure of A2J and J2A

3 Problem analysis

Anyone trying to build an abstract picture of this sort of algorithm and to reason about it will find that it stretches his or her powers of imagination to some degree. It helps to have a particular implementation in mind. In this paper we will use the structure of *zita_a2j* and *zita_j2a* as a framework. However, the analysis is more general and applies in other cases as well.

Figure 1 shows the structure of those applications. In this figure both are shown with the Jack client connected directly to the sound card used by the Jack server — this is the setup we would use to measure the latency (using e.g. *jack_delay*) in a round-trip involving both Jack's sound card and the additional one.

Taking *zita_a2j* as an example, the ALSA thread, as regards audio processing, does little more than transfer audio frames from the ALSA device to a lock-free buffer. It also provides some extra information which will be discussed later. Keeping the ALSA thread as simple as this allows it to run at a higher real-time priority than the Jack server, and with a shorter period time, and this in turn helps to obtain accurate timing information. The actual resampling and the control logic is performed in the process callback of the Jack client. The structure of *zita_j2a* is virually the mirror image of *zita_a2j*.

3.1 Building a model

What needs to do be done is to control the resampling ratio in such a way that on average the same number of samples enter and leave the lock-free buffer. We also want to do this using only

very small and smooth changes of the resampling ratio.

Monitoring the *actual* state of the buffer (the number of frames stored in it) won't work for several reasons. The most fundamental one is that we actually can't observe the buffer state in any reliable way from either side, as the other side could be modifying it at the same time. At best we could have an upper or lower bound. Also, this value doesn't change in a smooth way, but jumps each time a period is processed on either side. And these changes occur when the code of the Jack process callback or the ALSA thread actually runs, and this moment is not at all representative of the real timing of the audio samples. For example the process callback of the Jack client can run at any time between the start of the current cycle and the start of the next one, this just depends on the position of the client in the connection graph and on the CPU load of other clients.

The key to creating a working model is to take abstraction of the period based processing, including the random timing errors, and imagine the resampling algorithm as a continuous process.

Suppose we would have two *continuous* functions of time, $W(t)$ and $R(t)$ that provide the number of samples that have been written to resp. read from the buffer at any time t. Then if Δ is the required delay of the whole process, we could evaluate the error $W(t) - R(t) - \Delta$ in each process callback and use this to control the resampling ratio. This, with some refinements and extra functionality, is the basic algorithm used in *zita_a2j* and *zita_j2a*. Remains to create those two functions. Let $J(t)$ be the function on Jack's side and $A(t)$ the one on the ALSA side. Then for *zita_a2j* $J(t) = R(t)$ and $A(t) = W(t)$, and for *zita_j2a* $J(t) = W(t)$ and $A(t) = R(t)$.

On the Jack side, part of the solution is already available in the server. The DLL *(Delay Locked Loop)* that has been part of Jack since many years computes in each cycle a prediction of the start time of the next cycle, while removing most of the jitter due to random wakeup latency. This provides a smooth and continuous mapping between time (as measured by Jack's microsecond timer) and frame counts.

So we can implement $J(t)$ by just reading the timestamp provided by Jack's DLL into t_J, and summing the number of frames read from or written to the lock-free buffer in k_J. We don't actually need the function on Jack's side, since we are only interested in its value at the start of the current period.

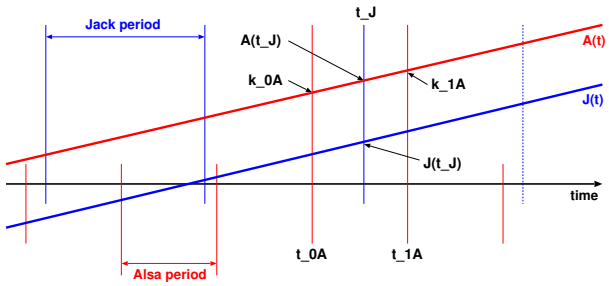


Figure 2: Delay calculation parameters

A similar DLL can be provided at the ALSA side. This requires reading the wakeup time of the ALSA device using Jack's microseconds timer and applying the DLL algorithm which is quite simple. To transfer the data to the Jack side a second lock-free queue is used. For each period the ALSA thread sends a message containing it current status, the computed timestamp for the next period, and the number of frames written to or read from the audio buffer. At the Jack side, during each process callback these messages are read and the frame counts are accumulated into a variable k_{A1}. The most recent data (t_{A1}, k_{A1}) is used, along with the same from the previous period, (t_{A0}, k_{A0}). Since the ALSA thread reports the wakeup time of its *next* cycle, the interval (t_{A0}, t_{A1}) includes the current wakeup time at the Jack side (or in the worst case one of the endpoints will be close), so a simple interpolation is all that is needed to compute $A(t_J)$. Figure 2 shows the relevant parameters for the case *zita_a2j*.

3.2 Resampler delay

The analysis so far has ignored the delay introduced by the resampling process itself. There are two aspects to this. First, this latency can be significant and it must be taken into account when defining the target value. Second, when using k_J, the number of frames transferred between the resampler and the audio queue, as the value of $J(t)$ we are actually making an error. $J(t)$ should increase by exactly the same delta in each period, the Jack period size either multiplied or divided by the resampling ratio, but it doesn't because it

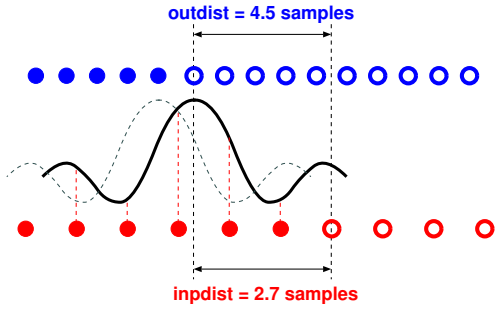


Figure 3: Resampler latency

is constrained to be integer. The sum will be exact on average, there is no long term accumulating error, but we are missing the fractional part.

That fractional part is actually represented by the internal state of the resampler. To see this we will use *zita-resampler* as an example. Constant bandwidth resampling works by evaluating a FIR filter (a near 'brickwall' lowpass wich corresponds to a windowed *sinc*() function in the time domain) for each output sample. The central peak of the impulse response corresponds to the current output sample, and the actual coefficients used depend on the position of the filter w.r.t. the input samples. This is shown in Fig. 3, using a very short filter. Actual resampling filters are much longer.

When *zita_resampler* finishes processing a block of frames it remains in a state ready to compute the next output sample, except that it may have to read one or more input samples before it can proceed. In Fig. 3 the bottom (red) dots represent input samples and the top (blue) ones the output. Solid dots are samples already used or computed. The filter is aligned with the next output sample. In this example one more input sample is required to compute the next output, the first non-solid one in the figure. This will not always be the case, for example if the previous call terminated because the output buffer was full it could be that the next output doesn't require a new input sample. But in any case the distance between the next input sample and the next output one is well-defined and it can be expressed in either the input or output sample rate. This value includes the fractional part that is missing from k_J.

The *Vresampler* class used in *zita_a2j* and *zita_j2a* provides a function *inpdist*() which returns the current value of this delay at the input sample rate, and this is used to correct the value of k_J. One may ask if such a small error actually matters. This depends on the nominal resampling ratio. If this is not the quotient of two small integers then the fractional error is a pseudo-random value and its effect willl be removed by the loop filter that controls the resampling ratio. The worst case results if the two sample rates are nominally the same. The error will be a sawtooth function with a frequency equal to the difference between the two actual sample rates. In this case the loop filter may not completely remove it. The effect was visible in test results of early implementations that did not include the correction in the error calculation.

3.3 Closing the loop

Combining the elements presented above we can now formulate the equations giving the delay error for both cases. Let γ be the resampling ratio, d_{res} be the value returned by the *inpdist*() member of the resampler, Δ the target delay value and $t = t_J$ the start time of the current cycle, then

$$\begin{aligned} E_{A2J} &= W(t) - R(t) + d_{res} - \Delta \\ E_{J2A} &= W(t) - R(t) + d_{res} * \gamma - \Delta \end{aligned}$$

Using the definitions of $W()$ and $R()$, and setting

$$d_A = A(t_J) = [k_{A1} - k_{A0}] \frac{t_j - t_{A0}}{t_{A1} - t_{A0}} \qquad (1)$$

these become

$$E_{A2J} = [k_{A0} - k_J] + d_A + d_{res} - \Delta \qquad (2)$$

$$E_{J2A} = [k_J - k_{A0}] - d_A + d_{res} * \gamma - \Delta \qquad (3)$$

This error is first processed by a second order lowpass filter and then becomes the input to the second order loop filter. This is very similar to the one used in Jack's DLL, see [Adriaensen, 2005]. The first filter is added to further reduce phase modulation by high frequency noise on the error value. Its bandwidth is 20 times that of the loop filter, so it does not affect stability. The resampler code adds another lowpass to smooth ratio changes. Also this one must be dimensioned so it does not affect operation of the loop.

The values k_J and k_{A1} are obtained by accumulating differences in each cycle. To make the equations above represent the actual round trip delay

error we need to initialise them with the correct values. Since k_{A0} is just k_{A1} from the previous cycle it needs no initialisation. A bit of arithmetic (which is left as an exercise for the reader and wich may tickle his or her powers of imagination a bit) will show that the correct initial values are

$$\begin{aligned} k_J &= -P_J/\gamma \\ k_{A1} &= P_A \end{aligned}$$

if the ALSA device is the input, and

$$\begin{aligned} k_J &= P_J * \gamma \\ k_{A1} &= -P_A \end{aligned}$$

in the other case, with P_J and P_A being the Jack and ALSA period sizes, and γ the resampling ratio.

The key to understand this is that at any time the delay must be the sum of the number of frames in both hardware buffers, the lock-free queue, and the resampler, while taking the different sample rates into consideration.

A related matter is to determine the target round-trip delay value. Some more (simple) arithmetic will show that

$$\begin{aligned} \Delta_{min,A2J} &= T_{res} + 2T_J + (1+c)T_A \\ \Delta_{min,J2A} &= T_{res} + 2T_J + 2T_A \end{aligned}$$

where T_{res} is the delay of the resampler, T_J and T_A the Jack and ALSA side period times, and c is the maximum expected wakeup latency of the ALSA thread, e.g. 1/2 when this thread is allowed to run half a cycle late. These values allow for worst case conditions, e.g. a Jack client running near the end of the cycle.

3.4 Improving the settling time

The system discussed so far will provide a constant processing delay, but with the loop running at is normal bandwidth (around 0.05 Hz in the current implementation) it could take a long time to reach the target value Δ. We can do two things to speed up convergence of the loop. One is to run the loop filter at a higher bandwidth initially. The other is to use the first delay error measurement to force a situation close to the required one, and then let the loop take care of the remaining error. This requires modifying the state of the lock-free queue. Note that once the system is running we can't set the number of frames in the queue to any specific value in a safe way (as the other side may be accessing the queue at the same time). The only thing we can do is force a *relative* change by either reading or writing a number of frames, and that is fact all we need.

If N is the delay error rounded to the nearest integer, then for the A2J case we set $k_J = k_J + N$ and read N frames from the queue, and for the J2A case we set $k_J = k_J - N$ and write $-N$ frames to the queue. Adjusting k_J removes most of the error from the model, and applying the corresponding change to the queue ensures that the model remains in sync with the actual situation.

Both example applications do remove the initial error in this way, then run the loop at higher bandwidth for the first 4 seconds.

4 Implementation details

4.1 Delay error calculation

The k_J, k_{A0} and k_{A1} values used in (1,2,3) are integers that are incremented by frame counts in each iteration, so they will overflow at some time. Since we only ever take *differences* of those, the overflow doesn't matter. But this part of the calculation *must* be done as a subtraction of integers without conversion to a floating point value, as suggested by the square brackets in the equations. The remaining parts can be done safely in floating point since these terms are recomputed each time and there will be no accumulating roundoff error.

Jack represent its microseconds timer as a 64-bit integer type. These are difficult to use in calculations so they are converted to double. To avoid loss of precision, the actual value of t_J, t_{A0} and t_{A1} is the microseconds time masked to 28 bits, and divided by 10^6 to obtain seconds. This representation will wrap around every 268.4 seconds or so. Again since we are always taking differences this is easy to detect and correct.

4.2 Error recovery and reporting

In case of a fatal error in the ALSA device or a timeout of the Jack server there is no alternative but to restart the loop initialisation. The two applications discussed here contain some state management code to allow this without having to restart the actual processes. This uses a third lock-free queue (not shown in Fig. 1) to send commands from the Jack side to the ALSA thread.

When restarting the loop will settle quite fast, as the previous resample rate correction can be reused.

The Jack1 server will occasionally skip some cycles while creating or removing clients or when making port connections. This can be detected and handled quite easily and without introducing any errors in the loop. The particular implementation of the lock-free audio queue allows it to recover from overflow or underflow conditions by just reading or writing the number of frames that were missed before. When doing this, the same adjustment is made to k_J to remove the error from the delay calculation.

A fourth lock-free queue is used to convey status and optional monitoring information for output in the main thread.

4.3 The lock-free queue

When recovering from skipped cycles the lock-free audio queue may be in an overflow or underflow condition. The same is true when removing the initial delay error as discussed in 3.4. Also, the value of N used there can be positive or negative.

The lock-free queue must implemented in a way that maintains correct read and write counters and the corresponding data pointers at all times. It must also allow logical read or write operations of any number of items, including negative ones.

Such an implementation need not be more complicated than some existing ones, and in fact it can be much simpler. The C++ class used in *zita_a2j* and *zita_j2a* uses a 2^N buffer size as would most. It maintains read and write counters as 32-bit integers. These are just incremented by the number of items the user claims to have read or written, without enforcing any limits. The read and write indices are the respective counters modulo the buffer size. Since the buffer size divides the 2^{32} range of the counters, these can be allowed to overflow without any consequence. It is the user's responsability to avoid buffer overflow or underflow if that matters, and the class provides the necessary information to make this easy, so no functionality is lost by this particular implementation.

5 Results

Figure 4 shows the result of a 1 kHz sinewave signal processed by *zita_j2a*, with the sample rates

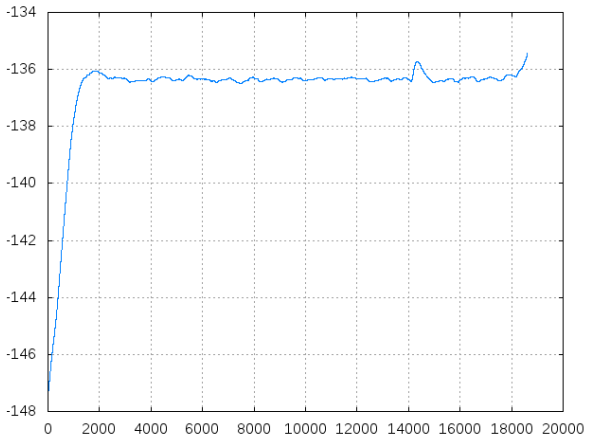


Figure 4: zita-j2a, 48.0 to 44.1 kHz

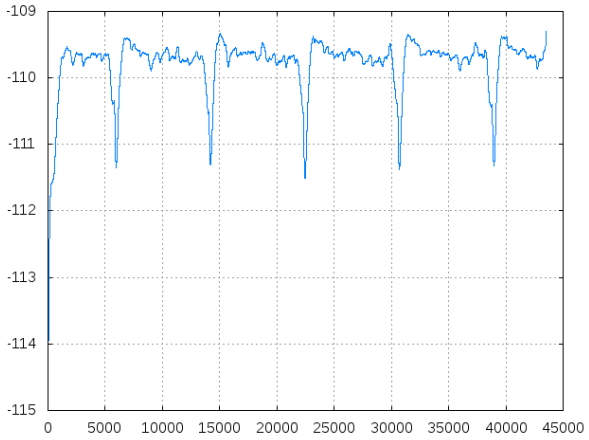


Figure 5: zita-j2a, 48.0 to 48.0 kHz

being 48 kHz at the Jack side and 44.1 kHz for the ALSA device. The Y axis is the phase of the output signal (w.r.t. to the generator) in degrees, the X-axis is in centiseconds. The loop stabilises in about 15 seconds. After that time, variations are less than 0.5 degrees peak-to-peak. One degree at 1 kHz corresponds to 2.78 microseconds. The small bump at around 145 seconds is the result of switching the desktop workspace. This measurement was done one a single CPU machine, running a standard (unpatched) kernel and having no HW video acceleration.

Figure 5 is the result of the same test but with both ends running at 48.0 kHz. There is a periodic dip every 83 seconds, which corresponds

to a frequency of around 0.012 Hz. This is the difference between the period frequencies of both soundcards. Every 83 seconds the interrupts of the two cards occur at almost the same time and compete for the CPU. This effect would probably not be visible on an SMP machine. It's harmless in practice as the delay changes are very small and smooth.

6 Acknowledgements

The results reported in this paper build on the work done by the Jack and ALSA developers. A particular thanks to the members of the respective mailing lists for the prompt answers to all my questions.

References

Fons Adriaensen. 2005. *Using a DLL to filter time*. Available at `http://kokkinizita.linuxaudio.org/papers/index.html`.

Signal Processing Libraries for Faust

Julius SMITH
Center for Computer Research in Music and Acoustics (CCRMA)
Music Dept., Stanford University
Stanford, CA 94306,
USA
jos@ccrma.stanford.edu

Abstract

Signal-processing tools written in the FAUST language are described. Developments in FAUST libraries, `oscillator.lib`, `filter.lib`, and `effect.lib` since LAC-2008 are summarized. A good collection of sinusoidal oscillators is included, as well as a large variety of digital filter structures, including means for specifying digital filters using analog coefficients (on the other side of a bilinear transform). Facilities for filter-bank design are described, including optional delay equalization for phase alignment in the filter-bank sum.

Keywords

FAUST, audio signal processing, filters, effects, oscillators

1 Introduction

The FAUST (Functional AUdio STream) language, developed at GRAME[1] [1; 2], is well known for its compact specification of signal-processing block diagrams, and its compilation into efficient C++ audio applications and plugins. Thanks to the use of architecture files that encapsulate platform-specific details, FAUST applications can be conveniently generated for a wide variety of host environments (Linux, Mac, Windows), and audio plugins can be generated for a wide variety of host applications such as Pd and SuperCollider, to name just two [3; 4; 5].

In the `architecture` subdirectory within the FAUST distribution, there are presently seven `.lib` files containing various utility functions. Possibly the most commonly used of these is `music.lib`, which also imports `math.lib`. Three other `.lib` files pertain more specifically to signal processing utilities:

- `oscillator.lib` — signal sources
- `filter.lib` — general-purpose digital filters
- `effect.lib` — digital audio effects

The remaining two FAUST library files are `maxmsp.lib`—a Max/MSP compatibility library, and `reduce.lib`—enabling function application across a signal in time, such as `maxn(n) = reduce(max,n)` to compute the maximum amplitude of a signal.

The directory `examples/faust-stk/` additionally contains `instrument.lib`, providing common utility functions for the FAUST-STK collection [6], such as envelope generators and table-lookup utilities.

The libraries `oscillator|filter|effect.lib` were first discussed at LAC-08 [7]. This paper provides an overview of developments since then and up to FAUST release version 0.9.46 (Dec. 2011).

2 Faust Library `oscillator.lib`

The purpose of `oscillator.lib` is to provide reference implementations of various elementary waveform generators, such as sinusoidal, sawtooth, and pulse-train, as well as other classic signals such as pink-noise, etc.

2.1 Sinusoid Generators

All sinusoidal oscillators in `oscillator.lib` are invoked via the same API as `osc(freq)` (defined in `music.lib`), where `freq` is the desired oscillation frequency in Hz. However, some provide *two* outputs instead of one when both "in-phase" and "quadrature" (sine and cosine) are available. All are *filter-based*. That is, they are implemented as lossless second-order filters driven by an *impulse* signal `[1,0,0,...]`, and they use no wave

[1] http://faust.grame.fr/

tables.[2] All algorithms have been previously published [8; 9; 10; 11].[3]

Presently, the following algorithms are implemented:

`oscb`	"biquad" two-pole filter section (impulse response)
`oscr`	2D vector rotation (second-order normalized ladder) provides sine and cosine outputs
`oscrs`	sine output of `oscr`
`oscrc`	cosine output of `oscr`
`oscs`	state variable osc., cosine output (modified coupled form resonator)
`oscw`	digital waveguide oscillator
`oscws`	sine output of `oscw`
`oscwc`	cosine output of `oscw`

The relative merits of each oscillator type are summarized below. Note that all differences have to do with finite numerical precision effects and dynamic range variations under time-varying conditions. The best overall choice depends on the situation.

- `oscb`, the impulsed direct-form biquad[4] is the fastest computationally, requiring only one multiplication and two additions per sample of output. However, as is well known, the amplitude of oscillation varies strongly with frequency, and it becomes numerically poor toward `freq=0` ("dc").
- `oscr`, the "2D vector rotation," requires four multiplies and two additions per sample. Its amplitude is invariant with respect to frequency, and it is good all the way down to dc. Since its coefficients are numerically inexact roundings of `s = sin(2*PI*freq/SR)` and `c = cos(2*PI*freq/SR)`, where `SR` denotes the sampling rate (defined in `music.lib`), there is long-term amplitude drift corresponding to the extent the identity $\mathtt{s}^2+\mathtt{c}^2 = 1$ is violated. This oscillator provides in-phase (cosine) and phase-quadrature (sine) outputs.
- `oscs`, based on the classic "state variable filter," [12, p. 530] and known as the "magic circle algorithm" in computer graphics, is quite fast, requiring only two multiplies and two additions per output sample. Its amplitude varies much less with frequency, and it too is good down to dc. There is no amplitude drift over time, so this one can be used for very long signal durations. On the other hand, there is some dependence of oscillation amplitude on frequency. At low frequencies, its two state variables are nearly in phase quadrature, but they become in-phase at SR/2. Thus, two outputs with approximately 90-degrees relative phase at low frequencies could be brought out. The output that is brought out is the "cosine" choice.
- `oscw`, the second-order digital waveguide oscillator, requires one multiply and three additions when frequency is constant, and another multiply when frequency is changing. Otherwise it has all of the good properties of `oscr` (except for internal dynamic range normalization), providing sine and cosine outputs in exact phase quadrature, and no dependence of amplitude on frequency. However, unlike `oscr`, `oscw` exhibits no amplitude drift while frequency is fixed. This is because it uses a "structurally lossless" algorithm derived by transformer coupling of normalized digital waveguides [10; 8].[5] A negative point relative to `oscr` is that numerical difficulties may arise below 10 Hz or so, implying that `oscw` is not a good choice for LFOs. Internally, the state variables of `oscw` require a larger dynamic range than those of `oscr`. It is likely that `oscw` would be the most economical choice for special-purpose VLSI.

2.2 Virtual Analog Waveforms

The following waveform generators are presently included, among others:

`imptrain(freq)`	periodic impulse train
`squarewave(freq)`	zero-mean square wave
`sawtooth(freq)`	alias-suppressed sawtooth
`sawN(freq)`	order N anti-aliased saw

[2] `osc(freq)` in `music.lib` uses a length 2^{16} wave table. The linearly interpolated variant `osci(freq)` adds linear interpolation.

[3] `https: //ccrma.stanford.edu/~jos/pasp/-Digital_Waveguide_Oscillator.html`

[4] `https://ccrma.stanford.edu/~jos/filters/-Direct_Form_II.html`

[5] `https: //ccrma.stanford.edu/~jos/pasp/-Digital_Waveguide_Oscillator.html`

The `sawtooth` and `sawN` algorithms are based on recently developed "Differentiated Polynomial Waveform" (DPW) methods for virtual analog waveform generation [13; 14]. The default case is `sawtooth = saw2`, where `saw2` is a differentiated parabolic waveform (order 2). More generally, `sawN` is based on a differentiated polynomial of order N. The higher the order, the less aliasing is incurred. Bandlimited square, triangle, and pulse-train are derived as linear filterings of bandlimited sawtooth in FAUST releases beyond 0.9.46.

2.3 Noise Generation

The basic white-noise generator, uniformly distributed between -1 and 1, is `noise`, defined in `music.lib`. Based on that, `oscillator.lib` also defines `pink_noise`, also called "$1/f$ noise" [15], implemented (approximately) as white noise through a three-pole, three-zero IIR filter that approximates a $1/f$ power response.[6] The third-order IIR filter was designed using `invfreqz` in Octave (matlab).

3 Faust Library `filter.lib`

Filter-related utilities are provided in `filter.lib`. The principal functions defined appear in Fig. 1, p. 4, and Fig. 2, p. 5. To save space, functions introduced at LAC-08 [7] are not repeated here (such as EKS string synthesizer elements, comb filters, cubic distortion overdrive, dc blockers, speaker bandpass, Crybaby wah pedal, etc.). The subsections below provide further discussion of various groups. The source is documented with comments and references so that anyone knowledgeable in basic digital filter theory [16] should be able to use it as a (terse) manual and starting point for further reading.

3.1 Direct-Form Digital Filters

The four direct-form digital filter structures have coefficients that appear in the filter transfer function.[7] The functions `tf1(b0,b1,a1)` and `tf2(b0,b1,b2,a1,a2)` specify first- and second-order (biquad) digital filter sections, respectively. Often larger filters are built by stringing first- and second-order sections in series and/or parallel. The FAUST language makes this especially easy to do.

The function `iir(bcoeffs,acoeffs)` allows specification of an arbitrary-order IIR digital filter in direct form. The arguments `bcoeffs` and `acoeffs` are each parallel signal banks that provide the coefficients, and they may be thought of as "lists" of coefficients. The *pattern matching* facility in FAUST allows recursive definition in terms of such lists. As an example, `tf2(b0,b1,b2,a1,a2)` can be alternatively specified as `iir((b0,b1,b2),(a1,a2))`. As usual in FAUST, the specification is compact:

```
iir(bv,av) = sub ~ fir(av) : fir(bv);
```

where `fir(bv)` specifies a general causal FIR digital filter, with `bv` the list ("vector") of FIR "tap" coefficients. It is given by

```
fir(bv,x)
  = sum(i,count(bv),take(i+1,bv) * x@i);
```

where `count` and `take` are defined in `math.lib`.

3.2 Ladder and Lattice Digital Filters

Ladder and lattice digital filters have superior numerical properties. Using the pattern-matching facility, it was possible to specify all four major types recursively in FAUST. A particularly valuable case is the *normalized ladder filter* [17] `iir_nl(bcoeffs,acoeffs)`, used as the basis for the super-robust biquad `tf2snp()`. While normalized biquads are straightforward to design (*e.g.*, `nlf2()` in `filter.lib`), the normalized ladder filter realizes rational transfer functions of any order (any number of poles and zeros) in terms of a power-normalized ladder structure. For an introduction and pointers to references, see [8] and `filter.lib`.[8]

3.3 Digital Filter Sections Specified as Analog Filter Sections

It is convenient to be able to specify basic filter section in terms of *analog* filter coefficients, as opposed to the usual digital-filter coefficients. This is easy to do in FAUST by including a built-in *bilinear transform*.[9] This makes use of the wonderful

[6] `https: //ccrma.stanford.edu/~jos/sasp/-Example_Pink_Noise_Analysis.html`

[7] `https: //ccrma.stanford.edu/~jos/filters/-filters/Four_Direct_Forms.html`

[8] `https: //ccrma.stanford.edu/~jos/pasp/-Conventional_Ladder_Filters.html`

[9] `https: //ccrma.stanford.edu/~jos/pasp/-Bilinear_Transformation.html`

Direct-Form Digital Filters	(§3.1, p. 3)
`fir(bcoeffs)`	general FIR digital filter
`iir(bcoeffs,acoeffs)`	general IIR digital filter
`tf1(b0,b1,a1)`	first-order direct-form digital filter = `iir((b0,b1),(a1))`
`tf2(b0,b1,b2,a1,a2)`	`iir((b0,b1,b2),(a1,a2))`
Lattice/Ladder Filters	(§3.2, p. 3)
`iir_lat2(bcoeffs,acoeffs)`	two-multiply lattice digital filter
`iir_kl(bcoeffs,acoeffs)`	Kelly-Lochbaum ladder digital filter
`iir_lat1(bcoeffs,acoeffs)`	one-multiply lattice digital filter
`iir_nl(bcoeffs,acoeffs)`	normalized ladder digital filter
`tf2np(b0,b1,b2,a1,a2)`	biquad based on stabilized 2nd-order normalized ladder
`nlf2(f,r)`	second-order normalized ladder digital filter, special API
Analog-Specified Filters	(§3.3, p. 3)
`tf2s(b2,b1,b0,a1,a0,w1)`	`tf2` specified via *s*-plane (analog) coefficients
`tf2snp(b2,b1,b0,a1,a0,w1)`	tf2s using a protected normalized ladder filter for `tf2`
`tf2sb(b2,b1,b0,a1,a0,w1,wc)`	`tf2s` plus a mapping to *bandpass* in the digital domain
`tf1sb(b1,b0,a0,w1,wc)`	same as `tf2sb` but for first-order filter sections
IIR Low/High/Band-Pass	(§3.4, p. 6)
`lowpass(N,fc)`	Nth-order Butterworth lowpass, -3 dB frequency at `fc` Hz
`highpass(N,fc)`	Nth-order Butterworth highpass, -3 dB frequency at `fc` Hz
`bandpass(Nh,fl,fu)`	Order `2*Nh` Butterworth bandpass, -3 dB frequencies `fl,fu` Hz
`bandstop(Nh,fl,fu)`	Order `2*Nh` Butterworth bandstop filter, -3 dB gain at `fl,fu` Hz
`lowpass3e(fc)`	3rd-order elliptic lowpass, 60 dB stopband rejection, 0.2 dB passband rip.
`lowpass6e(fc)`	6th-order elliptic lowpass, 80 dB stopband rejection, 0.2 dB passband rip.
`highpass6e(fc)`	highpass transformation of `lowpass6e` ($\omega \leftarrow 1/\omega$)
`bandpass12e(fl,fu)`	bandpass transformation of `lowpass6e`
`bandpass6e(fl,fu)`	bandpass transformation of `lowpass3e`
Shelfs, Peaking Equalizers	See FAUST example `parametric_eq.dsp`
`low_shelf1(L0,fx,x)`	1st-order shelf, dc gain `L0` dB, crossover to unity gain at `fx` Hz
`low_shelf1_l(G0,fx,x)`	dc gain `G0` (linear), crossover to unity gain at `fx` Hz
`low_shelf3(L0,fx,x)`	3rd-order low shelf
`low_shelf5(L0,fx,x)`	5th-order low shelf
`low_shelf`	`= low_shelf3; // default = third-order case`
`high_*`	same high-shelf cases as for `low_shelf`
`peak_eq(Lfx,fx,B)`	2nd-order "peaking equalizer", peak level `Lfx` dB, width `B` Hz at `fx` Hz
`peak_eq_cq(Lfx,fx,Q)`	Constant-Q 2nd-order peaking equalizer section, Q = fx/B
`peak_eq_rm(Lfx,fx,w)`	Regalia-Mitra 2nd-order peaking equalizer section, `w ~ PI*B/SR`
Fractional Delay Lines	(§3.6, p. 6)
`fdelayN(maxdelay, delay)`	`N`th-order FIR Lagrange-interpolated delay line, `N`=1,2,3,4
`fdelayNa(maxdelay, delay)`	`N`th-order IIR allpass-interpolated delay line, `N`=1,2,3,4

Figure 1: Functions defined in `filter.lib` since LAC-08. See the source code for full usage documentation and literature references.

Filter Banks	(§3.7, p. 6)
`mth_octave_analyzer(O,M,ftop,N)`	`N`-band octave filter-bank, `M` band-slices per octave, Butterworth band-split order `O` (not `0`, must be an odd integer), `N` = total number of bands (including dc and Nyquist), `ftop` = highest band-split crossover frequency (e.g., 20 kHz)
`mth_octave_analyzer6e(M,ftop,N)`	uses order 6 elliptic band-split filters
`mth_octave_filterbank(O,M,ftop,N)`	`mth_octave_analyzer` followed by delay equalizer
`mth_octave_filterbank_alt`	dc-inverted variant (cheaper for odd `O`)
`mth_octave_spectral_level`	spectrum analyzer using `mth_octave_analyzer(5)`, displays (in bar graphs) the average signal level in each spectral band
`mth_octave_spectral_level6e`	order 6 elliptic crossovers
`spectral_level`	`= mth_octave_spectral_level(2,10000,20); // simplest`
`half_octave_analyzer(N)`	`= mth_octave_analyzer6e(2,10000,N);`
`half_octave_filterbank(N)`	`= mth_octave_filterbank5(2,10000,N);`
`octave_filterbank(N)`	`= mth_octave_filterbank5(1,10000,N);`
`octave_analyzer(N)`	`= mth_octave_analyzer6e(1,10000,N);`
`analyzer(O,lfreqs)`	general analyzer, order `O` Butterworth crossovers at listed freqs
`filterbank(O,lfreqs)`	`analyzer(O,lfreqs) :` delay equalizer (allpass-complementary)
`filterbanki(O,lfreqs)`	Inverted-dc variant

Figure 2: Filter-bank functions defined in `filter.lib`. See the source code for full usage documentation and literature references.

feature of FAUST that if the coefficients are constant, all expressions will compile away to leave numerical digital-filter coefficients. On the other hand, if a slider-control, say, is providing an analog coefficient, the bilinear transform will be computed in real time (at the control rate) from the controller by the compiled result. Normally a one-pole smoother such as `smooth(0.99)` is used to interface the final computed coefficient into the filtering computation at the full sampling rate.

In particular, `tf2s(b2a,b1a,b0a,a1a,a0a,w1)` equals `tf2(b0d,b1d,b2d,a1d,a2d)` specified in the analog domain, where a last-letter 'a' means 'analog', and 'd' means 'digital'. (Note the opposite numbering of the coefficients, in conformance with typical notation.) Thus, the analog transfer function specified is

$$H(s) = \frac{b_{2,a}\,s^2 + b_{1,a}\,s + b_{0,a}}{s^2 + a_{1,a}\,s + a_{0,a}}.$$

The parameter `w1` is the digital frequency ω_d to which analog frequency $\omega_a = 1$ is mapped; it determines the frequency-scaling parameter of the bilinear transform. In lowpass or highpass filter design, the frequency mapping is applied to the cutoff frequency (-3 dB point).

Butterworth filters are particularly easy to specify in analog form [18; 19; 16], [10] because, for order N, all N zeros are at infinity and all N poles lie along a circle in the left-half s-plane. For example, the second-order Butterworth lowpass filter with its -3 dB frequency normalized to $\omega_a = 1$ is simply

$$H(s) = \frac{1}{s^2 + \sqrt{2}\,s + 1}$$

and can be specified as `tf2s(0,0,1,sqrt(2),1)`.

3.3.1 Normalization and Stability Protection

For extreme time-varying filtering applications, a practically useful variant named `tf2snp` is provided that implements `tf2s` using a normalized ladder filter (for decoupling signal and coefficient energy, §3.2) together with stability projection (easy to do in ladder/lattice digital filters by simply clipping their reflection coefficients to the range $(-1, 1)$). This is used in the most numerically robust Moog VCF implementation `moog_vcf_2bn` (`effect.lib`, §4).

[10] `https: //ccrma.stanford.edu/~jos/filters/-Butterworth_Lowpass_Design.html`

The example `vcf_wah_pedals.dsp` in the FAUST distribution provides a comparison of three Moog VCF implementations as well as the second-order Crybaby wah-pedal and a fourth-order wah-pedal based on the Moog VCF.

3.3.2 Bandpass Mapped Biquad

The function `tf2sb(b2,b1,b0,a1,a0,w1,wc)` is a bandpass mapping of the basic analog-specified biquad `tf2s`. In addition to the frequency-scaling parameter `w1` (which gets set to *half* the desired passband width in radians per second), there is a desired center-frequency parameter `wc` (also in rad/s). Thus, `tf2sb` implements a fourth-order *digital* bandpass filter section specified by the coefficients of a second-order *analog* lowpass prototype section. Such sections can be combined in series for higher orders. The order of mappings is (1) frequency scaling (to set lowpass cutoff `w1`), (2) bandpass mapping to `wc`, then (3) the bilinear transform, with the usual scale parameter `2*SR`, where `SR` denotes the sampling rate. The FAUST implementation for this was based on algebra carried out in `maxima`.

3.4 Butterworth Lo/Hi/Bandpass Filters

Butterworth lowpass and highpass filters of any order can be defined recursively in FAUST thanks again to the pattern-matching facility in the language. The elliptic (Cauer) filters[11] are special-cased because the pole locations are computed using the elliptic rational function, which is not available in typical computer math libraries. Such a function could of course be supplied as a foreign function in FAUST.

3.5 Shelf and Equalizer Sections

The low/high shelf and peaking equalizer sections implemented in `filter.lib` are described further in `filter.lib` and in [16].[12]

3.6 Lagrange/Thiran-Interpolated Fractional Delay Lines

Delay lines interpolated using higher-order FIR Lagrange interpolation are all used as follows:

```
fdelayN(maxdelay, delay, inputsignal)
```

where `N`=1,2,3, or 4 is the order of the Lagrange interpolation polynomial. Note that this API follows that of `fdelay` in `music.lib`. The requested delay should not be less than $(N-1)/2$ because the interpolating polynomial needs to be able to "reach" that far into the "past" when interpolating.

Delay lines interpolated using higher-order IIR allpass Thiran interpolation are all invoked as

```
fdelayNa(maxdelay, delay, inputsignal)
```

where `N`=1,2,3, or 4 is the order of the allpass interpolation filter. In this case, it is recommended that the requested delay be at least $N-1/2$ because an Nth-order allpass provides a delay of N samples as its coefficients approach zero. Note that delay arguments that are too small can produce an *unstable* allpass filter. For rapid delay modulations, Lagrange (FIR) interpolation is generally preferred. However, allpass interpolation introduces no gain distortion and may therefore be preferred in nearly lossless feedback loops.

3.7 Filter Banks

A *filter bank* splits its input signal into a bank of parallel signals, one for each spectral band. If the bandpass filters used to create the channel signals are carefully designed, one may sum the channel signals to get back the original input signal (possibly scaled and/or delayed). In this case, the filter bank is said to be a *Perfect Reconstruction* (PR) filter bank [20]. However, for purposes of spectrum analysis, in which only the channel signal powers are displayed, the PR condition is overkill. Therefore, the filter banks implemented in `filter.lib` are divided into *"analyzers"*, which do not have the PR property, and "filter banks" which are "allpass complementary". Allpass-complementary filter banks are reasonable choices for "graphic equalizer" applications. An allpass-complementary filter bank is PR when the allpass reduces to a pure delay and possible scaling. In this terminology, the filter banks in `filter.lib` are implemented as analyzers in cascade with delay equalizers that convert the analyzer to an allpass-complementary filter bank. Spectrum analyzer outputs should at least be nearly "power complementary", *i.e.*, the power spectra of the individual bands should at least approximately sum to the original power spectrum.

[11] `http://en.wikipedia.org/wiki/Elliptic_filter`

[12] `https: //ccrma.stanford.edu/~jos/filters/-Low_High_Shelf_Filters.html`

The typical filter bank or analyzer is constructed as a *dyadic filter bank*, meaning that it consists of a sequence of band-splits, forming a binary tree of lowpass/highpass filter sections. Since audio applications are presumed, only the lower band is split when going from one stage to the next.

In the FAUST distribution, both filter banks and spectrum analyzers are illustrated in the example `graphic_eq.dsp`. See also `spectral_level.dsp` which is a standalone spectrum analyzer (nice as a standalone JACK app).

The example `gate_compressor.dsp` included with the FAUST distribution exercises the gate and compression utilities.

Space limitations preclude further discussion here. Please see comments in `filter.lib` for further usage details.

4 Faust Library `effect.lib`

The modules in `effect.lib` classify as "digital audio effects". In general, they tend to be special-purpose *filters*, frequently nonlinear and/or time varying.

4.1 Moog Voltage Controlled Filters

New since the analog-form Moog VCF [7] is the implementation `moog_vcf_2b` of the ideal Moog VCF transfer function factored into second-order sections. As a result, its static frequency response is more accurate than `moog_vcf` which suffers from an unwanted one-sample delay in its feedback path. On the downside, its coefficient formulas are more complex when one or both parameters are varied. The `res` parameter of `moog_vcf_2b[n]` is the fourth root of that in `moog_vcf`, so, as the sampling rate approaches infinity, `moog_vcf(res,fr)` becomes equivalent to `moog_vcf_2b[n](res^4,fr)` (when `res` and `fr` are constant).

4.2 Artificial Reverberation

The reverberation modules in `effect.lib` are described in [8]. Of special note is the high-quality reverberator called `zita_rev1`, ported to FAUST from the C++ source of zita-rev1 written by Fons Adriaensen.[13] It combines Schroeder allpass and FDN reverberation techniques [8].[14]

[13] `http://kokkinizita.linuxaudio.org/linuxaudio/-zita-rev1-doc/quickguide.html`

[14] `https://ccrma.stanford.edu/~jos/pasp/Zita_Rev1.html`

5 Conclusion

Developments since LAC-08 for FAUST libraries `oscillator|filter|effect.lib` were outlined. The overall goal is to accumulate reference implementations of commonly used algorithms in music/audio signal processing, with a general preference for expressive parametric algorithms yielding the highest performed sound quality per unit of computation.

6 Acknowledgments

Special thanks to Yann Orlarey for contributing various improvements to the functions described in this paper and making others possible at all, particularly with respect to the use of pattern matching. Special thanks also to Albert Gräf for adding the pattern-matching facility to the FAUST compiler.

References

[1] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of FAUST", *Soft Computing*, vol. 8, no. 9, pp. 623–632, 2004.

[2] A. Gräf, "Term rewriting extension for the FAUST programming language", in *Proc. 8th Int. Linux Audio Conf. (LAC2010), Utrecht, http://lac.linuxaudio.org/*, 2010, `http://lac.linuxaudio.org/2010/papers/-30.pdf`.

[3] Y. Orlarey, A. Gräf, and S. Kersten, "DSP programming with FAUST, Q and SuperCollider", in *Proc. 4th Int. Linux Audio Conf. (LAC2006), ZKM Karlsruhe, http://lac.zkm.de/2006/proceedings.shtml*, 2006, pp. 39–40, `http://lac.zkm.de/2006/proceedings.shtml-#orlarey_et_al`.

[4] A. Gräf, "Interfacing Pure Data with FAUST", in *Proc. 5th Int. Linux Audio Conf. (LAC2007), TU Berlin, http://www.kgw.tu-berlin.de/~lac2007/-proceedings.shtml*, 2007, `http://www.kgw.tu-berlin.de/~lac2007/-papers/lac07_graef.pdf`.

[5] J. O. Smith, "Audio signal processing in FAUST", 2012, `https://ccrma.stanford.edu/~jos/aspf/`.

Moog VCF	See FAUST example `vcf_wah_pedals.dsp`
`moog_vcf(res,fr)`	analog-form Moog VCF `res` = corner-resonance amount between 0 (none) and 1 (max) `fr` = corner-resonance frequency in Hz (less than SR/6.3 or so)
`moog_vcf_2b(res,fr)`	Moog VCF implemented as two biquads (`tf2`)
`moog_vcf_2bn(res,fr)`	two protected, normalized-ladder biquads (`tf2np`)
Phasing and Flanging	See FAUST example `phaser_flanger.dsp`
`vibrato2_mono(...)`	modulated allpass-chain (see `effect.lib` for usage)
`phaser2_mono(...)`	phasing based on 2nd-order allpasses (see `effect.lib` for usage)
`phaser2_stereo(...)`	stereo phaser based on 2nd-order allpass chains
`flanger_mono(...)`	mono flanger
`flanger_stereo(...)`	stereo flanger
Envelopes/Compression/Expansion	See FAUST example `gate_compressor.dsp`
`amp_follower_ud(att,rel)`	`att` = attack time-constant (sec) going up
`amp_follower(rel)`	`rel` = release time = time-constant (sec) going down (att=0)
`autowah(level)`	`level` 0 to 1
`gate_mono(thresh,att,hold,rel)`	squelch signal when below `thresh` (in dB), for at least `hold` seconds
`gate_gain_mono(thresh,att,hold,rel,x)`	"gain computer"
`gate_stereo(thresh,att,hold,rel,x,y)`	two mono gates using same gain computer
`compressor_mono(ratio,thresh,att,rel)`	single-channel dynamic-range compression: `ratio` = compression ratio dB-in over dB-out above `thresh` `thresh` = dB level threshold above which compression kicks in
`compressor_stereo(...)`	stereo case, common gain computer
`limiter_1176_R4_mono`	= `compressor_mono(4,-6,0.0008,0.5);`
`limiter_1176_R4_stereo`	stereo case
Artificial Reverberation	See FAUST examples `freeverb|reverb_designer|zita_rev1.dsp`
`jcrev`, `satrev`	Historical early Schroeder reverberators
`fdnrev0(...)`	Feedback Delay Network (FDN) reverberator [8]
`prime_power_delays(N,pathmin,pathmax)`	utility for finding prime-power delays across a range
`zita_rev_fdn(f1,f2,t60dc,t60m,fsmax)`	order 8 FDN used in zita-rev1 - see `effect.lib` for usage
`zita_rev_stereo(...)`	stereo version of zita-rev1 - see `effect.lib` for usage
`zita_rev1_ambi(...)`	zita-rev1 in ambisonics mode
`mesh_square(N)`	`N` by `N` square digital waveguide mesh
Other Modules	
`stereo_width(w)`	stereo width effect based on the Blumlein Shuffler
`apnl(a1,a2,x)`	nonlinear allpass filter used in FAUST-STK [6]
`piano_dispersion_filter(M,B,f0)`	closed-form piano-string allpass by Rauhala et. al [21]

Figure 3: Functions defined in `effect.lib` since LAC-08.

[6] R. Michon and J. O. Smith, "FAUST-STK: A set of linear and nonlinear physical models for the FAUST programming language", in *Proc. 14th Int. Conf. Digital Audio Effects (DAFx-11), Paris, France, September 19–23*, 2011.

[7] J. O. Smith, "Virtual electric guitars and effects using FAUST and Octave", in *Proc. 6th Int. Linux Audio Conf. (LAC2008), http://lac.linuxaudio.org/*, 2008.

[8] J. O. Smith, *Physical Audio Signal Processing*, https://-ccrma.stanford.edu/~jos/pasp/, Dec. 2010, online book.

[9] J. Dattorro, "Effect design: Part 3: Oscillators: Sinusoidal and pseudonoise", *J. Audio Eng. Soc.*, vol. 50, no. 3, pp. 115–146, 2002.

[10] J. O. Smith and P. R. Cook, "The second-order digital waveguide oscillator", in *Proc. 1992 Int. Computer Music Conf., San Jose*. 1992, pp. 150–153, Computer Music Association, http://ccrma.stanford.edu/~jos/wgo/.

[11] J. W. Gordon and J. O. Smith, "A sine generation algorithm for VLSI applications", in *Proc. 1985 Int. Computer Music Conf., Vancouver*. 1985, Computer Music Association.

[12] H. Chamberlin, *Musical Applications of Microprocessors*, Hayden Book Co., Inc., New Jersey, 1980.

[13] V. Välimäki, "Discrete-time synthesis of the sawtooth waveform with reduced aliasing", *IEEE Signal Processing Letters*, vol. 12, no. 3, pp. 214–217, 2005.

[14] V. Välimäki, J. Nam, J. O. Smith, and J. S. Abel, "Alias-suppressed oscillators based on differentiated polynomial waveforms", *IEEE Trans. Audio, Speech, and Language Processing*, vol. 18, no. 5, May 2010.

[15] R. F. Voss and J. Clarke, "'1/f noise' in music: Music from 1/f noise", *J. Acoust. Soc. of Amer.*, vol. 63, no. 1, pp. 258–263, Jan. 1978.

[16] J. O. Smith, *Introduction to Digital Filters with Audio Applications*, http://-ccrma.stanford.edu/~jos/filters/, Sept. 2007, online book.

[17] A. H. Gray and J. D. Markel, "A normalized digital filter structure", *IEEE Trans. Acoustics, Speech, Signal Processing*, vol. ASSP-23, no. 3, pp. 268–277, June 1975.

[18] C. S. Burrus, *Digital Signal Processing and Digital Filter Design (Draft)*, Connexions, Sept. 2009, online book: http://cnx.org/content/col10598/latest/.

[19] T. W. Parks and C. S. Burrus, *Digital Filter Design*, John Wiley and Sons, Inc., New York, June 1987, contains FORTRAN software listings.

[20] P. P. Vaidyanathan, *Multirate Systems and Filter Banks*, Prentice-Hall, 1993.

[21] J. Rauhala and V. Välimäki, "Tunable dispersion filter design for piano synthesis", *IEEE Signal Processing Letters*, vol. 13, no. 5, pp. 253–256, May 2006.

The Integration of the PCSlib PD library in a Touch-Sensitive Interface with Musical Application

José Rafael Subía Valdez
IUNA - UNQ
Buenos Aires
Argentina
jsubiavaldez@gmail.com

Abstract

This paper describes the study and use of the PCSlib library for Pure Data and its implementation in the project "Interface Design for the development of a touch screen with musical application" [Causa, 2011] The project consists of a touch-sensitive interface that allows the drawing of musical gestures that are then mapped to a harmonic structure generated by the PCSlib library. Pure Data also is responsible of the translation and reproduction of the musical gestures via MIDI.

Keywords

Pitch Class Sets, XML, Musical Gesture, Harmonic Structure

1 Introduction

The creation of touch sensitive screens over the years, has led to many performance instruments and tools. However, this technology has not yet entered the music writing and composing for fixed media category such as scores. Touch Screen instruments like the *ReacTable* [1] or the *Lemur* [2] have existed for a few years now. These innovative instruments have broken down technological walls permitting the development of more interfaces including the one developed in this project. Nevertheless, the *ReacTable*, the *Kitara Digital Guitar* [3] and such, have exploited the "Real-Time" characteristic of these interfaces. Their use has been focalized as instruments to perform and not so much to write music. Some other developments that approach the same questions stated in this project have been scarcely documented. French composer, Philippe Leroux used a system that translated drawings entered in a *Wacom* [4] tablet to pitches resembling the inputted sketch [Vassilandonakis, 2008]. This project used OpenMusic [5] to do so. Nonetheless, while Leroux's system intends to capture human gesture such as hand writing to be used or to create music structures, this project explores the use of codified traditional nomenclature. "Ugarit" is a touch sensitive screen that allows the writing of music by entering codified drawings that represent specific and traditional musical gestures.The result is a graphic score that will only produce sound after the notes are entered. With the help of the *Pitch Class Sets* [Forte, 1974] theory and its implementation in Pure Data through the PCSlib library, "Ugarit" lets the user concentrate in writing musical gestures and building music pieces without preoccupying him or herself of the harmonic structure. "Ugarit" maps the drawings to a harmonic structure that the system previously creates allowing its operator to think the writing of music in a different way.

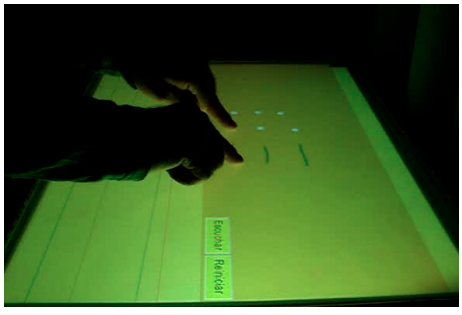


Figure 1: "Ugarit" Multitouch screen

"Ugarit" could also allows the teaching of mod-

[1] www.reactable.com
[2] www.jazzmutant.com/lemur_overview
[3] www.misadigital.com/guitars.htm
[4] www.wacom.com/
[5] http://repmus.ircam.fr/openmusic/home

ern music theories in a unique manner. Its cheap development, and its friendly interface, lets it to be assembled in schools everywhere. "Ugarit" would be perfect for teaching music for children or any kind of non-specialized public. Because it uses a modern approach to music paradigms, it would be updating the music theory taught to the public. It allows users to create music focusing in melodic contours and assembling musical gestures in time. Such ways of thinking music are the key to understand the creation of some of the modern music of the XXI century. It lets the user experiment with these nontraditional music concepts by taking care of complex harmonic structures internally. This permits the creation of "modern-like" music without the full knowledge of complex concepts needed to produce it.

2 Data Interpretation (Connection of the Touch Sensitive Interface and Pure Data)

2.1 Musical Data Entry

A team of four artists/programmers that were divided in two groups developed the complete system. Emiliano Causa and Sebastian G. Botasi developed the graphical part of the interface while Matías Romero Costas and the author of this paper programmed the audio part. This paper only describes in detail the implementation of a specific part of the program, but a brief schematic of the entire workflow is illustrated in the block diagram bellow. [fig. 2]

The musical data entry is accomplished through an XML[6] file generated by Processing[7] after the census and analysis of the touchscreen interface drawing. PD then reads the file through the "Detox" object. This object is developed as an external in the "jasch_lib" library developed by Jan Schacher. "Detox" allows reading the XML file informing the user when a "tag-tree" is opened or closed. The data is then routed through a series of abstractions that stores it in "coll" objects. Matías Romero Costas[8] developed this part of the program. It was later modified and completed with the part of the program that maps the values of the surface to the pitches of the harmonic structure, this will be explained later.

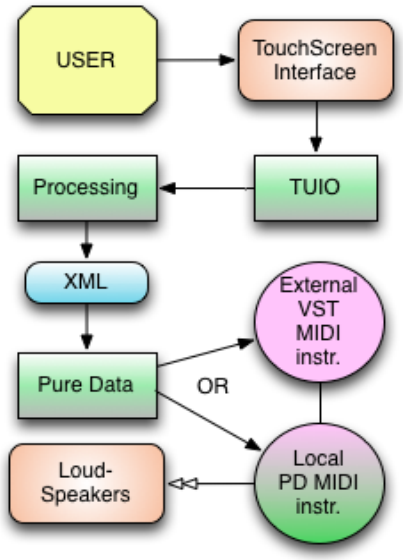


Figure 2: System workflow

2.2 Music Gesture players

The musical gestures implemented in the touch screen interface are "chord", "trill", "tremolo", "melody", "note", "arpeggio" and "glissando". All of them were implemented in a way that all action constituting the gesture, are united in a group of parameters. For this, the players assemble messages, used as buffers or temporary memories, that feed the "makenote" and "noteout" objects. "Makenote" and "Noteout" objects convert and send the processed midi messages to the corresponding outputs. These messages are created by receiving stored data from the "col" objects that deliver stored data in groups corresponding to gestures one at a time when the groups indexes are equivalent to playback time.

2.2.1 Tremolo and Trill

Essentially, the algorithm for trill and tremolo are the same since the gesture works the same way. A buffer generates two messages that store the two values to be interleaved at a certain speed during the duration of the event. The only difference is that the tremolo player. Sets the two messages with the two pitches entering from the screen, while the trill player only receives one pitch and trills with a semitone above it. This is accom-

[6]XML files are text files with especific formats that can be interpreted by different programs

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

[8]For more information, read documentation writen by the autor.

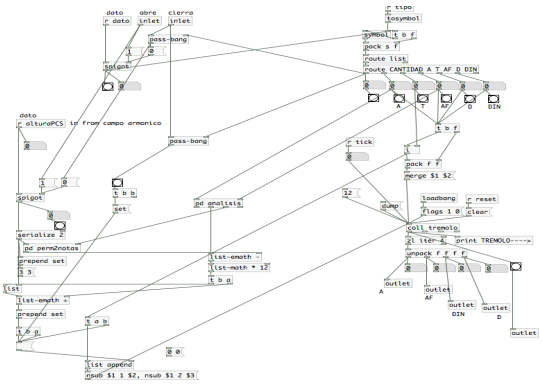


Figure 3: Tremolo Subpatch where the"coll" object is

plished by simply adding one to the entered pitch.

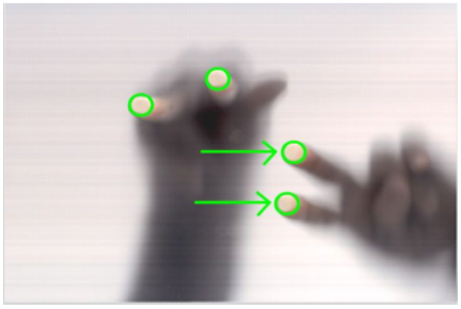


Figure 4: specific gesture to draw that produces a "tremolo"

2.2.2 Chord

The chord player uses a single line message as a buffer and builds one with all the pitches of the chord. The program must set the message correctly inserting the pitches and key velocities of the notes for the "makenote" object to play the chord properly. For this player, as well as for the "arpeggio" player, it was necessary to modify the abstractions where "coll" objects are, in order to filter the different start times of each individual note of the gesture entering from the surface and set only the time of the first note as the same for each. This was achieved with the abstraction "once" developed by Thomas Musil in Austria. It allows only the first value to pass acting as a gate.

2.2.3 Arpeggio

The Arpeggio playback, as well as the chord playback, results with the same modification of abstractions where the corresponding "coll" object is placed. In addition, the arpeggio involves using the start time of the first note and then an algorithm that plays the rest of the notes in succession. For this, we had to use 5 messages as temporary memory where the arpeggio notes (5 maximum) would be stored and then executed by a small counter that triggers them one by one in rapid succession, a fundamental characteristic of the arpeggio.

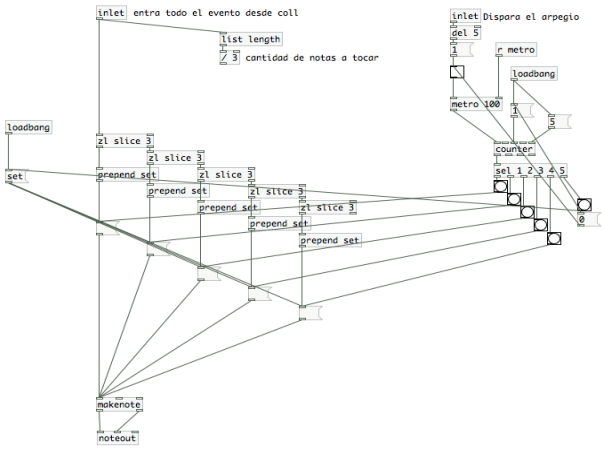


Figure 5: Arpeggio Player Subpatch

2.2.4 Glissandos

The glissando player is very simple. It always makes a linear interpolation between one note and another. It uses the note from which the glissando starts and the note to which it arrives and then interpolates these two pitches linearly using the "step" object, the object can also set the step size of the increase.

2.2.5 Melody and Notes

The two gestures "melody" and "note" use the same player called "NotaPlayer" (NotePlayer) since their respective abstractions where the "col" object are, delivers the pitches used in the musical gesture one by one. This allows the use of a single note player because the gesture may be divided temporarily without any problem and each note can have different and independent durations.

3 Pitch structuring using PCSlib

3.1 Basic Pitch Class Set theory background

Howard Hanson first introduced Pitch Class Set (PCS) theory in the 1960's. It was initially created as a new way to analyze and classify tonal

music but soon music theorists like Allen Forte and Robert Morris started to use it for atonal temperate music. The theory is based in the interval relations created between notes in a music piece. It allowed theorists to find coherent and close relations in harmonic structures of modern music that were difficult to find in those times.

The PCS theory uses the same analog Set Theory used in combinatorial algebra seen in mathematics. It allows classifying and studying relations between groups of notes that have the same interval characteristics. The theory establishes that each group of notes has a prime form, this permits a better way to classify groups of notes. To do this, the theory considers that all pitches should be enumerated from zero to eleven starting from C (see Table 1). This admits a better order for classifying same set types that only differ in the pitches involved (see Table 2). It also establishes that the octave relation is not important, meaning that if a Db4 is played and a F6 follows, the interval taken into consideration would be a major third, ignoring the two-octave difference.

C	C#	D	D#	E	F	F#	G	G#	A	A#	B
0	1	2	3	4	5	6	7	8	9	10	11

Table 1: Cromatic Scale enumerated from 0

Finally, the PCS theory allows mathematic operations in groups of pitches. Some simple operations include transposition, inversion and retrogradation. More complex operations are also viable and are the true potential of this theory applied to composition.

0	3	4	7	PCS 4-17 Prime Form
1	10	9	6	PCS 4-17 Inverted & Transposed

Table 2: Same Class Set with different Pitches, note that the same Set Class has the same intervals

3.2 Harmonic Structure Creation

The harmonic structure used is made with the "pcs_chain" object part of the PCSlib library [Di Liscia and Cetta, 2011a]. This object produces Pitch Class Set chains of the same set class [Di Liscia and Cetta, 2011b]. This part of the program is located in a subpatch and lets you choose a PCS of five or six notes. Then the object will split the PCS in two and will provide all possible

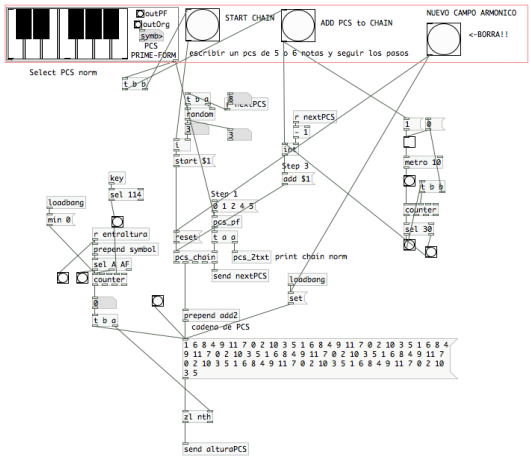


Figure 6: Patch that generates de Harminoc Structure with "pcs_chain"

combinations. The user then selects the partition to start the chain. The object provides a list of all possible partitions; the last option is always the best fit to saturate the total chromatic scale. The program uses this option to create the harmonic structure.[fig. 6]

3.3 Mapping of the harmonic structure to the inputted musical gesture

The harmonic structure will be essential to the color of any gesture entered. To achieve this, a compromise is necessary between the gesture, which is delivered in discrete MIDI values, and the pitches that will have to be changed to correctly fit the harmonic structure. This allows musical coherence of any piece of music written on the surface and allows the user greater freedom to concentrate on musical gesture without worrying about the pitches to use.

To accomplish this, we had to develop several algorithms that find the best accommodation of the pitches incoming from the gestures drawn on the surface to the harmonic structure. This way, the distortion between the drawing in the surface and the resulting pitches played is set to the minimum as possible. A clear example of such distortion would be the accommodating of the pitches from the drawn gesture to pitches that are too far apart from the registry and "ambitus" of the gesture entered. [fig. 7]

Through different algorithms the program is able to restructure the pitches inputted. Each solution was different for each musical gesture be-

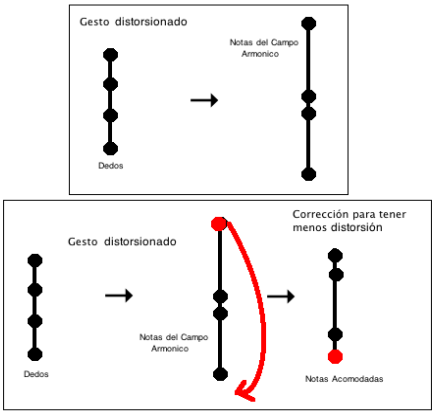


Figure 7: Diagram of how a part of the Permutation Algorithm works

cause the Pitch Class Sets theory relates to each gesture in a different manner. This means that the algorithm in the illustration above will not work to accommodate the pitches of the gesture called "melody". However, all the particular solutions to the problem have the same analysis system [Di Liscia and Cetta, 2011c] for the data entry from the surface. The analysis will help compare the pitches of the gesture to the pitches of the harmonic structure created. It will sort the data and extract the absolute pitches eliminating the octave relationship. On the other hand, it does keep track of the "octave" in which the pitches are subtracted from, finally with the help of the object "pcs_pf" it is able to find out to which set class the pitches entered form. All programming is achieved thanks to a set of abstractions included in "Pd-extended" known as "list-abs". These abstractions allow the manipulation of lists, a cornerstone in algorithmic programming for pitch relation manipulation.

3.3.1 Mapping of the gesture "chord" and "arpeggio" to PCS

The program that maps the music gesture "chord" and "arpeggio" is the same due to the fact that the "arpeggio" is a chord whose notes are deployed in time when played. The solution to the correct mapping has several steps. In addition to the analysis [fig. 8] previously explained, the program must compare the arrangement of the incoming pitches from the chord generated to the pitches previously generated in the harmonic structure.

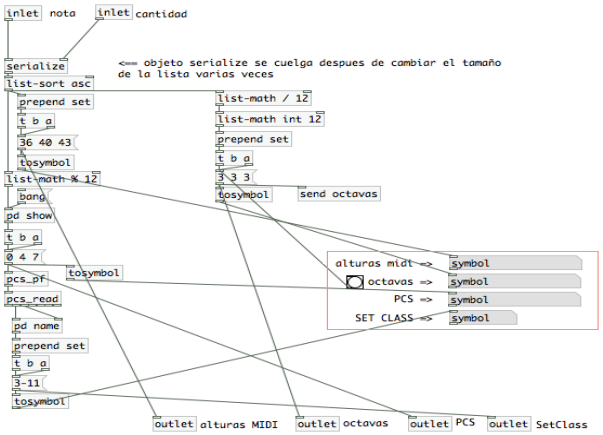


Figure 8: Patch that analizes the incoming MIDI pitches

This operation is performed by subtracting one chord to the other and finding the smallest difference in intervallic distance from one another. Then, the chord generated is swapped with the object "pcs_perm" until the permutation with least difference is found. This means that the chord that replaces the entrant is chosen according to which has the greatest similarity in the disposition of the entered pitches and the area covered by the chord. The program does not analyze deeper if there are several permutations with the same result, it chooses the first one with the least difference to optimize processing time. Finally, the program seeks the best accommodation for octave placement with an algorithm which states that if the difference in semitones between two notes is higher than 6, the interval can be inverted to bring one chord closer to the *ambitus* of the original inputted chord. Example of the algorithm, the 2 PCS are subtracted and then the intervals are added. (see Table 3 & Table 4).

3		5		7		10	Touch-screen
4		0		11		7	Harm. Struct.
1	+	5	+	4	+	3	= 13

Table 3: Interval Subtraction of PCS & addition of intervals

3		5		7		10	Touch-screen
0		4		7		11	Harm. Struct.
3	+	1	+	7	+	11	= 5 Best Option

Table 4: Interval Subtraction of PCS & addition of intervals (Best Option for Replacement)

3.3.2 Mapping of the gesture "glissando" and "tremolo" to PCS

Gestures "glissando" and "tremolo" were equally resolved because both receive data from the XML file the same way. The XML file delivers the initial and final pitch that the players need to reproduce the gestures, which must be accommodated to reproduce correctly the pitches entering from the harmonic structure. For this, the permutation algorithm explained in the above process is the same, but was simplified by omitting the "pcs_perm" object. The program compares the intervals entered form the XML file and the extracted from the harmonic structure. If the difference between the two intervals exceeds the augmented forth, the interval is inverted to better resemble the drawn gesture in the surface.

3.3.3 Mapping of the gesture "melodía" (melody) to PCS

The mapping of "melody" gesture to the previously designated PCS in the harmonic structure was accomplished in several steps. First, you must understand the concept of "melodic contour" and how the algorithm keeps its design and direction regardless of whether the area in which it develops is distorted. This is necessary because for the contour drawing to be maintained, it is more important to keep the direction of the intervals before the closeness of the notes.

The program creates a matrix where the two different PCS are entered, one incoming from the XML file and the other from the harmonic structure. The PCS are sorted according to a position index designated by the melody. The pitches are later rearranged in ascending order. First, an index number is allocated to each pitch of the melody entered from the XML file. For example, if you enter the PCS [3 0 7 9 5] as the melody contour, an index is given to each pitch producing a matrix like shown below. (see Table 5).

It is later re-order in ascending order for it to later be compared with the PCS entering from the harmonic structure. (see Table 6).

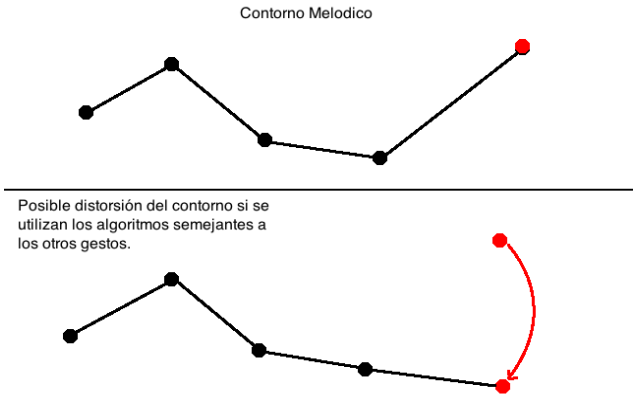


Figure 9: Diagram that shows the posible distortion of the gesture "melody"

0	1	2	3	4	Index (Sorted)
3	0	7	9	5	PCS from the XML

Table 5: Step 1 of the melodic contour mapping

1	0	4	2	3	Index (un-sorted)
0	3	5	7	9	PCS from the XML

Table 6: Step 2 of the melodic contour mapping

Now the PCS from the harmonic structure can be entered, it is sorted in ascending order and inserted into the matrix. For example, the PCS from the harmonic structure will be [4 11 6 2 8] that once re-ordered will look as follows: [2 4 6 8 11]. Now you can place that vector in the table and the matrix will be as follows (see Table 7).

1	0	4	2	3	Index (un-sorted)
0	3	5	7	9	PCS from the XML
2	4	6	8	11	PCS from Harm. Structure

Table 7: Step 3 of the melodic contour mapping

Thus we have the two PCS sorted from lowest to highest order and the index vector from the melodic order is now un-sorted. Then all that remains is to rearrange the matrix according to the first vector re-ordering from smallest to largest and extract the row number three. Resulting in the following. (see Table 8).

0	1	2	3	4	Index (Sorted)
3	0	7	9	5	PCS from the XML
4	8	11	2	6	PCS from Harm. Structure

Table 8: Step 4 of the melodic contour mapping

The PCS from the harmonic structure in the correct order to keep the melodic contour is [4 2 8 11 6]. As for the drawing of the melody, the result is the contour maintenance and so, it causes the least amount of distortion of the drawing entered from the surface.

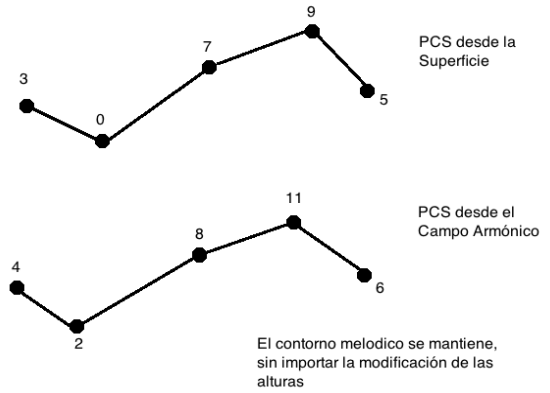


Figure 10: melodic contour maintained dispite the new pitches

All this part of program was development in the "abstraction" called "zzzzmelodia" and uses the "matrix" object included in the "iemmatrix" library. Finally, the new pitches are replaced in the "coll" object, which are then reproduced when the complete surface input is executed.

3.3.4 NO - Mapping of the gesture "Nota" (note) to PCS

The music gesture "nota" (note) was the only one left without mapping to a previously designated pitch entering from the harmonic structure. This is because the independence that "nota" has as a gesture and its immediate relation to a pitch, led to the conclusion that it was the only gesture totally free from the harmonic structure mapping process.

4 Conclusions

The Touch-Sensitive Interface with Musical Application was developed in 2011 and exhibited as a prototype on September 23, 2011. Throughout its development, objectives were achieved by solving problems step by step. Pure Data proved to be very versatile for data processing as implemented in this project even though it is difficult to achieve this type of programs in environments known as "max". The PCSlib library showed to be very complete and it allowed experimental approaches in the creation of harmonic structures with the use of the Pitch Class Sets theory. The touch sensitive surface named "Ugarit" opens new paradigms for cheap technology with extreme potential for teaching music in new ways. It could take music education a step closer to modern music by implementing PCS theory in the teaching of music and modern music theories by underlining the importance of musical gesture and intervallic relations.

The "Ugarit" was developed in a research program of a public university in Argentina. It is still in a complete experimental stage due to lack of income. Because the team depends entirely on public resources, the further development of the project is uncertain. However, its success after presenting it, proved to be a viable project. Now the developing team must wait and see what will become of this project. Most recently they have presented a complete report of the project and all the work done during this early stage. For more information about this project, such as requirements and building steps, it is encouraged to contact the author of this paper.

5 Acknowledgements

The Author would like to thank the entire team; Emiliano Causa, Sebastian G. Botasi & Matías Romero Costas. With whom he worked during the complete development of "Ugarit". And a special "thank you" to Dr. Pablo Di Liscia & Dr. Pablo Cetta for their guideness over the years.

PCSlib belongs to the research project "Musical Applications of sets and combinatorial matrices of set classes" Director Dr. Oscar Pablo Di Liscia and Dr. Pablo Cetta, in Quilmes National University.

Touch-Sensitive Interface with Musical Application belongs to the research project "Design and development of applications and interfaces for augmented reality for synthesis and digital audio processing" Part of the program PICTO-

ART. Director Carmelo Saitta and Pablo Cetta in Area Transdepartamental of Multimedia Arts from National College of Art (IUNA)

References

Emiliano Causa. 2011. *Diseño de interface para el desarrollo de una pantalla sensible al tacto con aplicación musical*. Revista de Investigación Multimedia (RIM), Buenos Aires, Argentina.

Pablo Di Liscia and Pablo Cetta. 2011a. *Composición asistida en entorno PD*. Revista de Investigación Multimedia (RIM), Buenos Aires, Argentina.

Pablo Di Liscia and Pablo Cetta. 2011b. *Elementos de Contrapunto Atonal*. EDUCA, Buenos Aires, Argentina.

Pablo Di Liscia and Pablo Cetta. 2011c. *Medidas de similitud entre sucesiones ordenadas de grados cromáticos*. Revista de Investigación Multimedia (RIM), Buenos Aires, Argentina.

Allen Forte. 1974. *The Structure of Atonal Music*. Yale University Press, London.

Miller Puckette. Pd documentation.

Yiorgos Vassilandonakis. 2008. An interview with philippe leroux.

Rite of the Earth – composition with frequency-based harmony and ambisonic sound projection.
LAC 2012, CCRMA Stanford

Krzysztof GAWLAS
University of Silesia
Cieszyn, Poland
gawl@o2.pl

Abstract
A multi part composition, called Rite of the Earth will be presented, utilizing the sounds of ceramic instruments built during the Academy of the Sounds of the Earth, a multidisciplinary artistic project held in the Artistic Department of Silesian University in Katowice (Poland). Most of the compositional process and sound synthesis was carried out in SuperCollider, mixing and spatializing in Ardour, with the use of tools by Fons Adriaensen.

Keywords
Sound recording, sound transformation and synthesis, music composition, multichannel spatialization.

1 Academy of Sounds of the Earth

Since 2007 the faculty of Ceramics, in the Institute of Fine Arts, Artistic Department, Silesian University in Katowice has been running a project called "Academy of the Sounds of the Earth", under direction of dr. hab. Małgorzata Skałuba-Krentowicz, exploring the possibilities of constructing non-traditional musical instruments utilizing ceramic techniques. Lecturers and students of the Institute of Fine Arts, Institute of Music and students of Ethnology put their attention on the properties of clay hardened by heat, its acoustic qualities and its sound. The project has been helping in experiencing the idea of creating art on the border between different sorts of artistic and scientific activities. A shared space of artistic language was created in relation to shape and sound in the fields of fine arts and music, enriching the field of research in building a universe of the sounds of ceramic instruments, sound objects and sound sculptures. The achievements of the project have been presented during several exhibitions in the form of concerts at the opening and closing. The present stage is an attempt to bring life to the instruments alone, to allow them to speak for themselves.

Fig. 1. The collection of instruments

2 Preparatory work

Many of the instruments, that were made during the project were built more to be visual objects than musical tools, so they do not meet the criteria that musicians apply to "regular" musical instruments – they are very quiet, play out of tune or are just very difficult to play. However according to the same qualities they seem to be very interesting as sound sources for electronic treatment: after amplification, detuning, retuning, filtering, spatializing and so forth, their sound may become distinctive, unusual or fascinating. The composer's task was to show the natural sound of the instruments and to go beyond using synthesis and transformation techniques.

2.1 Recording of sounds

As the music was from the beginning intended for ambisonic sound projection, most of the recordings were made in 2D B-format ambisonic, with omnidirectional Neumann KM-184 for W signal and two figure-of-eight Røde NT2A for X and Y signals respectively. Additionally some sounds were recorded with a pair of spaced omnidirectional microphones, which in some cases led to better results, as it provided a better sense of spaciousness.

2.2 Selecting the sounds as harmonic models

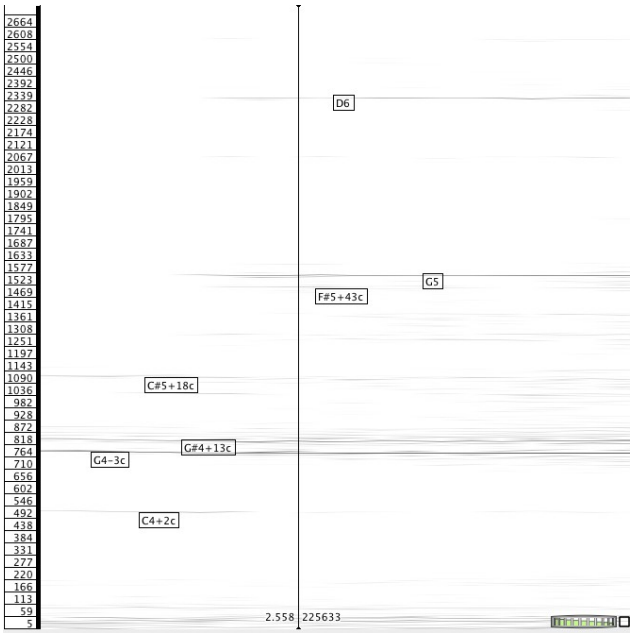


Fig. 2a. Spectrogram of example sound from Sonic Visualiser

After preliminary aural selection, the spectrum of interesting sounds was analyzed. This was done in Sonic Visualiser, which produces spectrograms, which can be tweaked to obtain easy readable results and allows the making of annotations directly onto the spectrogram. These analyses have been mainly used to visually inspect the inner structure of the sounds and to estimate which of the partials might be treated as the pitch base. Many of these sounds were then used as models for the construction of harmony. To import them into Open Music, where most of the musical transformations have been performed, there was the need of a program, that can export the result of the analysis to SDIF format. One of the programs that can do it is Spear [1], [2]. Spear uses a different analysis method (McAulay-Quatieri) and displays the results as sinusoid tracks with time-varying frequencies which can be edited. Unfortunately neither Spear nor OM work on Linux (Spear works under Wine), but both are free software. OM is open source, but relies on LispWorks, a commercial LISP implementation.

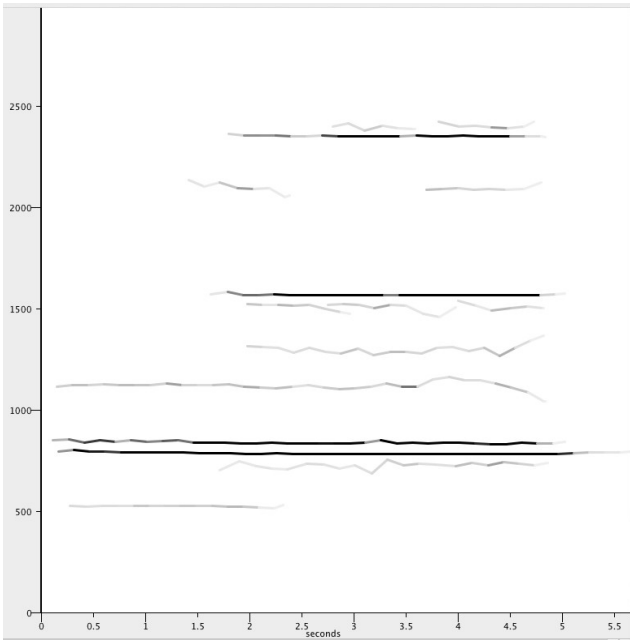


Fig. 2b. The same sound analysed in Spear

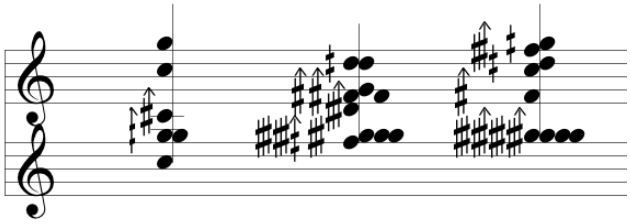


Fig. 2c. Resulting note pitches after SDIF file import in OpenMusic (higher staff is 2 octaves above lower, 1/8-th tone microtonal notation)

The example above (Fig. 2a-c) shows the analyses of the sound of a bowed ceramic bowl (using a violin bow) and the chord progression obtained from Spear/OpenMusic. An interesting aspect of this sound is a (little less than) minor second interval between strong partials.

3 Algorithmic composition sound synthesis and spatialization in SuperCollider

After achieving the desired results in OM the chords were pasted as arrays of frequencies into SuperCollider code. Algorithmic processes evoked here make harmonic (and sonoric) objects of unique authenticity from them .

Composer's Toolkit (Ctk) by Josh Parmenter [3], a SuperCollider library (Quark) for object-oriented composition was chosen as a syntax flavour for compositional and sound synthesis procedures. The class CtkScore stores Ctk events, which may be used in real time or non-real time without any change. That allows easy parameter tweaking and after achieving the desired results the rendering of soundfiles, just by changing `score.play` to `score.write`. CtkControl allows the use of envelopes or even generators in place of parameters.

To preserve the original sounds of ceramic instruments, synthesis techniques, which operate on soundfiles were chosen most frequently – sampling and granular synthesis, but some special or hybrid instruments also appear in this recording.

In Part 2 – "Carillon" to the sounds of chimes (ceramic rods forming a wind-chimes-like instrument) some harmonic FM (with integer ratios) was added to emphasize the pitch of the chimes.

In Part 1 – "Introduction and Mantra", the mantra part was synthesized as a resonant filter with frequency read from an ATS analysis file of a trumpet sound, excited by a shaker sound.

In some flute or trumpet sounds ring modulation or FM modulation has been used to add some artificial harmonics to the sound.

On drum sounds frequency shifting was added to change the pitch without much change in frequency band.

Thanks to AmbPan31 Ugen, by Fernando Lopez-Lezcano the soundfiles have been rendered directly to 3rd order Ambisonic format. 2D ambisonics have been used in this case, so the channels are: w, x, y, u, v, p, q. The "w" signal was modulated to simulate changes of distance and therefore as 8-th channel unmodulated "w" signal was added for feeding it later into reverb.

SuperCollider code was used to synthesize musical gestures as soundfiles, from which then compositions were put together in Ardour.

4 Montage and mixing

Generated soundfiles were then imported into Ardour. During mixing two ambisonic reverbs were added. The first was zitarev1 fed by unmodulated w signal, the second – jconvolver (both by Fons Adriaensen). Zitarev1 is the only reverb known to the author which has ambisonic output. It's a very high quality effect giving a musically desirable result. A specially prepared wxy impulse response was loaded into the jconvolver. This impulse response was made of the chimes clusters panned around the ambisonic soundfield. The result is a frequency dependant reverb, being both a spatial effect and a filter. Both reverbs were fed back into Ardour buses. Complete 2D 3rd order mix might be converted using Ambdec to the desired number of channels.

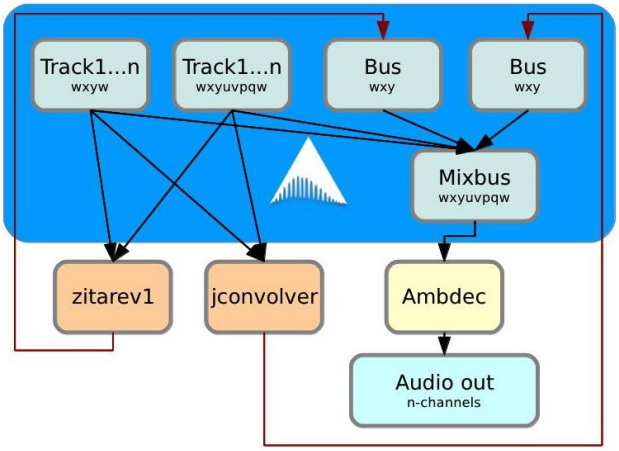


Fig. 4. Signal flow of the final mix

5 Conclusion

A personal achievement in computer music composition, sound design and spatialisation using free and open-source software has been presented. This view is very subjective but shows an example of how a complete system can be utilized. Hopefully this might be an inspiration for other composers and sound artists in search of an environment for their creative needs.

6 Acknowledgements

My thanks go to the originators of the Academy of the Sounds of the Earth, Małgorzata Skałuba-Krentowicz and Katarzyna Handzlik-Bąk, all the people participating in this project, and to all the developers of software I was able to use in the

course of my work on the project.

References

[1] M. Klingbeil. 2009. *Spectral Analysis, Editing, and Resynthesis: Methods and Applications*. Columbia University.

[2] C. Agon, G. Assayag, M. Laurson, C. Rueda. 1964. *Computer Assisted Composition at Ircam: PatchWork & OpenMusic*. Ircam, Paris. Sibelius Academy, Helsinski.

[3] B. Willkie and J. Parmenter. 2011. Non Realtime Synthesis and Object-Oriented Composition. In S. Wilson, D Cottle and N. Collins, editors, *The SuperCollider Book*, pages 125-138. MIT Press, Cambridge, Massachusetts, London, England.

Software

Sonic Visualiser:
http://www.sonicvisualiser.org/
Spear:
http://www.klingbeil.com/spear/
SuperCollider:
http://supercollider.sourceforge.net/
OpenMusic:
http://repmus.ircam.fr/openmusic/home
Ardour:
http://ardour.org/
jconv, zitarev1, ambdec:
http://kokkinizita.linuxaudio.org/

Minivosc - a minimal virtual oscillator driver for ALSA (Advanced Linux Sound Architecture)

Smilen Dimitrov and **Stefania Serafin**
Medialogy, Aalborg University Copenhagen
Lautrupvang 15
DK-2750 Ballerup,
Denmark,
{sd, sts}@create.aau.dk

Abstract

Understanding the construction and implementation of sound cards (as examples of digital audio hardware) can be a demanding task, requiring insight into both hardware and software issues. An important step towards this goal, is the understanding of audio drivers - and how they fit in the flow of execution of software instructions of the entire operating system.

The contribution of this project is in providing sample open-source code, and an online tutorial [1] for a mono, capture-only, audio driver - which is completely virtual; and as such, does not require any soundcard hardware. Thus, it may represent the simplest form of an audio driver under **ALSA**, available for introductory study; which can hopefully assist with a gradual, systematic understanding of **ALSA** drivers' architecture - and audio drivers in general.

Keywords

Sound card, audio, driver, ALSA, Linux

1 Introduction

Prospective students of digital audio hardware, could choose the sound card as a topic of study: on one hand, it has a clear, singular task of managing the PC's analog interface for playback and capture of digital audio data - as well as well-established expectations by consumer users in terms of its role; on the other hand, its understanding can be said to be cross-disciplinary, as it encompasses several (not necessarily overlapping) areas of design: analog and digital electronics related to soundcard hardware and PC bus interface implementation; PC operating system drivers; and high-level PC audio software.

Gaining a sufficient understanding of the interplay between these different domains in a working implementation can be an overwhelming task; thus, not surprisingly, the area of digital audio hardware design and implementation (including soundcards) is currently dominated by industry. Recent developments in open source software and hardware may lower the bar for entry of newcomer DIY enthusiast - however, the existence of many open source drivers for commercial cards doesn't necessarily ease the introductory study of a potential student.

In essence, an implementation of a soundcard will eventually demand dealing with the issue of an operating system *driver*. In the current situation, a prospective student is then faced with a 'chicken-and-egg' problem: proper understanding of drivers requires knowledge of the hardware (which the drivers were written for); and yet understanding the hardware, involves understanding of how the drivers are supposed to interface with it[1]. A straightforward way out, would be to study a 'virtual' driver - that is, a driver not related to an actual hardware; in that case, a student would be able to focus solely on the software interaction between the driver, and the high-level audio program that calls it. Unfortunately, in the case of the **ALSA** driver architecture for **Linux**, pre-existing examples of virtual drivers are in fact not trivial[2] - and, just as existing **ALSA** driver tutorials, *assume previous knowledge* of bus interfaces (and thus hardware).

The **minivosc** driver source code with the corresponding tutorial (on the **ALSA** website [1]) represents the simplest possible virtual **ALSA** driver, that does not require additional hardware. It has already led to the development of the driver used in the (possibly first) demonstration of an open soundcard system in **AudioArduino** [2] (and fur-

[1] and the lack of open card hardware designs for study makes this problem more difficult

[2] and may require existence of real soundcards on the system

ther used in [3]) - and as it limits the discussion to *only* the *software* interaction between driver and high-level software, disregarding issues in bus interfacing and hardware - it would represent a conceptually simpler entry level for a prospective student of sound card drivers.

2 Premise

Personal computer users working with audio typically rely on high-level *audio software* (from media players such as **VLC**, to more specialized software like **Pure Data**, or the wave editor **Audacity**[3]) to perform their needed tasks – and the *sound card* (as hardware) to provide an analog interface to and from audio equipment. This necessarily puts demands on the operating system of the PC, to provide a standardized way to access (what could be different types of) audio hardware. An operating system, in turn, would provide an audio or soundcard *driver* API (application programming interface), which should allow for programming of a driver that: abstracts some of the 'inner details' of the soundcard implementation; and exposes a standardized interface to the high-level audio software (that may want to utilize this driver). This, in principle, allows interfacing between software and hardware released by different vendors/publishers.

Earlier work like [4] attempts to provide a systematic approach to soundcard implementation; however, one clear conclusion from such a naïve approach is that: regardless of the capabilities of the hardware - one cannot achieve a fine control of timing required for audio, by using what corresponds to a simple 'user space' C program. Problems like these are typically solved within the driver programming framework of a given operating system - and as such, acquaintance with driver programming becomes a necessity for anyone aiming to understand development of digital audio hardware for personal computers. In terms of FLOSS[4] **GNU/Linux**- based operating systems, the current driver programming framework - as it relates to soundcards and audio - is provided by the Advanced Linux Sound Architecture (**ALSA**). **ALSA** supersedes the previous OSS (Open Sound System) as the default audio/soundcard driver framework for **Linux** (since version 2.6 of the kernel [5]), and it is the focus of this paper, and the eponymous **minivosc** driver (and tutorial). The **minivosc** driver was developed on **Ubuntu** 10.04 (Lucid), utilizing the 2.6.32 version of the **Linux** kernel; the code has been released as open source on Sourceforge, and it can be found by referring to the tutorial page [1].

2.1 Initial project issues

The **minivosc** project starts from the few readily available (and 'human-readable') resources related to introductory **ALSA** driver development: [8], [9], [10], and [11]. Most of these resources base their discussions on conceptual or undisclosed hardware, making them difficult to read for novices. On the other hand, there are few examples of virtual soundcard drivers, such as the driver source files **dummy.c** (in the **Linux** kernel source tree [12]) and **aloop-kernel.c** (in the **ALSA** source tree [13]); however, these drivers don't have much documentation, and can present a challenge for novices[5]. All these resources [8; 9; 10; 11; 12; 13] have been used as a basis here, to develop an example of a **mini**mal **v**irtual **osc**illator (**minivosc**) driver.

3 Architectural overview of PC audio

Even if the **minivosc** driver is a virtual one, one still needs an overview of the corresponding hardware architecture - also for understanding in what sense is this driver 'virtual'. As a simplified illustration, consider Fig. 1.

A driver will typically control transfers of data between the soundcard and the PC, based on instructions from high-level software. The direction from the soundcard to the PC is the *capture* direction; the opposite direction (from the PC to the soundcard) is the *playback* direction; a soundcard capable of delivering both data transfer directions

[3]Note that software like JACK - while it can be considered more 'low-level' than consumer audio software - is still intended to route data between 'devices'. Since it is the *driver* that provides this 'device' (as a OS construct that software can interface with) in the first place - drivers lay in a lower architectural layer than even software like JACK, and so involve different development considerations.

[4]free/libre/open source software

[5]the 'dummy' driver doesn't actually perform any memory transfers (which is, arguably, a key task for a driver), so it cannot be used as a basis for study – the 'loopback' driver is somewhat more complex than a basic introductory example, as it is intended to redirect streams between devices, and as such assumes some preexisting acquaintance with the **ALSA** architecture

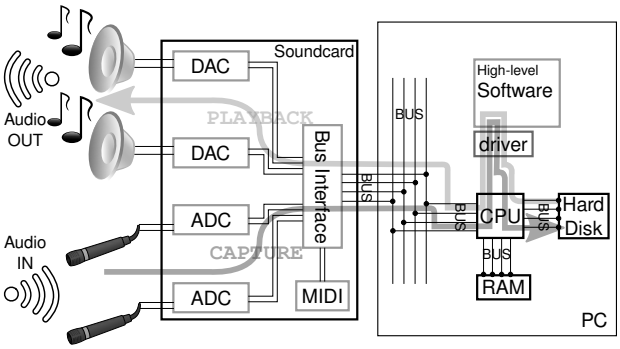


Figure 1: Simplified overview of the context of a PC soundcard driver (portions used from Open Clip Art library).

simultaneously can be said to be a *full-duplex* device.

While Fig. 1 shows the hard disk as (ultimately) both the source for the playback direction, and the destination for the capture direction - within this process, the CPU may use RAM memory at its discretion. In fact, the driver is typically exposed to pointers to byte arrays (buffers) in memory (in **ALSA** known as PCM (sub)streams [7, '*PCM (digital audio) interface*'], and named `dma_area`), that represent streams in each direction.

In terms of audio streams, Fig. 1 demonstrates a device capable of two mono (*or one stereo*) inputs, and two mono (*or one stereo*) outputs. Since audio devices like microphones (or amplifiers for speakers) typically interface through analog electronic signals - this implies that for each 'digital' input [or output] audio stream, a corresponding analog-to-digital (ADC) [or digital-to-analog (DAC)] converter hardware needs to be present on the soundcard[6].

As the main role of the soundcard is to provide an analog electronic audio interface to the PC - the role of the ADC and DAC hardware is, of course, central. However, the PC will typically interface to external hardware through a dedicated bus for this purpose[7]. This means, that some *bus interfacing* electronics - that will decode the signals from the PC, and provide signals that will drive (at the very least) the ADC/DAC converters - needs to be present on the soundcard as well[8].

An **ALSA** driver uses a particular terminology when addressing these architectural surroundings. The 'soundcard' on Fig. 1 will be considered to be a **card** by the driver[9]. One level deeper, things can get a bit more complicated: assuming that Fig. 1 represents a *stereo* soundcard, it would have one input stereo connector (attached to two ADCs), and one output stereo connector (attached to two DACs); an **ALSA** driver would correspondingly be informed about a card, that has one stereo input **device** (consisting of two **subdevice**s) - and one stereo output device (consisting, likewise, of two subdevices). Note that: "*..we use subdevices mainly for hardware which can mix several streams together* [14]" and "*typically, specifying a sound card and device will be sufficient to determine on which connector or set of connectors your audio signal will come out, or from which it is read... Subdevices are the most fine-grained objects ALSA can distinguish. The most frequently encountered cases are that a device has a separate subdevice for each channel or that there is only one subdevice altogether* [15]"

The **ALSA** driver is informed about such a hierarchical relationship (between *card*, *device*s and *subdevice*s) through structures (C `structs`, written by the driver author in the driver source files) - defined mostly through use of other structures, predefined by the **ALSA** framework (alias the **ALSA** 'middle layer'). The driver code, additionally, establishes a relationship between these structs, and the PCM stream data that will be assigned to each in memory; and connects these to predefined **ALSA** framework driver functions, which define the driver (and the corresponding hardware) behavior at runtime. Finally, Fig. 1 shows that other types of devices, such as a MIDI interface, can also be present on the soundcard. The **ALSA** framework

[6]Note, however, that this correspondence could, in principle, be solved by a *single* ADC or DAC element - along with a (de)multiplexer which would implement time-sharing of the element (for multiple channels)

[7]noting that, in principle, the buses used for hard-disks (such as *Integrated Drive Electronics (IDE)*) or RAM (known as *'Memory Interconnect'*) can be distinct

[8]For example, [4] describes a device that interfaces through the *Industry Standard Architecture (ISA)* bus - and uses standard TTL components (such as **74LS08**, **74LS688**, **74LS244**, etc) to implement a bus interface; [2] describes a device that interfaces through the *Universal Serial Bus (USB)* - and uses the **FT232** chip by FTDI to implement a bus interface

[9]noting that, in principle, the driver should be able to handle *multiple* cards; and be able to individually address each one

has facilities to address such needs too - as well as having a so-called *mixer* interface[10] - which will not be discussed here.

Application level From the PC perspective, a high-level audio software (audio application) is used, in first instance, to issue start and stop of audio playback or capture. When such a high-level command is issued by the user, the audio application communicates to the driver through the application-level API and: obtains a handle to the relevant structures; initializes and allocates variables; opens the PCM device; specifies hardware parameters[11] and type of access (interleaved or not) - and then starts with reading from (for capture) or writing to (for playback) the PCM device, by using **ALSA** API functions (such as `snd_pcm_writei`/`snd_pcm_writen` or `snd_pcm_readi`/`snd_pcm_readn`) [16]. The PCM device is representation of a source (or destination) of an audio stream[12]. The kernel responds to the application API calls by calling the respective code in the kernel driver, implemented using the kernel (**ALSA** driver) API [8].[13]

4 Concept of `minivosc`

A user would, arguably, expect to hear actual reproduced sound upon clicking 'play'; while recording, in principle, doesn't involve user sensations other than indication by the audio software (e.g. rendering of captured audio waveform). Taking this into account, it becomes clear that the stated purpose of **minivosc** - to be a 'virtual' driver (independent of any actual additional soundcard hardware) - can only be demonstrated in the *capture* direction[14]: as the driver simply has references to data arrays in memory, the effect of *playing back* (i.e., copying) data *to* non-existing hardware will be pretty much undetectable[15]. However, even with non-existing hardware, we can always write some sort of predefined or random data to the capture buffers in memory - which would result with visible incoming data in the high-level audio software (like when performing 'record' in **Audacity**).

To avoid the conceptualization problems of **ALSA** devices vs. subdevices, the **minivosc** driver is deliberately defined as a mono, 8-bit, capture-only driver, working at 8 kHz (the next-lowest[16] rate ALSA supports). The 8-bit resolution allows also for direct correspondence between: the digital representation of a single analog sample; and the storage unit of the corresponding arrays (buffers) in memory, which are defined as `char*`. Hence, one byte in memory buffer represents one analog sample, the next byte represents the next analog sample, etc. This allows for simplification of the process of wrapping data in a ring buffer, and thus easier grasping of the remaining key issues in **ALSA** driver implementation.

5 Driver structures

The **minivosc** driver contains four key structures - three of which are required by (and based on predefined types in) the **ALSA** framework:

- `struct` variable of type `snd_pcm_hardware` (required) - sets the allowed sample formats, sampling rates, number of channels, and buffering properties
- `struct` variable of type `snd_pcm_ops` (required) - assigns the actual functions, that should respond to predefined **ALSA** callbacks
- `struct` variable of type `platform_driver` (required) - named `minivosc_driver`, it describes the driver, and at the same time, determines the bus interface type
- `struct` variable of type `minivosc_device` - custom structure that contains all other parameters related

[10] which allows for, say, individual volume control directly from the main OS volume applet

[11] access type, sample format, sample rate, number of channels, number of periods and period size

[12] and it can have: "plughw" or "hw" interface; playback or capture direction; and standard (blocking), non-blocking and asynchronous modes (see also [7, 'PCM (digital audio) interface'])

[13] Note that the application doesn't have to talk to the driver directly; there could be intermediate layers, forming a **Linux** audio software stack (see [17]). However, in this paper, we focus solely on the perspective of the **ALSA** kernel driver.

[14] however, note that **aloop-kernel.c**[13], is also a 'virtual' driver, and yet works in both directions; however, since it is intended to 'loop back' audio data between applications and devices[18], the virtual setups possible can be reduced to the case when the 'loopback' driver routes one audio application's data written to its playback interface, back to its capture interface; and another audio application grabs data from the 'loopback' capture interface and writes it to disk.

[15] similar to, in **Linux** parlance, 'piping' data to `/dev/null`. While a specific consumer of such data could be programmed, that alone complicates the understanding of interaction between typical audio software and drivers

[16] The lowest **ALSA** rate being 5512 Hz, see **include/sound/pcm.h** in **Linux** source [19]

to the soundcard, as well as pointers to the digital audio (PCM) data in memory

The `minivosc_driver` struct variable defines the `_probe` and `_remove` functions, required for any **Linux** driver; however, by choosing the struct type, we also determine the type of bus this driver is supposed to interface through. For instance, a PCI soundcard driver would be of type `struct pci _driver`; whereas a USB soundcard driver would be of type `struct usb_driver` (see [1]). However, **minivosc** is defined as `platform_driver`, where "*platform devices are devices that typically appear as autonomous entities in the system* [20, 'platform.txt']" - and as such, it will not need actual hardware present on any bus on the PC, in order for the driver to be loaded completely[17].

The `snd_pcm_ops` type variable simply points to the actual functions that are to be executed as the predefined **ALSA** callbacks, which are discussed in the next section. The different fields in `snd_pcm _hardware` allow the device capabilities in terms of sampling resolution (i.e., analog sample format) and sampling rate to be specified. For this purpose, there are predefined bit-masks in **ALSA**'s **pcm.h** [19], such as `SNDRV_PCM_RATE_8000` or `SNDRV_ PCM_FMTBIT_U8` (for 8 kHz rate, or for sample format of 8-bit treated as unsigned byte, respectively). One should be aware that audio software may treat these specifications differently: for instance, having **arecord** capture from the **minivosc** driver, will result with an 8-bit, 8 kHz audio file - simply because that is the default format for **arecord**. On the other hand, **Audacity** in the same situation - while acknowledging the driver specifications - will also internally convert all captures to the default 'project settings', for which the minimum possible values are 8000 Hz and 16-bit [21].[18]

One of the most important structures is what we could call the 'main' *device* structure, here `minivosc_device`. It can also be a bit difficult to understand, especially since it is - in large part - up to the driver authors themselves to set up the structure, and its relationships to built-in **ALSA** structures. These relationships are of central interest, because a driver author *must* know the location of memory representing the digital audio streams (`snd_pcm_runtime->dma_area` in Fig. 2), in order to implement *any* digital audio functionality of the driver. And finding this memory location is not trivial - which is maybe best presented in graphical manner, as in Fig. 2, which shows a partial scope of the 'main' structure `minivosc_device` and its relationships.

On Fig. 2, only `minivosc_device` has been written as part of the driver code - all other structs (with darker backgrounds) are built-ins, provided by **ALSA**. Pointers are shown on left edge of boxes; self-contained struct variables are on the bottom edge[19]. Some relationships (such as `snd_pcm_substream->runtime` to `snd_pcm_runtime` pointing) are set up internally by **ALSA**; the relationships to the 'main device' structure (`minivosc_device`) have to be coded by the driver author. Further complication is that the authored relationships can *not* be established at the same spot in the driver code - as some structures become available only in specific **ALSA** callbacks.

This is a conceptual departure from the typical basic understanding of program execution - where a predetermined sequential execution of commands is assumed. Instead, driver programming may conceptually be closer to GUI programming, where the author typically writes *callback functions* that run whenever a user performs some action. Additionally, we can expect to encounter different amount of instances of some of these structs! For example, `snd_pcm_substream` can carry data for a given output connector, which could be stereo. So, if a stereo file is loaded in audio software, and 'play' is clicked - we could expect **ALSA** to pass a *single* `snd_pcm_substream`, carrying data for both channels, to our driver. However, if we are trying to play a 5.1 surround file, which em-

[17] which is not the default behavior for actual hardware drivers - they will simply not run some of their predefined callbacks, if the hardware is not present on the bus

[18] While these captures can be exported from **Audacity** as 8-bit, 8 kHz audio files - that process implies an *additional* conversion from the internal 16-bit format.

[19] Note, the **ALSA** struct boxes show only a small selection of the structs' actual members; while the 'main device' struct still contains some unused variables, leftover from starting example code. Connections are colored for legibility.

Unlike a more detailed UML diagram, a map like Fig. 2 helps only in a specific context: e.g., the driver is supposed to write to the `dma_area` when the `_timer_function` runs, however this function provides a reference to `minivosc_ device`; the map then allows for a quick overview of structure field relationship, so a direct pointer to the `dma_area` can be obtained for use within the function.

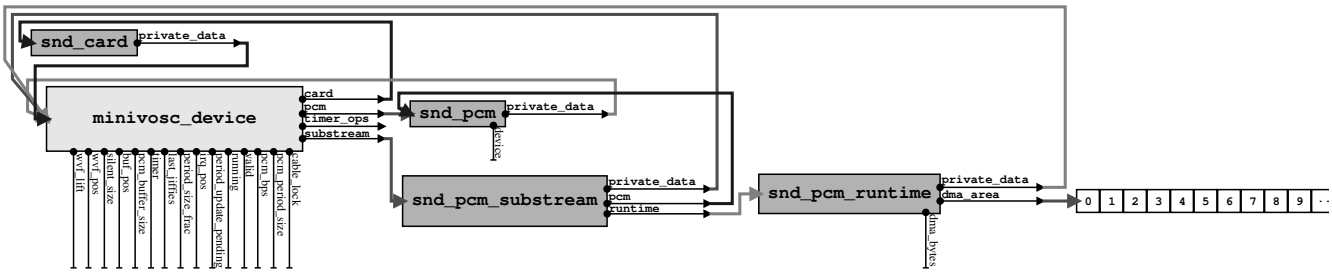


Figure 2: Partial 'structure relationship map' of the `minivosc` driver.

ploys 2 stereo and one mono connector - we should expect *three* `snd_pcm_substream`s to be passed to our driver. This could further confuse high-level programmer newcomers, that might expect to receive something like an *array* of substreams in such a case: instead, **ALSA** may call certain callbacks multiple times - and it is up to the driver author to store references to these substreams.

minivosc avoids these problems as a mono-only driver - thus within the code, we can expect only one instance of each struct shown on Fig. 2; and the reference to the only `snd_pcm_substream` can be found directly on the main 'device' struct, `minivosc_device`. This allows us easier focus on another important aspect of **ALSA** - the timing of execution of callbacks, which is necessary for understanding the driver initialization process in general.

6 Execution flow and driver functions

The device driver architecture of **Linux** specifies a driver model [20], and within that, certain callback functions that a driver should expose. In the case of **minivosc**, first the `__init` and `__exit` macros ([22, Chapter 2.4]) are implemented, as functions named `alsa_card_minivosc_init` and `alsa_card_minivosc_exit`. These functions run *when a driver module is loaded and unloaded*: the kernel will automatically load modules, built in the kernel, at boot time - while modules built 'out of tree' have to be loaded manually by the user, through the use of the **insmod** program. The `_init` function in **minivosc** registers the driver, and attempts to iterate through the attached soundcards. As **minivosc** is a 'platform' driver, and there is no actual hardware - the `_init`, in this case, is made to always result with detecting a single (virtual) 'card'. Next in line of predefined callbacks are `_probe` and `_remove` [20, 'driver.txt'], in **minivosc** implemented as `minivosc_probe` and `minivosc_remove`. In principle, they would run *when a (soundcard) hardware device is attached to/disconnected from the PC bus*: for instance, `_probe` would run when the user connects a USB soundcard to the PC by inserting the USB connector - if the driver is already loaded in memory. For permanently attached devices (think PCI soundcards), `_probe` would run immediately after `_init` detects the cards; thus, in the case of **minivosc**, `_probe` will run immediately after `_init`, at the moment when the driver is loaded (by using **insmod**).

The **minivosc** driver code informs the system about which are its init/exit functions, by use of `module_init`/`module_exit` facility (see [23, 'Chapter 2']); while it specifies which are its probe/remove functions through use of the `platform_driver` structure. Finally, last in line of predefined callbacks are the **ALSA** specific callbacks; the driver code tells the system which are these functions, through the predefined **ALSA** struct `snd_pcm_ops`.[20]

While **ALSA** may define more `snd_pcm_ops` callbacks [9], there are 8 of them being used in **minivosc**, by assigning them to functions: one, `snd_pcm_lib_ioctl`, being defined by **ALSA** – and seven `snd_pcm_ops` functions written as part of **minivosc**: `minivosc_pcm_open`, `minivosc_hw_params`, `minivosc_pcm_prepare`, `minivosc_pcm_trigger`, `minivosc_hw_free`, `minivosc_pcm_close`, `minivosc_pcm_pointer`. As clarification - here is the order of execution of above callbacks for the **minivosc** driver, for some common events:

- driver loading: `_init`, then `_probe`
- start of recording: `_open`, then `_hw_params`, then

[20] Note that the term 'PCM' is used in **ALSA** to refer *generally* to aspects related to digital audio - and *not* to the particular 'Pulse Code Modulation' method as known from electronics (although that is where the term derives from [7, 'PCM (digital audio) interface']).

_prepare, then _trigger
- end of recording: _trigger, then _hw_free, then _close
- driver unloading: _exit, then _remove

We already mentioned that for the **minivosc** driver, loading/unloading events happen when the insmod/rmmod commands are executed. 'Start of recording' event would be the moment when the 'record' button has been pressed in **Audacity**; or the moment when we run arecord from the command line – correspondingly, 'end of recording' event is when we hit the 'stop' button in **Audacity**; or when arecord exits (if, for instance, it has been set to capture for only a certain amount of time). However, note that – *even* with all of this in place – the actual performance of the driver in respect to digital audio is still *not defined*; memory buffer handling is also needed.

6.1 Audio data in memory (buffers) and related execution flow

As noted in Sec. 5 'Driver structures', one of the central issues in **ALSA** driver programming is the location in memory, where audio PCM data for each substream is kept - the dma_area field being a pointer to it. In principle, each substream can carry multi-channel data: for instance, a 16-bit sample would be represented as two consecutive bytes in the dma_area; while stereo samples could be interleaved [24]. Thus **ALSA** introduces the concept of frames [25], where a *frame* represents the size of one analog sample for all channels carried by a substream. As **minivosc** is specified as a mono 8-bit driver, we can be certain that each byte in its dma_area will represent a single sample - and that one frame will correspond to exactly one byte.

The approach to implementing the sampling rate that **minivosc** has (taken from [13]), is to use the **Linux** *system timer* ([26, 'Kernel Mechanisms'], [23, 'Chapter 6']). Note that standard **Linux** system timers are "*only supported at a resolution of 1 jiffy. The length of a jiffy is dependent on the value of HZ in the Linux kernel, and is 1 millisecond on i386* [27]". However, there also exist so-called *high-resolution timers* [28] (for their basic use in **ALSA**, see [12]).

6.2 The sound of minivosc - Driver execution modes

The driver writes in the dma_area capture buffer repeatedly (as controlled by timers), within the _xfer_buf function - or more precisely, within the minivosc_fill_capture_buf function called by it. In the **minivosc** code, three different variants can be chosen (at compile time), for copying a small predefined 'waveform grain' array repeatedly in the capture buffer, which results in an audible oscillation when the capture is played back (hence *oscillator* in the name). Note the need to 'wrap' the writing to the capture buffer array, since in **ALSA**, it is defined as a *circular* or *ring buffer* [24]. Finally, all of the three 'audio generation' algorithms can be commented, in which case the **minivosc** driver will simply write a constant value in the buffer. There is an additional facility, called 'buffermarks', which indicate the start and end of the current chunk, as well as the start and end of the dma_area - which can be used to visualize buffer sizes.

7 Conclusions

The main intent of **minivosc** is to serve as a basic introduction to one of the most difficult issues in soundcard driver programming: handling of digital audio. Given that many newcomers may have previous acquaintance with 'userland' programming, the conceptual differences from user-space to kernel programming (including debugging [1]) can be a major stumbling block. While a focus on capture only, 8-bit / 8 kHz mono driver leaves out many of the issues that are encountered in working with real soundcards, it can also be seen as a basis for discussion of [2], which demonstrates full-duplex mono @ 8-bit / 44.1 kHz (and can interface with stereo, 16-bit playback). Thus, the main contribution of this paper, driver code and tutorial would be in easing the learning curve of newcomers, interested in **ALSA** soundcard drivers, and digital audio in general.

8 Acknowledgments

The authors would like to thank the Medialogy department at Aalborg University in Copenhagen, for the support of this work as a part of a currently ongoing PhD project.

References

[1] S. Dimitrov, "Minivosc homepage," WWW: http://www.alsa-project.org/main/index.php/

Minivosc / http://imi.aau.dk/~sd/phd/index.php?title=Minivosc, 21 Dec 2010.

[2] S. Dimitrov and S. Serafin, "Audio Arduino - an ALSA (Advanced Linux Sound Architecture) audio driver for FTDI-based Arduinos," in *Proceedings of the 2011 conference on New interfaces for musical expression*, 2011.

[3] ——, "Towards an open sound card — a barebones FPGA board in context of PC-based digital audio," in *Proceedings of the 2011 Audio Mostly conference*, 2011.

[4] S. Dimitrov, "Extending the soundcard for use with generic DC sensors," in *NIME++ 2010: Proceedings of the International Conference on New Instruments for Musical Expression*, 2010, pp. 303–308.

[5] D. Phillips, "A User's Guide to ALSA | Linux Journal," WWW: http://www.linuxjournal.com/node/8234/print, 2005.

[6] git.alsa project.org, "alsa-kernel.git/tree - Documentation/," WWW: http://git.alsa-project.org/?p=alsa-kernel.git;a=tree;f=Documentation.

[7] J. Kysela, A. Bagnara, T. Iwai, and F. van de Pol, "ALSA project - the C library reference," WWW: http://www.alsa-project.org/alsa-doc/alsa-lib/index.html, 22 Dec 2010.

[8] T. Iwai, "The ALSA Driver API," WWW: http://www.alsa-project.org/~tiwai/alsa-driver-api/index.html, 21 Dec 2010.

[9] ——, "Writing an ALSA Driver," WWW: http://www.alsa-project.org/~tiwai/writing-an-alsa-driver/, 21 Dec 2010.

[10] B. Collins, "Writing an ALSA driver," WWW: http://ben-collins.blogspot.com/2010/04/writing-alsa-driver.html, 21 Dec 2010.

[11] S. K., "HowTo Asynchronous Playback - ALSA wiki," WWW: http://alsa.opensrc.org/index.php/HowTo_Asynchronous_Playback

[12] J. Kysela, "sound/drivers/dummy.c," WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=sound/drivers/dummy.c, 22 Dec 2010.

[13] J. Kysela, A. İnan, and T. Iwai, "drivers/aloop-kernel.c," WWW: http://git.alsa-project.org/?p=alsa-driver.git;a=blob_plain;f=drivers/aloop-kernel.c;hb=e0570c46e3c4563f38e44a25cfac1f07ff5a02a8, 2010.

[14] www.alsa project.org, "Asoundrc - AlsaProject," WWW: http://www.alsa-project.org/main/index.php/Asoundrc, 22 Dec 2010.

[15] V. Schatz, "A close look at ALSA," WWW: http://www.volkerschatz.com/noise/alsa.html, 2010.

[16] M. Nagorni, "ALSA Programming HOWTO v.0.0.8," WWW: http://www.suse.de/~mana/alsa090_howto.html, 15 May, 2011.

[17] G. Morrison, "Linux audio uncovered," *Linux Format magazine*, no. 130, pp. 52–55, April 2010, URL: http://www.tuxradar.com/content/how-it-works-linux-audio-explained.

[18] J. Kysela, "snd-aloop and alsaloop notes," WWW: http://people.redhat.com/~jkysela/RHEL5/loop/BACKGROUND, 22 Dec 2010.

[19] J. Kysela and A. Bagnara, "git.kernel.org - include/sound/pcm.h," WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=blob;f=include/sound/pcm.h;, 2010.

[20] git.kernel.org, "Documentation/driver-model," WWW: http://git.kernel.org/?p=linux/kernel/git/stable/linux-2.6.32.y.git;a=tree;f=Documentation/driver-model, 22 Dec 2010.

[21] wiki.audacityteam.org, "Bit Depth - Audacity Wiki," WWW: http://wiki.audacityteam.org/wiki/Bit_Depth, 22 Dec 2010.

[22] P. J. Salzman, M. Burian, and O. Pomerantz, *The Linux Kernel Module Programming Guide*. CreateSpace, 2009, URL: http://linux.die.net/lkmpg.

[23] A. Rubini and J. Corbet, *Linux device drivers*. O'Reilly Media, 2001, URL: http://www.xml.com/ldd/chapter/book.

[24] J. Tranter, "Introduction to Sound Programming with ALSA | Linux Journal," 2004, WWW: http://www.linuxjournal.com/article/6735?page=0,1.

[25] www.alsa project.org, "FramesPeriods - AlsaProject," WWW: http://www.alsa-project.org/main/index.php/FramesPeriods, 28 Dec 2010.

[26] D. Rusling, "The linux kernel," *The Linux Documentation Project*, 1996, URL: http://www.tldp.org/LDP/tlk/.

[27] elinux.org, "High Resolution Timers - eLinux.org," WWW: http://elinux.org/High_Resolution_Timers, 28 Dec 2010.

[28] T. Gleixner and D. Niehaus, "Hrtimers and beyond: Transforming the linux time subsystems," in *Proceedings of the Ottawa Linux Symposium, Ottawa, Ontario, Canada*, 2006, URL: http://www.kernel.org/doc/ols/2006/ols2006v1-pages-333-346.pdf.

[29] M. Johnson *et al.*, *Linux Kernel Hackers' Guide*. Johnson, 1993, URL: http://www.tldp.org/LDP/khg/HyperNews/get.

An Open-Source C++ Framework for Multithreaded Realtime Multichannel Audio Applications

Matthias GEIER[1], **Torben HOHN**[2] and **Sascha SPORS**[1]

[1]Quality & Usability Lab, TU Berlin
Ernst-Reuter-Platz 7
10587 Berlin, Germany
matthias.geier@tu-berlin.de
sascha.spors@tu-berlin.de

[2]Linutronix GmbH
Auf dem Berg 3
88690 Uhldingen, Germany
torbenh@linutronix.de

Abstract

An open-source C++ framework is introduced which facilitates the implementation of multithreaded realtime audio applications, especially ones with many input and output channels. Block-based audio processing is used.

The framework is platform-independent and different low-level audio backends can be used for both realtime and non-realtime operation. Support for further backends can be added easily.

Keywords

MIMO, realtime, multithreading, C++

1 Introduction

Most realtime audio processing applications which have to handle a high number of input and output channels are bound to a specific realtime audio framework. This is not a problem in itself, because there are several very good frameworks available. Some of them can be used on several platforms and some of them even have the possibility to switch between different audio backends.

Such frameworks are perfectly suitable to create stand-alone applications. But what if you want to compare the audio output of your program with the prototype algorithm you realized in your favorite software for numerical processing? Wouldn't it be nice to create a shared library for this software directly from your realtime C++ code? And while we are at it, wouldn't it be nice to use the same source code to create a plugin for your favorite graphical patching software?

Multichannel applications often have an embarrassingly high potential for parallel computation on multi-processor/multi-core computers, especially if they have many output channels. Wouldn't it be nice if parallel processing would be automatically provided in all the aforementioned scenarios?

To tackle these challenges, this paper presents a new C++ framework for interactive multiple-input/multiple-output (MIMO) audio applications. It is part of the *Audio Processing Framework* (APF)[1], which is free software released under the GNU General Public License (GPL)[2].

Applications created with this framework are automatically capable of multithreading. The number of audio threads is chosen by the user and does not change during runtime. The scheduling mechanism is simple and static, yet effective (see section 5). Applications are not bound to a specific audio driver. Audio backends for both realtime and non-realtime operation can be switched easily (at compile time) and new backends can be added by the user. This way, it is possible to implement audio algorithms for a realtime application, but the algorithms can nevertheless be evaluated block-by-block in a non-realtime manner. For more information about the audio backends and their usage see section 4.6.

2 Target Applications

The presented framework can be used for any block-based audio application with many input and output channels. The overall topology of the audio processing graph should not change too much during runtime, but the number of channels and other system parameters may change dynamically. Typical applications are sound field synthesis, multichannel echo cancelling and loudspeaker/microphone beamforming. The target systems range from stereo processing on a simple laptop to dedicated computer systems driv-

[1] http://tu-berlin.de/?id=apf
[2] http://gnu.org/copyleft/gpl.html

ing tens or even hundreds of loudspeaker channels with as many input channels. The first application using the framework is the *SoundScape Renderer* (SSR)[3], a tool for object-based spatial audio reproduction providing a variety of rendering algorithms [Geier et al., 2008].

3 Realtime & Non-Realtime Threads

An application based on the presented framework uses two different kinds of threads. The actual audio processing is mostly done in a callback function which is called successively – for every audio block – from the audio backend. The thread running this callback function is called *realtime thread* because only a given time period is available for the computation of each audio block. Therefore, only a certain fixed amount of calculations may be done, depending on block size, sampling rate, processor speed and other factors. Additionally, no blocking functions may be called in the realtime thread. Especially operations like allocating memory, reading and writing of files and sockets, creating and joining threads and waiting for mutexes have to be avoided. If processing of an audio block is not finished at the required time, the output data is not ready on time and has to be discarded, leading to errors in the output signal.

The other kind of thread is simply called *non-realtime thread*. Threads of this kind handle input from the user interface, read and write files, reserve and free memory, communicate via network and do anything else which is not related to audio processing.

The timing of the audio callback function is normally bound to the soundcard, which imposes the mentioned realtime constraints. However, – as described in section 4.6 – there is a special audio backend which can be used for offline processing. In this case, realtime-safety would not be needed anymore. The thread running the audio callback function should nevertheless be considered as realtime thread. Audio algorithms should be backend-agnostic and assume to be potentially used in a realtime context.

In interactive applications, information must be transferred from the non-realtime thread to the realtime thread. This should not be done by writing to and reading from the same memory location in both threads, respectively, because this can lead to data corruption. A mutex cannot be used either, because this would be realtime-unsafe. Instead, a lock-free queue is used (see section 4.3).

If more memory is needed for audio processing or if unused memory shall be freed, this may not be done in the realtime thread. Therefore, if new memory is needed, it is allocated and initialized in the non-realtime thread and then a pointer to the new data is transferred to the realtime thread in a realtime-safe way. This is realized by means of the realtime-safe list described in section 4.4.

4 Components

The framework is implemented in C++ using the Standard Template Library (STL). Many structural decisions are made at compile time to avoid unnecessary runtime overhead. In most cases, generic programming is preferred over classic object-oriented programming. Dynamic polymorphism is only used where actually needed.

The framework does not define a special data type for audio data, only the audio backends (see section 4.6) define a sample type (in most cases – but not necessarily – `float`). Within the audio algorithms themselves, arbitrary data types can be used.

The following sections describe the main components of the framework.

4.1 Lock-free Ringbuffer

The key component to make the framework both realtime-safe and thread-safe is a lock-free ringbuffer. It is available as the class `LockFreeFifo<Command*>` and it is only thread-safe for single-reader/single-writer access. This means that only one single thread is allowed to use the `push()` function to write data to the ringbuffer and only one other thread is allowed to use the `pop()` function to retrieve data from the ringbuffer. It has to be ensured by the programmer that each function is only used in the appropriate thread.

4.2 Command Queue

Two instances of the lock-free ringbuffer are used in the `CommandQueue`, a lock-free queue for arbitrary user-defined commands which can be defined by implementing the member functions `execute()` and `cleanup()`.

[3] `http://tu-berlin.de/?id=ssr`

Every command object is created and initialized in a non-realtime thread, all necessary memory is allocated at this time. A pointer to the fully constructed command object is pushed to an instance of `LockFreeFifo<Command*>`. The main realtime thread periodically (usually once per audio block) processes all items from this queue with the function `process_commands()`, which in turn invokes the virtual `execute()` function on each command object. Of course, the `execute()` function is not allowed to use any realtime-unsafe functions. After execution, it is not safe to deallocate the memory used by the command object within the realtime thread. Therefore, the command object is pushed into another instance of `LockFreeFifo<Command*>` to be deallocated in the non-realtime thread. Before that, the virtual `cleanup()` function is called from the non-realtime thread.

To add a command to the `CommandQueue`, the member function `push()` is used, the function `wait()` can be used to wait until the command is actually executed and cleaned up. No actual commands are defined by the `CommandQueue`, only the abstract base class `Command` from which arbitrary commands can be derived as long as they provide an implementation for the `execute()` and `cleanup()` member functions.

The transport of information from the realtime thread to the non-realtime thread also has to be triggered by the latter. For that, commands can be defined which query the information in the `execute()` function and provide it to the non-realtime thread in the `cleanup()` function.

Typically, an application holds exactly one instance of `CommandQueue` which is responsible for all the communication to and from the realtime thread. If there are several non-realtime threads – for example for a graphical user interface and a network interface which work in parallel – access to the `CommandQueue` has to be locked with a mutex. This is not done automatically.

4.3 Thread-safe Data Access

To avoid parallel writing and reading at the same memory location from a realtime and a non-realtime thread, respectively, the class `SharedData<T>` can be used. It wraps a single variable of any standard or user-defined type. Internally, it uses a reference to a `CommandQueue` to push a custom `Command` which holds a copy of the value that is assigned to the actual variable in the `execute()` function.

4.4 Realtime-safe List

To handle a dynamic number of audio channels, the realtime-safe data structure `RtList<Item*>` is used. In the non-realtime thread, list elements are created, initialized and added to the list with the `add()` function and elements are removed with the `rem()` function, whereby the deallocation of any memory also happens in the non-realtime thread. Communication between threads is again handled by means of the `CommandQueue`. In the realtime thread, the relevant functions of a `std::list` can be used, namely `begin()`, `end()`, `empty()` and `size()`.

When adding items to the `RtList`, ownership is passed to the list. That means the responsibility to destroy the object at an appropriate time is transferred to the list and user code may not call `delete` on an object owned by an `RtList`.

4.5 MIMO Processor

The class `MimoProcessor` is the base class for the signal processing part of an application. When deriving from `MimoProcessor`, several template arguments have to be used. The first one is the deriving class itself. This C++ idiom is called Curiously Recurring Template Pattern (CRTP) [Coplien, 1995] and it is used to achieve compile-time polymorphism. The rest of the template arguments are so-called policy classes. The user can choose from a given set of policies and even implement new ones. This programming strategy is called policy-based class design [Alexandrescu, 2001]. One of the policies selects the audio backend which is described in section 4.6.

The `MimoProcessor` incorporates one instance of `CommandQueue` for all realtime-safe and thread-safe operations. Instances of `RtList` and `SharedData` can be used with this queue.

4.6 Audio Backends

The `MimoProcessor` base class is not limited to a specific audio backend. By means of policy-based design, the audio backend can be specified as template argument at compile time.

Currently, the JACK Audio Connection Kit

(JACK)[4] is supported (`jack_policy`), support for PortAudio[5] is underway and further audio backends can be added easily.

A special policy class is the `pointer_policy`, which can be used with any C or C++ program; in this case, the audio callback function has to be called explicitly. This can be used in a realtime context, for example in an External for *Pure Data*[6] and *Max*[7] (utilizing the *flext* library). However, it can also be used in a non-realtime context for offline processing, for example in a MEX-file (shared library) for *Matlab*[8] and *GNU octave*[9] or in an application which reads all inputs from a multichannel audio file and writes all outputs to another multichannel file. Example code is available for all mentioned applications.

4.7 Crossfade

When doing block-based processing, the input signal is divided into consecutive blocks of audio data and it is assumed that all parameters of the algorithm are constant during one audio block.

Parameter changes only happen between blocks and this can very often lead to audible artifacts due to discontinuities in the resulting output signal. One way to reduce these errors is to calculate each block two times – once with the parameters of the previous block and once with the current parameters – and make a crossfade between the two resulting blocks. With large parameter changes and small blocksizes there may still be audible artifacts, but in most cases the transitions become inaudible.

The described crossfade functionality is built into the `MimoProcessor`. Only if parameters change – as noticed by the `CommandQueue` – the specified audio algorithm is processed twice and a crossfade is done automatically. In static scenarios the overhead of crossfading may not be wanted. Therefore, the whole functionality can be switched off at compile time.

5 Parallel Processing

The desired audio processing algorithm has to be organized in several lists of type `RtList<Item*>`. These lists hold polymorphic base class pointers and allow the execution of the virtual function `Item::process()` on each list item. To do the actual signal processing, item classes have to be defined – derived from the class `Item` – and they have to implement their signal processing algorithms in the `process()` function. By implementing the list items and assembling them into lists, the programmer can establish the overall audio processing graph. The different lists can be seen as *stages* of the algorithm. Lists are processed one after each other, items within one list are processed in parallel by several threads.

The `process()` function is called from a realtime thread and has to fulfill the following constraints. It may write data only to places where the item has exclusive access, either in member variables of the object itself or in dedicated memory areas elsewhere. It may read data from any of the `RtList`s except the list where its object belongs to, because other list elements could be written to at the same time by another realtime thread. The amount of memory used by a list item may not change during its lifetime. It must be allocated in the non-realtime thread and when the item is removed, its memory must also be deallocated in the non-realtime thread. This happens automatically if the `add()` and `rem()` functions of the `RtList` are called from the non-realtime thread.

Especially in applications with a dynamic number of channels, `RtList`s can be iterated over by means of the `begin()` and `end()` functions which reflect possible changes in the length of the list. In less dynamic scenarios, references to other objects – as long as they will not be removed during runtime – can also be specified at initialization of the list item.

Parallel processing is achieved by automatically executing different `process()` functions in different realtime threads which can run in parallel on multi-processor/multi-core computers. The number of realtime threads can be specified by the user, according to the available resources, and this number does not change during runtime. If N threads are requested, $N-1$ *worker threads* are created. The *main audio thread* – the one where the audio callback function is called from – is normally created and controlled by the audio backend.

[4] `http://jackaudio.org`
[5] `http://portaudio.com`
[6] `http://puredata.info`
[7] `http://cycling74.com/products/max`
[8] `http://mathworks.com/products/matlab`
[9] `http://gnu.org/software/octave`

The distribution of list elements to the audio threads is very simple. The main audio thread gets every N-th item starting with the first, the first worker thread gets every N-th item starting with the second and so on until the $(N-1)$th worker thread, which gets every N-th item starting with the N-th. This very basic scheduling mechanism may not be the best choice for arbitrary audio processing graphs, but it works quite well for the targeted application areas. It works best if the length of the lists is much bigger than the number of threads and if the amount of work done in the `process()` function is not too different in all items of a given list.

The main audio thread and the worker threads are synchronized with semaphores. Each worker thread does a very simple cycle of operations repeatedly. It waits until signaled from the main audio thread by a semaphore and then processes all items of the current list which are chosen by the aforementioned scheduling algorithm. When finished, it signals to the main audio thread and waits again until the next iteration. The cycle of the main audio thread can be defined by the user. Typically, several lists are processed one after each other with the function `_process_list()`. Within this function, the specified list is set as current list, the worker threads are signaled via their semaphores to start processing and the main audio thread itself then also processes the appropriate items of the current list. Afterwards, the main audio thread waits until all worker threads have signaled that they are done with their parts of the list.

6 Additional Components

As mentioned in the introduction, all presented components are part of the Audio Processing Framework (APF). Additionally, several other components are included in the APF, among them a delay line, a partitioned convolution engine (using the FFTW library[10]), several specialized iterator classes, math operations and many helper functions. For a full list of features and more detailed information see the online documentation[11].

[10] `http://fftw.org`

[11] `http://dev.qu.tu-berlin.de/projects/apf`

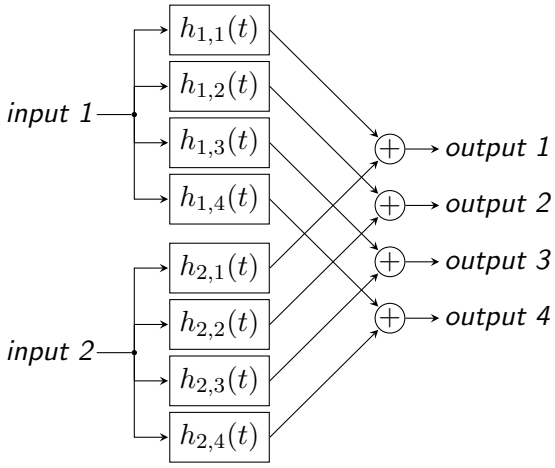


Figure 1: Example MIMO system with 2 input channels and 4 output channels. In the general case of N inputs and M outputs, $N \times M$ filters are needed.

7 An Example

Figure 1 shows a generic MIMO system with a filter between each input and each output. The main objective of the programmer is to identify which parts of the algorithm shall form separate objects, how they depend on other objects and which of these objects can be processed in parallel. In the example, three kinds of objects come to mind – the input channels, the filters and the additions (which can be combined with the output channels).

To implement the filter class, the convolver which was mentioned in the previous section can be used. A filter object is always bound to one specific input channel, so a reference can be specified at initialization and stored within the filter object. Because of that, the `process()` function of the filter object does not need to iterate over the whole list of inputs but can rather get the input data directly via the internal reference to the corresponding input channel. Input data is processed by the convolver and the result is written to a member variable inside the filter object. Apart from these processing instructions, the filter object must also provide means for the following stages of the algorithm to read the output data of the filter. This is normally done by providing (read-only) `begin()` and `end()` functions which return iterators to the data. Of course, the filter class has to be derived from the class `Item`

so that all filter objects can be added to a list of type `RtList<Item*>`. Having a dynamic number of inputs is no problem. Whenever an input is added, a whole set of filter objects – one for each output channel – has to be created an added to the list at once. To safely switch filter coefficients during runtime, the `CommandQueue` is used.

The other class which has to be defined in this example is for the addition, which can be combined with the output channels. Each addition corresponds to one unique output channel. The `process()` function of the addition object must iterate over all filter objects and find out which of the filter outputs must be added. Therefore, each filter object must contain a flag of some kind to be associated with a certain output channel. If the number of inputs is supposed to change dynamically, this information cannot be stored in the addition object itself because this would require dynamic memory allocation which is not allowed in a realtime context. Once all relevant filter data is added, the result can be written directly to the corresponding output.

Most of the work is done in the `process()` functions. The only thing left to do is to call `_process_list()` for each of the defined lists in the main audio callback function.

8 Acknowledgements

Thanks to Till Rettberg for many helpful suggestions and discussions. This work was partly funded by German Research Foundation (DFG) grant FOR 1557.

References

Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley.

James O. Coplien. 1995. The column without a name: Curiously recurring template patterns. *C++ Report*, 7(2):24–27.

Matthias Geier, Jens Ahrens, and Sascha Spors. 2008. The SoundScape Renderer: A unified spatial audio reproduction framework for arbitrary rendering methods. In *124th Convention of the Audio Engineering Society*.

Using the BeagleBoard as hardware to process sound

RAFAEL VEGA
El Software Ha Muerto
Medellín, Colombia
rvega@elsoftwarehamuerto.org

DANIEL GÓMEZ
Grupo Leonardo, Universidad ICESI
Cali, Colombia
dgomez@icesi.edu.co

Abstract

This paper describes the implementation of diverse layers of code to enable an open source embedded hardware device to run audio processing pure data patches. The evolution of this implementation is reviewed describing different approaches taken by the authors in order to find optimal software and hardware settings. Although some problems are detected when running specific patches, the system has a conjunction of features that are relevant for the floss audio community.

Keywords

Beagleboard, Puredata, sound processing, sound synthesis, opensource.

1 Introduction

Although there is a large number of open-source software tools for working with audio, there are few open-hardware alternatives for musical applications. There is a need to explore the possibilities of open hardware in conjunction with open source software to close the gap between musicians, music producers and the hardware used to create their sounds. It is foreseeable that the closer the sound workers get to the tools they use, the more expressiveness and creativity that can be achieved.

In the Latin American context, there has been an evolution of software design for musical and sonic scenarios along with different MIDI or OSC interfaces attached to laptops [5] [6] [12]. This experimental and DIY tools are designed and used in specific concert or installation situations. Due to their low cost and ease of production we have wondered whether these tools could be a solution for musicians that can barely afford a commercial electronic musical instrument. This project is the first step to explore such platform.

This paper describes the diverse approaches taken to build a stable platform for processing and synthesizing audio signals using an open source, low cost, portable computer such as the BeagleBoard. To achieve this goal, diverse software APIs were explored and the Linux stack, libpd [3] and JACK server were chosen for the current implementation. We present the work so far which is a DSP engine that can parse pd patches and run in a BeagleBoard. Although the project should include physical interfaces that allow for changing parameter values at run time, these features have not been implemented at this early stage.

2 Related work

Some open source tools for musicians exist with varying price ranges, programmability and scope. A common approach has been to develop hardware and firmware and release it as GPL or CC, so that users are able to experiment with the assembly, and possible tweaking of hardware and software. This approach is taken by projects such as Meeblip [7] a diy monophonic synthesizer that works with an atmega chip; Shruthi-1 [8] is another open source (GPL firmware and CC schematics) that runs on a custom mother board. The system that is closer to our approach is Satellite CCRMA [1] where a platform is designed to host PureData under a Linux architecture in the BeagleBoard. Their motivation is to support the creation of new instruments and installations guided by low cost, portability and an an existing community of users such as Pure Data community. Their project is aimed to enhance the longevity of new instruments and to become a possible standard platform for further developments. Although the hardware and software used are the

same as in our project, our approach differs on the methodology of the implementation described further.

3 Our Work

PureData was chosen as the language for the description of DSP processes for several reasons: It has been greatly adopted by programmers, sound designers, musicians and tinkerers to implement their audio processing algorithms with many of them using it on-stage (running on a laptop computer). It has an easier learning curve than text-based programming languages such as C++ and it's more suited for rapid implementation of algorithms than compiled languages. Finally, there is a large amount of patches already created by it's user community that can be leveraged by anyone for their creations.

On the hardware side, an inexpensive, portable and powerful computer was needed and having an open design would allow for future enhancements or alterations to the board. Some options such as the PandaBoard were considered but the open-ness, price range and community around the BeagleBoard made it the option of choice. The BeagleBoard project was started by a group of Texas Instruments employees in association with Digi-Key with open-source development in mind and is now supported by a very active community and available from a number of international distributors.

The decision was made to write a C++ program called XookyNabox that would parse a PD patch using an available library and one of the available Linux API's to access the input and output audio buffers. Three libraries were considered: PDAnywhere [4] [9], ZenGarden [11] and libpd [10]. PDAnywhere was discarded right away because of the fact that it uses fixed point arithmetic and the lack of active development around it. After taking a quick look at the implementation and API's for ZenGarden and libpd, ZG was chosen because the readability of the code and it's ease for embedding into C++ (and objective-C) projects. This gave way to the first version of XookyNabox.

The lower level, blocking ALSA API was used to interface with the hardware, along with a number of very simple PD patches. It ran but some synchronization issues were found where the sound output for simple oscillators was rendered at different frequencies than expected. The decision was made then to switch the ALSA API with the higher level and callback based API of PortAudio. This approach solved the synchronization issues but it was required that the program ran as a daemon so that it could be launched at startup in the BeagleBoard without user interaction. This became an issue and the the architecture of the program was changed again.

The next version of XookyNabox still used ZenGarden and was implemented as a JACK client. This time it ran as a daemon without issues but once slightly more complicated patches were used, some of the PD blocks did not run in the BeagleBoard. This, and the fact that many vanilla PD blocks were not implemented in ZenGarden, showed the necessity to use a more robust PD implementation. Enter libpd.

The final version of XookyNabox is implemented as a JACK client and instantiates libpd.

4 The approach that worked

A lightweight, minimalistic Linux distribution that was able to run on the BeagleBoard was needed and Angstrom Linux fit the bill [13]. Also, JACK, ALSA, and the JACK devel libraries are available pre-compiled in the default package repositories for Angstrom.

The Linux system was configured to start automatically at runlevel 3 (without a GUI) and JACK was set up to launch at system startup with a sample rate of 48KHz and a buffer size of 256 samples as suggested by the BeagleBoard user community:

```
jackd -d alsa -p 256 -n 4 -P hw:0 -C hw:0 -S -r 48000 &;
```

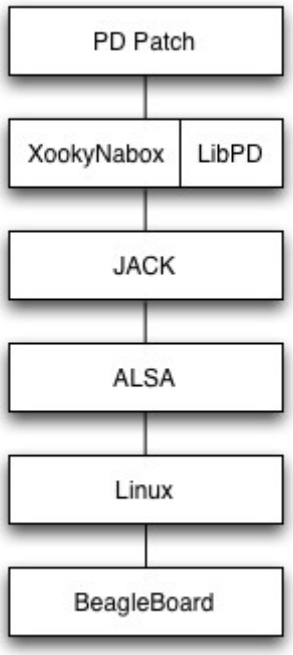


Image 1. Block diagram of the system.

Here's an overview of the implementation of the XookyNabox code. The interesting portion is the process function where mono input buffers from JACK are combined into an interleaved stereo buffer, it is then fed into libpd and the reverse process is applied to libpd's output buffer using a simple technique described in the Audio Programming Book by Boulanger and Lazzarini [2]:

```
//Interleaved io buffers:
float *output =
(float*)malloc(bufferLength*2*sizeof(float)
);
float *input =
(float*)malloc(bufferLength*2*sizeof(float)
);

// ========
// = MAIN =
// ========
int main (int argc, char *argv[]) {
    parseParameters(argc, argv);
    initLibPd();
    initJackAudioIO();

    // Keep the program alive.
    while(1){
        sleep(1);
    }

    return 0;
}

void parseParameters(int argc, char
*argv[]){
    // Show help message, retreive
parameters from argv,
    // make sure the last parameter is
actually a .pd file...
}

// =====================
// = INITIALIZE LIB PD =
// =====================
void initLibPd(){
    // Instantiate libpd, set all relevant
parameters, open the .pd
    //file and pass it to the libd object
}

// ========================
// = INITIALIZE AUDIO I/O =
// ========================
void initJackAudioIO(){
    // Create JACK client, register
callbacks, register
    //io ports and get sample rate from
JACK server.
}

// ========================
// = JACK AUDIO CALLBACK =
// ========================
int process(jack_nframes_t nframes, void
*arg){
    // Get pointers to the input and output
signals
    sample_t *in1 = (sample_t *)
jack_port_get_buffer(portI1, nframes);
    sample_t *in2 = (sample_t *)
jack_port_get_buffer(portI2, nframes);
    sample_t *out1 = (sample_t *)
jack_port_get_buffer(portO1, nframes);
    sample_t *out2 = (sample_t *)
jack_port_get_buffer(portO2, nframes);

    // Jack uses mono ports and pd expects
interleaved
    //stereo buffers.
    for(unsigned int i=0; i<nframes; i++){
        input[i*2] = *in1;
        input[(i*2)+1] = *in2;
        in1++;
        in2++;
    }
    // PD Magic!
    libpd_process_float(input, output);

    for(unsigned int i=0; i<nframes; i++){
        *out1 = output[i*2];
        *out2 = output[(i*2)+1];
        out1++;
        out2++;
    };  return 0;
}
```

It is worth mentioning that an iOS version of XookyNabox was written successfully as a CocoaTouch application that uses libpd and the CoreAudio API's.

5 The patches

The system was tested with basic signal processing and sound synthesis patches constructed exclusively using objects from pd-vanilla. The pd patches were transferred to the SD card that stores the file system for the Linux installation on the Beagle Board. Once the system is powered, the XookyNabox code fetches a file called “patch.pd” and tries to load it. We had successful experiences with some patches, but on the other hand, others did not load at all. The tested patches had no interactivity, they were just single processes that either generated or processed sound in an automatic fashion.

The list of successfully loaded processing patches includes ring, fm and am modulation, filtering and clipping. Successful synthesis patches include additive, subtractive and FM modulation, although envelopes were impossible to create. The problem with controlling the amplitudes over time was a crash of the XookyNabox program that occurred when a patch tried to set the phase of an osc~ object to any specific angle. The same problem occurred when using basic objects for creating envelopes (line, line~, vline~ and delay) so a limit to the testing emerged due to system failures.

The first debugging process was to make a list of objects that, if included, made patches crash, but, at the time of writing this paper, we just got started with checking for the different errors that came form the pdlib when loading the mentioned objects. At this point there is no clarity on the cause of the program crash.

6 Conclussions and future work

At the time of writing the paper, there are still complications in the loading of some pd patches related with specific connections and objects. This complications are below the XookyNabox code and go deeper into libpd in combination with the other JACK, ALSA and Angstrom settings. The actual possibilities of the system are limited and a thorough debugging has to be made.

Paralell to a debugging phase, the development of an interactive electronic bridge to allow communication of electronic sensors (accelerometers, potentiometers, sliders, buttons, etc) to control the patch in real time is a next step in the project.

Although there are some problems to solve, the use of pd patches in a portable, light and relatively low cost computer makes foreseeable a new generation of easily programmable customized audio hardware.

7 Acknowledgements

This paper is the consequence of a long term collaboration within different groups of enthusiasts and academics namely the AudioProgramming group in Medellín, the ACORDE research group, the LEONARDO research group, Julián Brolin Giraldo at the Un-Loquer hackerspace and the invaluable support from Juan Reyes.

References

[1] Edgar Berdahl, and Wendy Ju (2011) “Satellite CCRMA: A Musical Interaction and SoundSynthesis Platform” Proceedings of the New Interfaces for Musical Expression congreess 30 May–1 June 2011, Oslo, Norway.

Available online: https://ccrma.stanford.edu/~eberdahl/Papers/NIME2011SatelliteCCRMA.pdf

[2] Richard Boulanger and Victor Lazzarini (2010) “The Audio Programming Book” The MIT Press, 2010.

[3] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, Hans-Christoph Steiner “Embedding Pure Data with libpd” Pure Data Convention Weimar, Berlin 2011.

[4] Gunter Geiger (2003) PDa: Real Time Signal Processing and Sound Generation on Handheld Devices. Proceedings of the 2003 International Computer Music Conference (San

Francisco), International Computer Music Association, 2003.

[5] Pérez, J., & Jaramillo, J. (2011). MEII: Sistema interactivo para promover la construcción expresiva musical en niños de 4 a 8 años. Revista S&T, 9(17), 55-66. Cali: Universidad Icesi

Web Resources

[6] Juan Reyes, Sonare, https://ccrma.stanford.edu/~juanig/artes/sonare.html

[7] meeblip http://meeblip.noisepages.com

[8] shruthi-1 http://mutable-instruments.net/shruthi1

[9] Gunther Geiger, PDa, http://pd-anywhere.sourceforge.net

[10] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, Hans-Christoph Steiner, libpd, https://github.com/libpd/libpd

[11] Zen Garden https://github.com/mhroth/ZenGarden

[12] Leonrado Parra, FatChorizo, http://youtu.be/upDvvV0rGuM

[13] Linux Angstrom http://www.angstrom-distribution.org