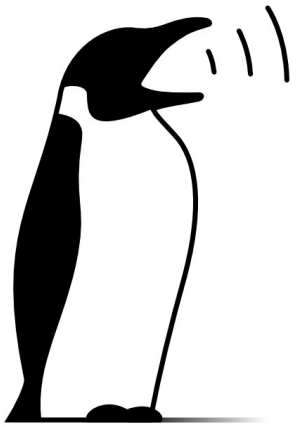
Proceedings of the

Linux Audio Conference 2015

April 9th - 12th, 2015

Johannes Gutenberg University (JGU)
Mainz, Germany



Musikinformatik & Medientechnik 53/2015
Institut für Kunstgeschichte und Musikwissenschaft, Bereich Musikinformatik

Published by
Johannes Gutenberg University (JGU) Mainz, Germany
IKM, Bereich Musikinformatik
April 2015
Editors: Frank Neumann, Albert Gräf

`http://lac.linuxaudio.org/2015`
Published as TR 53/2015 in the series "Musikinformatik & Medientechnik"
ISSN 0941-0309

Credits
Layout: Frank Neumann
Typesetting: LaTeX and pdfLaTeX
Logo Design: The Linuxaudio.org logo and its variations copyright © 2006 Thorsten Wilms

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

Printed in Germany

Partners and Sponsors

Linuxaudio.org

Johannes Gutenberg University

IKM

Institut für Kunstgeschichte
und Musikwissenschaft

Hochschule für Musik Mainz

Journalistisches Seminar

Grame - Centre national de création musicale

HKU

HKU University of the Arts, Utrecht

Bitwig - next generation music software
for Linux, Mac, Windows

Hörtech gGmbH, Oldenburg

MOD - step onto the future

Foreword

Welcome everyone to LAC 2015 in Mainz!

This is the 13th edition of LAC, the international conference with an informal, workshop-like atmosphere and a unique blend of scientific and technical papers, tutorials, sound installations and concerts centering on the free GNU/Linux operating system and open source software for audio, multimedia and musical applications. It is also the first time that the conference takes place at the Johannes Gutenberg University (JGU) in Mainz.

With about 36,500 students from some 130 nations, JGU is one of the ten largest universities in Germany. It combines almost all academic disciplines under one roof, including the Mainz University Medical Center, the School of Music, and the Mainz Academy of Arts, which is a unique feature in the German academic landscape. The JGU is also one of the oldest universities in Germany. Founded in 1477 during the era of Johannes Gutenberg and reopened after a 150-year break in 1946 by the French forces then based in Germany, Johannes Gutenberg University owes much to the man whose name it bears and his achievements. In keeping with the Gutenberg Spirit, the JGU has set itself the mission of "moving minds and crossing boundaries". Interdisciplinary discourse is therefore a hallmark of our university. This is also true for the Computer Music Research Group (Bereich Musikinformatik) of the JGU which hosts LAC 2015. Being founded in 1991, our research group has been among the first German academic institutions in this interdisciplinary field at the intersection of music, mathematics, computer science and media technology. In our media lab students are working almost exclusively with Linux, and in our research we are also devoted to contributing to the growing body of open source audio and computer music software. So our research group owes very much to Linux audio in general and the LAC in particular, and we are very proud to be this years' host of the conference.

Thanks to all the contributors who submitted papers and proposed workshops, installations and music, we will again have an interesting and varied program at the conference. There will be presentations about the latest Linux-related research and development in ambisonics, realtime aspects, plugins and modular systems, musical applications, sound synthesis, programming languages and tools, as well as numerous hands-on workshops about methods, techniques and both open-source and commercial software and hardware running on or using Linux. I am grateful that renowned digital signal processing expert Julius O. Smith III from the CCRMA at Stanford University will present a keynote lecture on emerging technologies for musical audio synthesis and effects. Another special event will be the award ceremony for Grame's 1st Faust Open-Source Software Competition, presented by Yann Orlarey during the conference. Throughout the day, there will be different sound installations and software demonstrations in the installation space and at the School of Music. In the evening there will be electroacoustic concerts in the "Roter Saal" at the School of Music, and the traditional Linux Sound Night will take place Saturday night at the "Baron", also on the university campus.

Again, this is the 13th LAC, so what could possibly go wrong? Many things in fact, if it wasn't for the tremendous help I had from friends and colleagues from the LAC team and our own university. In particular, I would like to thank Frank Neumann for organizing the paper submissions and peer reviews in the face of fairly tight deadlines, Jörn Nettingsmeier for implementing the live stream and taking care of the concerts, Robin Gareus for helping with the live stream, the website and the online program, Marc Groenewegen and his colleagues and students from the HKU who generously offered to help at the conference, and everybody who was involved in the paper and music reviews (too numerous to be mentioned here, so I have to refer you to the following pages for details). Thanks are also due to Klaus Pietschmann, head of our institute, for his encouragement and help to remove some (metaphoric) roadblocks, as well as Peter Kiefer, Moritz Reinisch and the administration of the School of Music for providing their concert hall and equipment, Karl N. Renner and his team at the department of Journalism who help us out with professional camera operators and equipment for the live stream, Nicole Labitzke and her staff at the JGU media center for providing additional audio equipment, and André Brinkmann and his team at our data center for their comprehensive support including server, audio and video equipment as well as network capacities.

Funding for the conference was generously provided by our research funding department. I would also like to thank our corporate supporters Bitwig (makers of the Bitwig Studio DAW software available on Linux, for which registered participants can win a license at the conference), HörTech (non-profit company for research and development concerning hearing, which provides us with equipment for the installation space) and MOD (makers of the universal effect pedal based on Linux, who deserve a special mention for providing funding for the Linux Sound Night), as well as the campus restaurant, bar and concert venue "Baron" for organizing and hosting the Linux Sound Night this year.

Last but not least, thanks are due to my own team members Daniel Gebhardt and Felicitas Volke, as well as our colleagues at the department of Musicology, in particular Thorsten Hindrichs and Gabriele Maurer, for their help organizing and conducting this event. Special thanks go to my wife Evelyn and my entire family for supporting me during the sometimes strenuous times leading up to the conference. This is a real team effort and without the help from all these people and facilities our small research group would never have been able to bring about a comprehensive and technically challenging event like this, so a big thank you to you all!

We hope that you will enjoy the conference and have a pleasant stay in Mainz!

Albert Gräf
Computer Music Research Group
JGU Mainz

Conference Organization Core Team

Albert Gräf
Frank Neumann
Robin Gareus

JGU | Bereich Musikinformatik

Daniel Gebhardt
Felicitas Volke

Conference Website and Design

Robin Gareus
Albert Gräf

Paper Administration and Proceedings

Frank Neumann

Concert Sound

Jörn Nettingsmeier

Stream Team

Robin Gareus
Frank Neumann
Janosch Gräf
Jörn Nettingsmeier

Camera Team

Dennis Banemann
Marisa Santos

JGU Team

Sabrina Vanessa Armbrüster	Lucas Djamba
Sebastian Gräf	Yannic Gräf
Igor Lukansky	Yana Prinsloo
Roberto Sattler	Esther Wilke
Sabrina Volke	

HKU Team

Karim Bizid	Joost van Dalen
Roald van Dillewijn	Akke Houben
Bass Jansson	Pjotr Lasschuit
Koen Pepping	Pieter Suurmond
Yuri Wilmering	Marijn Zwart

Special Thanks

Gianfranco Ceccolini	MOD
Giso Grimm	University of Oldenburg, Hearing4all
Marc Groenewegen	HKU
Harry van Haaren	OpenAV
Michèl Hammann	JGU, Media Center
Thorsten Hindrichs	JGU, Musicology Dept.
Peter Kiefer	JGU, School of Music
Katja Klein	JGU, School of Music
Friedhelm Krämer	JGU, AV-Technical Dept.
Mea Liedl	Bitwig GmbH
Daniela Mandrik	JGU, School of Music
Gabriele Maurer	JGU, Musicology Dept.
Yann Orlarey	Grame
Kristina Pfarr	JGU, Press and Public Relations
Jürgen Rein	JGU, Data Center
Moritz Reinisch	JGU, School of Music
Kathrin Voigt	JGU, Press and Public Relations
Rebecca Wessinghage	JGU, Journalism Dept.

... and to everyone else who supported us after the editorial deadline of this publication.

Review Committee

Fons Adriaensen	Huawei European Research Center, Germany
Tim Blechmann	Austria
Götz Dipper	ZKM | Institut für Musik und Akustik, Germany
John ffitch	United Kingdom
Robin Gareus	Université Paris 8, France
Albert Gräf	Johannes Gutenberg University (JGU) Mainz, Germany
Marc Groenewegen	HKU, Netherlands
Harry van Haaren	OpenAV, Ireland
Florian Hollerweger	MIT, United States
Jeremy Jongepier	linuxaudio.org, Netherlands
Victor Lazzarini	Maynooth University, Ireland
Kjetil Matheussen	NOTAM, Norway
Romain Michon	CCRMA - Stanford University, United States
Dave Phillips	United States
Peter Plessas	mur.at, Austria
Miller Puckette	UCSD, United States
Martin Rumori	Institute of Music and Performing Arts Graz, Austria
Bruno Ruviaro	Santa Clara University, United States
Funs Seelen	Sogyo, Netherlands
Julius O. Smith	CCRMA, United States
Pieter Suurmond	Hogeschool voor de Kunsten Utrecht, Netherlands
Steven Yi	Maynooth University, United States
IOhannes m zmölnig	Institute of Electronic Music and Acoustics Graz, Austria

Music Jury

Fons Adriaensen	Huawei European Research Center, Germany
Götz Dipper	ZKM | Institut für Musik und Akustik, Germany
Marc Groenewegen	HKU Utrecht, Netherlands
Bernd Härpfer	Hochschule für Musik Karlsruhe, Germany
Mark IJzerman	HKU Utrecht, Netherlands
Peter Kiefer	JGU Mainz, Germany
Victor Lazzarini	Maynooth University, Ireland
Björn Lindig	thefutarist.org, Germany
Than van Nispen tot Pannerden	HKU Utrecht, Netherlands
Yann Orlarey	Grame, France
Jochen Arne Otto	ZKM | Institut für Musik und Akustik, Germany
Hauke Schlüter	ton-und-gut.de, Germany
Pieter Suurmond	HKU Utrecht, Netherlands

Table of Contents

Toolbox for acoustic scene creation and rendering (TASCAR): Render methods and research applications

Giso GRIMM[a,b*] and **Joanna LUBERADZKA**[a] and
Tobias HERZKE[b] and **Volker HOHMANN**[a,b]
[a] Medizinische Physik and Cluster of Excellence Hearing4all
Universität Oldenburg, D-26111 Oldenburg, Germany
[b] HörTech gGmbH
Marie-Curie-Str. 2, D-26129 Oldenburg, Germany

Abstract

TASCAR is a toolbox for creation and rendering of dynamic acoustic scenes that allows direct user interaction and was developed for application in hearing aid research. This paper describes the simulation methods and shows two research applications in combination with motion tracking as an example. The first study investigated to what extent individual head movement strategies can be found in different listening tasks. The second study investigated the effect of presentation of dynamic acoustic cues on the postural stability of the listeners.

Keywords

Spatial audio, hearing research, motion tracking

1 Introduction

Hearing aids are evolving from simple amplifiers to complex processing devices. Algorithms in hearing devices, e.g., directional microphones, direction of arrival estimators, or binaural noise reduction, depend on the spatial properties of the surrounding acoustic environment [Hamacher et al., 2005]. Several studies show a large performance gap between laboratory measurements and real life experience, attributed to a changed user behavior [Smeds et al., 2006] as well as oversimplification of the test environment [Cord et al., 2004; Bentler, 2005]. To bridge this gap, a reproduction of complex listening environments in the laboratory is desired. To allow for a systematic evaluation of hearing device performance, these virtual acoustic environments need to be scalable and reproducible. There are several requirements for a virtual acoustic environment to make it suitable for hearing research. For human listening a high plausibility of the environments and a reproduction of the relevant perceptual cues is required. For machine listening and processing in multi-microphone hearing devices, a correct reproduction of relevant physical properties is needed. For an ecologically valid evaluation of hearing devices, the virtual acoustic environments need to reflect relevant every-day scenarios. Additionally, to assess limitations of hearing devices, realistic but challenging environments are required. In both cases, the reproduction need to allow for listener movements in the environment and may contain moving sources.

Existing virtual acoustic environment engines often target authentic simulations for room acoustics (e.g., EASE, ODEON), resulting in a large complexity. They typically render impulse responses for off-line analysis or auralization. Other tools, e.g., the SoundScapeRenderer [Ahrens et al., 2008], do not provide all features required here, such as room simulation and diffuse source handling. Therefore, a toolbox for acoustic scene creation and rendering (TASCAR) was developed as a Linux audio application. The aim of TASCAR is to interactively reproduce time varying complex listening environments via loudspeakers or headphones. For a seamless integration into existing measurement tools of psycho-acoustics and audiology, low-delay real-time processing of external audio streams in the time domain is applied, and interactive modification of the geometry is possible. TASCAR consists of a standalone application for the acoustic simulation, and a set of command line programs and Octave/Matlab scripts for recording from and playing to jack ports, and measuring impulse responses.

The simulation methods and implementation are described in section 2. Two research applications of TASCAR in combination with motion tracking are shown as an example. The first study (section 3.1) investigates to what extent individual head movement strategies can be found in different listening tasks. Results indicate that individual strategies exist in natural listening tasks, but task specific behavior can be found in tasks which include localization. The second study (section 3.2) investigates the effect

* g.grimm@uni-oldenburg.de

of presentation of dynamic acoustic cues on the postural stability of the listeners. Test subjects performed a stepping test while imposed with stationary or spatially dynamic sounds. Results show that in the absence of visual cues the spatial dynamics of acoustic stimuli have a significant effect on postural stability.

2 TASCAR: Methods and implementation

The implementation of TASCAR utilizes the jack audio connection kit [Davis and Hohn, 2003]. Audio content is exchanged between different components of TASCAR via jack ports. The jack time line is used as a base of all time-varying features. Audio signals are processed block-wise in the time domain. A rough signal and data flow chart of TASCAR is shown in Figure 1.

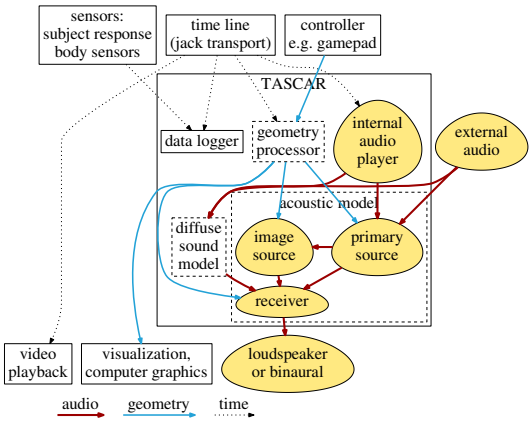


Figure 1: Schematic audio and control signal flow chart of TASCAR in a typical hearing research subjective test application.

The structure of TASCAR can be divided into three major components: Audio content is delivered by an audio player module. It provides a non-blocking method of accessing sound file portions. Audio content can also be delivered by external sources, e.g., from physical sources, audio workstations, or any other jack client. The second major block is the geometry processing of the virtual acoustic environment. The last block is the acoustic model, i.e., the combination of audio content and geometry information into an acoustic environment in a given render format.

2.1 Geometry processing

An acoustic scene in TASCAR consists of objects of several types: Sound sources, receivers, reflectors, and dedicated sound portals for coupled room simulations [Grimm et al., 2014]. All object types have trajectories defined by location in Cartesian coordinates and orientation on ZYX-Euler-coordinates. These trajectories are linearly interpolated between sparse time samples; the location is interpolated either in Cartesian coordinates, or in spherical coordinates relative to the origin. The orientation is interpolated in Euler coordinates. The geometry is updated once in each processing cycle.

Sound source objects can consist of multiple "sound vertices", either as vertices of a rigid body, i.e., following the orientation of the object, or as a chain, i.e., at a given distance on the trajectory. Each "sound vertex" is a primary source.

$\mathbf{p}_{src}$ is the primary source position, $\mathbf{p}_{rec}$ is the receiver position, and O_{rec} is the rotation matrix of the receiver. Then $\mathbf{p}_{rel} = \mathrm{O}_{rec}^{-1}(\mathbf{p}_{src} - \mathbf{p}_{rec})$ is the position of the sound source relative to the receiver, and $r = ||\mathbf{p}_{rel}||$ is the distance between source and receiver.

Reflectors can consist of polygon meshes with one or more faces. For each mesh, reflection properties can be defined. For a first order image source model, each pair of primary source and reflector face creates an image source. For higher order image source models, also the image sources of lower orders are taken into account. A schematic sketch of the image model geometry is shown in Figure 2. The image source position $\mathbf{p}_{img}$ is determined by the closest point on the (infinite) reflector plane $\mathbf{p}_{cut}$ to the source $\mathbf{p}_{src}$: $\mathbf{p}_{img} = 2\mathbf{p}_{cut} - \mathbf{p}_{src}$.

The image source position is independent of the receiver position. However, the visibility of an image source depends on the receiver position and the reflector dimension. If the intersection point of the connection from the image source to the receiver with the reflector plane $\mathbf{p}_{is}$ is within the reflector boundaries, the image source is visible, and a specular reflection is applied. If $\mathbf{p}_{is}$ is not within the reflector boundaries, the effective image source position is shifted into the direction of the closest point on the boundary to $\mathbf{p}_{is}$, and an "edge reflection" is applied. The differences between these two reflection types in terms of audio processing are described in section 2.2.2.

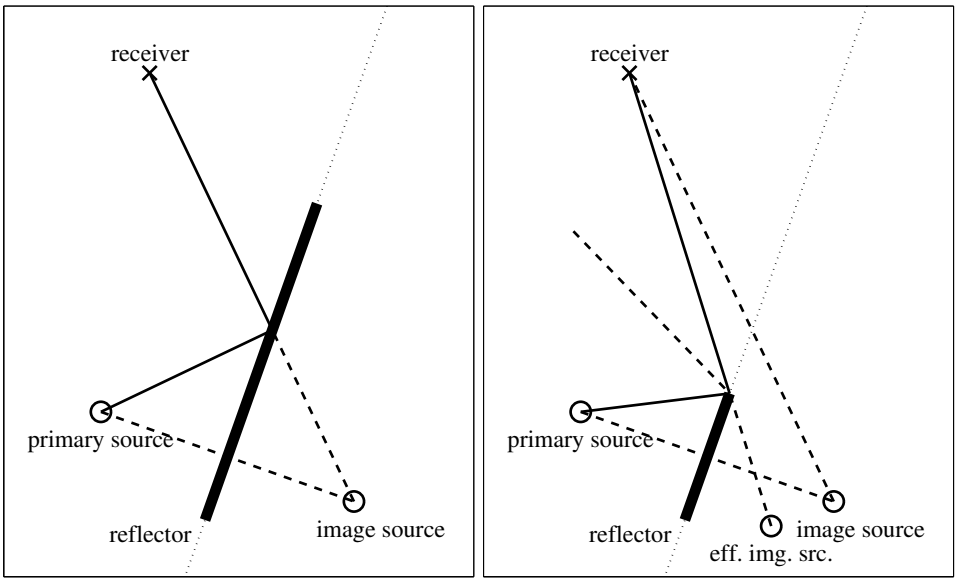


Figure 2: Schematic sketch of the image model geometry. Left panel: "specular" reflection, i.e., the image source is visible within the reflector; right panel: edge reflection.

2.2 Acoustic model

For each pair of receiver and sound source – primary or image source – an acoustic model is calculated. The acoustic model can be split into the transmission model, which depends only on the distance between source and receiver, an image source model, which depends on the reflection properties of the reflecting surfaces as well as on the "visibility" of the reflected image source, and a receiver model, which encodes the direction of the sound source relative to the receiver into the render output format.

2.2.1 Transmission model

The transmission model consists of air absorption, and a time-varying delay line for a simulation of Doppler-shift and time-varying comb-filter effects.

Point sources follow a $1/r$ sound pressure law, i.e., doubling the distance r results in half of the sound pressure. Air absorption is approximated by a simple first order low-pass filter model with the filter coefficients controlled by the distance:

$$y_k = a_1 y_{k-1} + (1 - a_1) x_k \quad (1)$$

$$a_1 = e^{-\frac{r f_s}{c\alpha}}, \quad (2)$$

where c is the speed of sound, x_k is the source signal at the sample k, and y_k is the filtered signal. The empiric constant $\alpha = 7782$ was manually adjusted to provide sensible values for distances below 50 meters. This approach is very similar to that of [Huopaniemi et al., 1997] who used a FIR filter to model the frequency response at certain distances. However, in this approach the distance parameter r can be varied dynamically.

The time varying delay line uses nearest neighbor interpolation[1].

2.2.2 Image source model

Early reflections are modeled using an image source model. In opposite to most commonly used models (e.g., [Allen and Berkley, 1979]) which calculate impulse responses for a rectangular enclosure ("shoebox model"), reflections are simulated for each reflecting polygon-shaped surface.

With finite reflectors, it is distinguished between a "specular" reflection, when the image source is visible from the receiver position within the reflector, and an "edge" reflection, when the image source would not be "visible". In both cases, the source signal is filtered with a first order low pass filter[2] determined by a reflectivity coefficient ρ, and a damping coefficient δ:

$$y_k = \delta y_{k-1} + \rho x_k \quad (3)$$

For "edge" reflections, the effective image source is shifted that it appears from the di-

[1] Other interpolation methods are planned.

[2] In later versions of TASCAR the reflection filter will be controlled by frequency-dependent absorption coefficients to avoid the sample rate dependency.

rection of a point on the reflector edge which is closest to $\mathbf{p}_{is}$. If receiver or sound source are behind the reflector, the image source is not rendered.

2.2.3 Receiver model

A receiver encodes the output of the transmission model of each sound source into the output format, based on the relative position between sound source and receiver, $\mathbf{p}_{k,rel}$. Each receiver owns one jack output port for each output channel n; the number of channels depends on the receiver type and configuration. The receiver output signal $z_k(n)$ for the output channel n and sound source k is

$$z_k(n) = w(\mathbf{p}_{k,rel}, n) y_k \tag{4}$$

with the transmission model output signal y_k. $w(\mathbf{p}_{rel}, n)$ are the driving weights for each output channel. The mixed output signal of the whole virtual acoustic environment is the sum of $z_k(n)$ across all sources k, plus the diffuse sound signals decoded for the respective receiver type (see section 2.3 for more details).

Several receiver types are implemented: Virtual omni-directional microphones simply return the output without directional processing, $w = 1$. Simple virtual cardioid microphones apply a gain g depending on the angle θ between source and receiver:

$$w = \frac{1}{2}\left(\cos(\theta) + 1\right) \tag{5}$$

For reproduction via multichannel loudspeaker arrays, receiver types with one output channel for each loudspeaker can be used. A "nearest speaker" receiver is a set of virtual loudspeakers at given positions in space (typically matched with the physical loudspeaker setup). The driving weights for each virtual loudspeaker are 1 for the least angular distance between the virtual loudspeaker and $\mathbf{p}_{rel}$, and 0 for all other channels. Other receiver types are horizontal and full-periphonic 3rd order Ambisonics [Daniel, 2001], VBAP [Pulkki, 1997], and "basic" as well as "in-phase" ambisonic panning [Neukom, 2007].

Since the geometry is updated only once in each processing block, all receiver types interpolate their driving weights so that the processed geometry is matched at the end of each block. For some receiver types, e.g., 3rd order Ambisonics, this may lead to a spatial blurring of the sources if the angular movement within one processing block is large compared to the spatial resolution of the receiver type.

2.3 Diffuse sources and reverberation

Diffuse sources, e.g., background signals, or diffuse reverberation [Wendt et al., 2014], are added in first order ambisonics (FOA) format. No distance law is applied to diffuse sound sources; instead, they have a rectangular spatial range box, i.e., they are only rendered if the receiver is within their range box, with a von-Hann ramp at the boundaries of the range box. Position and orientation of the range box can vary with time. The diffuse source signal is rotated by the difference between receiver orientation and box orientation. Each receiver type provides also a method to render FOA signals to the receiver-specific output format.

2.4 Further components

Besides the open source core of TASCAR in form of a command line application[3], a set of extension modules is commercially developed by HörTech gGmbH. These components include a graphical user interface, a time aligned data logging system for open sound control (OSC) messages, interfaces for motion trackers and electro-oculography, and specialized content controllers.

3 Example research applications

In this section, two studies related to hearing aid research which are based on TASCAR are briefly described, to illustrate possible applications.

3.1 Individualized head motion strategies

The hypothesis of this study was that task-specific head movement strategies can be measured on an individual basis. Head movements in a natural listening environment were assessed. A panel discussion with four talkers in a simulated room with early reflections was played back via an eight-channel loudspeaker array, using 3rd order Ambisonics. Head movements were recorded with the time aligned data logger using a wireless inertial measurement unit and a converter to OSC messages.

Figure 3 shows five individual head orientation trajectories. Systematic differences can be observed: Whereas one subject (green line) performs a searching motion, i.e., modulation

[3] https://github.com/gisogrimm/tascar

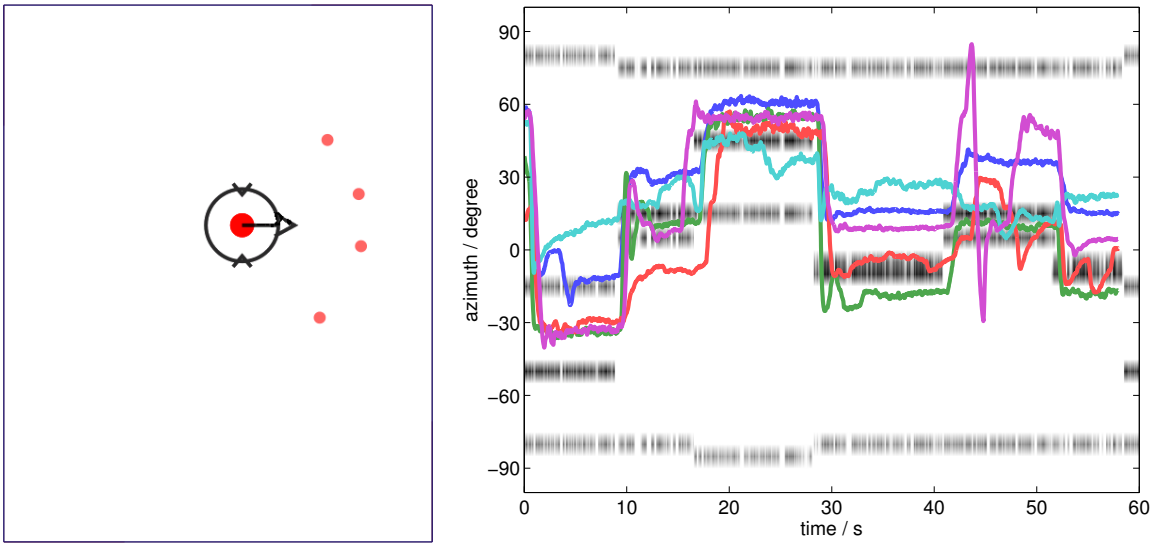


Figure 3: Intensity of a panel discussion in a room as a function of time and azimuth (shades of gray) with five individual head orientations.

around the final position, at each change of talkers, other subjects adapt slower to source position changes. One subject (blue line) shows a constant offset, possibly indicating a better-ear listening strategy.

3.2 Postural stability

Some hearing aid users feel disturbed by fast-acting automatics of hearing aids and the potentially resulting quickly changing binaural cues. To prepare the ground for further investigations of this effect, the second study assessed the effect of spatially dynamic acoustic cues on the postural stability [Büsing et al., 2015]. It is based on an experiment in which it was shown that the presence of a stationary sound can improve the postural stability in the absence of visual cues [Zhong and Yost, 2013]. A Fukuda stepping test was performed, in which the subjects were asked to step 100 steps in a fixed position. The subject drift was taken as a measure of postural stability.

In this study with 10 young participants with normal vision and hearing, the factors *vision* (open or closed eyes), *stimulus* (static or moving) and *spatial complexity* (two sources or many sources) on postural stability were analyzed. The stimuli were rendered with TASCAR; the factors *stimulus* and *spatial complexity* were realized by alternative virtual environments. The environment with low complexity was a kitchen scene with a frying pan and a clock, either rendered statically or with a sinusoidal rotate around the listener. The complex environment was a virtual amusement park, either from a carousel perspective or from a static position. The subjects were tracked with the microsoft kinect skeleton tracking library. The positions of the modeled nodes were send from the windows PC via OSC to the TASCAR data logger. The body rotation was measured as the rotation of the shoulder skeleton nodes. The results are shown in Figure 4. Vision has the largest effect on the body rotation; with open eyes the average body rotation during the test is small, independent from the stimulus and complexity condition. However, without visual cues, the spatially dynamic complex scene leads to a significantly higher body rotation than the corresponding complex static scene.

4 Conclusions

To bridge the gap between laboratory results and real-life experience in the domain of hearing research and hearing device evaluation, a tool for acoustic scene creation and rendering (TASCAR) was developed. The tool focuses on a reproduction of perceptual cues and physical properties of the sound field which are relevant for typical applications in hearing device research. Simplifications allow for computational efficiency. The implementation utilizes the jack audio connection kit, resulting in a large flexibility.

To compute the sound at a given position of the receiver, the signal coming from each source

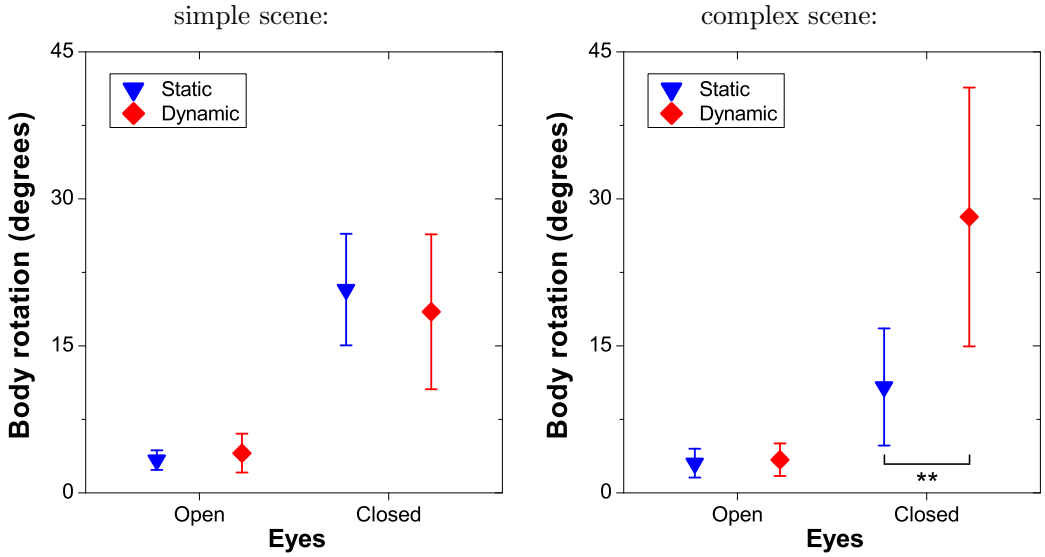


Figure 4: Body rotation in a Fukuda stepping test in a simple scene (left panel) and a complex scene (right panel). In the absence of visual cues, the dynamic cues (red diamonds) have a significant effect on the body rotation in the complex scene.

– primary or image source – is computed based on the transmission model, i.e., depending on the distance. The receiver output signal is computed depending on the type of the receiver and the angle between source and receiver. The receiver signals from all sources are added up and combined with diffuse sounds, resulting in the sound of a virtual acoustic environment in a given point.

Two studies based on the spatial audio reproduction of TASCAR demonstrate its applicability as a research tool for reproduction of spatially dynamic acoustic environments.

5 Acknowledgments

This study was funded by the research group DFG for1732. We thank Isabel Büsing and Tobias Neher from the group "Individualized Hearing Devices" for providing the data on postural stability and the good collaboration.

References

J. Ahrens, M. Geier, and S. Spors. 2008. The soundscape renderer: A unified spatial audio reproduction framework for arbitrary rendering methods. In *Audio Engineering Society Convention 124*. Audio Engineering Society.

J. B. Allen and D. A. Berkley. 1979. Image method for efficiently simulating small-room acoustics. *The Journal of the Acoustical Society of America*, 65:943.

R. A. Bentler. 2005. Effectiveness of directional microphones and noise reduction schemes in hearing aids: A systematic review of the evidence. *Journal of the American Academy of Audiology*, 16(7):473–484.

I. Büsing, G. Grimm, and T. Neher. 2015. Einfluss von räumlich-dynamischen Schallinformationen auf das Gleichgewichtsvermögen (influence of spatially dynamic acoustic cues on postural stability). In *18. Jahrestagung der Deutschen Gesellschaft für Audiologie*, Bochum, Germany.

M. T. Cord, R. K. Surr, B. E. Walden, and O. Dyrlund. 2004. Relationship between laboratory measures of directional advantage and everyday success with directional microphone hearing aids. *Journal of the American Academy of Audiology*, 15(5):353–364.

J. Daniel. 2001. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scnes sonores complexes dans un contexte multimdia*. Ph.D. thesis, Université Pierre et Marie Curie (Paris VI), Paris.

P. Davis and T. Hohn. 2003. Jack audio connection kit. In *Proc. of the Linux Audio Developer Conference. ZKM Karlsruhe*.

G. Grimm, T. Wendt, V. Hohmann, and

S. Ewert. 2014. Implementation and perceptual evaluation of a simulation method for coupled rooms in higher order ambisonics. In *Proceedings of the EAA Joint Symposium on Auralization and Ambisonics*, Berlin.

V. Hamacher, J. Chalupper, J. Eggers, E. Fischer, U. Kornagel, H. Puder, and U. Rass. 2005. Signal processing in high-end hearing aids: state of the art, challenges, and future trends. *EURASIP Journal on Applied Signal Processing*, 2005:2915–2929.

J. Huopaniemi, L. Savioja, and M. Karjalainen. 1997. Modeling of reflections and air absorption in acoustical spaces a digital filter design approach. In *Applications of Signal Processing to Audio and Acoustics, 1997. 1997 IEEE ASSP Workshop on*, pages 4–pp. IEEE.

M. Neukom. 2007. Ambisonic panning. In *Audio Engineering Society Convention 123*, 10.

V. Pulkki. 1997. Virtual sound source positioning using vector base amplitude panning. *J. Audio Eng. Soc*, 45(6):456–466.

K. Smeds, G. Keidser, J. Zakis, H. Dillon, A. Leijon, F. Grant, E. Convery, and C. Brew. 2006. Preferred overall loudness. ii: Listening through hearing aids in field and laboratory tests. *International Journal of Audiology*, 45(1):12–25.

T. Wendt, S. van der Par, and S. Ewert. 2014. Perceptual and room acoustical evaluation of a computational efficient binaural room impulse response simulation method. In *Proceedings of the EAA Joint Symposium on Auralization and Ambisonics*, Berlin.

X. Zhong and W. A. Yost. 2013. Relationship between postural stability and spatial hearing. *Journal of the American Academy of Audiology*, 24(9):782–788.

An Update on the Development of OpenMixer

Elliot KERMIT-CANFIELD and **Fernando LOPEZ-LEZCANO**
CCRMA (Center for Computer Research in Music and Acoustics),
Stanford University
{kermit|nando}@ccrma.stanford.edu

Abstract

This paper serves to update the community on the development of OpenMixer, an open source, multichannel routing platform designed for CCRMA. Serving as the replacement for a digital mixer, OpenMixer provides routing control, Ambisonics decoders, digital room correction, and level metering to the CCRMA Listening Room's 22.4 channel, full sphere speaker array.

Keywords

Multichannel Diffusion, Spatialization, Ambisonics, Open Source Audio Software, 3D Audio

1 Introduction

The Listening Room at the Center for Computer Research in Music and Acoustics (CCRMA) is a versatile studio space serving the CCRMA community. Featuring a 22.4 channel spherical speaker array and a 32 speaker wave field synthesis system, the Listening Room is an important facility for research, music composition, and 3D sound diffusion among other uses. As conventional mixers are limited in configurability, channel count, and usability, we have implemented a software routing system—called OpenMixer—to control the primary speaker system.

Although conceived in 2006, the implementation of OpenMixer began in 2009. Since then, the capabilities of the Listening Room developed—there are more speakers, faster computers, and more users. To meet the demands, OpenMixer has been under continuous development. This paper serves as an update to its current status and extends Lopez-Lezcano and Sadural's 2010 LAC paper [1].

1.1 Original Implementation

OpenMixer is our hardware and software solution to the problem of routing, mixing and controlling the diffusion of multiple multichannel sources over a large number of speakers, and was tailored to the Listening Room studio at CCRMA. The control hardware is comprised of standard PC computer components housed in a fan-less case and off-the-shelf RME audio sound cards, plus external Solid Stage Logic AlphaLink MADI AX and Aphex 141 boxes for A/D and D/A conversion. The control software runs on Gnu/Linux and is a collection of free, open source software programs coordinated from a master control program written in the SuperCollider programming language. Inputs to the system include 3 ADAT connections to and from the Listening Room Linux workstation (a regular CCRMA Linux computer that can be used for music or sound work and has direct access to all speakers), 2 additional ADAT I/Os for external digital sources, 16 line level analog inputs, 8 microphone preamplifier inputs, 8 analog inputs for a media player, and 4 ethernet jacks connected to a separate network interface (which can also provide Internet access) that enable computers which have NetJack easy access to all speakers and features of the system.

At the time the previous paper was written, the Listening Room was equipped with only 16 speakers in a 4 + 8 + 4 configuration, which could either be addressed independently or be fed the signals of an Ambisonics decoder (second order horizontal, first order vertical) integrated into the system. Two Behringer USB control surfaces (BCF200 and BCR200) provided a simple and easy-to-use user interface.

1.2 Changes

The following sections outline most of the changes to OpenMixer since 2009, both in hardware and in the underlying control software and functionality.

2 Hardware Upgrades

2.1 Speakers

The number of speakers in the Listening Room has grown over time. The last major upgrade happened in March 2011, when the number of main speakers was changed from 16 to 22 (23 if we count an additional "center channel" which is only used for more accurate 5.1 or 7.1 playback). The spatial configuration was changed from 4 + 8 + 4 to 1 + 6 + 8 + 6 + 1, which enabled us to cover the sphere more uniformly and thus decode full periphonic third order Ambisonics with high accuracy (see figs. 1 to 3). In addition, four subwoofers were added, which required adding an additional eight channel ADAT D/A converter to the equipment rack.

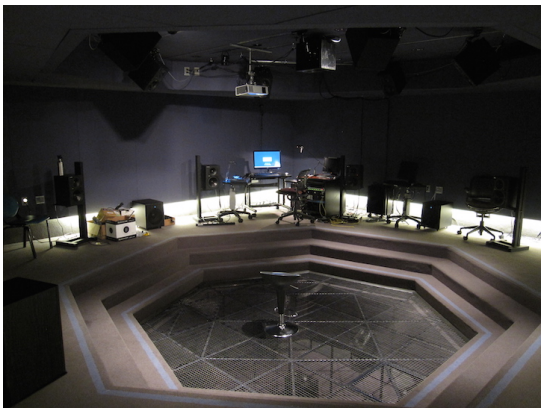


Figure 1: The Listening Room

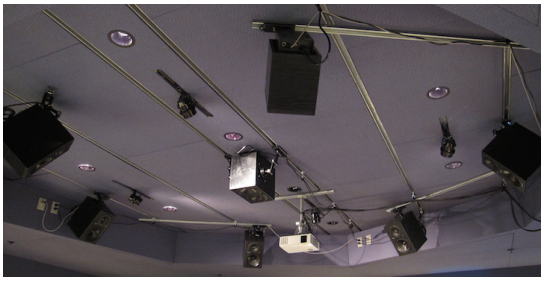


Figure 2: 7 elevated speakers

2.2 Computer and Peripherals

One of CCRMA's fan-less Linux machines is at the heart of the Listening Room—OpenMixer system. As we needed more processing power to support some of the planned expansions of the OpenMixer system, we migrated the hardware from an old quad core Q9550 CPU running

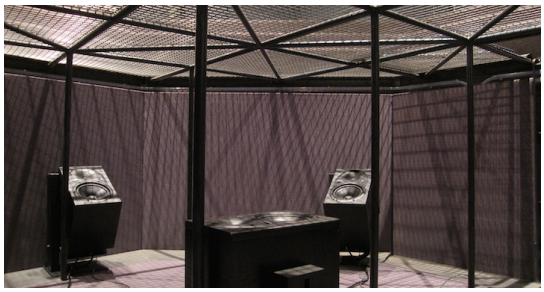


Figure 3: 7 speakers below the grid floor

at 2.83GHz, an EP45-DS3R Gigabyte motherboard, and 8G of RAM to a newer quad core i7-3770K CPU running at 3.50GHz and using a DZ77GA-70K Intel motherboard with 32G of RAM. At the same time we moved from the old Fedora 14 platform to Fedora 20 (new real time (RT) patched kernel and newer versions of all packages). An updated hardware signal-flow chart can be seen in fig. 4.

The upgrade was anything but boring. Although much faster, the upgraded computer refused to work with the PCI audio cards used by the old hardware. We were using two RME cards in tandem—one RME Hammerfall DSP MADI PCI that interfaced with the main SSL A/D D/A box and an RME 9652 PCI card for additional ADAT ports. The upgraded motherboard included two legacy PCI slots for that purpose, but we were never able to have both cards working at the same time. The second card would fail to get interrupts delivered to it. This was difficult to debug and we spent countless hours switching cards around. We eventually had to move to newer PCI express cards with equivalent functionality (an RME HDSPe MADI and an RME RayDAT PCIe).

Even then, the technique of aggregating both cards into a composite ALSA device using *.asoundrc* did not work anymore. The newer kernel, ALSA drivers, and hardware could not get the card periods aligned, even though the sound cards are synced using word clock. JACK would immediately start generating xruns when started on the composite device—each card by itself would work fine. To get around this problem we tried running JACK on only one of the cards and connecting the second card to the system. We did this through either JACK's `audioadapter` internal client or `alsa_in` and `alsa_out` clients. In both cases, the DSP load was too high even for the upgraded hardware

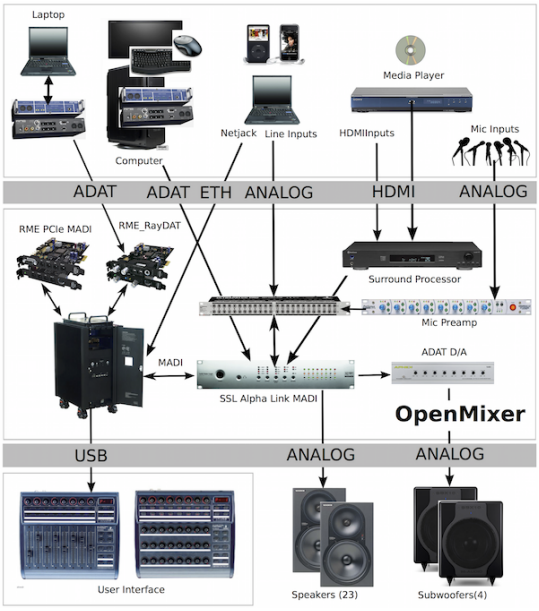


Figure 4: System hardware diagram

platform.

We finally used Fons Adriaensen's Zita-ajbridge to add the second card as a separate JACK client [2]. As in the previous solutions, this performs unnecessary sampling rate conversion of the second card inputs and outputs, but it was the only way to get the system to work reliably and the additional load was acceptable (between 9 and 10% of a core each for 64 input and output channels). Another twist to the story is that we could not get the MADI card to run as the master card when using the RayDAT as a secondary audio card. We had to run the RayDAT as the master, which increased the channel count to sample-rate conversion as well as the load. All these sound driver peculiarities need serious debugging, but we just did not have the time to do that.

For cases like this one it would be nice to have an option in Zita-ajbridge to not do sampling rate conversion at all, as the cards run in sync through word clock.

3 Software Upgrades

3.1 SuperCollider and Supernova

The majority of OpenMixer runs inside SuperCollider, a programming environment specifically designed for audio applications [3]. In addition to being well supported by a large developer/user community, SuperCollider is extendable through custom UGens and plugins and handles multichannel audio, MIDI, and OSC in a native and intuitive fashion.[1] SuperCollider itself is really two programs—an interpreted language (`sclang`) controlling a separate synthesis server (`scsynth`) running as a separate unix process.

In this new version of OpenMixer, we switched the synthesis server to `SuperNova`, an alternative server written by Tim Blechmann which supports automatic parallelization of the DSP load across multiple cores [4]. This simplified the software considerably, as before this, we were performing load balancing across cores by instantiating several instances of `scsynth` and controlling which server was chosen for particular tasks by hand.

When the OpenMixer computer boots, SuperCollider runs automatically as a boot service and starts all other ancillary software. The main task of SuperCollider is routing audio from multiple sources (Linux workstation, ADAT, analog, BluRay, mic preamps, and networks sources) to the speaker array, as well at to the Linux workstation and networks sources for recording. This represents a complex many-to-many routing relationship through the use of software buses (see fig. 5). In addition to routing straight from a channel to some number of speakers (each with independent gain control), OpenMixer also supports decoding of up to third order Ambisonics audio streams. Another important task that SuperCollider performs is as a task manager for subsidiary programs such as JACK and Jconvolver. If any of OpenMixer's auxiliary processes dies, the system restarts them automatically.[2]

3.2 System software

Together with the hardware changes, we upgraded the operating system to Fedora 20 and a new RT patched kernel [5]. To get the best real-time performance from the new system, we needed to perform a few new tweaks to its configuration.

We found that the Intel i7-3770K processor together with the new kernel (currently 3.14.x with the latest RT patch) used the new `intel_pstate` frequency scaling driver instead of the conventional

[1] We could only imagine the horrors of programming OpenMixer in C or C++!

[2] Except in very bad situations...

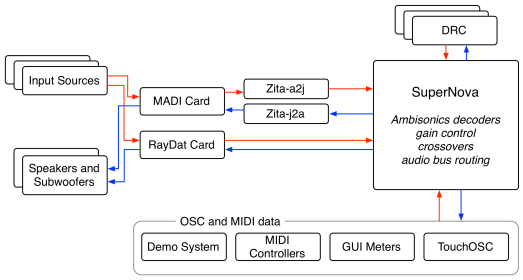


Figure 5: Simplified OpenMixer software routing diagram

governor-based scaling cpupower software. The `intel_pstate` driver can be controlled through variables in the `/sys` filesystem (`/sys/devices/system/cpu/intel_pstate`). The following incantation was used to tell the driver to try to run all cores at the maximum speed all the time:

```
echo 100 > /sys/devices/system/cpu/
  intel_pstate/min_perf_pct
```

We also disabled the `thermald` daemon which we normally run in all our Linux workstations, as it also tweaks the speed of the processor cores if the thermal load exceeds preset limits. We know what the load on the processor is, so there is no risk of thermal overloads. In both cases, we want the core speed to be pegged to the maximum available. It looks like the scheduler is not entirely aware of the current speed of each core and can migrate a computationally heavy task from a core running at full speed to one that is idling (and running at a very low clock frequency). The `intel_pstate` driver will of course adjust the speed of the slow core, but not fast enough to avoid an xrun in some cases.

We also had to disable hyper-threading (HT) in the BIOS. This disables the additional "fake" cores and brings down the total number from eight to four (but those four are real cores—we will use a cheaper Intel i5 processors which lacks hyper-threading for future deployments). Without these changes, we experienced occasional xruns in JACK. HT creates two logical cores out of each physical core by storing extra state information that enables both logical cores to appear to be independent processors. The potential improvement in performance is only realized by highly threaded applications and tops out at about 20% in ideal conditions. This comes at the cost of increased thread scheduling latency. We don't know how much latency HT actually adds, but it seems that it is enough in our current mix of tasks and threads to negate any advantage in raw processing power HT might offer [6–8].

The start-up scripts that boot OpenMixer have also changed. The boot activation of the software now happens through a static `systemd` service. The service executes a script that tweaks root level access configuration settings and then starts the main start-up script as the "openmixer" user. This in turn starts a private X server for future GUI extensions and transfers control to `sclang` (the SuperCollider language). The script also re-starts `sclang` if for some reason `sclang` dies.

3.3 User Interface

The user interface in the previous version of OpenMixer was limited to two Behringer control surfaces to access most common settings. This was a deliberate choice to keep the interface simple and mode-less, where every button, fader, and knob had only one clearly labeled function. Most of the interface has not changed but some new functions have been added (see fig. 6). For example, the master level control knobs also double as level meters for each bank of 8 channels (you can select the function of the knobs), there is a "DRC on/off" button that can be used to disable the correction filters on each speaker, we now have a monitor option (with optional pre-fader level) that routes any input back to the Linux workstation, we implemented a speaker test function to quickly make sure all speakers are operational and also a software reset that reinitializes OpenMixer.

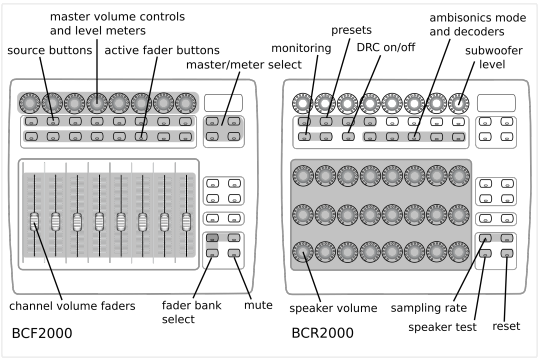


Figure 6: User interface

We have also employed SuperCollider's QT

GUI objects to build a graphic user interface so that a monitor connected to the control computer can display relevant information. For now, we have only implemented input and output VU level meters, but this change opens the door to future upgrades that should make OpenMixer even simpler to use, specially if we use a video screen with touch control.

OpenMixer is internally controlled through OSC messages. We have been therefore experimenting with applications like TouchOSC on tablets or smart phones to be able to control OpenMixer remotely through simple apps installed in user's equipment [9].

3.4 Subwoofer Processing

The addition of subwoofers to the Listening Room required us to implement proper crossovers in software. We used LinkwitzRiley 4^{th} order crossovers implemented as SuperCollider UGens that are part of the instruments that do software mixing and routing within OpenMixer.

3.5 Ambisonics Decoding with ADT and ATK

First through third order Ambisonics decoders are naively supported by OpenMixer. In the previous version of OpenMixer, Ambisonics decoders were provided by running `ambdec_cli` as a subprocess of the master SuperCollider program, with decoder matrices tuned to our speaker arrangement kindly supplied by Fons Adriansen. A stereo UHJ decoder was also provided by running Jconvolver as another subprocess with the supplied UHJ preset configuration.

In the new version of OpenMixer, we have switched to using decoders calculated by the Ambisonics Decoder Toolbox (ADT), written by Aaron Heller and Eric Benjamin [10–12]. ADT consists of Matlab/GNU Octave scripts that calculate Ambisonics decoders though a variety of techniques. In critical listening tests, the decoders have proved to perform very closely to the previous hand-tuned ones. In addition to generating Ambdec configuration files, ADT writes the decoders as Faust programs which can then be compiled to create native SuperCollider UGens [13]. For stereo UHJ decoding, we have switched to the Ambisonic Toolkit (ATK) UGens, written primarily by Joseph Anderson [14]. We have been able to fold Ambisonics decoding entirely into SuperCollider which minimizes the complexity of the code, the number of external programs used, JACK graph complexity, and context switches.

3.6 Digital Room Correction

From the start, OpenMixer calibrates all speakers for uniform sound pressure level and delay at the center of the room. While the new location of the speakers in the studio covers the sphere with better resolution, they are not really suspended in a free field condition. In particular, the speakers located under the grid floor (below the listener) have a different tonal quality due to the physical construction of the "pit" in which they are located. Furthermore, not all speakers are the same exact model (although they all share the same high frequency drivers). We have Mackie HR824s at ear level and smaller Mackie HR624s for the elevated and lowered speakers. The result of this led to incorrect and sometimes confusing rendering, especially when decoding full sphere Ambisonics (manifested through a tendency of sounds to be "pulled" towards the ceiling).

In the new version of OpenMixer, we have implemented digital room and speaker correction using Denis Sbragion's DRC software package, as described by Joern Nettingsmeier [15]. By recording the impulse responses of each speaker at the listening position, inverse FIR filters can be calculated to even out small differences in speaker impulse response due to the speakers themselves and their location in the room.[3] This, coupled with the very dry acoustics of the studio itself makes for an accurate and even reproduction of sound over the whole sphere around the listening position. The only trade off is an increase of approximately 10 milliseconds in the latency of the system. We have added an on/off switch to be able to bypass the DRC generated correction filters and access the speakers directly if necessary.

Jconvolver, written by Fons Adriaensen, is used as an external subprocess of the main SuperCollider program and provides an efficient, real-time convolution engine that performs the filtering [17].

3.7 Adding HDMI inputs

We recently had the need to allow users to connect HDMI audio based equipment (specifically game consoles—for research, of course—

[3] We used 20 second logarithmic chirps, recorded and analyzed with Fons Adriaensen's Aliki [16].

and also laptops). This adds another way to connect equipment to the system and can be used to play back 5.1 or 7.1 content (easier in those cases than using NetJack or analog interfaces). We bought an Outlaw Model 975 surround processor, a low cost unit that provides both HDMI switching and decoding of the most common compressed surround formats (Dolby Digital, DTS, Dolby TrueHD, and DTS-HD Master Audio, etc). The analog outputs of the unit were connected to the "player" analog interface, and the existing Oppo BluRay player was connected to one of the Outlaw's HDMI inputs.

The only gripe is that the unit cannot (yet) be controlled from OpenMixer itself. It would be nice to be able to control everything from the OpenMixer control surfaces so that the user does not need to know which HDMI input is which and how the surround processor is controlled. We could use an infrared remote control, or inquire if the RS232 interface included in the unit can be used for control in addition to firmware upgrades.

3.8 Demo Mode

OpenMixer makes the Listening Room easily configurable, however, it requires some effort between walking into the studio and hearing sound. In the past, demonstrations of the Listening Room's sound system—which are quite frequent—typically included hunting for saved Ardour sessions to recall and some amount of guesswork to find projects spread across CCRMA's filesystem. To simplify this process, we implemented a demo mode that highlights the capabilities of the system in an easy to use way. We used a Korg nanoKONTROL2 as a control surface that is patched directly to OpenMixer. This controller has eight channel strips (three buttons, a knob and a fader) and transport control buttons. We have pre-rendered an assortment of multi-channel works for the Listening Room's speaker configuration that can be easily triggered through the nanoKONTROL2.

With minimal effort, someone can listen to a curated set of mixes of great variety. We have included selections that show off different types of 3D diffusion (e.g., quad, 5.1, octophonic, third order periphonic Ambisonics, etc.) as well as mixing style (e.g., "concert hall-esque," "fully immersive," "academic electroacoustic," etc.). One such piece, a recording mixed through Ambisonics, is rendered both in full third order Ambisonics and stereo UHJ in a time-synced way so the "immersive-ness" of the decoders can be audified and evaluated.

4 Future Work

Although OpenMixer was originally written for the CCRMA Listening Room, we are working to parameterize the scripts so that it is more portable.[4] Once appropriately parameterized, we envision OpenMixer as a useful tool for other people controlling multichannel speaker arrays.[5] We are in the process of implementing OpenMixer in our small concert hall, the Stage, that is equipped with a 16.8 channel 3D speaker system. OpenMixer will make it possible to easily move pieces from the Listening Room environment to the concert hall. This is especially true in the Ambisonics domain, as new decoder technologies implemented in ADT make it relatively easy to design effective 3D decoders for dome speaker configurations such as the one we have on the Stage. CCRMA concerts frequently feature large speaker arrays (up to 24.8) which are controlled with similar systems. Making OpenMixer a general solution would simplify these large scale productions.

Much work remains to be done in the software itself, in particular, to use the new X-based GUI interface and to control the whole system through OSC from tablets and smart phones.

In the Listening Room itself, we have plans to add more subwoofers (for a total of 8) to increase both the fidelity and localization resolution of low frequencies.

As the technology gets faster and the number of speakers grow, OpenMixer will most likely remain a work-in-progress.

We originally intended to use JackTrip as an additional input source for remote performances, but the feature did not see much demand and it was impossible to make JackTrip work (without changes to the source code) as a reliable subprocess to the SuperCollider code. We plan to use the currently unimplemented JackTrip inputs to house Fons Adriansen's Zitanjbridge (network-JACK) JACK clients, thus

[4] Naturally, this is a challenging task as speaker configuration, sound cards, input sources, etc. are different for each system.

[5] More information about OpenMixer can be found at https://ccrma.stanford.edu/software/openmixer.

providing a way to connect to the Listening Room from remote locations [18].

5 Conclusions

OpenMixer is certainly not the only solution for controlling 3D speaker arrays at CCRMA, but it has fulfilled its purpose to make the Listening Room configurable for many different uses. The latest inclusion of DRC-based impulse response calibration has significantly improved the quality of sound diffusion in the Listening Room, and is being considered for other listening spaces at CCRMA.

6 Acknowledgments

The success of OpenMixer is largely due to the support of the CCRMA community. Many thanks to Chris Chafe and Jonathan Berger for embracing the Listening Room in their classes and projects. OpenMixer is built on free, open source software and would never have existed without all who contributed to those projects.

References

[1] F. Lopez-Lezcano and J. Sadural, "Openmixer: a routing mixer for multichannel studios," *Linux Audio Conference*, 2010.

[2] F. Adriaensen, "Zita-ajbridge, alsa-jack and jack-alsa resampler." `http://kokkinizita.linuxaudio.org/linuxaudio/`.

[3] J. McCartney, "Supercollider: real-time audio synthesis and algorithmic composition." `http://supercollider.sourceforge.net/`.

[4] T. Blechmann, "Supernova: a multiprocessor aware real-time audio synthesis engine for supercollider," Master's thesis, TU Wien, 2011.

[5] Various, "Real-time linux patches." `https://rt.wiki.kernel.org/index.php/Main_Page/`.

[6] A. Valles, "Performance insights to intel hyper-threading technology." `https://software.intel.com/en-us/articles/performance-insights-to-intel-hyper-threading-technology`.

[7] T. Leng, R. Ali, J. Hsieh, V. Mashayekhi, and R. Rooholamini, "An empirical study of hyper-threading in high performance computing clusters," *Linux Clusters Institute Conference*, 2002.

[8] Calomel.org, "Network tuning and performance: a simple guide to enhancing network speeds." `https://calomel.org/network_performance.html`.

[9] Hexler.net, "Touchosc: Modular osc and midi control surface for iphone/ipod touch/ipad." `http://hexler.net/software/touchosc`.

[10] A. Heller and E. Benjamin, "Ambisonic decoder toolkit." `http://www.ai.sri.com/ajh/ambisonics/`.

[11] A. Heller, E. Benjamin, and R. Lee, "A toolkit for the design of ambisonic decoders," *Proceedings of the Linux Audio Conference 2012*, 2012.

[12] A. Heller and E. Benjamin, "The ambisonics decoder toolbox: Extensions for partial-coverage loudspeaker arrays," *Proceedings of the Linux Audio Conference 2014*, 2014.

[13] Y. Orlarey, "Faust: the quick path from ideas to efficient dsp." `http://faust.grame.fr/`.

[14] J. Anderson, "The ambisonic toolkit: tools for soundfield-kernel composition." `http://www.ambisonictoolkit.net/`.

[15] D. Sbragion, "Drc: Digital room correction." `http://drc-fir.sourceforge.net/`.

[16] F. Adriaensen, "Aliki, impulse response measurement tool." `http://kokkinizita.linuxaudio.org/linuxaudio/`.

[17] F. Adriaensen, "Jconvolver, a convolution engine." `http://kokkinizita.linuxaudio.org/linuxaudio/`.

[18] F. Adriaensen, "Zita-njbridge, network-jack and jack-alsa resampler." `http://kokkinizita.linuxaudio.org/linuxaudio/`.

Low Delay Audio Streaming for a 3D Audio Recording System

Marzena Malczewska, Tomasz Żernicki and **Piotr Szczechowiak**
Zylia Sp. z o. o.
Umultowska 85
Poznań, Poland,
{marzena.malczewska, tomasz.zernicki, piotr.szczechowiak}@zylia.pl

Abstract

This paper presents a prototype of a 3D audio recording system named AudioSense which uses Wireless Acoustic Sensors to capture spatial audio. The sound is recorded in real-time by microphones embedded in each sensor device and streamed to a Processing Unit for 3D audio compression. One of the key problems in systems which stream audio data is end-to-end latency. This paper is focused on analyzing a set of chosen parameters of the Opus codec in order to obtain the minimal delay. Experimental results on the prototype system have shown that it is possible to achieve below 10ms of end-to-end audio delay with the use of the Opus codec.

Keywords

audio streaming, opus codec, low latency streaming, wireless sensor network, spatial audio

1 Introduction

The area of 3D audio and object based audio is currently a hot research topic as evidenced by a large number of research papers and new emerging standards such as MPEG-H 3D Audio. The majority of the research efforts in this area are concentrated on the audio processing and rendering side. The problem of 3D audio recording is getting less attention in the literature.

Channel-based method of spatial sound production assume that the number of microphones used during the recording is directly proportional to the number of loud speakers used during sound rendering. This can lead to large numbers of microphones in case of multiple audio channels recordings. In addition, proper setting and tuning of microphones in the field can be a tedious task which requires many resources in terms of time and manpower. Current distributed recording systems use wired microphones, which makes it difficult to deploy and use the system in certain environments.

To solve the limitations of current spatial audio recording systems, the AudioSense system is being developed. It introduces object-based sound representation and wireless audio streaming. The system can be described as a Wireless Acoustic Sensor Network (WASN) [Bertrand, 2011; Akyildiz et al., 2007]. In this system individual sound sources (objects) are extracted from a sound mixture using sound source separation techniques (e.g Independent Component Analysis [Comon, 1994]). Object based audio representation gives high flexibility in terms of sound rendering (easy rendering for headphones, stereo, 5.1, 7.1 systems) and enables interactive manipulation of individual sound objects during playback.

The proposed AudioSense system has many possible applications and can be used for both indoor and outdoor audio recordings. The system can be used in teleconference applications to add the possibility to identify speakers by speech direction. It can be also used for wildlife monitoring and live TV broadcasts from the field. The AudioSense technology has applications in surveillance systems where it can identify and track objects based on sound processing. The system can be also applied in the entertainment industry in case of movies, games and virtual reality applications that require immersive 3D sound.

Realisation of the AudioSense system is a challenging task that requires overcoming major challenges in such areas as audio streaming, audio coding, sound sources separation and audio synchronisation. This paper is focusing on designing a low delay audio streaming mechanism that meets the strict requirements of the AudioSense system.

The AudioSense system consists of battery operated devices with low processing capabilities. Therefore audio recording, coding and streaming has to be performed with energy efficiency in mind. Live applications of the system require also low latency audio streaming that is reliable and allows simultaneous streaming of data from multiple devices over the wireless

medium.

In order to minimise the end-to-end delay it is necessary to optimise the audio recording process, use a low latency audio codec and streaming method. This paper shows the design of the system that tries to achieve this goal. It presents the technologies and design choices made to implement a low delay audio streaming system on off-the shelf embedded devices. The results achieved during the performance evaluation of the system show that it is possible to achieve a low end-to-end delay of 10ms for wireless audio streaming within the AudioSense system.

The rest of the paper is organised as follows. Section 2 presents the related work in the area of spatial audio recording systems. Section 3 describes the architecture of the AudioSense system together with hardware and software components implemented to build the first prototype of the system. The results achieved during the performance evaluation phase are presented in Section 4. Finally Section 5 concludes the paper and describes the lessons learnt from implementing a low delay audio streaming system.

2 Related work

Spatial audio recording systems are gaining on popularity with the introduction of 3D audio systems and technologies that can reproduce truly immersive sound. Majority of existing systems for spatial audio recording use wired microphones [Gallo et al., 2007] to capture the virtual sound stage. This fact limits drastically the mobility of such systems and increases significantly their deployment time.

In the literature one can find also wireless systems for distributed audio recording like [Taysi et al., 2010] and [Pham et al., 2014]. The main problem with such systems is that the wireless sensor network devices are equipped with low quality microphones, amplifiers and A/D converters due to the low cost and high energy efficiency of the system. Sounds recorded with such systems have insufficient quality for many audio applications.

One of the systems that tries to overcome the problems of low cost WASNs is WiLMA [Schörkhuber et al., 2014]. The system introduces a wireless microphone array that offers high quality audio recording and processing. WiLMA enables connection of up to 4 professional microphones to each sensor module and provides wireless synchronisation for audio recordings. Similar approach to system design is presented also in [Mennill et al., 2012] where a distributed microphone array system is used for environmental monitoring and animals recording. This system is also based on battery operated sensors and uses GPS for accurate synchronisation of the recordings.

One of the limitations of spatial audio recording systems presented in [Mennill et al., 2012] is that the system does not offer continuous real-time wireless audio streaming. All the recordings are stored on local flash memory of the sensor devices. The AudioSense system takes the next step in spatial audio recording systems by providing low delay wireless streaming capabilities and audio representation in the object-based format. These features open up the door for a whole new range of audio applications that can be realised with the use of the AudioSense system.

3 System overview

The proposed architecture of the AudioSense system is presented in Figure 1. From the functional side the system can be divided into two parts. The first part consists of Acoustic Sensors that form a wireless network responsible for audio recording. The second part includes an embedded device which performs 3D audio processing. Each device in the wireless sensor network has one or several microphones, A/D converter and performs initial audio compression. Compressed audio signals are transmitted through the Gateway to the Processing Unit. The Gateway serves as an interface between the wireless and the wired part of the system. After reception of the audio signals the Processing Unit performs aggregation of the individual streams.

Each of the streams is decoded and synchronised with each other. In the next step the process of sound sources separation is performed to generate individual audio objects [Salaün et al., 2014], [Ozerov et al., 2012]. These objects are then used in the process of 3D audio coding (e.g. MPEG-H 3D Audio [ISO/IEC WD 23008-3, 2014]. Finally the encoded audio is transmitted over the Internet to the client side where the sound rendering is performed.

3.1 Hardware components

From the hardware perspective the Acoustic Sensor prototype consists of a Beaglebone Black [Coley, 2014] with an Audio Cape board [BeagleBoard, 2012] and a dedicated microphone

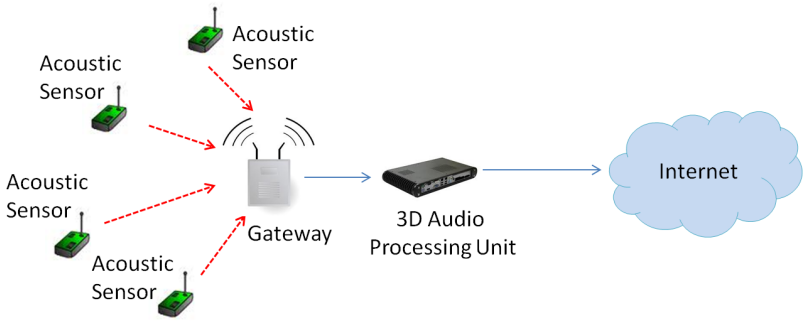


Figure 1: Architecture of the AudioSense system.

board designed in-house. Beaglebone Black is a low-power single board computer based on 1GHz ARM Cortex A8 CPU. The board provides only one 12-bit analog-to-digital converter which is not sufficient for any professional audio applications. Hence the usage of an Audio Cape (6 channels of up to 96 kHz sampling at 24 bit) is required to improve the quality of the recorded sound. For the microphone board a pair of Monacor MCE-4000 electret omnidirectional microphones is selected due to high signal to noise ratio and very good sensitivity. Each microphone is connected to a low noise operational amplifier - MCP6021. The amplified acoustic signal is passed on to the Audio Cape where analog to digital conversion is executed. Next, the digital data in one of available formats (e. g. S16LE) is sent to the Beaglebone Black. Each acoustic sensor is equipped also with a wireless interface compatible with the IEEE 802.11 a,b,g,n standards. The first version of the prototypical Acoustic Sensor is presented in Fig. 2.

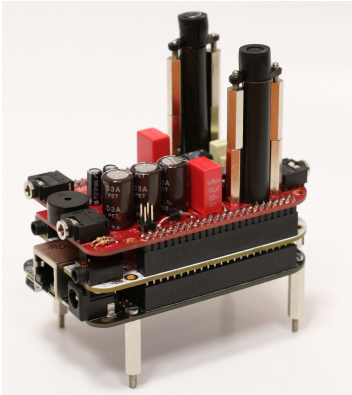


Figure 2: First version of the Acoustic Sensor prototype.

3.2 Software components

The prototype of Audio Sensor is running Debian Jessie Linux with kernel version 3.8.13. In order to implement audio processing on the device, the Gstreamer framework is utilised. Sound capturing, coding and streaming are all implemented within a single Gstreamer v1.4 pipeline. Figure 3 illustrates Gstreamer pipelines implemented on both the Acoustic Sensor and the 3D Audio Processing Unit.

The pipeline on the Acoustic Sensor side is responsible for capturing audio samples using the ALSA plugin and encoding them with the Opus [Valin et al., 2013] encoder. Next, every packet is encapsulated in the RTP packet and sent via UDP to the Processing Unit.

On the Processing Unit side each received packet is processed by the depayloader and Opus decoder. This processing is performed for each stream independently. Next step is the separation plugin, which gets n streams and, after performing the process of sound sources separation, generates m audio objects. The processed data is passed on to the multiqueue and then interleaved to form a multichannel wave file. For this purpose a new Gstreamer module is implemented called *WavNChEnc*. The stream generated by the module is then passed to the MPEG-H 3D Audio codec which generates a single .mp4 file.

To measure time of encoding, the measurements points were set right before and after Opus encoder. Respectively, on the Processing Unit the points were set before and after the Opus decoder. Streaming time was measured with the measurement points set just before UDP transceiver in Acoustic Sensor and right after RTP depayloader in the Processing Unit.

One of the key aspects in networked audio

systems is audio synchronisation. In order to provide synchronisation of the recorded audio streams, a separate synchronisation module is implemented. The synchronisation method applied is a hybrid approach based on reference broadcast that uses the ideas presented in [Elson et al., 2002] and [Budnikov et al., 2004]. Using this hybrid synchronisation method it is possible to achieve a synchronisation error of around 200μs.

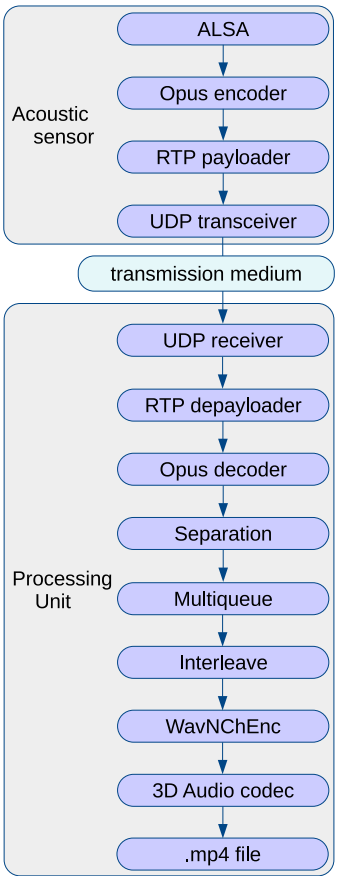


Figure 3: Gstreamer pipelines implemented on the Acoustic Sensor and the 3D Audio Processing Unit side.

4 Performance evaluation

4.1 Test scenarios

The performance of the designed 3D Audio recording system was evaluated using several test scenarios. The main goal of the experiments was to adjust and optimise hardware and software components of the system to achieve minimal streaming delay for different audio bitrates and network setups. Audio streaming latency was measured in an end-to-end manner which includes:

- Audio encoding time - time needed to encode one whole buffer of data by the Opus encoder. Such measurements were executed for different codec parameters which have the highest impact on the encoding time (e.g. bitrate, complexity, frame-size).
- Audio decoding time - measurement of decoding time for the same set of parameters as in the case of audio encoding.
- Audio transmission time - packets latency measurement when streaming wirelessly over Wi-Fi (IEEE 802.11n).

For the Audio streaming tests the Opus parameters were constant while the network setup was different in each experiment. The system was tested with several Acoustic Sensors in the network. In each of the cases the distance between the Acoustic Sensors and the 3D Audio Processing Unit was different to test the system in different working conditions. In addition to latency tests the experiments included also CPU usage measurements for Opus encoding and decoding.

Impact of selected parameters of the Opus codec on the quality of sound was not the subject of our tests. Several tests were performed in the past and are well described in [Hoene et al., 2011].

4.2 Results

This subsection presents experimental results achieved by measuring the end-to-end audio streaming delay in the AudioSense system. The first set of tests was performed to measure the encoding delay of the Opus codec in order to find the optimal codec parameters that enable minimal processing latency. Three parameters of the codec were identified as possible candidates for processing delay optimisation:

- *Complexity* - is defined as a trade-off between processing complexity and quality/bitrate. This parameter is selected using an integer from 0 to 10, where 0 is the lowest complexity and 10 is the highest. In the experiments fixed values of 0, 3, 6 and 10 were used to check what is the influence of complexity on the processing delay.
- *Frame size* - Opus has fixed frame durations of 2.5, 5, 10, 20, 40, and 60 ms. Increase in the frame duration has influence

on coding efficiency improvement but the gain becomes small for frame sizes above 20 ms.

- *Bitrate* - Opus supports different bitrates in the range between 6 kbit/s and 510 kbit/s. Higher bitrate results in higher quality audio and lower latency in packets delivery at the cost of increased bandwidth.

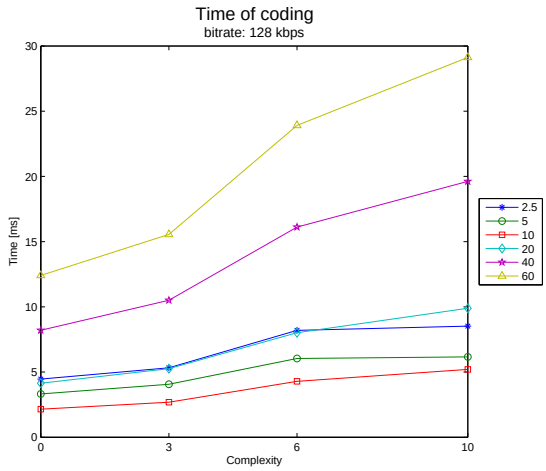


Figure 4: Opus encoding time for different values of *complexity* and *frame-size*.

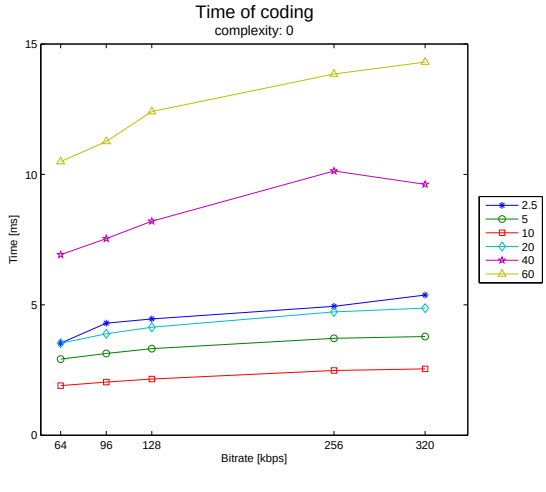


Figure 5: Opus encoding time for different values of *bitrate* and *frame-size*.

Figure 4 shows Opus encoding delay for different complexity and frame size settings (different colors on the figure correspond to different frame sizes). For all measurements the bitrate remained constant at the level of 128 kbit/s. It is clearly visible that the average encoding latency increases with higher complexity values. The difference is especially visible for higher frame durations of 40 and 60 ms where the latency is 3 to 6 times higher than in the case of smaller frame sizes. Therefore the best values of frame size in case of the AudioSense system are below 20ms where the encoding delay is smaller than 10ms. Surprisingly the frame size of 2.5ms is providing a similar encoding delay as in the case when the frame duration is set to 20ms. In terms of complexity the optimal value is 3 with frame size of 10 ms.

The influence of different audio bitrates on the Opus encoding delay is illustrated in Figure 5. The complexity parameter in all cases is fixed at 0. The experiments are performed for five audio bitrates (64, 96, 128, 256 and 320 kbit/s) and the same frame size values as in the previous test. The graph shows that significant increase in processing time is visible for larger values of frame size (40 and 60 ms). It is evident that the bitrate change has much smaller effect on encoding time than the change of the complexity parameter. For frame size values below 20ms the change of bitrate has very small effect on the encoding delay - only 1ms increase when changing the bitrate from 64kbit/s to 320 kbit/s. This experiment shows once again that the frame size of 10ms provides the optimal setting in terms of Opus encoding latency.

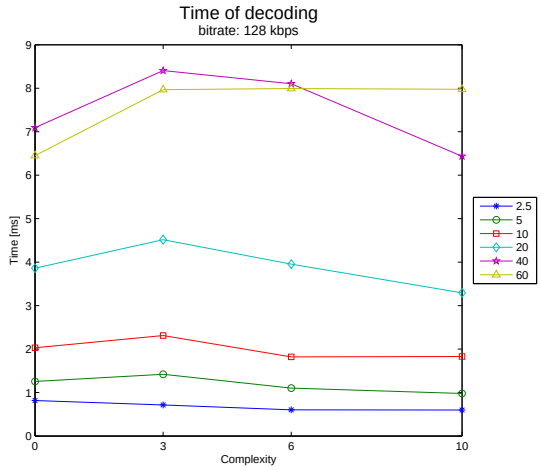


Figure 6: Opus decoding time for different values of *complexity* and *frame-size*.

Opus decoding latency is tested in a similar manner as in the case of encoding. Figure 6 shows the impact of the complexity parameter on the decoding time for different frame duration values. The audio bitrate is fixed at 128 kbit/s. As can be seen in Figure 6 the com-

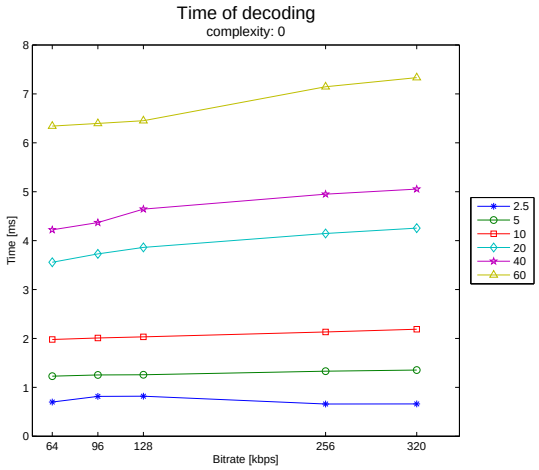


Figure 7: Opus decoding time for different values of *bitrate* and *frame-size*.

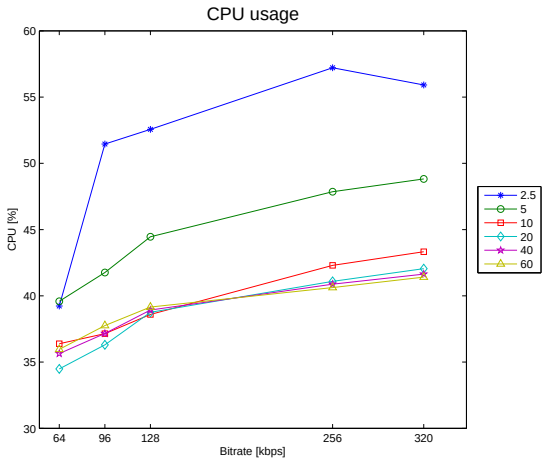


Figure 8: CPU usage for different values of *bitrate* and *frame-size*.

plexity parameter has a small impact on the overall audio decoding time. For smaller values of frame size (10ms and below) the decoding time remains the same for different complexity values. The difference is visible only in case of larger frame size values (20, 40 and 60ms) where the decoding delay can increase or decrease by around 1ms with the change of codec complexity.

Figure 7 presents Opus decoding times with respect to different audio bitrates. In all cases the complexity parameter is set to 0. It is clearly visible that audio bitrate has very small impact on the decoding time. The main parameter that has the biggest influence on decoding time is frame duration. From the point of view of Opus decoding, the best performance in terms of execution time can be achieved for the smallest possible values of frame size: 2.5 and 5ms.

The AudioSense system consists of battery operated sensor devices therefore the power consumption during codec operation is an important parameter that can limit the total operation time of the system. Figure 8 presents the CPU usage on the Acoustic Sensor while performing coding and decoding using the Opus codec. The measurements are performed for six different audio bitrates and six frame durations. In all cases the CPU operation remains between 35% and 57%. Highest CPU usage is recorded for the smallest frame size value (2.5ms). For all frame sizes between 10ms and 60ms the CPU usage stays on the same levels. The influence of audio bitrate on CPU processing is not significant as changing the bitrate from 64 kbit/s to 320 kbit/s increases the CPU usage by 7% on average. The complexity parameter of the Opus codec has a stronger influence on the CPU processing than audio bitrate change. Changing the complexity from 0 to 3 increases the CPU usage by 10% on average. Switching from 3 to 6 adds another 10% of CPU processing. It is recommended to set the complexity on 3 or lower in order to keep the CPU usage below the level of 50%. From the energy efficiency point of view the optimal frame size is equal to 10ms.

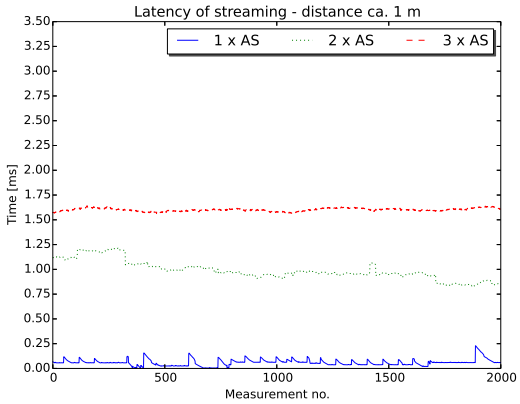


Figure 9: Streaming delay measurements with a distance of 1m between devices.

Audio encoding and decoding adds a significant delay in the audio processing pipeline of the AudioSense system. The third factor that adds an additional delay is audio streaming over the wireless channel. In order to mea-

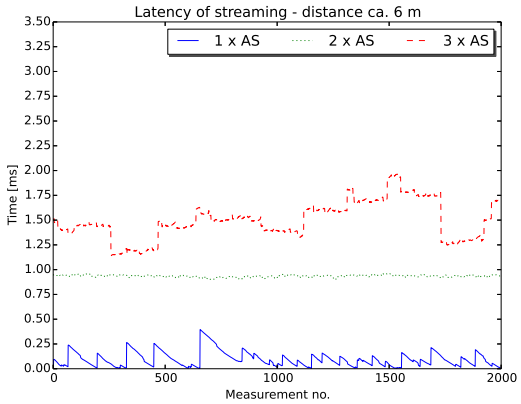


Figure 10: Streaming delay measurements with a distance of 6m between devices.

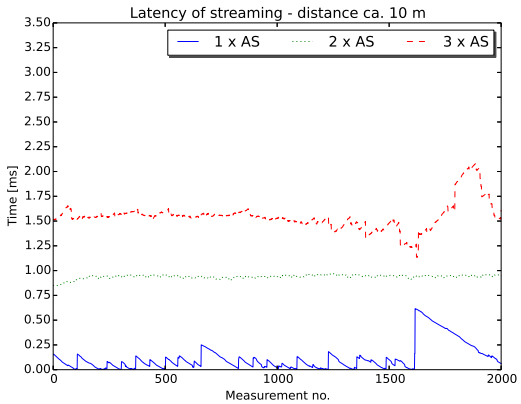


Figure 11: Streaming delay measurements with a distance of 10m between devices.

sure streaming latency over Wi-Fi several experiments are performed using the prototype AudioSense system.

All the tests are made with the same parameters of the Opus codec: sampling rate 48kHz, bitrate 128 kbit/s, complexity 0, frame size 10ms. Figure 9 presents the first set of experiments where the network consists of one, two or three Acoustic Sensors (AS). In all cases the distance between the 3D Audio Processing Unit and Acoustic Sensors is equal to 1m. The measurements are taken over 2000 audio samples. The streaming is performed under Line of Sight (LOS) conditions using the 802.11 n mode. It is clearly visible in Figure 9 that the streaming delay is the lowest (around 70μs) when there is only one Acoustic Sensor in the network. Addition of the second sensor that sends simultaneously audio data to the processing unit increases significantly the overall packets delivery time to around 1ms on average. The network with three Acoustic Sensors increases the delay even further to around 1.6ms.

Figures 10 and 11 show the results of the same experiment as above but under different network conditions. The distance between the devices is increased to 6m and 10m respectively. The streaming is performed under Non Line of Sight (NLOS) conditions. The results demonstrate that the increase in distance between devices has small influence on the average audio streaming delay. The average delay for packet reception remains at the same levels in all three sets of tests. The main difference can be noticed in the jitter levels which are much higher when using the system in NLOS conditions.

5 Conclusions

This paper presents the architecture of the AudioSense system which is designed to record and process spatial audio. All the hardware and software components of the prototype implementation of the system are described in detail. The result of the work is a wireless acoustic sensor network capable of distributed sound recording in an object-based audio format.

The paper focuses also on the development of a low delay audio streaming technique which meets the strict requirements of the AudioSense system. For this purpose the Gstreamer framework is utilised together with the Opus codec. The optimal working parameters for the codec are selected through experimental evaluation and the end-to-end delay is measured for different setups of the wireless network. The results demonstrate that it is possible to achieve an average delay below 10ms for coding, transmission and decoding of the audio signal in a wireless system of several Acoustic Sensors.

For the future work it would be interesting to test the system on a larger scale with parallel transmissions from many Acoustic Sensors. The capacity of the system and transmission delay can be further optimised by utilising wireless streaming in the 802.11 ac standard. For the needs of sound sources separation it will be beneficial to apply a hardware based synchronisation method which would limit the synchronisation error to several μs.

6 Acknowledgements

This work was supported by National Centre for Research and Development (NCBiR), Poland,

"Leader" programme.

References

Adapteva, 2014. *Parallella Reference Manual 13.11.25.*

Ian F Akyildiz, Tommaso Melodia, and Kaushik R Chowdury. 2007. Wireless multimedia sensor networks: A survey. *Wireless Communications, IEEE*, 14(6):32–39.

BeagleBoard, 2012. *BeagleBone Audio Cape Revision A1 System Reference Manual*, October.

Alexander Bertrand. 2011. Applications and trends in wireless acoustic sensor networks: a signal processing perspective. In *Communications and Vehicular Technology in the Benelux (SCVT), 2011 18th IEEE Symposium on*, pages 1–6. IEEE.

D. Budnikov, I. Chikalov, S. Egorychev, I. Kozintsev, and R. Lienhart. 2004. Providing common i/o clock for wireless distributed platforms. In *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*, volume 3, pages iii–909–12 vol.3, May.

Gerald Coley, 2014. *BeagleBone Black System Reference Manual - Revision B*, January.

Pierre Comon. 1994. Independent component analysis, a new concept? *Signal processing*, 36(3):287–314.

Jeremy Elson, Lewis Girod, and Deborah Estrin. 2002. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation*, OSDI '02, pages 147–163, New York, NY, USA. ACM.

Emmanuel Gallo, Nicolas Tsingos, and Guillaume Lemaitre. 2007. 3d-audio matting, post-editing and re-rendering from field recordings. *EURASIP: Journal on Advances in Signal Processing.* Special issue on Spatial Sound and Virtual Acoustics.

Christian Hoene, Jean-Marc Valin, Koen Vos, and Jan Skoglund. 2011. Summary of opus listening test results. Technical report, IETF.

ISO/IEC WD 23008-3. 2014. Information technology - High efficiency coding and media delivery in heterogeneous environments - Part 3: 3D audio. Technical report, International Organization for Standardization / International Electrotechnical Commission, International Telecommunications Union – Telecommunication, January.

Daniel J Mennill, Matthew Battiston, David R Wilson, Jennifer R Foote, and Stephanie M Doucet. 2012. Field test of an affordable, portable, wireless microphone array for spatial monitoring of animal ecology and behaviour. *Methods in Ecology and Evolution*, 3(4):704–712.

Alexey Ozerov, Emmanuel Vincent, and Frédéric Bimbot. 2012. A General Flexible Framework for the Handling of Prior Information in Audio Source Separation. *IEEE Transactions on Audio, Speech and Language Processing*, 20(4):1118 – 1133, May. 16.

Congduc Pham, Philippe Cousin, and Arnaud Carer. 2014. Real-time on-demand multi-hop audio streaming with low-resource sensor motes. In *Local Computer Networks Workshops (LCN Workshops), 2014 IEEE 39th Conference on*, pages 539–543. IEEE.

Yann Salaün, Emmanuel Vincent, Nancy Bertin, Nathan Souviraà-Labastie, Xabier Jaureguiberry, Dung T. Tran, and Frédéric Bimbot. 2014. The Flexible Audio Source Separation Toolbox Version 2.0. May.

Christian Schörkhuber, Markus Zaunschirm, and IO-hannes Zmölnig. 2014. Wilma-wireless largescale microphone array. In *Linux Audio Conference*, volume 2014.

Z. Cihan Taysi, M. Amac Guvensan, and Tommaso Melodia. 2010. Tinyears: Spying on house appliances with audio sensor nodes. In *Proceedings of the second ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '10, pages 31–36, New York, NY, USA. Association for Computing Machinery.

Jean-Marc Valin, Gregory Maxwell, Timothy B Terriberry, and Koen Vos. 2013. High-quality, low-delay music coding in the opus codec. In *Audio Engineering Society (AES) Convention no. 135.* Audio Engineering Society.

Postrum II: A Posture Aid for Trumpet Players

Mat DALGLEISH
Department of Music and Music Technology, Faculty of Arts, University of Wolverhampton
The Performance Hub, Walsall Campus,
Walsall, UK, WS13BD
m.dalgleish2@wlv.ac.uk

Chris PAYNE
Department of Music and Music Technology, Faculty of Arts, University of Wolverhampton
The Performance Hub, Walsall Campus,
Walsall, UK, WS13BD
c.c.payne@wlv.ac.uk

Steve SPENCER
Department of Music and Music Technology, Faculty of Arts, University of Wolverhampton
The Performance Hub, Walsall Campus,
Walsall, UK, WS13BD
steve.spencer@wlv.ac.uk

Abstract

While brass pedagogy has traditionally focussed on sound output, the importance of bodily posture to both short-term performance and longer-term injury prevention is now widely recognized. Postrum II is a Linux-based system for trumpet players that performs real-time analysis of posture and uses a combination of visual and haptic feedback to try to correct any posture issues that are found. Issues underpinning the design of the system are discussed, the transition from Mac OS X to Ubuntu detailed, and some possibilities for future work suggested.

Keywords

Brass pedagogy, posture, calm technology, Pd-extended, Ubuntu.

1 Introduction

Wright [1] notes that while brass instrument playing involves the combination and coordination of multiple facets, tone production is relatively poorly understood and often considered dauntingly complex. Perhaps as a result of these difficulties, trumpet pedagogy has traditionally focused on sound production with little consideration of the body. As a result, the role of the body of the player tended either to go unrecognized, or the ability of the body to intuitively find appropriate techniques has been assumed [2].

Over time this lack of concerted engagement with the body has proved problematic; there has been increasing recognition that bodily posture is important, not only in terms of sound production and performance, but also longer-term injury prevention. For example, Kelly [3], Whitener [2] and Dornbusch [4] note that poor posture can degrade respiratory function, stamina, embouchure and tone. Others have suggested that poor posture in trumpet players can lead to back, shoulder and neck pain [5] [6], and muscular weakness [4]. Indeed, similar problems have been identified in other musicians, from pianists to string players, and a number of clinics established specifically to deal with musicians' injuries [7].

As a result, the literature of brass pedagogy has sought to identify the typical posture problems found in trumpet players and arrived at a consensus regarding ideal alignment of the body. Based on this literature, we introduce a posture aid that analyzes the posture of a standing or seated player in real-time and, if necessary, applies corrective haptic and visual feedback. In particular, we describe how this system builds on previous work that utilized haptic feedback only, and our transition from Mac OS X to Ubuntu Linux.

2 Optimal Posture

Drawing on the literature of brass pedagogy described above, three distinct types of posture issue can be identified (see Figure 1).

Figure 1: Optimal posture (far left) compared to three common types of posture issue.

Within the figure above, the first image (from left to right) demonstrates optimal posture

allowing the lungs and ribcage freedom to operate. The second image shows the head rotated forward, thereby restricting the flow of air out from the neck and back of throat. The third image shows both the head rotated forward and the sternum collapsed, inhibiting respiration. Finally, the fourth shows excessive sideways twisting of the body [8].

Of course, not all trumpet players stand; some adopt a seated playing position. However, the posture issues experienced are closely related and pragmatically almost identical to those that occur while standing. Indeed, Jacobs [1] has referred to a position that he termed "standing while sitting" that would provide the best position for the lungs to function and support the tone of the instrument.

3 System Design

3.1 Previous System and its Discontents

Our previous prototype consisted of a Microsoft Kinect 3-D camera and Synapse application for posture analysis, a mapping layer created in the MaxMSP visual programming environment, and two 2x2 vibrotactile arrays built around Arduino microcontrollers [8]. The combination of Kinect camera, Synapse application and MaxMSP required that the system run on Mac OS X (specifically, Mac OS 10.9 on a Macbook Pro). However, the expense of these underlying technologies limited the potential to build multiple instances of the system. This is desirable as it makes it possible to test the system on several users simultaneously, or in different locations at the same time.

3.2 Transition to Ubuntu

The new system supplements simplified haptic feedback with visual feedback via an ambient projection. It also adds the ability to capture time-stamped audio for subsequent analysis. Thus, it becomes possible to compare posture to sound output over time. However, perhaps most significant change, at least in terms of implementation, is the move from Mac OS to Ubuntu Linux.

Some elements of this transition are reasonably straightforward. For instance, the Open Source and cross-platform Pure Data-extended (Pd-extended) provides a near like-for-like replacement for the MaxMSP. Both are derived from the Max family of languages developed at IRCAM in Paris and offer similar functionality, to the point that they even share many object names [9].

3.3 System Overview

The Postrum II system consists of three layers:

- input (camera and audio)
- analysis and mapping
- output (visual and haptic feedback)

3.3.1 Input

A generic USB webcam is initialized as a video4linux (V4L) device. The [pix_video] object then is used to grab live video from the V4L device at a resolution of 640x480 pixels and 30 frames per second (FPS). CD-quality mono audio is also collected from a microphone via the [adc~] object, timestamped, and recorded to disk using the [writesf~] object. As explained in the future work section of this paper, this data is not yet fully utilized, but has rich potential.

3.3.2 Analysis and Mapping

The analysis and mapping layer sits between the input and output layers and is primarily built in Pd-extended. Each frame of video is passed into a real-time analysis sub-patch that implements a combination of computer vision processes. Firstly, background subtraction is used to capture a player-specific reference image of optimal posture and isolate the player from her surroundings. Next, the [pix_movement] object creates a black and white bitmap of the difference between an average of the most recent frame and the reference image (an average of several frames can be used if smoothing is required). The difference between the two highlights areas of the body that have departed from the optimal posture, and from there the type of posture issue is identified (based on the types detailed above), as well as the degree of deviation from the ideal.

This posture data is converted and then dispatched as Open Sound Control (OSC) messages to the feedback layer. The figure below (Figure 2) outlines the data flow through the Postrum II system, showing its three distinct layers.

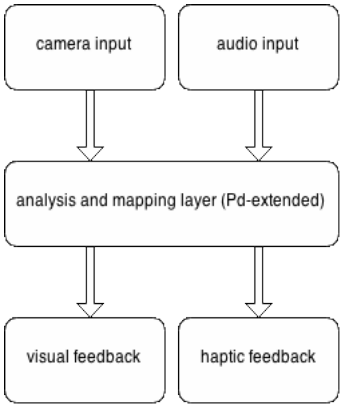


Figure 2: Flow diagram of the main elements of the Postrum II system

3.3.3 Output

The concept of calm technology was developed by Weiser and Brown [11]. It concerns the shifting of interaction to the periphery of attention in an attempt to reduce information overload. Informed by this concept, in this research, an ambient visual display is used to indicate departures from the optimal posture. When departures from the ideal are minor or short-lived, the aim is to inform but not significantly distract the player from other musical tasks by occupying only the periphery of their attention (see Figure 3). In particular, a key design principle is that the visual feedback should be sympathetic to sight reading; that it should not require the eyes to be taken off a musical score in order for it to be processed by the player.

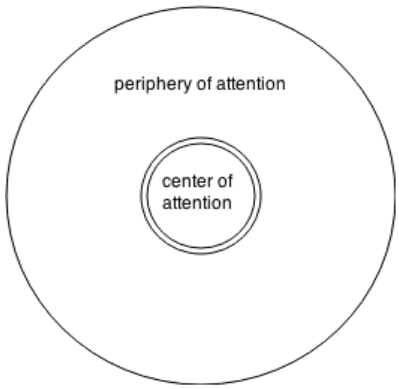


Figure 3: Principle of the visual feedback.

The position and color of the projection around the periphery indicate the type of posture issue and its severity. The ambient display is split into five areas (see Figure 3). Colors range from green (optimal posture) through to red (severe posture issue). When the entire display is green the player's posture is optimal. The area directly above the head (A) turning red indicates that the head has rotated forward; the areas below this (B) turning red denotes that the sternum has collapsed; the left or right sides of the display (C) turning red indicates that the body is twisted to one side or the other. If optimal posture is not resumed within a few seconds, a second state is entered. In this state, visual feedback remains but is progressively supplemented by haptic feedback to more concertedly attract the attention of the user. The amplitude of the haptic feedback is proportional to the extent and duration of the postural deviation.

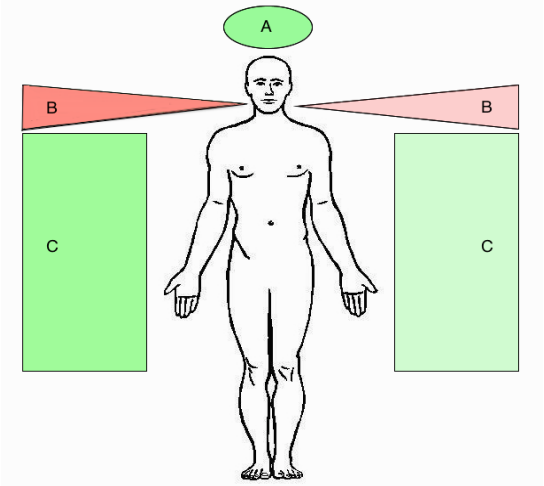


Figure 4: Visual feedback in Postrum II.

The visual feedback component is implemented in the open source Processing programming language. It is displayed on a large television screen or, preferably, projection onto the wall of the practice room. Compared to the first Postrum system, the haptic feedback component is greatly simplified. It consists of a single 2.5cm vibration motor mounted on the torso (just above the waist) using a soft and elastic band. It is controlled by an Arduino microcontroller via a simple H-bridge. The Arduino in turn connects to the host computer via a wired USB connection. The amplitude of the vibration motor is able to be continuously varied by means of pulse-width modulation.

3.4 Related Work

Our earlier system aside [8], to the best of our knowledge, previous posture aids aimed specifically at brass players have been passive mechanical devices only. The *Shulman System for Brass* [13] rests on the sternum and holds the trumpet in an optimal position in front of the player. The *ERGObrass* [14] supports the weight

of the instrument on a rod attached to the floor or to the player's belt, thereby freeing up the arms, shoulders, and upper body.

Work more similar to ours exists in other musical domains. For instance, the *Music Jacket* [15] is a wearable, real-time system for novice violin players that uses a camera to track the position of the bowing arm and haptic feedback to "nudge" the player into adopting good posture habits. The *Integrated Vibrotactiles* interface [16] is also aimed at violinists. In a similar manner to the Music Jacket, it provides the player with real-time haptic feedback that aims to foster good movement and posture within a 3-D space.

4 Discussion and Future Work

The Postrum II solves some of the design issues identified in our first prototype. In particular, it attempts to reduce the cognitive demands placed on the player when only minor postural deviations are identified, and thus impinge less on the ability to carry out attention-heavy musical tasks.

At present, Postrum II uses a standard Ubuntu distribution. It would be interesting to compare its performance and ease of use to that of a specialized audiovisual variant. It would also be a relatively small step from our current system to one that is able to run on the Raspberry Pi single board computer (SBC). This would not only further reduce the cost of the system, but enable the technology to "disappear" into, or be hidden inside, the fabric of the practice room.

A particularly interesting possibility for future work lies in comparing posture data and the (recorded) sound output of the instrument to look for correlation between the two. Another possibility concerns studying the effects of tiredness and fatigue on the posture of expert trumpet players. While these players may usually initially present with good posture, the effect of long (and potentially tiring) practice sessions has so far been little explored.

References

[1] Wright, B. Y. and Eric, D. 2009. *Physics, Pedagogy, and the Art of Trumpet Players.*

[2] S. Whitener. 1990. *A Complete Guide to Brass: Instruments and Pedagogy*. Schirmer Books, New York.

[3] K. Kelly. 1983. The Dynamics of Breathing: a Medical/Musical Analysis with Arnold Jacobs and David Cugell MD. In *The Instrumentalist*, volume December 1983: 6–12.

[4] J. Dornbusch. *Dental Aspects in Playing Trumpet, Trombone, Horn, Tuba and Others*. http://www.jazzer.de/bmd/homehalt.htm

[5] F. G. Campos. 2005. *Trumpet Technique*, pages 30–34, Oxford University Press, New York.

[6] S. Varney. *Alexander Technique: A Balm for Back Pain?* http://www.npr.org/2011/03/28/134861319/alexander-technique-a-balm-for-back-pain

[7] J. van der Linden, Y. Rogers, C. Taylor and M. Dalgleish. 2011. Technology inspired design for pervasive healthcare. In *Workshop on User-Centered Design of Pervasive Healthcare Applications,* Pervasive Healthcare conference 2011, Dublin, Ireland.

[8] M. Dalgleish and S. Spencer. 2014. Postrum: Developing Good Posture in Trumpet Players Through Directional Haptic Feedback. In Proceedings of the *9th Conference on Interdisciplinary Musicology (CIM14)*, 4-6 December 2014, Berlin, Germany.

[9] neupert. Pd for Max users. http://puredata.info/docs/tutorials/PdForMaxUsers

[10] M. Peach. OSC Objects for Pure Data. http://puredata.info/Members/martinrp/OSCobjects/

[11] M. Weiser and J. S. Brown. 1996. The Coming Age of Calm Technology.

[12] no author. *The Shulman System breaks down at Goodrich concert.* http://williamsrecord.com/2004/03/02/the-shulman-system-breaks-down-at-goodrich-concert/

[13] A. Heinonen. *A Relaxing Revolution*. http://www.ergobrass.com/trpt/eng/ITG%20Journal.pdf

[14] J. van der Linden, E. Schoonderwaldt, J. Bird, and R. Johnson. 2011. *MusicJacket - combining motion capture and vibrotactile feedback to teach violin bowing*. In *IEEE Transactions on Instrumentation and Measurement*, volume 60(1), pages 104-113.

[15] T. Grosshauser and T. Hermann. 2009. *Augmented Haptics - An Interactive Feedback System for Musicians.* In *Haptic and Audio Interaction Design 4th International Conference*, Proceedings. *Lecture Notes in Computer Science, 5763*. M. E. Altinsoy and S. Merchel (Eds), pages 100–08. Springer, Berlin.

Faust audio DSP language in the Web

Stephane LETZ and **Sarah DENOUX** and **Yann ORLAREY** and **Dominique FOBER**
GRAME
11, cours de Verdun (GENSOUL)
69002 LYON,
FRANCE,
{letz, sdenoux, orlarey, fober}@grame.fr

Abstract

With the advent of both HTML5 and the Web Audio API (a high-level JavaScript API for audio processing and synthesis) interesting audio applications can now be developed for the Web. The Web Audio API offers a set of fast predefined audio nodes as well as customizable *ScriptProcessor* node, allowing developers to add their own javascript audio processing code.

Several projects are developing abstractions on top of the Web Audio API to extend its capabilities, and offer more complex unit generators, DSP effects libraries, or adapted syntax. This paper brings another approach based on the use of the FAUST audio DSP language to develop additional nodes to be used as basic audio DSP blocks in the Web Audio graph.

Different methods have been explored: going from an experimental version that embeds the complete FAUST native compilation chain (based on *libfaust + LLVM*) in the browser, to more portable solutions using JavaScript or the much more efficient *asm.js* version. Embedding the FAUST compiler itself as a pure JavaScript library (produced using *Emscripten*) will also be described.

The advantages and issues of each approach will be discussed and some benchmarks will be given.

Keywords

Web Audio API, FAUST, Domain Specific Language, DSP, real-time

1 Introduction

This paper demonstrates how an efficient compilation chain from FAUST to the Web Audio API can be done, allowing the available FAUST programs and libraries to be immediately used in a browser.

Section 2 describes the Web Audio API and how it can be extended and targeted by Domain Specific Languages. Section 3 describes the FAUST language and its mechanisms to be deployed on a large variety of platforms. Section 4 exposes the compilation chain and the multiple target languages available from a unique DSP specification. In the context of the Web Audio API, section 5 presents the different approaches experimented to deploy Faust DSP programs on the Web. Section 6 exposes some use cases, and finally some results and benchmarks are given in section 6.1.

2 Programming audio in the Web

2.1 Web Audio API

The Web Audio API [12] specification describes a high-level JavaScript API for processing and synthesizing audio in Web applications. The conception model is based on an audio routing graph, where a number of AudioNode objects are connected together to program the global audio computation.

The actual processing is executed in the underlying implementation [1] for native nodes, but direct JavaScript processing and synthesis is also supported using the *ScriptProcessorNode*.

2.2 Native nodes

The initial idea of the specification is to give developers a set of highly optimized *native* nodes, implementing the commonly needed functions: playing buffers, filtering, panning, convolution etc. The nodes are connected to create an audio graph, to be processed by the underlying audio real-time rendering layer.

2.3 JavaScript ScriptProcessorNode

The ScriptProcessorNode interface allows the generation, processing, or analyzing of audio using JavaScript. It is an AudioNode audio-processing module that is linked to two buffers, one containing the input audio data, one containing the processed output audio data.

An event, implementing the AudioProcessingEvent interface, is sent to the object each time the input buffer contains new data, and the event handler terminates when it has filled the output buffer with data.

[1] typically optimized assembly or C/C++ code

This is the *hook* given to developers to add new low level DSP processing capabilities to the system.

2.4 Programming over the Web Audio API

Various JavaScript DSP libraries or musical languages, have been developed over the years ([4], [6], [8], [10]) to extend, abstract and empower the capabilities of the official API. They offer users a richer set of audio DSP algorithms and sound models to be directly used in JavaScript code.

When following this path, developments have to be restarted from scratch, or by adapting already written code (often in more real-time friendly languages like C/C++) into JavaScript.

An interesting alternative has recently been developed by the Csound team [11]: by using the C/C++ to JavaScript Emscripten [3] compiler, the complete C written Csound runtime and DSP language (so including a large number of sound opcodes and DSP algorithms) is now available in the context of the Web Audio API. Using an automatic C/C++ to JavaScript compilation chain opens interesting possibilities to ease the deployment of well-known and mature code base on the Web.

3 FAUST language description

FAUST [Functional Audio Stream] [1] [2] is a functional, synchronous, domain-specific programming language specifically designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing music languages like Max[2], PureData, Supercollider, etc., is that programs are not interpreted, but fully compiled. FAUST provides a high-level alternative to hand-written C/C++ to implement efficient sample-level DSP algorithms.

One can think of FAUST as a *specification language*. It aims at providing the user with an adequate notation to describe *signal processors* from a mathematical point of view. This specification is free, as much as possible, from implementation details. It is the role of the FAUST compiler to automatically provide the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs[3] taking care of generating the most efficient code. The compiler also offers various options to control the generated code, including options to do fully automatic parallelization and to take advantage of multicore architectures.

From a syntactic point of view FAUST is a textual language, but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (: , ~ <: :>) [1].

Here is an example of how to write a noise generator in FAUST:

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;
process = noise
        * vslider("Volume",0,0,1,0.1);
```

3.1 Language deployment

Being a specification language, the FAUST code tells nothing about the audio drivers or the GUI toolkit to be used. It is the role of the *architecture file* to describe how to relate the DSP code to the external world. Additional generic code is added to connect the DSP computation itself to audio inputs/outputs, and to control parameters, which could be buttons, sliders, num entries etc. in a standard user interface, or any kind of control using a remote protocol like OSC or HTTP.

This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max-MSP externals, PD externals, VST plugins, CoreAudio or JACK standalone applications, etc.).

4 FAUST compilation chain

4.1 Static compilation chain

The current version of the FAUST compiler (*faust1*) produces DSP code as a C++ class, to be inserted in an architecture file. The resulting file is finally compiled with a regular C++ compiler to obtain an executable program or plug-in (Figure 1).

The produced application is structured as shown in Figure 2. The DSP becomes an audio computation module linked to the user interface and the audio driver.

[2]the *gen* object added in Max6 now creates compiled code from a patch-like representation, using the same LLVM based technology

[3]In faust1, faust2 branch allows to compile for different languages

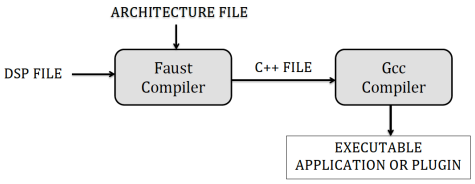


Figure 1: Steps of FAUST compilation chain

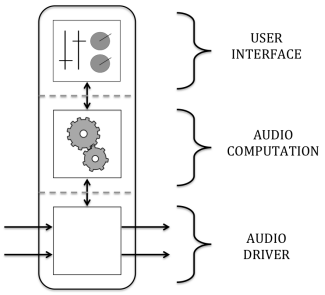


Figure 2: FAUST application structure

4.2 Multiple backends

Faust2 development branch uses an intermediate FIR representation (FAUST Imperative Representation), which can be translated to several output languages.

The FIR language describes the computation performed on the samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and define the necessary control structures (*for* and *while* loops, *if* structure etc.). The *language of signals* (internal to the FAUST compiler) is now compiled in FIR intermediate language.

To generate various output languages, several backends have been developed: for C, C++, Java, JavaScript, asm.js, and LLVM IR (Figure 3). The native LLVM based compilation chain is particularly interesting: it provides direct compilation of a DSP source into executable code in memory, bypassing the external compiler requirement.

4.3 LLVM

LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure, designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages. Executable code is produced dynamically using a "Just In Time" compiler from a specific code representation, called LLVM IR. Clang, the "LLVM native" C/C++/Objective-C compiler is a front-end for LLVM Compiler.

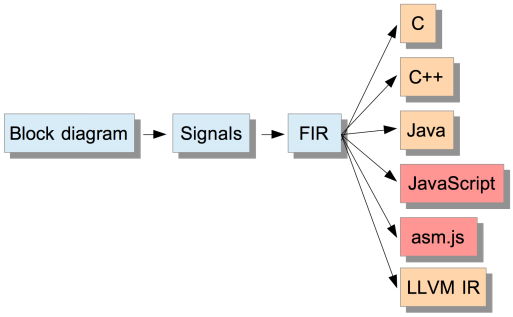


Figure 3: Faust2 compilation chain

It can, for instance, convert a C or C++ source file into LLVM IR code.

Domain-specific languages like FAUST can easily target the LLVM IR. This has been done by developing a special LLVM IR backend in the FAUST compiler.

4.4 Dynamic compilation chain

The complete chain goes from the DSP source code, compiled in LLVM IR using the LLVM back-end, to finally produce the executable code using the LLVM JIT [5]. All steps take place in memory, getting rid of the classical file based approaches. Pointers to executable functions can be retrieved from the resulting LLVM module and the code directly called with the appropriate parameters (Figure 4).

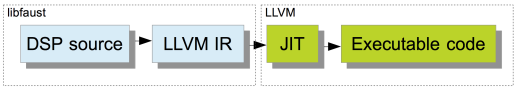


Figure 4: libfaust + LLVM dynamic compilation chain

4.5 FAUST compiler as a library

In the faust2 development branch, the FAUST compiler has been packaged as a library called *libfaust*, published with an associated API [5] that imitates the concept of oriented-object languages, like C++:

- given a FAUST source code (as a file or a string), calling the *createDSPFactory* function runs the compilation chain (FAUST + LLVM JIT) and generates the "prototype" of the class, as a *llvm-dsp-factory* pointer.
- next, the *createDSPInstance* function, corresponding to the *new className* of C++,

instantiates a *llvm-dsp* pointer, to be activated and controlled through its interface, and connected to the audio drivers.

Having the compiler available as a library opens new interesting possibilities explored in the FaustLive [9] application. DSP source code can be compiled on the fly and will run at native speed.

5 Using FAUST compiler in the Web

We have tested and implemented two different methods to use the FAUST compilation chain in the Web:

- the first one consists in embedding the *libfaust + LLVM* native compilation chain directly in the browser. Starting from the FAUST DSP source, a native WebAudio node will be compiled on the fly, to be used like any regular native node. The set of all control parameters will be exposed as WebAudio AudioParams objects.
- an alternative and more portable method purely stays at JavaScript level, using asm.js and Emscripten. Starting from the FAUST DSP source, a highly optimized asm.js based ScripProcessor node will be produced. The set of all control parameters will be exposed to control the DSP node.

Both approaches have advantages and issues that will be explained in detail in the following sections.

5.1 Native FAUST DSP Web Audio node

Embedding the *libfaust + LLVM* compilation chain has been experimented by "hacking" the WebKit open-source browser and by plugging the FAUST compiler in its Web Audio subproject.

A new native C++ *FaustNode* (sub-class of base class AudioNode) has been added to the set of native Web Audio nodes[4]. This node takes the DSP source code as a string parameter, compiles it on the fly to native executable code, and activates it:

```
var dsp
  = context.createFaustNode(code);
```

[4]This work was done in summer 2012 with the generous help of Chris Rogers, working at Google at that time.

As a native node, it can be used like any other regular native node and connected to other nodes in the graph:

```
dsp.connect(context.destination);
```

The FAUST source code usually contains an abstract description of its user interface, described in terms of buttons, sliders, bargraphs..., to be "interpreted" and displayed by an actual user interface builder component. This user interface can be obtained as a JSON description, that can be decoded to implement the UI themselves to control the node's parameters:

```
var json = dsp.json();
```

Internal control parameters of the DSP can be retrieved as a list of AudioParams, to be used like regular ones:

```
var num_params
  = dsp.numberOfAudioParams();
var audio_param
  = dsp.getAudioParam(0);
audio_param.setValue(0.5);
```

Instead of directly accessing the given parameter, another possibility is to use the following generic function, taking a complete access "path" to the parameter, and a given value:

```
dsp.setAudioParamValue("/wet",0.5);
```

Embedding the FAUST compiler in a browser is quite efficient, since the native executable code runs in the real-time audio thread that computes the audio graph rendering. But more general deployment and acceptance would require convincing the Web Audio community to embed a DSL language for audio processing in all browsers.

5.2 Compiling to JavaScript

More portable solutions have to use the ScriptProcessorNode node, directly producing JavaScript code to be executed in the node.

5.2.1 JavaScript backend

A pure JavaScript backend has been added to FAUST in 2012 to produce standard JavaScript code. The DSP class definition is then wrapped with a generic JavaScript file in order to get a fully working Web Audio ScriptProcessorNode.

5.2.2 Results

Two main problems have been discovered with this approach:

- for some of its computations, the FAUST compiler relies on pure 32 bits integral mathematical operations. Since JavaScript stores numbers as floating-point values according to the IEEE-754 Standard, this kind of computation can't produce the expected result. Thus, some DSP effects (like *noise* generation that uses a wrapping 32 bits integer division) do not work correctly.
- since standard JavaScript is not really suited to implement fast DSP code, the generated program is significantly slower compared to the native C/C++ or LLVM versions. The resulting audio nodes are usable only when the programmed DSP code is simple enough, but more demanding algorithms (like physical models) can usually not be used.

5.3 Compiling to asm.js JavaScript

Started in 2011 to facilitate the port of large C/C++ code base in JavaScript, Mozilla developers have started the *Emscripten* compiler project, based on LLVM technology, that generates JavaScript from C/C++ code.

Later on, they designed *asm.js*, a completely typed subset of JavaScript, statically compilable, garbage-collection free, that can be highly optimized by the compilation chain embedded in recent Web browsers. It is then possible to reach performances similar to pure native code[5]

Mainly designed to manipulate simple types like floating point or integer numbers, *asm.js* language is particularly of interest for audio code. Two successive developments have been carried out with this approach.

5.3.1 Using Emscripten compiler

Starting from the FAUST DSP generated C++ class, the Emscripten compiler translates it to JavaScript. Additional wrapping JavaScript code connects the Emscripten runtime memory manager and makes the generated code become a ScriptProcessorNode node to be used in the audio graph (Figure 5). This method has been successfully developed and demonstrated by Myles Boris [7].

Although this approach performs rather well, it requires the Emscripten tool chain to be installed on the user machine. A more integrated system has been later on developed.

[5]In the best cases, asm.js code is said to be only 2 or 3 times slower than pure native code, see `http://kripken.github.io/mloc_emscripten_talk`

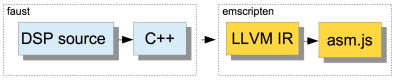


Figure 5: FAUST to asm.js (using Emscripten) static compilation chain

5.3.2 Developing a direct asm.js backend

A pure asm.js backend has been added to the faust2 branch, bypassing the Emscripten compilation chain (Figure 6).

The backend produces the asm.js module as well as some additional helper JavaScript functions, to be wrapped by generic JavaScript to become a completely usable Web Audio node. Heap memory code to be used with the asm.js module, and connection with compiled helper functions is managed by the wrapping code.

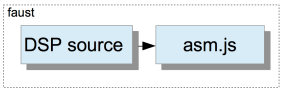


Figure 6: FAUST to asm.js (using FIR backend) static compilation chain

A new DSP instance is created using the following code, taking the Web Audio context and a given "buffer_size" as parameters:

```
var dsp
  = faust.karplus(context, buffer_size);
```

The user interface can be obtained as a JSON description, that can be decoded to implement the UI themselves to control the node's parameters:

```
var json = dsp.json();
```

The instance can be used with the following code:

```
dsp.start();
dsp.connect(context.destination);
dsp.setValue(path_to_control, val);
```

5.4 Embedding the JavaScript FAUST compiler in the browser

Thanks to the Emscripten compiler, the FAUST compiler itself can be compiled to asm.js JavaScript. This has been done by compiling the *libfaust* C++ library to the *libfaust.js* JavaScript library (Figure 7), that exports a unique entry point:

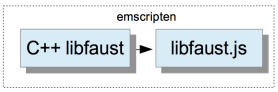


Figure 7: Compiling C++ libfaust to libfaust.js with Emscripten

- *createAsmCDSPFactoryFromString(...)* allows to create a DSP factory from a given DSP program as a source string and a set of compilations parameters, uses the asm.js backend, and produces the complete asm.js module and additional pure JavaScript methods as a string.
- then calling JavaScript "eval" function on this string *compiles* it in the browser. The dynamically created asm.js module and additional pure JavaScript methods (Figure 8) can then be used.

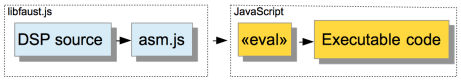


Figure 8: libfaust.js + asm.js dynamic compilation chain

This internal code in then wrapped with additional JavaScript code. A DSP "factory" will be created from the DSP source code with the following code:

```
var factory
  = faust.createDSPFactory(code);
```

A fully working DSP "instance" as a Web Audio node is then created with the code:

```
var dsp
  = faust.createDSPInstance(factory,
                            context,
                            buf_size);
```

The user interface can be retrieved as a JSON description:

```
var json = dsp.json();
```

The instance can be used with the following code:

```
dsp.start();
dsp.connect(context.destination);
dsp.setValue(path_to_control, val);
```

6 Use cases

Using the previously explained technologies, three different use cases have been experimented:

- compiling self-contained ready to use Web Audio nodes (see section 6.1)
- using FAUST static compilation chain to produce HTML pages with DSP code (see section 6.2)
- using the FAUST dynamic compilation chain to directly *program DSP* in the Web (see section 6.3).

6.1 Programming Web Audio nodes with FAUST

Self contained ready to use Web Audio nodes can be produced using the *faust2asmjs* script, using the static compilation chain explained in section 5.2. The script basically calls the FAUST compiler targeting the asm.js backend with the appropriate architecture file, that wraps the produced code with generic JavaScript to be usable in the Web Audio API context (Figure 9).

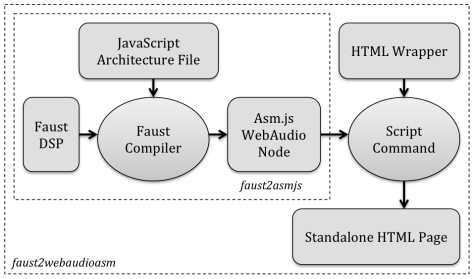


Figure 9: faust2amsjs and faust2webaudioasm compilation chains

6.2 Deploying FAUST DSP examples in the Web

Using the *faust2webaudioasm* script, a DSP source file can be compiled to a self-contained ready to run HTML page (Figure 10), using the static compilation chain (see section 5.2 and Figure 9).

The FAUST compiler targeting the asm.js backend with the appropriate architecture file is called. The asm.js + JavaScript WebAudio node is then wrapped in a more complex HTML code template, and the final HTML page is obtained. Adding the *-links* parameter to the script makes the HTML page also contains links

to the original DSP textual file, as well as the block-diagram SVG representation.

Thus it becomes quite simple to publish DSP algorithms, helping it wider usage of the FAUST DSL approach.

Figure 10: Example of SVG based user interface generated from the JSON description

6.3 Programming DSP in the Web

Having the FAUST compiler itself as a library in the browser opens interesting capabilities:

- "light" FAUST IDE allowing users to test the language can be easily developed on the Web, completing the more full featured FaustLive application [9].
- combining existing DSP sources published as HTML pages, to create new DSP programs to be directly tested and used in the Web, or possibly exported to any native platform supported by the FaustWeb external compilation service. This has been demonstrated by Sarah Denoux [13].

7 Tests and benchmarks

The three previously described approaches have been tested on a 4 cores MacBook Pro 2,3 GHz.

7.1 Benchmarks

The Web Audio API is still a fresh specification. Its implementation in different browsers on different platforms is not always complete or stable. Comparing the previously described approaches has been quite challenging, mainly because of slight differences of behavior or interaction with the underlying operating system.

The proposed benchmarks have been done by simply comparing the application CPU use with some heavy FAUST programs, using the "Activity Monitor" tool included in OSX. Three different DSP programs have been tested.

Since the various presented methods could not be developed in a same browser, we had to use two different ones. Native version is tested in the "hacked" WebKit application, JavaScript and asm.js using Firefox version 32.0.3.

Effect	native	JavaScript	asm.js
cubic_distortion	6.0 %	45 %	28 %
harpe	2.7 %	50 %	8 %
kisanaWD	4 %	over 100%	14 %

Table 1: Global CPU use of the application tested on a MacBook Pro 2,3 GHz

Even with this limited testing method, some interesting results emerge. The native chain (based on *libfaust + LLVM*) is clearly the fastest, the asm.js based one is usable in a lot of real world use cases. The JavaScript version performs poorly, and is even not usable because of CPU overuse in a lot of examples (like "kisanaWD" here).

7.2 Known issues and perspective

Although the previously described developments show some promising results, they are still several issues to be solved:

- code for pure JavaScript and asm.js generated nodes is executed in the main thread. So it may suffer from interferences with the UI computation or possibly garbage collection. Moreover latency is added since an additional buffer is used in the audio chain. Thus real-time guaranties may not be met typically resulting in audio glitches [6].
- a specific problem has been discovered when audio computation produces "denormal" float values: on Intel processors, CPU performances degrade a lot [7].
- on the contrary, the "native" version is much more stable, has less latency since the computation is done in the real-time thread with no added buffer, but is much more difficult to deploy and maintain [8].

[6] A possible solution to this problem by moving the ScriptProcessorNode code in audio worker threads has been recently discussed in the W3C Audio working list, see `http://webaudio.github.io/web-audio-api`

[7] The problem has been reported and should be solved at the JavaScript language definition level.

[8] A port in Firefox is in progress.

8 Conclusion

The FAUST audio DSP language can now be used to easily develop new audio nodes in the Web Audio model, and use them in an audio graph. Complete HTML pages with a working user interface can also be generated. Having the dynamic compilation chain (either in native or pure JavaScript mode) directly available in the browser is also interesting to further explore.

Even if the Web Audio approach starts to mature, there are still some problematic issues, for instance float samples denormalization problem, or non real-time guaranties while rendering the ScriptProcessorNode JavaScript code.

The recent discussion on the *Audio Workers* model opens perspectives for a better rendering scheme. Basically the JavaScipt audio code will be moved to the real-time audio thread, and communications to get/set parameter values will be done from/to the main thread.

It remains to be tested how the compilation of DSP to Web Audio nodes from a high-level DSL language like FAUST or Csound will benefit from it.

Acknowledgments

This work has been done under the FEEVER project [ANR-13-BS02-0008] supported by the "Agence Nationale pour la Recherche".

References

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

[2] S. Letz, Y. Orlarey and D. Fober, "Work Stealing Scheduler for Automatic Parallelization in Faust", *Linux Audio Conference*, 2010.

[3] A. Zakai, "Emscripten: an LLVM to JavaScript compiler", *In Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications, pages 301–312. ACM* , 2011.

[4] H. Choi, J.Berger, "Waax: Web Audio API extension", *In Proceedings of the Thirteenth New Interfaces for Musical Expression Conference.*, 2013.

[5] S. Letz, Y. Orlarey and D. Fober, "Comment embarquer le compilateur Faust dans vos applications ?", *Journees d'Informatique Musicale*, 2013.

[6] C. Roberts, G. Wakefield, and M. Wright, "The Web Browser as Synthesizer and Interface". *New Interfaces for Musical Expression conference (NIME)*, 2013.

[7] M. Borins, "From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten", *Linux Audio Conference*, 2014.

[8] C. Clark, A. Tindale, "Flocking: a framework for declarative music-making on the Web", *International Computer Music Conference*, 2014.

[9] S. Denoux, S. Letz, Y. Orlarey and D. Fober, "FAUSTLIVE Just-In-Time Faust Compiler... and much more", *Linux Audio Conference*, 2014.

[10] J. Kalliokoski, "audiolib.js, a powerful toolkit for audio written in JS", `https://github.com/jussi-kalliokoski/audiolib.js/`

[11] V. Lazzarini, E. Costello, S. Yi and J. Fitch, "Csound on the Web", *Linux Audio Conference*, 2014.

[12] WebAudioAPI reference description, `http://webaudio.github.io/web-audio-api/`

[13] S. Denoux, Y. Orlarey, S. Letz, and D. Fober, "Compose with Faust in the Web", *Web Audio Conference*, IRCAM & Mozilla Paris, France 2015.

AVTK - the UI Toolkit behind OpenAV Software

Harry VAN HAAREN
OpenAV Productions,
Co. Clare,
Ireland.
harryhaaren@gmail.com

Abstract

AVTK[1] is a small lightweight user interface toolkit designed for building custom graphical user interfaces interfaces. It was conceived particularly to build LV2 plugin GUIs, however it can be equally useful in standalone programs.

Focusing on user experience, AVTK promotes ease of use for novices yet affords power-users the most efficient interactivity as possible.

The author feels this is particularly important in live-performance software, where user-experience and creativity are in close proximity.

Keywords

User Experience, User Interface, LV2 Plugins.

1 Introduction

AVTK is a C++ user interface library. It is designed specifically for creating LV2 plugins[2], but can be used to build normal user interfaces too. OpenAV created AVTK out of the need for a flexible and powerful way of creating custom user interfaces for audio plugins.

In the background section, common issues when building user interfaces for plugins are presented. The implementation of the AVTK user interface library is then shown, and code samples are provided, providing an example of how to build a minimal interface. Finally we discuss how the common problems presented in the background are solved in this implementation.

2 Background

When creating an embeddable user interface for a plugin, issues arise that do not apply to writing normal user interfaces. These issues arise from the interaction between the user interface of the host program and plugin.

2.1 Embedding

Embedding is the process of showing a user interface created in one toolkit inside the window of another toolkit. This is a complex problem and many different solutions exist. Many solutions are platform dependant, ie: they only work on Linux, Mac OsX or Windows.

PUGL[3] is a cross platform library which "supports embedding and is suitable for use in plugins". AVTK uses PUGL to embed into host program's windows, and it can create a standalone window too if embedding isn't desired.

2.2 Modal Popup Windows

Many general purpose toolkits have the functionality to create modal popup windows. This type of window can cause problems when embedding a plugin UI in a host window.

The first problem is that the popup is set to be "above" its parent window, however the parent window is generally already set to be "above" the host program. This can cause the popup window to be below the parent window, however modal popup windows don't allow the user to interact with the parent. The user is force to manually bring the popup window to the front, and interact with it. This has a negative impact on the user experience (UX).

The second problem is one of stalling the UI thread of the host program. When a modal popup dialog is created, many user interface toolkits wait for the user to interact with it, before continuing the execution. This stalls the host programs UI thread, as that thread created the dialog. The end result is that as the popup window is shown, the host programs UI is frozen. Again, a negative impact on UX.

2.3 User Experience

When using a plugin user interface, user experience is of very high importance. In the context of AVTK, the user is most likely creating music, or involved in a creative process of some description. To ensure the best UX for both novice and power users, the normal user interaction concepts are augmented in AVTK. Care is taken to ensure that these augmented interaction possi-

bilities do not cause confusion to novice users.

Transparently augmenting the interaction with a widget provides power users with better UX, and ultimately in the user achieving their goal more efficiently. The scroll wheel on the mouse, and modifiers keys are used in order to augment the interface for efficency.

3 The Implementation

The implementation of AVTK draws from experience gained while developing custom interfaces using other toolkits.

The main window is treated as one large canvas, and widgets are drawn into this canvas. As widgets can be transparent, creating layered interfaces becomes easy.

3.1 Dependencies

During the design stage cross-platform libraries were chosen, in order to ensure portability. Cairo[4] is used for all drawing routines, while PUGL provides access the window and input events. PicoJSON[5] is used for parsing JSON, while tinydir[6] provides access to the filesystem.

3.2 Widget Class

The Widget class is the core of AVTK. The Widget class has virtual functions that a developer can override to customize behaviour. The following is an excerpt of the `Widget` class:

Listing 1: Widget Class

```
class Widget
{
public:
  Widget( Avtk::UI* ui,
          int x,
          int y,
          int w,
          int h,
          string label );

  // draw and handle events
  virtual void draw(cairo_t* cr);
  virtual int handle(PuglEvent* e);

  // set value on widget
  float value();
  void  value( float v );

  // change notification callback
  void (*callback)(Widget*, void*);
  void* callbackUD;
};
```

The most important function is draw(), which paints the widget to screen. The handle() function deals with user input using the cross-platform `PuglEvent` abstraction.

The `value()` functions set and get the value of the widget. A callback function can be provided to be notified of activity on a particular widget instance.

3.3 Minimal UI

The code in listing 2 shows a minimal AVTK user interface with a single button. Notice that we override the `widgetValueCB()`function, and that the callback function of any child widgets is automatically routed to the `UI` instance.

Listing 2: Minimal Demo UI

```
class DemoUI : public Avtk::UI
{
  public:
  DemoUI(PuglNativeWindow parent = 0) :
    Avtk::UI( 400, 240, parent )
  {
    new Avtk::Button( this,
      50, 20, 300, 200, "Button" );
  }

  void widgetValueCB(Avtk::Widget* wid)
  {
    printf("Widget %s with value %f\n",
      wid->label(), wid->value() );
  }
};

int main()
{
  return DemoUI().run();
}
```

3.4 Custom Widget Creation

In order to customize a widget one simply derives from the `Widget` base class, and overrides the `draw()` method. The Cairo library is used to draw widgets, and a `cairo_t*` context is passed into the draw() function. Calling the desired Cairo functions will draw the widget.

The `value()` function of the `Widget` base class can be called to get the current value of the widget - allowing easy drawing of the widget in its current state.

The `handle()` function can be overriden in case the custom widget requires non-default user input handling.

3.5 Modal Widgets

Modal popups are implemented as a popup *widget*, instead of its own window in AVTK. This is to keep UX consistent, and avoid the pitfalls of modal windows as described in the background section.

As the calling thread should not be stalled while the popup is shown, AVTK treats a popup widget like a normal widget. The exception is

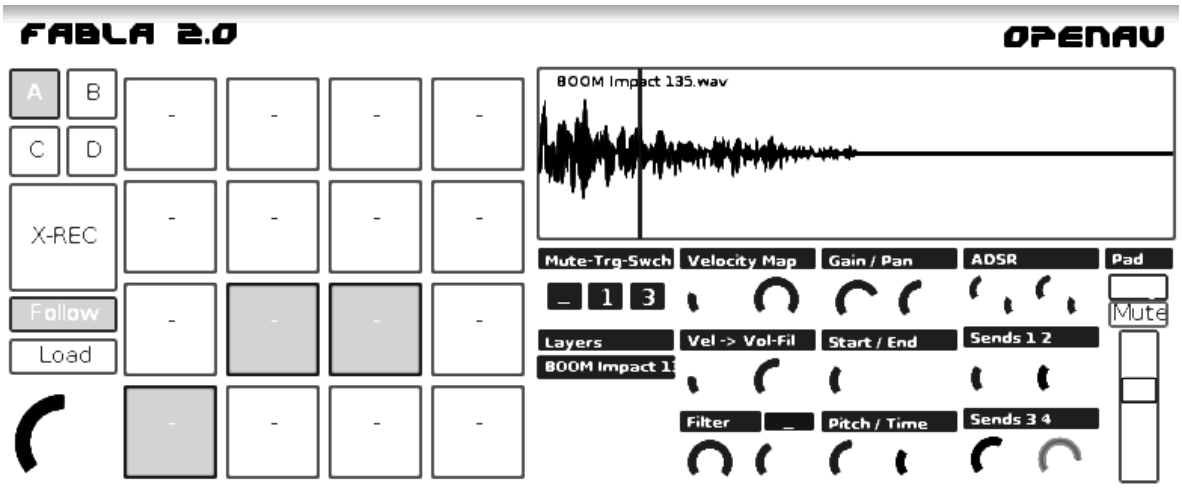


Figure 1: The Fabla2.0 interface, powered by AVTK. Edited for print.

event handling, which allows the modal widget to disable interaction with the other widgets while it is shown.

Figure 2 shows two user interfaces one of which has a modal dialog shown. Note the dialog is positioned close to the cursor for ease of interacting with it.

3.6 Theming

The theme engine in AVTK is geared towards providing a variety of themes to a widget, and being able to change them on the fly. Themes are loaded from JSON files at runtime, allowing easy modification.

Theme files provide colours for "use-cases". Some examples are background, foreground, background-dark etc. This approach allows using the same theme file on any widget, and the widget adapts the colours in the theme to what is being drawn. This lightweight approach to themeing allows fast prototyping, and scales to having many different themes available for each widget.

Fig. 1 shows an interface, and various widgets themed to show the user the effect of the widget in question. In this particular screenshot changing the "Bank" changes the primary colour of the interface, indicating the change to the user.

4 Special Features

This section describes special features of AVTK. The interaction between the power user and the interface is where the design of AVTK flourishes. Many widgets afford operation with hotkeys, right-clicks and drag-n-drop areas. The following sections discuss where the interaction between user and interface has been augmented.

4.1 Right Click and Default Values

User input from the mouse buttons is leveraged in almost every widget in order to allow efficient resetting of controls. Any `Widget` that uses its `value()` is enabled with a right-click feature to reset to the default value. Calling `defaultValue()` sets a new default value for the widget.

Right-click to reset to default value can be disabled, using the `Widget::rClickMode()` function.

4.2 Groups and Scroll events

`Groups` can be used to provide radio-button style selection of a particular widget in the group (see Fig 2. for widget examples).

This is particularly useful in scrollable areas, which are commonly used in applications to select a particular option from a range of predetermined options.

Using input from the scroll wheel is an intuitive mapping - but there is a conflict when a `Group` is in a `Scroll` widget as the Scroll widget uses the scroll event. In order to achieve an optimal workflow, a `Ctrl+Scroll` action is added to a `Group` allowing the user to navigate group.

The efficiency of navigating a widget list is improved for power users familiar with the `Ctrl+Scroll` hotkey, while consistency is maintained in how widgets behave regardless of if they are in a scrollable area or not.

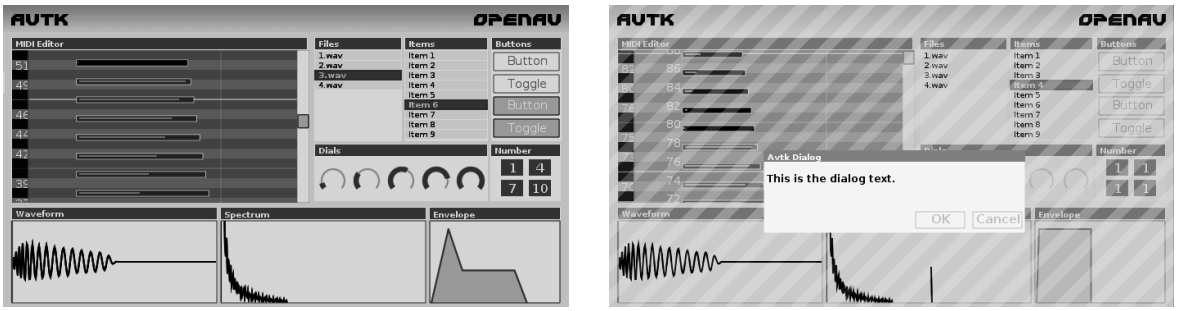


Figure 2: Showcase of AVTK widgets, a popup Dialog "stripes out" the background.

4.3 Filenames

When the user is searching for a particular audio file in a list of similar files, the filenames often have an identical start causing long filenames. This causes a workflow where the user must scroll horizontally and vertically in order to select the desired file.

In an attempt to solve the workflow issue as above, the AVTK file-browser includes an option to hide the common start of a filename. And example to illustrate this is as follows:

```
// Actual Filenames
samplepack_kick_heavy.wav
samplepack_kick_click.wav
samplepack_snare_snappy.wav

// AVTK Filenames
kick_heavy
kick_click
snare_snappy
```

The example above shows three filenames with the common prefix `samplepack_` followed by the information user requires, followed by a filetype extension.

In this example, the common prefix `samplepack_` has been removed, and the extension is removed for readability (It is noted that many general purpose toolkits already hide the file extension).

4.4 Testing of User Interfaces

When an issue is found in a program by a user, often the first step a developer takes is to attempt to reproduce the issue. The user is asked to describe what steps will reproduce the issue, and the developer mimics them in order to find the cause of the issue.

To improve the workflow in finding UI issues, user input is recorded, and then replayed on the developer's computer.

AVTK serializes the input events from the user to a JSON file, which is uploaded to the developer. They then replay the events, automatically mimicing the users input.

It should be noted that the developer must have the same version of the software as the user as pixel co-ordinates are stored in the event stream. It follows that if the user interface is re-arranged, the users actions may no longer achieve the same result.

5 Conclusion

AVTK is a small lightweight user interface toolkit, targeting developers who wish to build custom user interfaces. It solves common issues other toolkits have when embedding as a plugin UI by utilizing a more appropriate design for the use-case.

It has some special features geared towards power-users, and has a theme engine that allows developers create prototypes quickly. An event recording and playback mechanism is included to aid finding issues users are having with the software in question.

6 Future Work

OpenAV intends to continue using AVTK to build interfaces for plugins and standalone software. Currently the Fabla 2.0 sampler is the only complex project using AVTK.

Future work includes expanding the available widgets as necessary for multi-media centric software, and testing on all major platforms.

7 References

[1] http://openavproductions.com/avtk
[2] http://lv2plug.in
[3] http://drobilla.net/software/pugl
[4] http://cairographics.org
[5] http://github.com/kazuho/picojson
[6] http://github.com/cxong/tinydir

Ingen: A Meta-Modular Plugin Environment

David E. Robillard
School of Computer Science, Carleton University
1125 Colonel By Drive
Ottawa ON K1S 5B6
Canada
d@drobilla.net

Abstract

This paper introduces Ingen, a polyphonic modular host for LV2 plugins that is itself an LV2 plugin. Ingen is a client/server system with strict separation between client(s) and the audio engine. This allows for many different configurations, such as a monolithic JACK application, a plugin in another host, or a remote-controlled network service. Unlike systems which compile or export plugins, Ingen itself runs in other hosts with all editing facilities available. This allows users to place a dynamic patching environment anywhere a host supports LV2 plugins. Graphs are natively saved in LV2 format, so users can develop and share plugins with others, without any programming skills.

Keywords

LV2, JACK, plugin, modular, synthesizer

1 Introduction

The Free Software world has long had powerful visual programming environments like Pure Data [Puckette, 1996], and higher level modular synthesizers like Alsa Modular Synth [Nagorni, 2003]. However, most existing software modular environments (or simply *modulars*) do not integrate as well as possible with other projects. Due to the limitations of popular plugin APIs, most existing modulars are primarily designed around built-in *internals*, and use a different interface themselves (e.g. running only as an application). This situation results in much effort spent building components that are not widely useful across applications.

There are several typical forms for audio processors: a stand-alone software application, a software plugin, or a remote device. Each has advantages depending on the situation. Remote control is necessary for hardware to integrate with a software environment, and increasingly popular for software due to the pervasiveness of tablets, powerful controllers, and networks. An ideal system must be controllable from any location to fit well in all these scenarios. Consequently, the same must be true of the plugins hosted within it.

This leads to the elegant conclusion that the ideal form of a modular plugin host, and the ideal form of a plugin within it, are one and the same. Ingen is an exercise in chasing this ideal: a modular host that has exactly the same external form as the plugins used within it. The practical benefit of such a design is that the user can build a device anywhere in the system where plugins are supported. This makes it possible to work around limitations in programs or the lack of an available plugin to solve the necessary problem. By making it simple for users to share their creations, the community at large can benefit from the pool of plugins created by users who would not have done so if writing code was required.

Ingen takes advantage of the LV2 plugin API's extensibility to achieve these goals. The two have a symbiotic relationship: when Ingen itself requires an API advancement, the improvement ideally becomes standardised in LV2. Other plugins may then use this API, resulting in more powerful plugins for use in Ingen, or other hosts. Likewise, Ingen benefits from advancements originally designed for other plugins.

This paper introduces Ingen as a useful tool for users, and shares the general conclusions reached over the years that led to its design, and consequently the design of many aspects of LV2.

2 Features and Philosophy

2.1 Internals Considered Harmful

Ingen is designed around the principal that generic plugins should be used wherever possible: internals are a symptom of an inadequate plugin API. With an open and extensible specification like LV2, these limitations can be addressed so progress needn't be stalled. This avoids a walled garden effect where a large amount of effort is spent on internals that only work in one program.

The minimalist ideal is for Ingen to have no internals whatsoever, but currently a few are required for tasks that are beyond the capabilities of generic plugins. In particular, LV2 currently lacks

polyphonic voice control, so the *Note* internal performs voice allocation, sending controls to particular voices based on MIDI input. Perhaps in the future there will be sufficient developer interest in (visibly) polyphonic plugins to develop an LV2 extension which will eliminate the need for a special voice allocation internal.

2.2 Messages Considered Wonderful

The common combination of GUI-centric design and direct memory access between plugins and their GUIs results in several problems. In particular, a large subset of many plugins' functionality is inaccessible except via the custom GUI. This severely limits the power of the system to intelligently automate or otherwise control plugins. In addition, shared access to mutable data from multiple threads is an infamously error-prone situation, made even more difficult with the addition of real-time requirements. It is all too common for a poorly written GUI to cause audio drop-outs, or crash the plugin entirely. Direct access to plugin internals is occasionally necessary (for visualisation in particular), but is an inherently flawed approach to plugin control in general.

The solution to this problem is a classic one: separate the plugin and its user interface, and have the two communicate via messages. If these messages are meaningful (i.e. not opaque), the plugin can be controlled in the same way from any source: the GUI, the host, other plugins, scripts, and so on. Traditionally, MIDI is used for this purpose, but MIDI has significant limitations. Ingen supports sending arbitrary messages between hosted plugins and their UIs (including MIDI), and is controlled entirely via messages itself.

Control via standard and portable messages is the key to building audio components that can be deployed in many different scenarios. Designing the system fundamentally around this principle (rather than "bolting on" partial support for remote control) ensures that all interfaces to the system enjoy the same power.

2.3 Polyphony

Though inspired by modular synthesizers, Ingen does not seek to emulate the limitations of hardware. Polyphony in particular is an important feature where software has a distinct advantage. This is an area where extreme minimalism is counterproductive: though it is possible to build a polyphonic synth manually in a monophonic modular by replicating voices, this is a burden on the user. Instead, Ingen implements polyphony internally. Nodes can simply be flagged as polyphonic, and they will be replicated as necessary. Polyphony[1] is a property of the containing graph, i.e. if a graph has polyphony p, all nodes in that graph have either 1 or p voices. Any connection between polyphonic ports is a polyphonic connection, and any connection from a polyphonic port to a monophonic port mixes down all voices.

2.4 Data Types

Ingen supports many data types, including audio, "control voltage" (CV, audio-rate numeric controls), and events in any format such as MIDI. A port transmits either signals or sequences: audio and CV are the only signal types, everything else is a sequence. Sequences are a series of "events" or "messages" transmitted in-band with audio. LADSPA-style control ports are control-rate signals from the point of view of the plugin, but in Ingen are exposed as sequences of floating point numbers to allow control changes to be transmitted with sample accuracy.

The ability to work with many data types is powerful, but requires the user to understand the types of different ports. Ingen distinguishes port data types by colour, and also adds hint symbols as shown in Table 1. Signal and sequence ports are distinguished by shape: signal ports have rounded borders (suggesting *continuous*), and sequence ports have bevelled borders (suggesting *discrete*). A symbol is also composed on the type hint, for example, a real number signal (CV) is tagged '$\tilde{\mathbb{R}}$', and real number messages are tagged '$\ddot{\mathbb{R}}$'.

Symbol(s)	Data type
$\sim$	Audio (floating point)
$\mathbb{R}$	Real (floating point)
$\mathbb{Z}$	Integer
$\mathbb{M}$	MIDI
$\square, \boxtimes$	Boolean
$=$	Patch message

Table 1: Type hint symbols for ports.

Despite the many different data types, Ingen attempts to preserve the "anything to anywhere" ability of classic modular synthesizers wherever possible. For example, a float message output can be connected to a CV input; Ingen will automatically write the CV buffer as if the signal were continuous.

2.5 Inter-Plugin Communication

Many plugins must communicate both audio and messages, a typical example being MIDI synthesizers. Both are transmitted in Ingen in the same context to allow sample-accurate real-time message

[1] As opposed to the boolean polyphon*ic*.

handling, and avoid threading issues. This is distinct from some systems, such as Pd, where message transmission follows different rules than signal transmission. In other words, messages in Ingen are in-band with audio signals.

The benefit of this approach is a single consistent concept of real-time: plugins have one `run()` method which processes all inputs and emits all outputs synchronously. However, some plugins must perform non-real-time operations in response to messages. For example, a sampler plugin may need to load samples from disk.

The LV2 worker extension solves this problem. The worker extension provides a simple API for plugins to schedule a callback to be called "soon" in a non-real-time thread, and a mechanism for replying back to the audio thread in a later cycle. This makes it possible for plugins to perform non real-time operations, but the API is designed such that its use is inherently real-time safe, and plugins do not need to use any non-portable threading libraries. Having this mechanism implemented by the host has performance benefits as well, for example, the host can share one ring buffer and worker thread for all plugins. This can dramatically reduce the memory consumption when many plugins are loaded.

3 Architecture

3.1 Model

Ingen uses a simple data model to describe all components of a graph. Each object has a unique path (like `/fx/verb1`) and a set of key:value properties. Keys are URIs (making state meaningful), and values may have any type. Essentially, everything is a hierarchical tree of dictionaries.

The use of a consistent data model allows for a very simple protocol to perform a large number of operations. Rather than adding "commands" to the interface for every new feature, changes are implemented in terms of property changes. Only a few methods are required to allow arbitrary property changes, so this allows for a powerful yet stable protocol. There are no issues with breaking the number or order of arguments, since properties have no order. New information can be added freely without requiring any changes to old code.

3.2 Protocol

The Ingen protocol itself is very similar. Messages are built from LV2 Atoms [Robillard, 2014], particularly "Object"[2] which is a dictionary with URI keys and any type of value.

[2]This is an "object" in the JSON sense, not as in object-oriented programming.

The LV2 Patch extension defines several messages, similar to HTTP and DAV methods, which can be used to access and manipulate the graph. The simplest is a `Get`, which requests a description of the given subject:

```
[
  a patch:Get ;
  patch:subject </osc> ;
]
```

The response describes the subject in the same format, in this case the plugin instance, or *block*:

```
</osc>
    a ingen:Block ;
    lv2:prototype <urn:someplugin> ;
    ingen:canvasX 42.0 ;
    ingen:canvasY 24.0 .
```

Manipulation is similar. For example, a `Put` message can be used to create the above block:

```
[
  a patch:Put ;
  patch:subject </osc> ;
  patch:body [
    a ingen:Block ;
    lv2:prototype <urn:someplugin> ;
    ingen:canvasX 42.0 ;
    ingen:canvasY 24.0 ;
  ]
]
```

Syntactically, this says "I am a `Put` message, with subject `/osc`, and body `[ a ingen:Block ...]`". The definition of `patch:Put` and the associated properties gives us the meaning: "*put* this block at `/osc`".

The short names here are abbreviations of URIs, for example, `patch:Put` expands to `http://lv2plug.in/ns/ext/patch#Put`. URIs are used here to provide a global namespace, but when properly documented, also provide transparency. For example, the above URI leads to documentation which describes the meaning of a `patch:Put`. This documentation is also machine readable to support intelligent tools. For example, a `patch:Put` *must* have one `patch:subject` property, and the same tools used for LV2 plugin validation can ensure this restriction is obeyed. Note, however, that no Internet access is involved in handling messages; properly documenting URIs is simply a best practice for convenience and tool support.

There are similar messages to delete elements, set properties (including control values), and so on. All messages are defined in the LV2 Patch extension, which is also used by some plugins for control (for example, the LV2 example sampler uses this vocabulary to load samples).

3.2.1 Serialisation

Conceptually, Ingen uses the same protocol everywhere. However, the above text serialization would only be used over a network, or shown for debugging purposes. When running in the same process, messages are instead serialised as binary LV2 atoms for increased performance. These two encodings are conceptually identical and differ only in representation. Similarly, plugins inside Ingen which communicate with atoms are connected directly, with no serialisation.

The same textual serialisation used in the remote protocol is used when saving graphs. Conceptually, the Ingen protocol can be considered a stream of patches to the saved graph (hence the name of the LV2 Patch extension). For example, the description of the `/osc` block returned by the server in Section 3.2 could be a literal snippet of a saved graph file. Ports use the standard LV2 vocabulary, so Ingen graphs can be loaded by applications with LV2 support, with no special Ingen support required.

3.3 Event Handling

Building and manipulating a graph of plugins requires operations that are not real-time safe. To allow live editing without dropouts, Ingen must avoid all such operations (such as memory allocation or mutex locking) in the audio thread.

Conveniently, message-based control lends itself to an event-oriented implementation, which makes for an elegant solution to this problem. All operations in Ingen are implemented as events which are triggered by the receipt of some message. An event has three phases:

1. Pre-Process: Upon receipt of the message, perform any non-real-time operations necessary before the change can be applied (e.g. instantiate a plugin). When finished, push the event into a queue for the audio thread.

2. Process: In the audio thread, apply the changes prepared in the pre-process stage (e.g. insert an instantiated plugin into a graph). After this, the change is effectively complete. When finished, push the event (including references to any resources that need to be freed) into a queue for post-processing.

3. Post-Process: Clean up any necessary resources, and broadcast the change to all clients.

4 Examples

The most straightforward use of a modular is to build chains of effects plugins. Though simple, even this provides an improvement over what is easily achievable in hosts with a strictly linear signal path. For example, processing the left and right channels separately in a DAW like Ardour can be achieved this way without complicating the session's bus routing.

More interesting is to custom-build instruments. Figure 1 shows an example of an extremely simple polyphonic synthesizer, with only one envelope, saw oscillator, and low pass filter.

Ingen allows graphs to be nested, and has no restrictions on the type or number of ports present. For example, Figre 2 demonstrates adding sidechain compression to a synthesizer graph.

It can be useful to combine existing high-level plugins with more low level components. For example, Figure 3 shows a graph which contains multiple instruments. A MIDI filter [Gareus, 2014] plugin is used to send automatic chords to an electric piano, while the input note is sent to a synthesizer.

5 Future Directions

Ingen is currently useful as an environment for hosting plugins with flexible routing. Its architecture allows it to function in a diverse range of environments, which has been the focus of development to date.

One goal for future development is to become a more powerful programming environment. Since plugins are free to communicate with arbitrary messages, the necessary infrastructure is already available, but an appropriate set of plugins is missing. Existing systems like Max/MSP and Pd are very mature in this respect, but use a different model than Ingen and LV2. In particular, it will be interesting to investigate how to exploit *meaningful* messages to provide a powerful modular programming environment.

References

Robin Gareus. 2014. midifilter.lv2. https://github.com/x42/midifilter.lv2.

Matthias Nagorni. 2003. Alsa Modular Synth. http://alsamodular.sourceforge.net/.

Miller Puckette. 1996. Pure Data: Another integrated computer music environment. *Proceedings of the Second Intercollege Computer Music Concerts*, pages 37–41.

David Robillard. 2014. LV2 Atoms: A data model for real-time audio plugins. In *Linux Audio Conference 2014*.

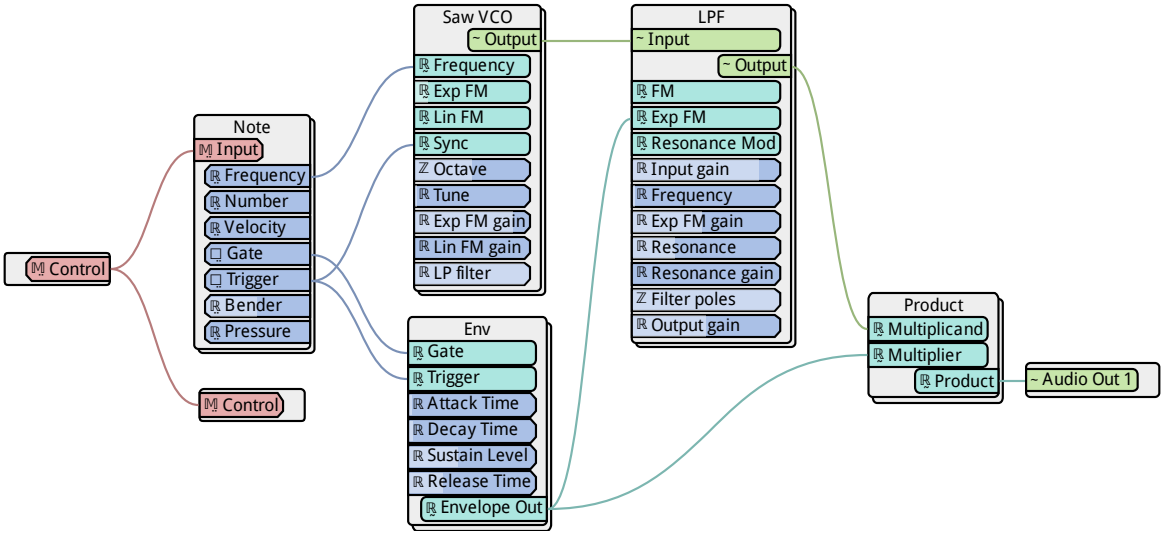


Figure 1: A simple polyphonic synthesizer.

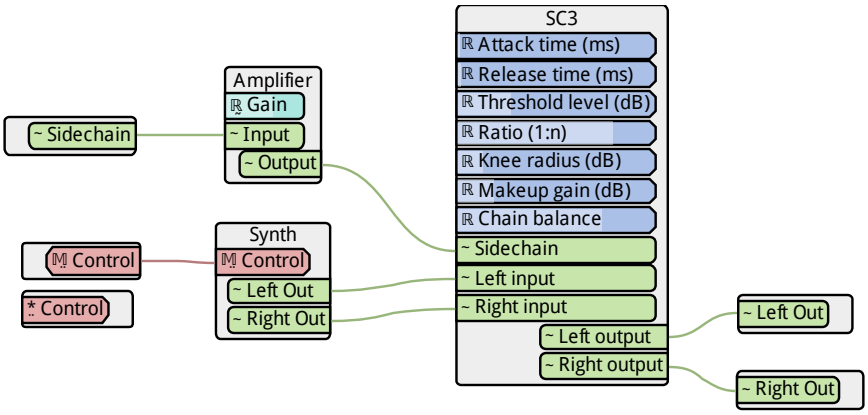


Figure 2: Adding sidechain compression to a synthesizer. The "Synth" block shown here is a nested Ingen graph (which can be edited by double-clicking in the interface).

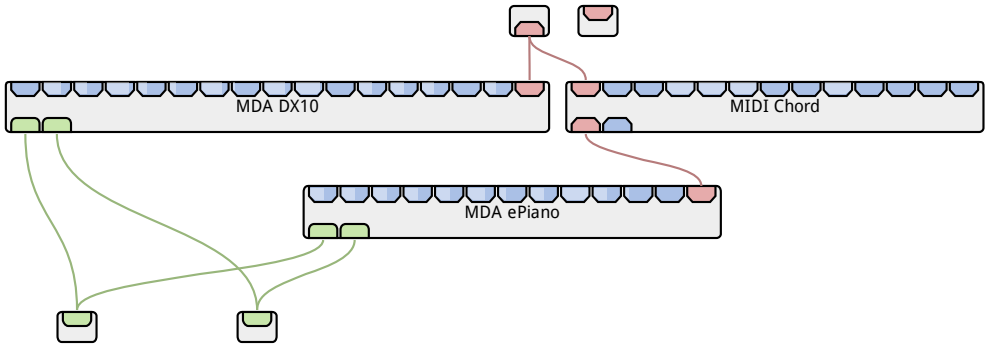


Figure 3: Simple MIDI processing to play the input root note on a DX10, and accompanying chords on an ePiano. Shown in vertical mode.

Segment Synthesizer

Andre SKLENAR
Kakapo Electronics s.r.o.
Na Slupi 5
Prague, Czech Republic, 128 00
andre.sklenar@gmail.com

Michal SYKORA
Kakapo Electronics s.r.o.
Na Slupi 5
Prague, Czech Republic, 128 00
michal.sykora.sk@gmail.com

Martin PATERA
Kakapo Electronics s.r.o.
Donatellova 5
Prague, Czech Republic, 100 00
mzstic@gmail.com

Amir HAMMAD
Kakapo Electronics s.r.o.
Janka Krala 18
Nitra, Slovakia, 94901
amir.ozk@gmail.com

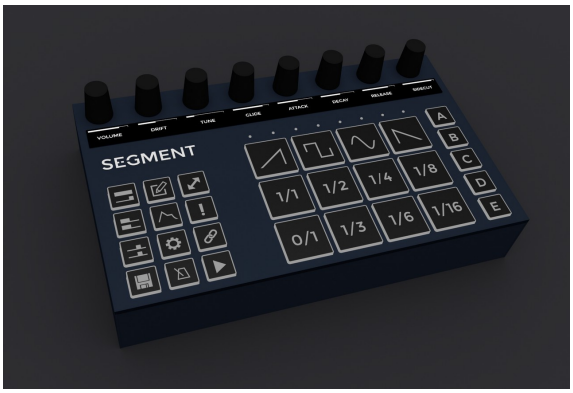
Abstract

This work aims to describe the new hardware *Segment Synthesizer*, its development process, cover a description of the original *Segment Synthesis* and its properties, advantages and limitations. It will also clarify the way the hardware controls and the user interface were designed, Segment's connectivity and its audio features.

Segment's accompanying features, such as the *Segment Cloud* and the *Segment Manager* will also be covered.

Keywords

Segment, Synthesis, Open Source, Open Hardware, Synthesizer



1 Introduction

Segment Synthesizer is an open source and open hardware monophonic digital hardware synthesizer. It features an original synthesis method invented and designed by Andre Sklenar called *Segment Synthesis*. It has no keyboard, so the user has to connect their own or drive *Segment* with a different input. A beats per minute - synced, low frequency oscillator modulated low pass filter is present on *Segment*, directly controllable by primary silicone pads on the front panel. Segment contains several other features, such as *drift* for intentional parameter instability or a flexible parameter routing.

From the hardware perspective, *Segment* runs bare-metal on an ARM Cortex-M4 chip, features a wide-ranging connectivity and supports multiple data protocols.

The first accompanying software is the *Segment Cloud*, where *Segment* users will be able to share their presets along with a short sound context example.

The second accompanying software application is the software that allows for *Segment* - computer communication for up/downloading presets and firmware and creating backups. It will also allow the users to browse the cloud and up/download content directly from/to *Segment*.

2 Segment Synthesis

Segment Synthesis is an original synthesis method aiming to create very thick and harmonically rich timbres in a most simple and elegant way.

2.1 The basic principle

The main idea behind *Segment Synthesis* is that only half of a period of a primitive waveform[1] is needed to get enough timbral information about the waveform. When applying this approach to the waveform construction, after the first half period of a waveform is rendered, another first half of a different primitive waveform is rendered, but one octave higher. When that is finished rendering, we continue with another half period of any primitive waveform, one octave higher again and so forth.

We call each of these waveform bits 'segments' – hence the name of the synthesis.

This basic idea is the main drive behind segment synthesis. Obviously, the approach described above is not practical, so fixed limitations and a specific implementation had to be adopted.

The closest relative of Segment Synthesis is Xenakis' Gendy Synthesis[2].

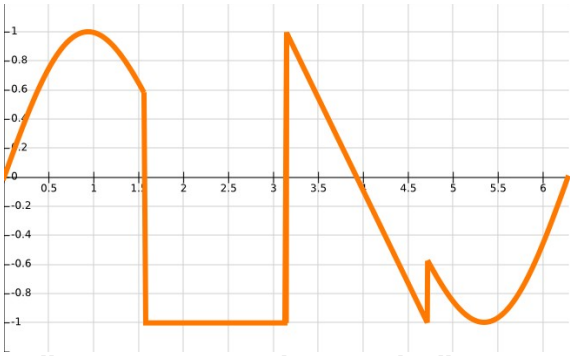


Illustration 1: Simple example illustration of Segment Synthesis (no frequency modulation).

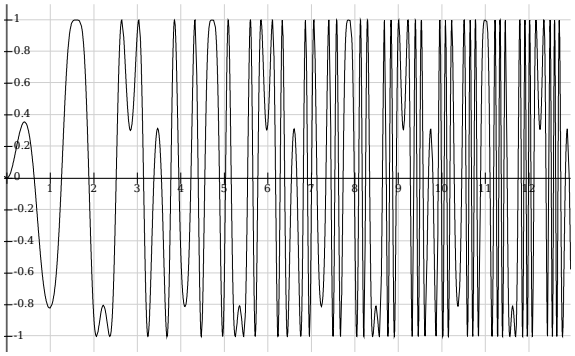
2.2 Frequency modulation

To further increase the harmonic spectrum employed in today's music production, a harsh approach was undertaken. Each segment is frequency modulated by itself using this equation (in this case the primitive waveform is a sine wave):

$$y = sin(x * sin(nx))$$

Where x is the phase accumulator running from 0 to 2π and n is the frequency modulation coefficient.

Should the phase accumulator be allowed to run indefinitely, this approach would produce a vastly non-periodic waveform:



Because of this and other practical reasons, the phase accumulator is reset to 0 when $x>2\pi$. This way the output is always periodic.

2.3 Segment implementation

Segment implements this synthesis method by providing the musician with 6 segments, where the waveform of each segment can be set by blending between one of the four primitive waveforms (as described earlier), the amount of frequency modulation, panning and volume. The first segment is a whole period. Second and third segment occur in the same time frame, but can be set independently and cover half a period and tune on double the fundamental frequency.

The fourth and fifth segments cover the time frame from ½ the period to ¾ the period and play 3 octaves higher. Again, the user can set their waveform, amount of frequency modulation, stereo panning and volume independently.

The sixth and final segment renders from ¾ of the period until the end and plays four octaves up.

This particular *Segment Synthesis* implementation is neither the best or the only one working. It is simply the implementation we have tuned to our liking and is one of many possible combinations.

2.4 Practical results

Musicians always seek new ways to craft their sounds. As a direct result, various post-processing is involved in our search for harmonically rich synthesizer patches. It is not uncommon to distort and saturate almost every channel, add choruses, double the tracks and detune their synthesizer voices to reach that goal.

Segment Synthesis tries to tackle this issue right at its root – the actual synthesis.

Direct comparisons with usual additive/subtractive synthesizers show that even when not employing frequency modulation, using the same amount of oscillators, *Segment* produces a much richer harmonic spectrum[3].

Due to a very high number of harmonics, the fundamental frequency can be tuned down way below 20Hz, while still retaining a significant amount of sonic information, which were previously multiple-aliases. This further widens the range of possible timbres.

2.5 Pitfalls

Due to *Segment Synthesis*' sharp edges (similar to hard-sync), *Segment* gets into aliasing very easily. The number of harmonics is so high, 2x oversampling did not produce satisfactory results and this was already over the edge of the computational power available. As this is clearly a problem, more research is needed on this topic.

Another, this time aesthetic, problem is that sometimes it is difficult to fit Segment into existing music because of the high amount of harmonics. Again, more research is needed to gain knowledge about how to control the amount and content of higher harmonics.

3 Auxiliary audio features

Because no synthesizer would suffice with just raw synthesis, Segment has several functions to make it a full-featured synthesizer device.

3.1 Beats per minute synced low frequency oscillator modulated low pass filter

Segment builds on a simple notion that most of today's popular music is grid-based. Based on that, we have implemented a low pass filter controlled by a low frequency oscillator which is permanently synchronized with the main tempo. This tempo can be tapped, set manually or received via any of the communication protocols we support (more on that later).

3.1.1 User interface

To control the LFO, which is one of Segment's main features, Segment has 12 primary silicone pads. These are in a 4x3 arrangement, where the top 4 change the LFO waveform from one of 4 primitive waveforms (as described earlier).

Two bottom rows allow the user to trigger different BPM-synced LFO speeds based on main tempo divisions. These are none, a whole note, a quarter note, an eight note, a triplet, a sextolet and so on.

Because the main tempo is running constantly, the musician never gets off beat with his LFOs. One can visualize this as having 8 running LFOs in parallel and the user is only choosing through which the signal is routed. The user still may manually stop the playhead if there is a need – such as before a tempo change during live performance, the musician can pre-tap the tempo and start on the first beat.

3.2 Sidecut filter

The sidecut filter is a single parameter with no filtering at the top position (0.5 on scale from 0 to 1). The filter progressively acts as a low pass

filter for n<0.5 and as a high pass filter for n>0.5. The filter is 4 pole with a resonance parameter. This filter also modifies the main low frequency oscillator-controlled low pass filter by modifying its upper (or lower) limits.

3.3 Sounds and presets

You can store 12 presets in total. This limitation is not due to memory limits, but because of playability and easy preset recalling during a live performance. When pressing the 'preset' button, the 12 primary pads' function changes to preset selectors. Because there is an RGB LED under each pad, the presets can be identified by user-chosen color coding (set by a single encoder as HSV). Each of these presets include 5 *sounds.* The sound buttons are to the left of 12 primary pads and these select one of 5 available *synthesis setups*. All parameters are stored preset-wise, except for the sounds. To sum up – there are 12 presets, each having 5 sounds.

3.4 Playability

Segment's layout is designed in a way so it can be played with a single hand, leaving the other hand to other controllers or keyboards. With this approach, the musician can easily change low frequency oscillator speed, low frequency oscillator waveform and sound simultaneously for each note played.

3.5 Drift

To make the sounds more organic and alive, we implemented a feature we call *drift.* Modifying one parameter (which internally modifies several of them), the user varies the amount of 'instability' of the synthesis parameters. This is done using a recursive filter:

$$x = x + (T_x - x)/n$$

Where x is *current parameter value*, T_x is *target parameter value* and n is the *drift speed coefficient.* The *target parameter value* is randomly rolled at a given interval. All these and the maximum distance of the *target parameter value* from the user-set parameter value are modified by the *drift* parameter. This way the *current parameter value* randomly drifts around its user-set pivot point, creating slight or drastic timbre changes in time.

4 Hardware

The Segment core runs bare-metal on an ARM Cortex-M4 clocked at 168MHz. There are two more ARM Cortex-M0 chips, one for controlling the displays and encoders and one controlling the buttons and pads. The chips communicate via an SPI bus. The code utilises the libopencm3 peripheral library. The audio engine runs on 48kHz at 32-bit and the DAC operates on 48kHz at 24-bit.

4.1 No keyboard

Segment has no included keyboard to play it. There are several reasons for this decision – first, most electronic musicians already own (sometimes a lot of) keyboards and controllers. It seemed redundant to provide them with more as there is only a limited amount of devices you can put on your table. If the user doesn't have a keyboard, he or she can either get one or drive *Segment* from a computer, a sequencer etc. Second, we all like different keys – touch response, size, number of octaves and so forth and we don't believe in a 'one size fits all' philosophy regarding keyboards. Lastly, the absence of the keyboards allowed us to cut down the costs greatly.

4.2 Auxiliary board

The top auxiliary board consists of 8 rotary encoders and 8 monochromatic OLED displays below each of them. The displays show the current function of each encoder depending on the control layer the user is in and the current value of that parameter on a bar graph.

4.3 Control buttons

To the left of the 12 primary pads,

there are 3x4 buttons that either select the layer the user is in or have a single use (such as tap tempo). The three main layers are the segment synthesis layers, which select which segment's parameters are being modified by the encoders.

4.3.1 Linker

The linker button (a small chain symbol) triggers the linker layer, which allows the user to link parameters together or route them from and to any port

4.3.2 Scale

The scale button (with a resize symbol) modifies the amount of modulation applied to parameters via external sources.

4.4 The user button

The user button's function (marked with an exclamation mark) can be defined by the user. This button can have any function from any layer or one that is not currently assigned anywhere, such as the killswitch.

4.5 User layer

When no layer is selected on the control buttons, the *user layer* is selected. The user can pull any 8 parameters from any layers to the *user layer* so they are immediately accessible. The user layer is stored preset-wise, because with some presets one wants to have hands-on control over different parameters than with others. This comes in handy and tries to get closer to the 'one knob per function' approach of the analog synthesizers.

5 Connectivity

Regarding audio, *Segment* provides a stereo balanced XLR output and a stereo 6.3mm headphone jack.

Furthermore, *Segment* has 2 USB MIDI inputs, so the user can connect their USB MIDI controllers and keyboards as *Segment* acts as a USB host.

Segment also acts as a class-compliant USB MIDI device (in and out) and support OSC.

Due to some degree of backward-compatibility, the device provides the user with a MIDI input in the form of a 3.5mm jack.

It is important to note that every parameter on the device is exposed via a MIDI CC.

Because we like analog and some of us have some modular gear we like to work with, *Segment* has 4 control voltage inputs and 2 control voltage outputs. These can be mapped to any parameter and also linked-through (CV in – USB MIDI out) so *Segment* can act as a CV-USB-MIDI converter.

Segment also contains UART input and output exposed on a 3.5mm stereo jack. The user can connect their DIY gadgets and control *Segment* with anything they can come up with.

6 The computer software

A specialised software (written in Java for cross-platformity) can be downloaded. The users can manage their presets, name them, tag them, backup and create them. The software also provides means for flashing updated or different versions of the firmware and uses a custom protocol due to the computational limits of the onboard MCU. From within the application, the user is also able to reach the Segment Cloud.

6.1 The Segment Cloud

The *Segment Cloud* is a cloud-based service providing for a platform where the users can share, comment, tag and download the presets/sounds either to their local database or to *Segment* directly.

Since *Segment* has no USB Audio interface, short sound snippets of around 8 seconds will be rendered to audio files on the server based on user's MIDI input. The synthesis code is mostly portable, so the server will emulate exactly the same audio output *Segment* would produce.

7 Tools used during development

The programming was mostly done with Geany and Eclipse, compiling with GCC.

Java programming was done with NetBeans, PCB design was made possible by gEDA and we used FreeCAD for CAD parts designs.

8 Licencing

Since the team believes in open source and open hardware, the whole code, PCB designs and CAD drawings will be available under GPLv3 licence. We have chosen the GPLv3 licence because of its 'virality' which allows us to protect ourselves from big corporations, but not restricting the users to study, modify, use and contribute to the code.

None of *Segment*'s parts are patented and we do not consider doing that anytime in the future.

Acknowledgements

We would like to thank all who provided us with valuable advice, especially the #lad and #libopencm3 channel at FreeNode on IRC.

1 Saw, sine, square and reverse saw.
2 Stochastic Synthesis: Origins and Extensions, Sergio Luque, 2006
3 http://www.segmentsynth.com/synthesis/

Sound Synthesis with Periodic Linear Time-Varying Filters

Antonio GOULART
Marcelo QUEIROZ
Computer Science Department
Sonology Research Center
University of São Paulo
São Paulo - Brazil
{ag, mqz}@ime.usp.br

Joseph TIMONEY
Victor LAZZARINI
Sound and Digital Music Technology Group
National University of Ireland - Maynooth
Maynooth - Co. Kildare - Ireland
{joseph.timoney, victor.lazzarini}@nuim.ie

Abstract

In this survey we explore implementations of recently proposed distortion synthesis techniques, namely the Feedback Amplitude Modulation (1st and 2nd order cases) and Allpass filter coefficient modulation. These techniques are based on Periodic Linear Time-Varying systems, in which we operate by finding a suitable modulation function to obtain a desired spectrum. In order to illustrate this survey and encourage exploration of these new techniques we present examples in the Csound language.

Keywords

allpass filter coefficient modulation, feedback amplitude modulation, periodic linear time-varying systems, distortion synthesis

1 Introduction

Knowledge of a good set of synthesis techniques is a must for electronic / electroacoustic / computer musicians, sound designers and synthesiser builders. Audio effects plugins developers also benefit from dealing with synthesis techniques that can also be used in an audio effect context, such as the ones that will be presented.

In a previous survey, classic and recent distortion synthesis techniques along with their implementations were presented at the LAC [Lazzarini, 2009]. Waveshaping [LeBrun, 1979], phaseshaping [Ishibashi, 1987], and summation formulae [Moorer, 1976], some of the classic approaches to distortion synthesis, were analysed mathematically and demonstrated in Csound code. More recent or unusual approaches like asymmetrical FM [Palamin et al., 1988], Phase Aligned Formants [Puckette, 1995], and Modified FM synthesis [Lazzarini and Timoney, 2010] were also exposed. Other, more common distortion techniques, are also widely presented in computer music textbooks [Dodge and Jerse, 1997] [Moore, 1990] [Puckette, 2007].

In this paper, we address some recent techniques which extend the distortion synthesis family, namely techniques based on Periodic Linear Time-Varying (PLTV) systems. With these new approaches we operate by modulating the coefficients of a filter, and it was shown [Pekonen, 2008] that it results in a kind of dynamic version of phase distortion [Ishibashi, 1987]. A thorough study of time-varying systems applied to these techniques [Cherniakov, 2003] [Timoney et al., 2014] discloses the appropriate tools for properly understanding and using these new synthesis processes. These systems are intended to operate differently than audio effects such as time-varying delay lines used in flanging [Zolzer, 2011] and allpass filters used to obtain variable fractional delays [Pekonen et al., 2010]. Additionally, they do not replace another type of variation within musical systems where users manually change (e.g. with a knob) parameter values to achieve a sonic effect, such as mentioned in [Wishnick, 2014].

Our motivation for presenting this survey comes from the fact that most audio programmers' work is supported by classic Linear Time-Invariant (LTI) systems theory [Oppenheim and Schafer, 1975], but Linear Time Varying (LTV) systems theory is less well covered in the literature [Huang and Aggarwal, 1982]. Also, time-varying tools were indeed considered a long time ago [Layzer, 1971] [Risset, 1969] for audio applications within our context, but only recently are being tackled in a more comprehensive fashion. A good understanding of LTV theory can bring us new kinds of synthesis/effects techniques and different ways for implementing established ones (which can bring nice variations or reduce computational costs).

We will dedicate each of the next sections to one technique and then conclude the text. The aim of this paper is not to delve into the theory of LTV systems; our intention, instead, is to present another look at some techniques and also provide code[1] for their implementation.

[1] `http://www.ime.usp.br/~ag/dl/lac15-code.zip`

2 Allpass coefficient modulation

An allpass filter contains poles and zeros at reciprocal distances from the origin, so their effects on all frequencies are balanced [Moore, 1990]. A typical 1st order invariant allpass would have a fixed coefficient, but a time-varying one has the transfer function

$$H(z,t) = \frac{-m(t) + z^{-1}}{1 - m(t)z^{-1}}, \quad (1)$$

with $m(t)$ as the modulating function which drives the coefficient. Using this filter, a time-varying phase distortion given by [Timoney et al., 2009] [Laakso et al., 1996]

$$\phi(\omega,t) = -\omega + 2\tan^{-1}\left(\frac{-m(t)\sin(\omega)}{1 - m(t)\cos(\omega)}\right) \quad (2)$$

is introduced in the signal. If we know how we want the phase to be distorted – in other words, if we know $\phi(\omega,t)$ – we can use the approximation $\tan(x) \approx x$ [Timoney et al., 2009] and determine the modulating signal as

$$m(t) = \frac{-(\phi(\omega,t) + \omega)}{2\sin(\omega) - (\phi(\omega,t) + \omega)\cos(\omega)}. \quad (3)$$

The allpass filtering described can be implemented with the difference equation [Lazzarini et al., 2009b]

$$y(n) = x(n-1) - m(n)(x(n) - y(n-1)). \quad (4)$$

The condition $|m(n)| < 1, \forall n$, assures stability [Cherniakov, 2003] and a DC offset

$$DC(n) = \frac{1 - m(n)}{1 + m(n)} \quad (5)$$

is introduced in the signal [Pekonen, 2008].

2.1 Classic phase distortion emulation

In [Pekonen, 2008] [Lazzarini et al., 2009b] [Timoney et al., 2014], we find as an example for this technique the emulation of the classic phase distortion [Ishibashi, 1987]. The phase distortion algorithm consists of reading a cosine table in an unusual way. Instead of getting indexes from a regular phase generator (which goes from 0 to 1 in a period related to the chosen fundamental frequency) multiplied by the table size, we add another function to the phase generator values and then read the cosine wavetable. In order to get an approximation of a sawtooth from a cosine, we must read the rising part of the cosine in a shorter time, and take more time to read the decaying portion. The function we must add to the phase generator, in this case, is drawn on the upper panel of Figure 1 and is given by [Lazzarini et al., 2009b]

$$g(x) = \begin{cases} (\frac{1}{2} - d)\frac{x}{d}, & x < d \\ (\frac{1}{2} - d)\frac{1-x}{1-d}, & x \geq d, \end{cases} \quad (6)$$

where d is how long it takes to read from the start of the cosine table up to its maximum, and the smaller it is the more abrupt the rising ramp will be and the more distortion we will obtain.

If we want to implement $g(x)$ using a modulated allpass filter we will get better results if $\phi(\omega,t)$ goes from $-\omega$ to $-\pi$ [Lazzarini et al., 2009b], so we make

$$\phi(\omega,t) = \frac{g(t)((1-2d)\pi - \omega)}{(1-2d)\pi} - (1-2d)\pi - \omega. \quad (7)$$

The resulting modulation function is shown on the lower panel of Figure 1. On Figures 2 and 3 we can see the waveforms and spectra of the sawtooths generated with both the original technique and the modulated allpass based one. Notice that the missing components of the classic phase distortion technique are actually present in the spectrum of the modulated allpass output. We can also see and hear that the latter spectrum is richer. The code for implementing this example is given in Listing 1, where we can see two instruments; the first one works as a synthesis instrument, distorting the sinusoid and producing richer spectra; the second one applies the same technique as an audio effect, distorting the sound of a pre-recorded flute (any sound file can be used, or even a microphone input).

Listing 1: Classic phase distortion emulation as synthesis (instrument 1) and technique used as an audio effect (instrument 2)

```
<CsoundSynthesizer>

<CsOptions>
-o dac
</CsOptions>

<CsInstruments>

0dbfs=1

```

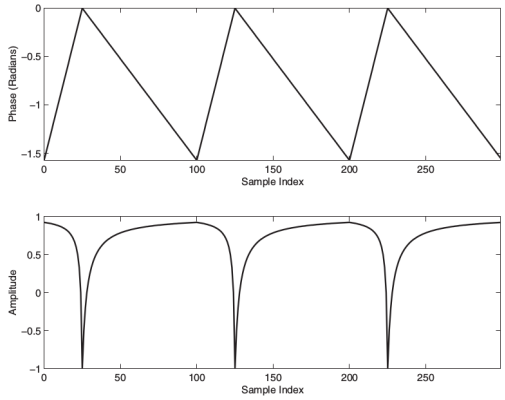


Figure 1: Phase distortion function (upper panel) and coefficient modulation function (lower panel). Source: [Timoney et al., 2014]

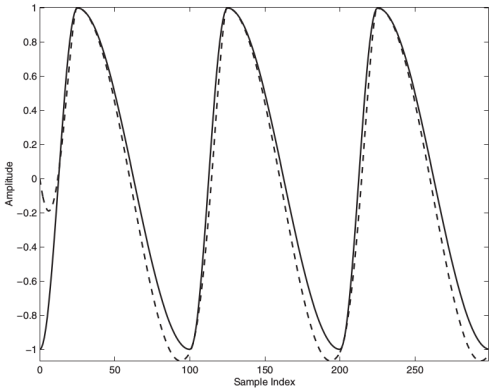


Figure 2: Waveforms generated with the classic technique (solid line) and the modulated allpass (dashed line). Source: [Timoney et al., 2014]

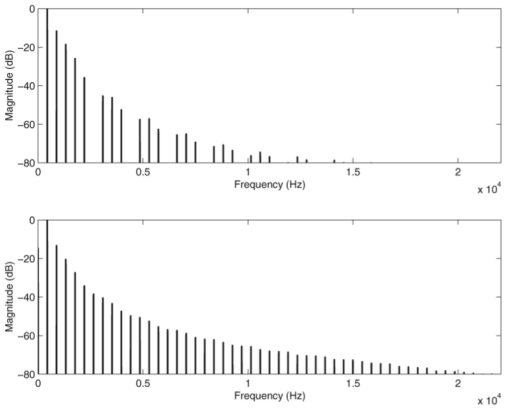


Figure 3: Spectra obtained with the classic technique (upper panel) and with modulated allpass (lower panel). Source: [Timoney et al., 2014]

```
11 /* audio-rate coeff allpass */
12 opcode Allpass,a,aa
13   adel init 0
14   setksmps 1
15   audio,acoef xin
16   aw = audio + acoef*adel
17   ay = -aw*acoef + adel
18   adel = aw
19   xout ay
20 endop
21
22
23 /* PD function */
24 /* inflection point */
25 gip = 0.1
26
27 /* ftgen producing line segments  */
28 ipdfun ftgen 1,0,16384,7,0,16384*gip
       ,1,16384*(1-gip),0
29
30 /*
31  instr 1:
32  PD synthesis
33  using sine wave input
34 */
35
36 instr 1
37   ifr = p5
38   iamp = p4
39
40   /* regular phase generator */
41   aph phasor ifr
42   /* phase distortion signal */
43   apd tablei aph, 1, 1, 0, 1
44
45   /* scaling of pd signal (eq.7)*/
46   iw = 2*$M_PI*ifr/sr /*omega value*/
47   ia = (1 - 2*gip)*$M_PI
48   apd = apd*ia
49   apd = apd*(ia - iw)/ia - ia - iw
50
51   /* coefficient modulation function
52      obtained from phase distortion
53      signal (equation 3) */
54   /* envelope for modulation */
55   kmod linseg 0,1,1,p3-2,1,1,0
56   amod = -kmod*(apd + iw)/(2*sin(iw)
       - (apd+iw)*cos(iw))
57
58   /* sine input */
59   asin tablei aph,-1,1,0,1
60
61   /* allpass */
62   asig Allpass asin,amod
63
64   /* envelope */
65   aout linenr asig * iamp, 0.01, 0.1,
         0.01
66
67   outs aout, aout
68 endin
69
70 /*
71  instr 2:
72  PD adaptive synthesis
73  using a monophonic instr input
```

```
74 */
75
76 instr 2
77   iamp = p4
78   /* input signal */
79   afl diskin2 "flutec3.wav",1,0,1
80
81   /* pitch tracking */
82   kfr,kamp ptrack afl,2048
83   kfr port kfr,0.01
84
85   /* master phase signal */
86   aph phasor kfr
87   /* phase distortion signal */
88   apd tablei aph,1,1,0,1
89
90   /* scaling of pd signal */
91   kw = 2*$M_PI*kfr/sr
92   ia = (1 - 2*gip)*$M_PI
93
94   apd = apd*ia
95   apd = apd*(ia - kw)/ia - ia - kw
96
97   /* envelope for modulation */
98   kmod linseg 0,1,1,p3-2,1,1,0
99
100  amod = -kmod*(apd + kw)/(2*sin(kw)
         - (apd+kw)*cos(kw))
101
102  asig Allpass afl,amod
103  aout linenr asig * iamp, 0.01, 0.1,
         0.01
104
105  outs aout, aout
106 endin
107
108 </CsInstruments>
109
110
111 <CsScore>
112
113 /* uncomment lines to
114    run instruments */
115 i 1 0 10 0.5 440
116 ;i 2 0 10 0.5
117
118
119 </CsScore>
120
121
122 </CsoundSynthesizer>
```

2.2 Finding a distortion function

Now we present a new example with the allpass coefficient modulation technique to bring more insight. In this example, instead of choosing a desired waveform and then finding out how to modulate the allpass coefficient, we choose an arbitrary signal to distort the phase. Keeping in mind that we should generate a signal within the appropriate range, that is [-1,1], any signal a priori can be considered for the process.

So we embarked on an experiment that involved the summing of partials in order to find a function to distort the phase of an input sinusoid and thus get a more musically interesting timbre. The expression we used to create our example is given by

$$y(n) = 0.4\cos(f_0) + 0.4\cos\left(2f_0 - \frac{\pi}{3}\right) + 0.35\cos\left(3f_0 + \frac{\pi}{7}\right) + 0.3\cos\left(4f_0 + \frac{4\pi}{3}\right) \quad (8)$$

In order to shift the output of Equation 8 to the appropriate range we use the scaling

$$y_s(n) = -\frac{\pi}{2}\frac{(y(n)+1)}{2}. \quad (9)$$

Listing 2 presents Csound code for implementing this new example. The upper panel of Figure 4 shows the distortion function for this example, while the lower panel shows the allpass modulation function resulting after applying Equation 3. Figure 5 shows the waveform and spectrum obtained. Albeit this was a naive example, it demonstrates in an intuitive way how a phase function could be easily generated using additive synthesis and then used to drive the allpass filter coefficient after applying Equations 9 and then 3, highlighting another musical possibility for using the allpass filter.

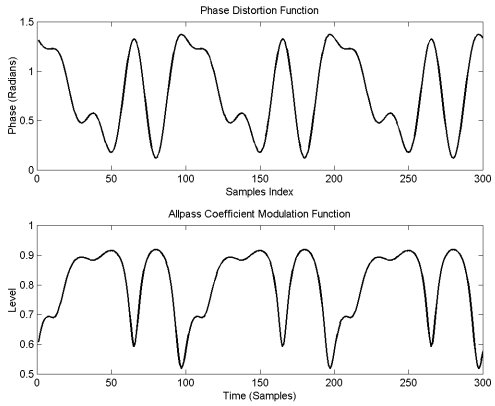


Figure 4: Phase distortion (upper panel) and resultant modulation (lower panel) functions.

Listing 2: Implementation of an arbitrary function as a phase distorter.

```
1 <CsoundSynthesizer>
2
3 <CsOptions>
```

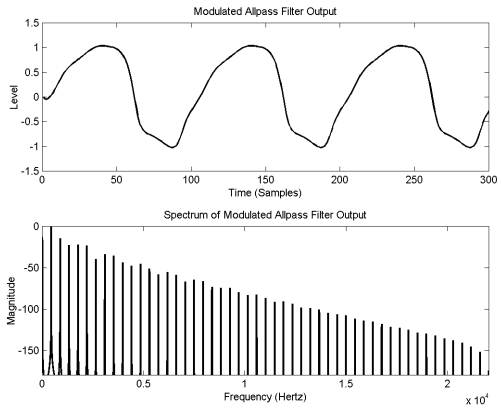


Figure 5: Waveform and spectrum obtained with arbitrary function

```
 4 -o dac
 5 </CsOptions>
 6
 7 <CsInstruments>
 8 0dbfs=1
 9
10 /* audio-rate coeff allpass */
11 opcode Allpass,a,aa
12   adel init 0
13   setksmps 1
14   audio,acoef xin
15   aw = audio + acoef*adel
16   ay = -aw*acoef + adel
17   adel = aw
18   xout ay
19 endop
20
21 /* PD function */
22 ipdfun ftgen 1, 0, 16384, 9, 1, 0.4,
       -90, 2, 0.4, -150, 3, 0.35,
       180/7-90, 4, 0.3, 4*180/3-90
23
24 instr 1
25
26   ifr = p5
27   iamp = p4
28
29   iw = 2*$M_PI*ifr/sr ;omega
30
31   aph phasor ifr  ;regular phase
32   /* phase distortion signal */
33   apd tablei aph, 1, 1, 0, 1
34
35   /* coef mod signal (eq.3)*/
36   apd = -0.5*$M_PI*(apd+1)/2 ;scaling
37   amod = -1*(apd + iw)/(2*sin(iw) - (
         apd+iw)*cos(iw))
38
39   /* sine input */
40   asin tablei aph,-1,1,0,1
41   /* allpass */
42   asig Allpass asin,amod
43   /* envelope */
44   aout linenr asig*iamp,0.01,0.1,0.01
45
46   outs aout, aout
47 endin
48
49 </CsInstruments>
50
51 <CsScore>
52 i 1 0 10 0.25 440
53 </CsScore>
54
55 </CsoundSynthesizer>
```

3 Feedback amplitude modulation

The Feedback Amplitude Modulation was mentioned by Layzer (1971) and described and implemented by Risset (1969) in his catalogue example #510, but a rigorous mathematical analysis of the technique was lacking. Since 2009 there was a renewed research interest in exploiting its musical possibilities [Lazzarini et al., 2009a], [Kleimola et al., 2011], [Lazzarini et al., 2011].

First of all we will review its 1st order case. The basic idea is to modulate the amplitude of an oscillator using its previous output, as in

$$y(n) = \cos(\omega_0 n)[1 + y(n-1)], \qquad (10)$$

with $\omega_0 = 2\pi f_0$ and the initial condition $y(n) = 0$ for $n \leq 0$.

A first analysis [Kleimola et al., 2011] is made expanding Equation 10 as

$$\begin{aligned} y(n) = \cos(\omega_0 n) + \\ \cos(\omega_0 n)\cos(\omega_0[n-1]) + \\ \cos(\omega_0 n)\cos(\omega_0[n-1])\cos(\omega_0[n-2]) + \\ \ldots \end{aligned} \qquad (11)$$

$$y(n) = \sum_{k=0}^{\infty} \prod_{m=0}^{k} \cos(\omega_0[n-m]), \qquad (12)$$

showing that the resultant signal is composed by harmonics of the fundamental f_0. A more interesting case is when we can control the amount of feedback in the system, so we introduce the feedback parameter β and Equation 10 becomes

$$y(n) = \cos(\omega_0 n)[1 + \beta y(n-1)]. \qquad (13)$$

The feedback parameter can be interpreted to be similar to modulation index of conventional FM synthesis, and its influence on FBAM's

spectral evolution is shown in Figure 6, borrowed from [Kleimola et al., 2011].

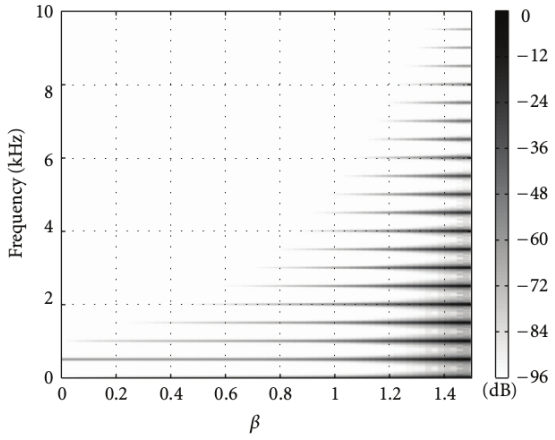


Figure 6: β influence on FBAM spectrum. Source: [Kleimola et al., 2011]

The interpretation of the system as a time-varying filter enables a better analysis. The structure for the 1st order case is

$$y(n) = x(n) + a(n)y(n-1), \tag{14}$$

with

$$x(n) = a(n) = \cos(\omega_0 n), \tag{15}$$

so we have a coefficient modulated IIR filter. As in the previous section, the modulation will create the dynamic phase distortion, generating new partials, and in this case the filter is not allpass, so it has a time-varying non-flat magnitude response.

Applying the equations for stability analysis by Cherniakov (2003) a condition for stability of this system was derived as

$$\left| \beta \prod_{m=1}^{N} \cos(\omega_0 m) \right| < 1. \tag{16}$$

An expression for the maximum value of beta was proposed by Kleimola et al. (2011) as

$$\beta_{max} \approx 1.9986 - 0.00003532(f_0 - 27.5), \tag{17}$$

and they also showed that as the value of β is increased, the presence of significant components in the output due to aliasing can be observed before the system becomes unstable. Listing 3 presents code for the 1st order FBAM implementation.

Listing 3: 1st order FBAM

```
<CsoundSynthesizer>

<CsOptions>
-o dac
</CsOptions>

<CsInstruments>

0dbfs= 1

opcode FBAM, a, kkki
  ; set vector size to 1 sample
  setksmps 1
  ay init 0 ; y[0] = 0
  ka, kf, kb, ifn   xin
  /* current sample +
  weighted previous sample */
  ay oscili ka + kb*ay, kf, ifn
  xout ay
endop

instr 1
        ; amp, freq, beta
  a1 FBAM 0.5, 440, 0.7, 1
  out a1
endin

</CsInstruments>

<CsScore>
f1 0 16384 10 1
i1 0 5
</CsScore>

</CsoundSynthesizer>
```

The 2nd order FBAM is obtained using two previous outputs in the modulation, each with its own feedback parameter. The system equation is then given by

$$y(n) = \cos(\omega_0 n)[1 + \beta_1 y(n-1) + \beta_2 y(n-2)]. \tag{18}$$

It was shown [Lazzarini et al., 2011] that with this system we can get a narrower pulse, and thus a richer spectral output for the FBAM system, as shown in Figure 7, borrowed from [Lazzarini et al., 2011]. Code for implementing the 2nd order FBAM is presented in Listing 4.

Listing 4: 2nd order FBAM

```
<CsoundSynthesizer>

<CsOptions>
-o dac
</CsOptions>

<CsInstruments>
ksmps = 10
0dbfs = 1
nchnls = 2
```

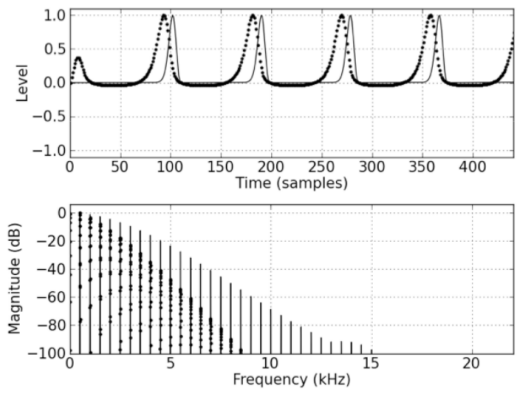


Figure 7: Comparison between 1st (in dots) and 2nd order (solid line) FBAM for 500 Hz fundamental. Source: [Lazzarini et al., 2011]

```
11
12 opcode Fbam2,a,akk
13   setksmps 1
14   asigm1 init 0
15   asigm2 init 0
16   ain,kb1,kb2 xin
17   asig =  ain*(1 + kb1*asigm1 + kb2*
         asigm2)
18   asigm2 = asigm1
19   asigm1 = asig
20   xout asig
21 endop
22
23 instr 1
24   kb1 = 0.7
25   kb2 = 0.7
26   ; sinusoidal input
27   ain oscili 1, 440, -1, 0.25
28   asig Fbam2 ain,kb1,kb2
29   asig balance asig, ain
30   outs asig,asig
31 endin
32
33 </CsInstruments>
34
35 <CsScore>
36 i 1 0 5
37 </CsScore>
38
39 </CsoundSynthesizer>
```

For a deeper analysis we can understand the system as a 2nd order PLTV system by rewriting the system equation as

$$y(n) = x(n)+\beta_1 a_1(n)y(n-1)+\beta_2 a_2(n)y(n-2), \quad (19)$$

with $x(n) = a_1(n) = a_2(n) = \cos(\omega_0 n)$. This is the simplest example but the analysis equations are complicated for this second order system. However we can also uncouple the input and modulation signals as independent streams, and treat the whole system as the combination of two first order units to reduce the difficulty of the analysis.

Despite the initial results about the 2nd order FBAM that were already reported, we plan to proceed with more thorough investigations especially regarding its stability and to determine any consequent restrictions it might have on the coefficient modulation waveform.

4 Conclusions

In this survey we presented some techniques for sound synthesis derived from the concept of phase distortion. Results can be used to generate approximations of classic sawtooth oscillators and more timbrally-involved FM-like spectra. Implementations in Csound, which are easily translated to other languages, were presented in order to promote the techniques among our community and audio tools developers.

The application of time-varying systems is not new in general signal processing, but only recently is the theory behind these systems being explored for time-varying digital audio filter systems. This paper shows there is a significant potential for a number of novel techniques based on PLTV systems. The field is still open to investigation, in particular with regards to 2nd and higher order systems.

We hope that this brief survey invites more musicians and technicians to explore the nice retro motivated sounds obtained with these systems, maintaining the interest in distortion synthesis/effects techniques. We hope to encourage the sharing of ideas and principles around these techniques, such as, for instance, nice distortion functions that could be used either in synthesis or acoustic instruments processing.

5 Acknowledgements

The research leading to this paper was partially supported by CAPES (proc. number 8868-14-0) and Science Foundation Ireland (ISCA-Brazil).

References

Mickhail Cherniakov. 2003. *An introduction to parametric digital filters and oscillators.* John Wiley & Sons Ltd, England.

Charles Dodge and Thomas Jerse. 1997. *Computer Music: Synthesis, composition and performance.* Schirmer Books, New York, NY, USA, segunda edition.

N.C. Huang and J.K. Aggarwal. 1982. Time varying digital processing: a review. In *Proceedings IEEE Int. Symp. Cas.*, pages 659–662, Rome, Italy.

Masanori Ishibashi. 1987. Electronic musical instrument (patent us 4658691), April.

Jari Kleimola, Victor Lazzarini, Vesa Valimaki, and Joseph Timoney. 2011. Feedback amplitude modulation synthesis. *EURASIP Journal on Advances in Signal Processing*, 2011(434378).

T. I. Laakso, V. Valimaki, M. Karjalainen, and U. Laine. 1996. Splitting the unit delay - tools for fractional delay filter design. *IEEE Signal Processing Magazine*, 13(1):30–60.

A. Layzer. 1971. Some idiosyncratic aspects of computer synthesized sound. In *Proceedings of the Annual Conference American Society of University Composers*, pages 27–39.

Victor Lazzarini and Joseph Timoney. 2010. Theory and practice of modified frequency modulation synthesis. *Journal of the Audio Engineering Society*, 58(6):459–471.

Victor Lazzarini, Joseph Timoney, Jari Kleimola, and Vesa Valimaki. 2009a. Five variations on a feedback theme. In *Proceedings of the International Conference on Digital Audio Effects (DAFx-09)*, Como, Italy.

Victor Lazzarini, Joseph Timoney, Jussi Pekonem, and Vesa Valimaki. 2009b. Adaptive phase distortion synthesis. In *Proceedings of the International Conference of Digital Audio Effects*, pages 28–35.

Victor Lazzarini, Jari Kleimola, Joseph Timoney, and Vesa Valimaki. 2011. Aspects of second-order feedback am synthesis. In *Proceedings of the International Computer Music Conference*, University of Huddersfield, UK.

Victor Lazzarini. 2009. A distortion synthesis tutorial. In *Proceedings of the Linux Audio Conference*, pages 12–20.

Marc LeBrun. 1979. Digital waveshaping synthesis. *Journal of the Audio Engineering Society*, 27(4):250–266.

F. Richard Moore. 1990. *Elements of computer music.* Prentice Hall, Englewood Cliffs, New Jersey, USA.

James A. Moorer. 1976. The synthesis of complex audio spectra by means of discrete summation formulas. *Journal of the Audio Engineering Society*, 24(9):717–727.

A.V. Oppenheim and R.W. Schafer. 1975. *Digital signal processing.* Prentice Hall, New Jersey, USA.

Jean-Pierre Palamin, Philippe Palamin, and André Ronveaux. 1988. A method of generating and controlling musical asymmetrical spectra. *Journal of the Audio Engineering Society*, 36(9):671–685.

J. Pekonen, V. Valimaki, J. Nam, J.O. Smith, and J.S. Abel. 2010. Variable fractional delay filters in bandlimited oscillator algorithms for music synthesis. In *International Conference on Green Circuits and Systems (ICGCS)*, pages 148–153, June.

J. Pekonen. 2008. Coefficient modulated first-order allpass filter as a distortion effect. In *Proceedings of the International Conference on Digital Audio Effects (DAFx-08)*, pages 83–87, Espoo, Finland.

Miller Puckette. 1995. Formant-based audio synthesis using non-linear distortion. *Journal of the Audio Engineering Society*, 43(1):40–47.

Miller Puckette. 2007. *The theory and technique of electronic music.* World Scientific Press, River Edge, NJ.

Jean Claude Risset. 1969. *An introductory catalogue of computer synthesized sounds.* Bell Laboratories, Murray Hill, New Jersey.

J. Timoney, V. Lazzarini, J. Pekonen, and V. Valimaki. 2009. Spectrally rich phase distortion sound synthesis using an allpass filter. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICCASP-09)*, pages 293–296, Taipei, Taiwan.

Joseph Timoney, Jussi Pekonen, Victor Lazzarini, and Vesa Valimaki. 2014. Dynamic signal phase distortion using coefficient-modulated allpass filters. *Journal of the Audio Engineering Society*, 62(9).

Aaron Wishnick. 2014. Time-varying filters for musical applications. In *Proceedings of the 17th International Conference on Digital Audio Effects (DAFx-14)*, pages 69–76, Erlangen, Germany.

Udo Zolzer, editor. 2011. *DAFX: Digital Audio Effects.* John Wiley & Sons, UK.

Timing issues in desktop audio playback infrastructure

Alexander Patrakov
SkyDNS LLC
Office 500, House 2, Kulibina Street,
620137 Yekaterinburg,
Russia
patrakov@gmail.com

Abstract

In year 2008, a feature with the name "timer-based scheduling" (also known as "glitch-free") has been introduced into PulseAudio in order to solve the conflicting requirements of low latency for VoIP applications and low amount of CPU time wasted for handling interrupts while playing music. The novel (at that time) idea was to use timer interrupts instead of sound card interrupts in order to overcome the limitation that the ALSA period size cannot be reconfigured dynamically. This idea turned out to hit some corner cases, and workarounds had to be added to PulseAudio. Despite its age, the implementation of the idea is still not 100% correct. This paper explains why it is the case and what can be done to improve the situation.

Keywords

PulseAudio, timing, rewinds

1 Introduction

Traditionally, ALSA playback chain was based on ALSA plugins sitting on top of the hardware device (represented by the "hw" plugin).

Hardware devices have circular buffers in memory, and there are two pointers that point to this buffer: the hardware pointer and the application pointer. The hardware periodically reads a sample pointed to by the hardware pointer and sends it to analog or digital outputs, then increments this pointer. The application writes samples to the memory area pointed to by the application pointer and moves it past the just-written samples.

There are multiple mechanisms provided to the application to write sound data to the sound card: classical unix-style writes via `snd_pcm_write*()` functions, mmap-based access, and the dangerous callback-based API. Still, with any of them, the end result is the same: application pointer tracks the first unwritten position in the soundcard buffer.

If the hardware pointer crosses the application pointer, an underrun happens. To avoid underruns, an application must supply new audio samples in a timely manner.

The hardware notifies the kernel when the hardware pointer crosses some predefined positions (period boundaries) in the circular buffer. There are, again, multiple mechanisms (blocking writes, `poll()`) how these notifications can be passed to the application, so that it doesn't have to busy-wait.

A whole lot of other behavior (format-conversion, resampling, mixing) is provided on top of the raw hardware devices by means of ALSA plugins [1]. The general idea (a circular buffer with hardware and application pointers and per-period wakeups), however, remains. As a result, applications can transparently use a large subset of ALSA API when working with such plugins.

This playback model was popular in the dmix era, and thus applications developed during that time gained dependency on some of the properties of this model. E.g., an assumption is common that wakeups due to the audio device happen regularly (exactly once per period) and can be used as a clock. Another common assumption is that the default buffer and period sizes are suitable for the application's purpose. In fact, there was no way to change them programmatically in the default "plug:dmix" setup.

There is certain latency (influenced by the audio buffer size) between the time when an audio sample is written to the API and when it is actually played back through the speakers. Low latencies are generally expected when an application reacts to user input. E.g., when a user changes equalizer settings in the audio player, they should take effect immediately. This is even more important for games: a gunshot sound should be heard as soon as the shot is made. Voice over IP applications are also sensitive to latency. So, with the traditional playback model, due to the fact that the latency is fixed, low latencies are generally used.

On the other hand, low latencies are *not* optimal for music players. First, low latencies make applications sensitive to process scheduler decisions, increasing the chance of audio dropouts. Second, low latency means high rate of interrupts from the sound card and application wakeups, which is bad for power saving. So, music players and games are under two conflicting requirements related to latency.

2 Timer-based scheduling

The traditional solution was to accept frequent (as required for the worst case) wakeups as a necessary evil, because there is no way to reconfigure buffer and period size on the fly. However, in some cases, a

better (albeit more complex) solution exists to this conflict of requirements. The solution ("timer-based scheduling" [2], implemented in PulseAudio [3] and CRAS [4]), involves the use of a dynamically reconfigurable timer instead of sound card interrupts as a source of wakeups.

PulseAudio has a client-server architecture. The server interacts with ALSA devices and performs mixing and routing of sound data received from client applications. Each time the timer fires, the sound card is asked about its current playback position, and, based on this information a decision is made how much data to request from applications in order to maintain their desired latency and to avoid underruns. Note that, if there are low-latency applications playing, the buffer will never be full.

A new stream can appear at any time, or an application can request volume change of an existing stream. The server is expected to deal with such requests quickly, i.e. without waiting for the already-buffered sounds to play out. Indeed, waiting for these sounds could take more than a second, which is too much. Thus, the server has to discard already-mixed samples from the sound card buffer and replace them with a new version, which takes the new stream or the volume change into account. Such operation is called a "rewind". It is an essential ingredient in an implementation of a dynamic-latency sound server, unless a tight limit is placed on the amount of buffered audio data. PulseAudio rewinds. CRAS doesn't, but Chrome/Chromium never requests latency high enough to cause a problem.

As the timer-based approach is more complex than the traditional approach, there are more questions to be answered by the implementation (such as PulseAudio) and parameters to be decided upon.

- Total sound card buffer size.
- The amount of time to sleep after writing sound data.
- The amount of old data to leave in the buffer "just in case" when rewinding.
- What latency limits to export to clients.
- How much data to ask from a client at a time.

3 Buffer and timing constraints

PulseAudio uses a large buffer (up to 2 seconds, if the hardware allows) by default. This is good for the purpose of providing high latencies for music players and thus for reducing the rate of CPU wakeups. The default can be overridden with the `tsched_buffer_size` parameter that is accepted by `module_udev_detect`, `module_alsa_card`, `module_alsa_sink` and `module_alsa_source`. The unit of the `tsched_buffer_size` parameter is microseconds.

This default, however, poses a problem[1] if any part of the audio processing pipeline inside the PulseAudio process turns out to be CPU-intensive. Examples of CPU-intensive steps include conversion to a compressed format such as DTS. When PulseAudio is running under Valgrind, or on a weak embedded CPU, even resampling becomes a problem.

The problem is related to the fact that PulseAudio only has a finite budget of time it can run with real-time priority without making blocking system calls. `rtkit` contains a hard-coded limit that doesn't allow expanding this budget past 200 ms. This limit exists for safety reasons, because a misbehaving real-time application can otherwise wedge the whole system. Thus, in the worst case (which always happens at the start of a high-latency stream) PulseAudio has to finish its processing of two seconds of audio in 200 ms, or it gets killed.

The situation is further aggravated by the fact that the `cpufreq` subsystem considers "low" (i.e. less than 80%) load as an excuse to keep the CPU frequency at the lowest possible value.

A solution that a user affected by the problem can apply is to set the buffer size to a lower value, such as 200 ms.

4 Wakeup timings

In the traditional timing model, the application usually is woken up once per period. The period size comes from application settings or from the defaults. There is not much that can be done beyond that (e.g. in response to underruns), because buffer and period sizes are not dynamically reconfigurable.

With timer-based scheduling, better reaction to underruns is possible, and PulseAudio implements that. It looks at the sink's latency (which is just the amount of time until it underruns unless supplied with new data), subtracts the scheduling watermark, and sleeps for that time. The default watermark is 20 ms. It is increased if an underrun or a near-underrun happens, and decreased if sufficient time has passed without such bad events[2].

This logic is further complicated by the fact that the requested latency is specified in the sound card's clock domain, while sleeping is done using the system clock domain. If the sample rate reported by the card is not precise, then these two values can differ. PulseAudio contains a "smoother"[3] that takes timestamps in both clock domains, estimates the actual sample rate, and then converts the intervals as needed.

[1] `https://plus.google.com/+ColinGuthrie/posts/EG7nT9TXTpd`

[2] See `http://cgit.freedesktop.org/pulseaudio/pulseaudio/tree/src/modules/alsa/alsa-sink.c`, functions `check_left_to_play()`, `decrease_watermark()` and `increase_watermark()`

[3] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/tree/src/pulsecore/time-smoother.c`

The traditional solution for mapping between soundcard and system clock domains would be using a delay-locked loop with a filter containing some integrators in it [5]. This solution works for JACK, but cannot be employed in PulseAudio, because the timestamp reports are assumed to be regular in time, which is valid only if the traditional period-based timing scheme is used. Therefore, the smoother in PulseAudio uses a 10-second window and builds a least-squares linear approximation between the sample count and the wall-clock timestamp based on data within that window.

A special rule (that cuts the sleeping time in half) is applied until one buffer worth of sound data is played. This is needed because some sound cards contain a hardware FIFO queue that consumes the initial portion of data much faster than one would expect according to the size of that portion and the sample rate.

All of the above assumes that the sound card can accurately report its hardware pointer at arbitrary point in time. However, this assumption is false on cards that do double-buffering of audio data transfers. Such cards can be distinguished using the `snd_pcm_hw_params_is_batch()` ALSA API function. Typically, such batch cards provide position reports that are accurate to only one period, and timer-based scheduling makes a period as large as possible to avoid useless CPU wakeups from the interrupts originating from the sound card. Since position reports are totally inaccurate, one just cannot obtain an estimation of time-to-sleep accurate up to 20 ms.

Currently, PulseAudio disables timer-based scheduling on batch cards[4], because it cannot save the CPU from unneeded wakeups.

CRAS does not have this watermark-based logic and does not use the mapping between soundcard and system clock domains for the purpose of wakeup timing. Indeed, CRAS doesn't have to do so, because it respects the client's idea how many frames should remain in the soundcard buffer when asking for more data, instead of asking as late as possible, and thus stays far from any edge cases.

5 Rewinds

As already explained, rewinds are needed in order to provide low-latency reaction to unpredictable events such as new streams and volume changes, while keeping the average latency high in order to save power. Rewind handling is an especially problematic area, with a lot of code written but never properly tested.

5.1 Rewind-related APIs

Both ALSA and PulseAudio offer APIs that let applications rewind their audio streams.

As already mentioned, ALSA's view on playback devices is based on the notion of a circular buffer in memory, with the hardware pointer and the application pointer associated with it. Here are the rewind-related API functions offered by ALSA: `snd_pcm_rewindable()`, `snd_pcm_rewind()`.

The `snd_pcm_rewind()` function tries to move the application pointer backwards by the specified number of samples, and returns the (possibly lower) number of samples that the pointer has actually been moved by. Of course, attempting to request rewinding into the already-played portion of the buffer does not make sense. The `snd_pcm_rewindable()` function returns the maximum safe amount of rewindable samples[5].

PulseAudio does not base its playback model on a mmap-able circular buffer. Instead, it has one stream-oriented function that clients use to submit samples: `pa_stream_write()`.

Seeking is done at the same time as writing new samples, using the last two arguments. Unlike ALSA, which supports only rewinds relative to the application pointer ("write index", as PulseAudio calls it), PulseAudio API can also be used to rewind to an absolute position in time, or relatively to the "read index" (the sample that is currently being played). The raw read index and write index can be obtained in the `pa_timing_info` structure via the `pa_stream_update_timing_info()` and `pa_stream_get_timing_info()` pair of functions.

OSS does not support rewinds in ways other than the (deprecated) mmap interface, which only works on top of raw hardware devices. I.e. no resampling, no channel remixing, only exclusive access to the sound card.

JACK, SDL, libao and the `waveOut` family of Windows APIs do not support rewinds at all. Android's AudioTrack API and CRAS don't support them, either.

5.2 Testing rewinds

Rewind operations can be used by software only if they actually work as described. E.g., a perfect implementation of rewinds needs to ensure that, after rewinding over some samples and writing exactly the same samples back, the audible result is exactly the same as if the rewind didn't happen at all.

Currently, for ALSA, the most common application that does a lot of rewinds is PulseAudio, and it does that only in response to dynamic events such as new stream appearing or volume changing, where a user already more-or-less expects a glitch and thus may not realize that something is wrong. So, in order to really ensure that rewinds work, a more systematic testing methodology is needed.

A simple ALSA-based program[6] has been thus written that exercises the `snd_pcm_rewind()` func-

[4] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/commit/?id=826c8f69d34ef49e86fe0ab6c93c1ffba8916131`

[5] There are disagreements on the intended meaning of the word "safe".

[6] `http://permalink.gmane.org/gmane.linux.alsa.devel/122179`

tion in such a way that is impossible to confuse the correct operation and a glitch. The program uses a buffer with four periods. After the initial filling of the buffer with silence, the application uses blocking writes, as follows. Each time it gets a chance to write a period worth of samples, it rewinds one period, writes one period of silence and one period of square waves. Therefore, if rewinds are implemented correctly, the hardware pointer only sees silence, and nothing should be heard from this application. Any non-silent output (without the application complaining that the rewind did not yield the expected result, and without near-underruns) is an indication of a bug somewhere.

Hardware ALSA devices pass this simple test. Many other devices currently don't.

It would also be nice, for similar reasons, to test the correct operation of the `snd_pcm_rewindable()` function. However, no valid test can be devised at this time, because there are disagreements about the semantics of the return value. As this function only recently stopped crashing on some plugins[7], PulseAudio does not use it, and uses an approximation based on `snd_pcm_avail_delay()` instead.

5.3 Causes of incomplete rewindability in hardware

Currently, for the "hw" plugin the `.rewindable` callback is implemented, effectively, as a difference between the application pointer and the hardware pointer. That is, "you can rewind up to the hardware pointer". However, the hardware pointer information is only updated either on period boundaries, or on explicit request (via `snd_pcm_avail()`) from the application. Failure to perform such request would lead to alsa-lib basing its calculations on an outdated value of the hardware pointer, and, thus, to the overestimated results for `snd_pcm_rewindable()`.

There is a disagreement whether the `snd_pcm_rewindable()` function should indeed return the difference between the application pointer and the hardware pointer. PulseAudio contains a safeguard that does not allow the rewound application pointer to come too close to the hardware pointer, because[8]

> *some DMA controllers go nuts (such as breaking the stream, causing interrupt storms, or something else seriously buggy) when trying to write to data that the DMA controller is just about to transfer.*

The default safeguard is the largest of 256 bytes or 1.33 ms.

On some cards, the hardware pointer position is not known exactly. For example, `ymfpci` updates its hardware pointer using a timer that fires every 5 ms. Therefore, the hardware pointer position reported to userspace may lag behind the real one by up to 5 ms, and the number of rewindable samples may be also overestimated by the same amount.

Also, the hardware itself may report the hardware pointer position imprecisely. E.g., on common Intel HD Audio controllers, the granularity of the reported pointer position (as measured by calling `snd_pcm_avail()` and `snd_pcm_rewindable()` repeatedly) is 32 or 64 bytes. This is probably related to the DMA block size.

An idea was expressed that alsa-lib should take the above sources of uncertainty over the hardware pointer position or DMA engine weirdness into account when returning the number of rewindable samples.

An opposite viewpoint is expressed by Clemens Ladisch[9]:

> *It would make sense to report the pointer update granularity, but not to adjust the return value of* `snd_pcm_avail/rewindable()`.

However, on many cards, the pointer update granularity is simply unknown. A set of patches has been posted by Pierre-Louis Bossart[10] (and later merged) that are expected to help assessing the granularity of pointer updates at runtime. Still, nobody so far has tried to use the information exposed by these patches in PulseAudio.

5.4 Rewindability of self-contained ALSA plugins

User-grade programs (i.e. everything except sound servers) usually don't talk to hw devices. Instead, they use ALSA PCM plugins for functionality like mixing, channel remapping, sample rate conversion and software-based volume control. There are also more exotic plugins for tasks like software AC3 or DTS encoding, or spectrum equalization. Finally, there are plugins that allow ALSA programs to talk to sound servers like PulseAudio or JACK.

The implementation of rewinds is the simplest in plugins where each sample sent to the slave is determined only by the corresponding input sample. That is, the plugin never looks at non-current sample and doesn't keep any state. In this case, the implementation of the rewind operation should just rewind the slave by the same amount of samples. Also, to answer the question "how many samples can be rewound safely", the plugin should just ask the slave and forward the answer. Here are the plugins where this logic or a simple variation of it applies: `alaw`, `asym`, `copy`, `empty`, `hooks`, `lfloat`, `linear`, `mmap_emul`, `mulaw`, `multi`, `route`. These plugins are indeed rewindable. The `softvol` plugin is rewindable for the same reasons as long as nobody changes the volume.

[7] Most of the fixes are in alsa-lib 1.0.28 and one is in 1.0.29

[8] `http://permalink.gmane.org/gmane.linux.alsa.devel/127256`

[9] `http://permalink.gmane.org/gmane.linux.alsa.devel/127290`

[10] `http://permalink.gmane.org/gmane.linux.alsa.devel/133961`

The `iec958` plugin is used by some old cards (such as ATIIXP and CMI8338) to convert raw PCM to IEC958 frames and back. Each IEC958 subframe corresponds to one audio sample and one channel. Besides the PCM sample itself, the subframe contains a preamble, and one bit for each of Validity (for DAC), User data, Channel status and Parity. Different subframes use different types of preamble. This is needed to distinguish between left and right channels, as well as to mark the beginning of user data and channel status. The whole audio block contains 384 subframes.

Therefore, the plugin needs to keep a simple internal state: the number of subframes sent since the last subframe with the Z-type preamble (which is used to mark the left-channel subframe which also contains the beginning of the first channel status word). Before alsa-lib 1.0.28, rewinds didn't affect the state. Therefore, right after a rewind, a wrong type of preamble was used, and wrong bits (not continuing what was sent before) of channel status were sent down the link. This could cause a momentary resynchronization glitch on some receivers. As of alsa-lib 1.0.28, this is fixed by updating the state after each rewind, and thus the `iec958` plugin fully supports rewinds now.

The `adpcm` plugin converts between linear PCM and IMA ADPCM, which is only useful for ancient ISA cards. Again, the conversion is not stateless: the per-channel state includes the predicted sample value and the step size index, and is updated at each new sample according to simple table-based rules. As of alsa-lib 1.0.29, rewinds don't change the state. It is a bug. To solve it, one has to make this state per-sample per-channel, organized in a circular buffer similarly to the sound samples. This is not done yet.

The `dmix` and `dshare` plugins currently fail the rewind-correctness test for unknown reason. On them, the `snd_pcm_rewind()` function returns exactly the same number of samples as requested, however, the test program produces non-silent output.

As `dsnoop` is a capture-only plugin, it is not reviewed here. The `share` plugin could not be tested due to unrelated bugs, but, according to the source code, its `.rewindable` callback always returns 0.

5.5 External plugins

ALSA comes with two SDKs for building third-party plugins. The `ioplug` framework is for building plugins that output sound to some external systems, and `extplug` is for building filters. Also there is a `ladspa` plugin that wraps, well, third-party LADSPA plugins. There are two big problems in this area.

First, the plugin is not notified about rewinds at all. There is simply no such callback in the `snd_pcm_ioplug_callback` and `snd_pcm_extplug_callback` structures. The common code (wrongly) pretends that ioplug-based plugins are fully rewindable, but the rewind operation merely moves the application pointer back by the specified number of frames, and returns that number. Extplug-based plugins, as well as the ladspa plugin, simply forward rewind-related requests to the slave.

An important special case is that rewinds do not work (i.e. do nothing, "successfully") in the `pulse` ALSA plugin, even though native PulseAudio API does support rewinds.

But maybe it is possible to detect rewinds even without the corresponding callbacks?

For ioplug-based plugins, it may be possible to figure out from within the `.transfer` callback if there was any rewind operation between the previous call and the current call, by looking at the application pointer in the `snd_pcm_ioplug` structure. This may be sufficient to implement rewinds in the `pulse` plugin, but, as this approach does not allow to figure out the real amount of rewindable samples, it is a bad idea.

Extplug-based plugins don't have any access to their own application pointer, because it is hidden behind a private `snd_pcm_extplug_priv` structure. So they just don't have any chance to handle rewinds properly.

To solve the problem mentioned above, it would be necessary to add new callbacks. But this would cause the second issue, which is much worse. Imagine that someone has to implement these callbacks. Many ioplug/extplug-based plugins wrap external libraries. In order to implement rewinds, a plugin would have to tell the library to restore its old state. Mission impossible: these third-party libraries (as well as LADSPA API), in the vast majority of cases, don't have API functions that save and restore the state. I.e. this is the same problem as above, but one layer deeper and thus beyond our control. Besides, there is physically no way to e.g. undo sending of Bluetooth packets. As a result, rewinds just cannot be implemented correctly in the majority of ioplug/extplug-based plugins, and it was, as it seems, a mistake to offer them.

An interesting exception to the above non-rewindability rule is the `jack` plugin, especially since JACK itself is non-rewindable. The trick is that the plugin creates a real-time thread, and the JACK callback is invoked in the context of this thread, exchanging the samples with the JACK server. This looks very much like a real sound card, which periodically reads samples from the memory buffer. The `.rewindable` callback still yields a questionable result, though, by not taking into account the hardware pointer position uncertainty, which is one JACK period in this case.

It may be theoretically possible to extend this "low-latency worker thread" idea to other `ioplug/extplug/ladspa` plugin types – i.e. to create a thread just for the purpose of calling the `.transfer` callback instead of calling it when the client writes data. A natural period for calling this callback would be one slave period, but then the

resulting minimum latency would be three slave periods (if the slave allows using two periods), which is one period more than without this thread.

The current understanding is that, instead of adding an extra level of buffering in ALSA for plugin rewindability, it may be a better idea to teach PulseAudio to identify non-rewindable ALSA devices as such, and deal with them as appropriate. Indeed, this low-latency worker thread creates frequent wakeups and thus nullifies the primary motivation behind timer-based scheduling anyway.

5.6 Rate plugin

The rate plugin converts the sample rate of the audio data. The process is based on the idea to find a digital representation of the same analog signal that is represented by the sequence of input samples. Due to Shannon's sampling theorem, a perfect resampler should reject frequencies higher than half of the lower sample rate, and pass all lower frequencies through. Therefore, its time response can be described by the appropriately scaled sinc function [6]. The sinc function, however, has infinite support and thus has to be windowed or approximated by some other function with a finite support in order to become useful. Such approximations introduce distortions in the resampled sound: components with frequencies below the ideal cut-off frequency get attenuated, and also "aliased" content (with frequencies not present in the input signal) appears in the output. There are several libraries that implement audio resampling, using different approximations, and thus having different quality and speed.

It follows from the description above that each output sample is influenced by several input samples, and that each input sample affects several output samples. So the process of sample rate conversion is stateful.

The rate plugin delegates the process of sample rate conversion to a pluggable external converter. Alsa-lib itself contains a very simple (and low-quality) converter based on linear interpolation. Alsa-plugins contain converters based on the Speex resampler, the ffmpeg resampler, and libsamplerate.

This architecture suffers from the same limitations as discussed above for extplug. Namely:

- there are no rewind-related callbacks in the `snd_pcm_rate_ops` structure;
- none of the underlying libraries supports rewinds explicitly, or allows to save and restore (or otherwise alter) its state programmatically.

Therefore, in the current architecture, the rate plugin cannot be rewindable. And indeed, it isn't rewindable, as of alsa-lib 1.0.28.

The same objections apply to resamplers used by PulseAudio, and there is already a bug[11] reported by a user who noticed imperfect stitching of resampled audio before and after the volume change of an unrelated stream.

This situation is far from ideal, especially since sample rate conversion is a very common part of the audio processing pipeline. An important difference here from the `ioplug`/`extplug`/`ladspa` case is that there is, in fact, no task to wrap arbitrary third-party libraries, especially since none of the existing resampler libraries are actually suitable. The process of sample rate conversion is well-defined mathematically, the set of input samples affecting a given output sample is known, so it is possible to write a rewindable windowed-sinc resampler implementation from scratch. But nobody did it so far.

5.7 PulseAudio virtual sinks

Some PulseAudio virtual sinks (e.g., `module-ladspa-sink` and `module-echo-cancel`) perform non-trivial audio processing and keep state. PulseAudio sink API includes the rewind operation, and plugins generally supply it. However, the implementation either only moves pointers, or resets the filter completely (because the backend library is not rewindable), which is wrong. To fix this, one needs to remove the "reset the filter" recommendation in the `module-virtual-sink` template module, and explain how to express the fact that the virtual sink is not rewindable.

To be fair, the recently-submitted LFE filter patchset by David Henningsson[12] takes rewinds into account.

5.8 Dealing with non-rewindable devices

PulseAudio currently contains some logic[13] that disables timer-based scheduling and rewinds on `ioplug` plugins such as `a52`. However, the code does not match the (extplug-based) `dca` plugin as non-rewindable, and needs to be updated.

Initially, the idea was to fix the `snd_pcm_rewindable()` ALSA API function so that it always returns 0 for non-rewindable plugins (which is a good idea anyway) and use it. However, there are two reasons why this solution cannot work.

First, `snd_pcm_rewindable()` works only when the buffer size is already set. As already explained, rewinds are needed only in order to compensate the unacceptably-high latency implied by a large buffer. If rewinds are impossible, the buffer must be small. So, in order to see whether we need a small buffer, we would need to set the buffer size already.

Second, without sending a test sound, it does not actually help to distinguish a rewindable device from a non-rewindable one. Indeed, a rewindable-in-principle device with an empty buffer (and the

[11] `https://bugs.freedesktop.org/show_bug.cgi?id=50113`

[12] `http://lists.freedesktop.org/archives/pulseaudio-discuss/2015-January/023042.html`

[13] `http://cgit.freedesktop.org/pulseaudio/pulseaudio/commit/?id=cb55b00ccd25d965b1222e74375aee05427a449b`

buffer is initially empty) cannot be rewound.

Thus, a new API is needed in order to test whether an ALSA PCM is rewindable at all. Such API (`snd_pcm_hw_params_can_rewind()`) has been added in April 2008, but removed[14] 10 days later, because it was thought (wrongly) that the `snd_pcm_rewindable()` API function is more useful.

6 Client-side timing

Some legacy applications (e.g. many ALSA-based media players) rely on the audio subsystem as a source of timing. In particular, they expect the wakeups to come in a regular fashion, in strict accordance to the period size. To satisfy such legacy clients, PulseAudio has a special `PA_STREAM_EARLY_REQUESTS` flag that can be specified when creating a stream. Without this flag, requests will be made as late as possible. The `pulse` ALSA plugin always sets this flag.

On a system where timer-based scheduling works, and the CPU scheduler behaves reasonably, this flag usually works as expected. Indeed, due to the ability to program the timer for an arbitrary interval, PulseAudio can emulate any period size to the client.

Problems begin[15] when PulseAudio decides not to use timer-based scheduling (e.g., due to a batch card). In this case, PulseAudio uses the period size that is supported by the sound card and is close to the one specified in the `daemon.conf` file. Now suppose that the stream is moved to a different sound card that does not support this period size. As PulseAudio only wakes up and requests data from the client only on interrupts from the sound card, it no longer can wake up the client precisely when needed. In theory, clients are notified when buffer metrics change, and can adapt, but, in practice, no client handles this seriously. Worse, wrapper libraries such as alsa-lib and SDL cannot handle this easily, as they don't have the notion of dynamic changes of buffer metrics in their client API.

A separate question is what to do with clients like Wine or QEMU that (for various legitimate reasons) request very low latencies that are impossible to satisfy with the default buffer and period sizes. To add insult to the injury, the `pulse` ALSA plugin accepts almost any period size and, due to lazy creation of the PulseAudio stream, has no way to tell the client that the requested buffer and period sizes were actually not used.

The arguments listed above highlight the fact that PulseAudio, in non-tsched mode, does not perform adequate isolation of clients from the actual sound hardware, in terms of the supported and advertised period sizes. The bug can be fixed by asking sound data from a client using a separate timer, not based on soundcard interrupts, and possibly lying to the client about the total latency where regularity of requests matters more than exact latency estimation.

7 Conclusion

Timer-based scheduling does solve the real problem that it is intended to solve: it achieves dynamic latency, which should be good for power saving. If no resampling or other stateful audio processing is used, it "just works" on simple devices that DMA one sample at a time and report their DMA position precisely. On devices with more complex buffering models, it runs into corner cases described in this paper. But none of the listed problems look unsolvable – after all, there is always a possibility to fall back to the traditional period-based playback model. And there is indeed development work ongoing to provide more detailed timing information, to implement rewinds correctly in new PulseAudio effects, and to make other improvements – which is a good thing.

8 Acknowledgements

The author would like to thank Lennart Poettering for writing PulseAudio, and the current developers for continuing with the project.

References

[1] ALSA project – the C library reference. PCM (digital audio) plugins. `http://www.alsa-project.org/alsa-doc/alsa-lib/pcm_plugins.html`.

[2] Lennart Poettering. 2008. What's cooking in PulseAudio's glitch-free branch. `http://0pointer.de/blog/projects/pulse-glitch-free.html`.

[3] www.freedesktop.org. PulseAudio. `http://www.freedesktop.org/wiki/Software/PulseAudio/`.

[4] www.chromium.org. CRAS: Chromium OS Audio Server. `http://www.chromium.org/chromium-os/chromiumos-design-docs/cras-chromeos-audio-server`.

[5] Fons Adriaensen. 2005. Using a DLL to filter time. `http://kokkinizita.linuxaudio.org/papers/usingdll.pdf`.

[6] Julius O. Smith, 2015. Digital Audio Resampling Home Page, "Theory of Operation" section. `http://www-ccrma.stanford.edu/~jos/resample/Theory_Operation.html`.

[14] `http://git.alsa-project.org/?p=alsa-lib.git;a=commitdiff;h=c88672d86fe713e8f049df895fc3b64c472fbf5d`

[15] `https://bugs.freedesktop.org/show_bug.cgi?id=66962`, wrongly closed as fixed at the time of this writing

Embedded Sound Synthesis

Victor LAZZARINI, Joseph TIMONEY and **Shane BYRNE**
Sound and Digital Music Technology Group
Maynooth University
Maynooth, Co.Kildare
Ireland,
{Victor.Lazzarini, Joseph.Timoney, Shane.Byrne.2011}@nuim.ie

Abstract

This article introduces the use of the Intel Galileo development board as a platform for sound synthesis and processing, in conjunction with the Csound sound and music computing system. The board includes an Arduino-compatible electronics interface, and runs an embedded systems version of the Linux operating system. The paper describes the relevant hardware and software environment. It introduces a port of Csound, which includes custom frontends that take some advantage of the board capabilities. As a case study, a MIDI synthesizer is explored as one of the many potential applications of the system. Further possibilities of the technology for Ubiquitous Music are also discussed, which use the various interfacing facilities present on the Galileo.

Keywords

Embedded systems, sound synthesis and processing, music programming, Ubiquitous Music

1 Introduction

The Intel Galileo board[1] (fig.1) is an embedded systems development board based on the Quark System-on-Chip (SoC), which includes an Arduino-like functionality (and compatibility with some existing extension shields and software). The board can be used as a straight replacement for the Arduino Uno boards, with a customised Arduino Integrated Development Environment (IDE) that allows programming of applications (sketches) using the Wiring library. The Galileo, however, runs under a Linux-based operating system, and thus allows other modes of application that are not restricted to Arduino IDE sketches, and which can take more complete advantage of the board capabilities.

In this article, we examine the use of the Galileo board for sound synthesis and processing, with the Csound[ffitch et al., 2014][Boulanger, 2000] sound and music computing system. We demonstrate the scalability of Csound, which has been shown to run on a great variety of platforms, from supercomputers[2] to mobile[Lazzarini et al., 2012][Yi and Lazzarini, 2012] and web[Lazzarini et al., 2014], and now on embedded systems such as the Galileo. The hardware and software combination discussed in this paper has the potential of opening up a variety of new applications for electronic music composers and performers.

As a case study, we have developed a complete software image for the system, which allows it to be booted up as an outboard MIDI synthesizer. This paper is organised as follows: we first describe the hardware and software environment that is available to Galileo developers. We then discuss the details of the port of the Csound system, and its custom frontend that takes advantage of the board's Arduino-like capabilities. This is followed by a report on our case study, the MIDI synthesizer. Finally, we propose some further applications of the technology.

2 Galileo hardware and software

Galileo boards have been produced under two slightly different hardware configurations, namely, original (GEN1, pictured in fig.1), and a revised specification (GEN2, pictured in fig.2)[3]. They generally run under custom, specially-designed, Linux for embedded systems images, created and supported by the Yocto Project[4]. The board can be booted up from the flash memory (containing a minimal/small linux image), or from the SD card, which can contain more complete operating system images.

[1] `http://arduino.cc/en/ArduinoCertified/IntelGalileo`

[2] Csound was used as part of two class C projects at the Irish Centre for High-Performance Computing (ICHEC) exploring parallel processing for audio

[3] `http://www.intel.ie/content/www/ie/en/do-it-yourself/galileo-maker-quark-board.html`

[4] `https://www.yoctoproject.org`

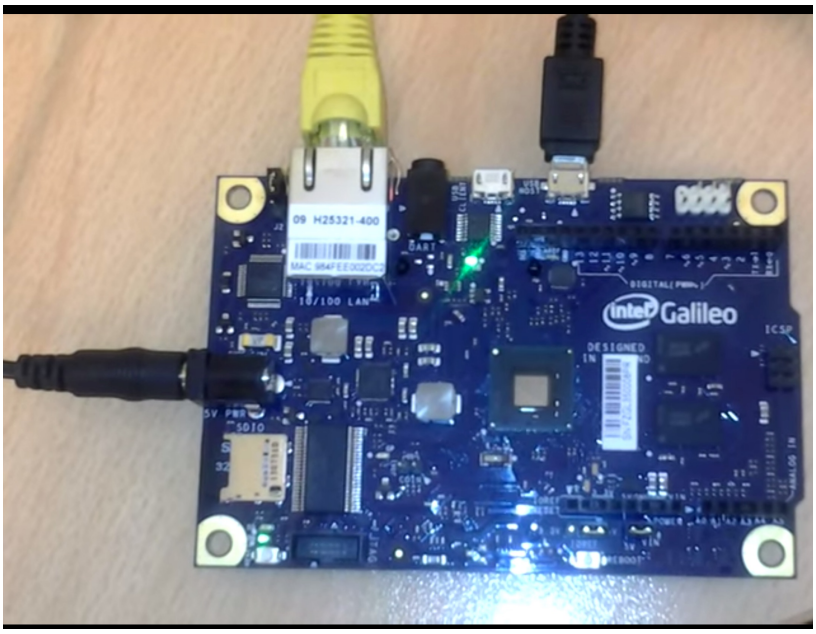


Figure 1: The Intel Galileo (GEN1) with ethernet and USB connections

2.1 Hardware specifications

The two share some basic attributes that include a Quark processor, which has the same instruction set to the Pentium, or i586, CPU, and contains a single core running at 400 MHz (also known as 'Clanton')[5], 10/100Mbit ethernet, PCI Express, USB 2.0 device and host interfaces, and microSD card reader. GEN1 boards have a 3.5 mm RS-232 connector, whereas GEN2 replaces this with a 6-pin Transistor-Transistor Logic (TTL) Universal Asynchronous Receiver-Transmitter (UART) header that is compatible with standard adaptors.

The Galileo uses the standard Arduino pin layout, which includes 20 General-Purpose Input/Output (GPIO) pins (6 multiplexed as analog inputs), plus power and Serial Peripheral Interface (SPI) headers. The hardware implementation of these is different in GEN1 and GEN2. In the former, an external GPIO expander chip (Cypress CY8C9540A) is used to control most of the pins in the shield, with only two Quark GPIOs connected directly (accessible via a multiplexer switch). The latter has 12 Quark GPIOs fully accessible to the headers, and uses a different GPIO expander chip layout (3 NXP PCAL9535A), mostly to control multiplexing (leaving eight available for input/output functions). The Quark GPIOs allow faster switching performance, through a dedicated software interface. Analog IOs are implemented in GEN1 via an Analog Devices AD7298 ADC IC, providing 12 bits of resolution, and in GEN2 via a Texas Instruments ADS108S102 IIO-ADC, which is 10-bit (scaled to a 12-bit range for compatibility purposes). Pulse-width modulation (PWM) is also implemented differently on the GEN2 board, providing higher resolution.

[5] http://ark.intel.com/products/79084/Intel-Quark-SoC-X1000-16K-Cache-400-MHz

2.2 Software systems

The board is generally run under a specially-built Linux OS image, although a Debian-based system has also recently been tested, and Microsoft has also provided a cut-down version of Windows 8 for it. We will discuss here the original Linux software that has been designed for the board. There are two types of Linux images that are used in the Galileo, based on different versions of the standard C library. The smaller image, mostly meant to be run from the limited space in the board flash memory, is built with uClibc. This library was originally designed for embedded Linux systems not using memory management units, but also runs on standard Linux. The other type of image is based on eglibc, which was designed for embedded systems but is generally compatible with the standard glibc. This library is more suit-

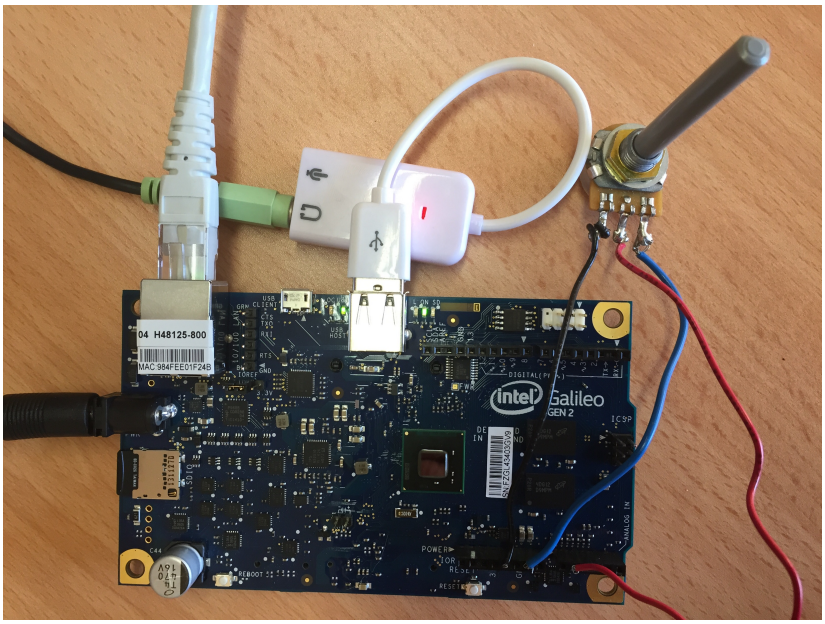


Figure 2: The Intel Galileo (GEN2) with ethernet, USB audio and a potentiometer connected to analog input 1 (pin A1)

able for SD card-based installations with no size constraints, as it is more fully-featured and provides better performance than uClibc. Software built with the Arduino IDE normally depends on uClibc, and therefore will not run on an eglibc image (although the Arduino IDE can be rebuilt from source to target this).

2.2.1 Development environment

Although it is possible to include all the development tools and use the board itself to build software, it is more advisable to set up a cross-compiling toolchain on a host computer. This is done by building an image and the toolchain from the sources, through a Linux Board Support Package (BSP) provided by Intel for the Clanton platform. The BSP is a collection of scripts (shell scripts, python scripts, recipes, etc) built with the resources provided by the Yocto Project, that allows us to build full Linux-based operating systems for specific embedded platforms. It uses the `bitbake` tool to collect all the information in the various scripts, download from sources, patch, build, and install the operating system software. The Galileo Yocto BSP can be used to build a fully-functional eglibc-based Linux standard base distribution, and a Software Developer Kit (SDK) containing a gcc/g++ toolchain. Most importantly for us, this Linux image contains the alsa library, and with it, we can access soundcards connected via the USB or PCI Express interfaces.

3 Csound for the Galileo

A fully-functional port of Csound was built for the Galileo board using the cross-compilation environment described above. The only two basic dependencies for Csound are libsndfile (for soundfile access) and the ALSA library (for realtime audio and MIDI). Although the image built with the provided BSP contains both, there are still a few issues to be resolved before we can build the system. Firstly, the cross-compilation environment installation does not appear to include the ALSA headers, so these need to be copied manually from the sysroot in the Yocto build to the installed toolchain sysroot.

Secondly, the libsndfile originally provided by the Yocto build is broken, as it depends on large file offset support that is not provided as standard by the system. So we have to modify the bitbake build recipe (provided in `./poky/meta/recipes-multimedia/libsndfile/libsndfile1_1.0.25.bb`) to configure the build with `-D_FILE_OFFSET_BITS=64`, and rebuild the image and SDK. With this in place, we can proceed to build Csound in the usual manner, using the CMake tools.

3.1 Custom frontends

In order to access the basic Arduino-like functionality of the Galileo, specific frontends were developed: `gcsound` (GEN1) and `gcsound2` (GEN2). This functionality can be divided into two groups: access to analog inputs, and access to the GPIO digital input and output. Such connections to the pins on the board is accomplished via the Linux Sysfs interface. This provides access to GPIOs via a number of files under `/sys/class/gpio` (for digital IO) and `/sys/bus/iio/devices/iio:device0/`. GPIOs need to be exported first by writing their number to the `/sys/class/gpio/export` file, and their direction ("in" or "out") needs to be written to `/sys/class/gpioN/direction`, where N is the GPIO number. Then its value (0,1) can be read/written to `/sys/class/gpioN/value`. The analog inputs can be accessed by reading the `/sys/bus/iio/devices/iio:device0/in_voltageN_raw` file (values in the range 0 - 4095), where N is the analog input number (0-5). The interface expects text (ASCII) characters as it was originally designed to work with `echo` and `cat`. Note that Sysfs is a regular interface for all GPIOs, and in order to take advantage of the fast IO provided by the GPIOs directly connected to Quark, a different interface (through `/dev/uio0` and `ioctl()` calls) is required. This has not yet been implemented in the two custom frontends.

3.2 Analog inputs

The analog inputs on the Galileo board are marked A0-A6. These pins are reserved for this purpose by the `gcsound` and `gcsound2` programs, ie. they cannot be used for digital input and output (although there is software support for this function). These inputs are offered to Csound orchestra in the software bus channels named as "analog$N$" where N is the analog port number. The application can then read these channels as required (using `chnget`). Access to the analog IO is present throughout the performance, continuously, and is implemented asynchronously (ie. non-blocking).

The signal is delivered as a floating-point number normalised between 0 and 1 (corresponding to a 0 - 5V input range). For instance, to access a potentiometer connected to the A1 analog input (as shown in fig.2), we use

```
ksig chnget "analog1"
```

3.3 Digital input and output

The remaining 14 pins can be used for general-purpose digital input or output. Access is provided as requested, through blocking reading/writing operations. This functionality is implemented as new opcodes in the system:

```
ival  gpin inum
kval  gpin inum
      gpout ival, inum
      gpout kval, inum
```

where `ival` and `kval` are the GPIO values (0 or 1), and `inum` is the GPIO number.

3.4 Pins and signals

Depending on the board version (GEN1, GEN2), digital signals at the various pins are mapped to different GPIO numbers. Access to some of these require the switching of one or more multiplex controls (also identified by specific GPIO numbers). The mapping of pins to GPIOs and multiplexers is shown below for the two versions of the Galileo board.

3.4.1 GEN1 board GPIO mapping

Table 1 can be used as reference for the GEN1 board pins and GPIO numbers used. Pins 4 - 9 are directly connected, other pins will require a multiplexer selector to be set before use. Most of the GPIO sources are provided through the Cypress CY8C9540A I/O Expander; two sources connected directly to the Quark SoC are available through pins 2 & 3 (as indicated on Table 1).

For the pins that require multiplexing, the `gpout` opcode will need to be used to select the correct source before accessing the pin from that source. For instance to access the GPIO for pin 0 and set it to 1, we have to use

```
gpout 1, 40
gpout 1, 50
```

so that GPIO 40 accesses the multiplex selector, selecting the source as GPIO 50, and we then set this to 1.

3.4.2 GEN2 board GPIO mapping

GEN2 board has a significantly different mapping scheme, as it employs a different hardware setup. Most of the 14 GPIO digital IO pins are connected directly to the Quark SoC, and so they are controlled in a slightly different way. In/out direction needs to be switched on for each pin by a separate GPIO setting. Most pins are multiplexed, so they also need to be

Table 1: Pin to GPIO mapping, Galileo GEN1

pin	mux selector, value	source/function
0	40, 0	UART0 RXD (/dev/ttyS0)
	40, 1	50 (GPIO)
1	41, 0	UART0 TXD (/dev/ttyS0)
	41, 1	51 (GPIO)
2	31, 0	14 (Quark GPIO)
	31, 1	32 (GPIO)
3	30, 0	15 (Quark GPIO)
	30, 1	18 (GPIO)
4	-	28 (GPIO)
5	-	17 (GPIO)
6	-	24 (GPIO)
7	-	27 (GPIO)
8	-	26 (GPIO)
9	-	19 (GPIO)
10	42, 0	SPI1_CS (Quark)
	42, 1	16 (GPIO)
11	43, 0	SPI1_MOSI (Quark)
	43, 1	25 (GPIO)
12	54, 0	SPI_MISO (Quark)
	54, 1	38 (GPIO)
13	55, 0	SPI_SCK (Quark)
	55, 1	39 (GPIO)

switched on via another GPIO. In addition, the board has pullup/pulldown 22k resistors connected to the pins that can be switched on and off, also through GPIOs. Table 2 shows the mapping for each pin and their associated GPIO numbers

The GPIOs controlling the direction of the Quark GPIO pins are set as 0 = output, 1 = input. Note that this is not necessary for the two PCAL9535A GPIO (pins 7 & 8). The resistor GPIO selectors are set as 0 = pulldown, 1 = pullup; if they are set to input, the resistor is disconnected. These allow the voltage to lower to the ground, or to rise to the operating voltage (5V), when pins are disconnected.

Multiplex selection works as with GEN1, by accessing and setting the relevant GPIO, and if there are mulitplexers involved, both need to be used. For example, to make the onboard led (which is connected to pin 13) light up, we need to set GPIO 46 to 0 to select the Quark GPIO, then 30 to 0 (output direction), and finally set 7 to 1 (to turn it on). Using this, an instrument that is equivalent to the Arduino *blink* sketch can be written as:

```
instr blink
  kcnt init 0
  kLed init 0
  gpout 46, 0
  gpout 30, 0

  if kcnt == 100 then
     kLed = (kLed == 0 ?  1 : 0)
     gpout kLed, 7
     kcnt = 0
   endif
  kcnt += 1
endin
```

3.5 Other possibilities

Presently, the Csound frontends `gcsound` and `gcsound2` do not implement other Arduino-like capabilities which are available in the system. These include PWM output, access to SPIO and Inter-Integrated Circuit (I2C) busses, fast GPIO, and tty interfaces. It is envisaged that some of these will be explored in the near future for specific applications. For instance, we plan to take advantage of SPIO connections to provide integrated audio DAC/ADC capabilities, either via custom or commercially-available shields. Fast GPIO will be made available alongside the regular Sysfs implementation, and

Table 2: Pin to GPIO mapping, Galileo GEN2

pin	mux 1, value	mux 2, value	dir	22k res	source/function
0	-	-	-	-	UART0 RXD (/dev/ttyS0)
	-	-	32	33	11 (Quark GPIO)
1	45, 1	-	-	-	UART0 TXD (/dev/ttyS0)
	45, 0	-	28	29	12 (Quark GPIO)
2	77, 1	-	-	-	UART1 RXD (/dev/ttyS1)
	77, 0	-	34	35	13 (Quark GPIO)
	77, 0	-	-	35	61 (PCAL9535A GPIO)
3	76, 1	-	-	-	UART1 TXD (/dev/ttyS1)
	76, 0	64, 0	16	17	14 (Quark GPIO)
	76, 0	64, 0	-	17	62 (PCAL9535A GPIO)
4	-	-	36	37	6 (Quark GPIO)
5	66, 0	-	18	19	0 (Quark GPIO)
6	68, 0	-	20	21	1 (Quark GPIO)
7	-	-	-	39	38 (PCAL9535A GPIO)
8	-	-	-	41	40 (PCAL9535A GPIO)
9	70 0	-	22	23	4 (Quark GPIO)
10	70, 0	-	26	27	10 (Quark GPIO)
11	44, 1	72, 0	-	-	SPI_MOSI (spidev1.0)
	44, 0	72, 0	24	25	5 (Quark GPIO)
12	-	-	-	-	SPI_MISO (spidev1.0)
	-	-	42	43	15 (Quark GPIO)
13	46, 1	-	-	-	SPI_SCK (spidev1.0)
	46, 0	-	30	31	5 (Quark GPIO)

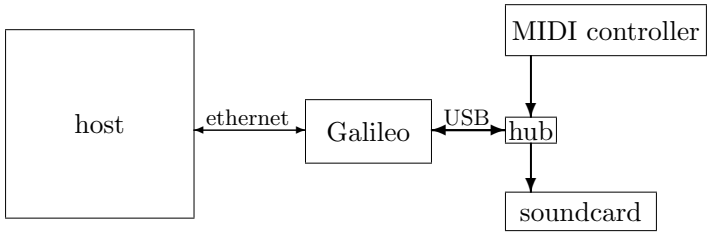


Figure 3: The Galileo Synthesizer layout

PWM will also be added to extend the output capabilities of the Galileo Csound implementation.

4 Case study: MIDI synthesizer

As a case study to assess the potential of the Galileo board for music-making, a fully-fledged MIDI synthesizer was developed. This employs Csound as the sound engine, interfacing with external USB audio and MIDI hardware. A Csound image for the board was developed, using the principles outlined in the previous sections, and including a realtime preemptive kernel. This image can be simply copied into any microSD card compatible with the board (sizes between 2 and 32GB). It will then be ready to use. It contains the standard Csound command-line frontend `csound`, the custom frontends `gcsound` and `gcsound2`, the Csound library (plus some plugin opcodes),

and the basic MIDI-based orchestra (midisynthesizer.csd).

These files are located in the `/home/root` directory. On boot up, the board will start a Csound process and load midisynthesizer.csd. This is set to use the default audio USB card and a MIDI input device, and it contains three instruments, accessible via MIDI program change messages. The three instruments are

1. Supersaw synth: a simple design based on five detuned sawtooth oscillators[Timoney et al., 2014]. Modulation controls detuning, CC 02 controls envelope attack, and CC 03 controls envelope release.

2. Pluck string: a Karplus-Strong-like[Karplus and Strong, 1983] instrument. Modulation controls brightness, CC 02 controls envelope attack, and CC 03 controls envelope release.

3. Voice: a ModFM[Lazzarini and Timoney, 2010] formant synthesizer. Modulation controls vowel types, CC 02 controls attack, and CC 03 controls envelope release.

Programs are set circularly to these three instruments (PGM mod3), and the synthesizer works in multimode, ie. it is possible to assign different programs to different channels. Although in this case study, we did not explore the possibilities offered by the Arduino-like electronics interfaces, these can be easily incorporated in the synthesizer design.

4.1 Testing the synthesizer

There is only a single host USB port on the board (there is also a client USB port, but this is used to connect to it from a host computer), so to use both an audio IO card and a MIDI controller, a hub is required. We tested the Galileo Synthesizer with a dynex USB hub, to which an M-Audio Oxygen 25 keyboard, and Behringer U-Control audio interface were connected. The synthesizer is generally very responsive, with low-latency audio output. Depending on the instrument, different polyphony limits apply. The vocal synthesizer is monophonic, but both the Supersaw and the Pluck string instruments allow up to six concomitant voices.

A video showing the synthesizer in action can be seen in

`http://youtu.be/ogYdJsKKxJk`

4.2 Connecting to the board

The board is fully accessible via `ssh`, through the use of a DHCP server (which can itself be run in the host computer) (fig.3). This can be used to debug, develop, and add new instruments to the existing ones. In this case, from the host terminal,

$ ssh root@<IP address>

In order to locate the IP address for the board, you need to have access to the network router, where you should see it listed alongside the board MAC address. You can find the board MACaddress on the top part of the ethernet socket. Once logged in, `vi` is available for simple editing. Files can also be copied to and from the board via `scp`. Host connection to the board is not required for synthesizer operation, as the board boots up into a Csound process directly. However, all USB connections should be present before booting the system.

5 Further applications

The technology described in this article has significant potential for further exploitation, beyond the simple case study discussed above. In particular, it has a direct application as a platform for Ubiquitous Music research and practice[Keller et al., 2015].

5.1 Portable live-electronics platform

For composers who employ live-electronics rigs regularly, having small, portable devices that can be used to interact with other performers on stage is very useful. With the Galileo+Csound system, it is possible to design various concert setups, with one or more boards, to deliver sound synthesis, processing, and audio playback. Due to the small size of these devices, they can be integrated in wearable components, or into augmented mechanic-electronic instruments.

5.2 Programmable effects units

The Galileo+Csound system can also be the basis for general-purpose audio processing "boxes", as fully programmable effects units. Because of the presence of a complete music programming language, it is possible to go beyond the usual categories of effects, and include more advanced signal processing algorithms, including frequency-domain methods such as the phase vocoder and sinusoidal modelling. Custom controls can be added via the electronics

interfacing capabilities offered by the board, as well as the usual MIDI and Open Sound Control (OSC).

5.3 Internet of Things sound devices

One of the original targets for the development of Galileo is to meet the potential of the Internet of Things (IoT) concept. Because of its networking capabilities (built-in ethernet, and easy wifi implementation via a PCI card), the board can be used as a remote sound device. It can run small web servers, node.js, and similar services, which can be linked up with the Csound sound synthesis engine. In addition, Csound can work as a networked sound server, which is capable of accepting control directly via UDP messages, and/or the OSC protocol.

5.4 Low-cost cluster computing for audio

With the built-in ethernet, it is possible to design a low-cost cluster, with the use of a network switch. The custom Linux OS image described in the earlier sections of this article also includes the OpenMPI library, which is an implementation of the Message Passing Interface (MPI), a standard technology for cluster networking. With this it is possible to construct a Cluster-based Csound frontend, that takes advantage of parallel processing to implement a high-performance audio engine. Such a setup would most likely be low cost in comparison to other comparable alternatives.

6 Conclusions

This paper reported on the implementation of an audio processing system using the Intel Galileo development board and Csound, running under a customised version of Linux for embedded devices. After a detailed discussion of the relevant hardware and software environment, we have explored the porting of Csound and the development of customised frontends to access the electronics interfacing capabilities of the board. A case study based on a simple MIDI synthesizer was used to demonstrate the platform as a viable sound and music programming environment. The article concluded with a number of possible application scenarios. While we have concentrated on a specific embedded platform, the ideas and principles discussed here can be applied elsewhere. In particular, we hope to develop similar systems for the Intel Edison[6] in the near future.

7 Acknowledgements

We would like to acknowledge the support of Intel Ireland, who very kindly supplied our research group with GEN1 and GEN2 Galileo development boards.

References

Richard J. Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

John ffitch, Steven Yi, and Victor Lazzarini. 2014. Csound on GitHub. `http://csound.github.io`.

Kevin Karplus and Alex Strong. 1983. Digital Synthesis of Plucked String and Drum Timbres. *Computer Music Journal*, 7(2):43–55.

Damian Keller, Victor Lazzarini, and Marcelo Pimenta (eds.). 2015. *Ubiquitous Music*. Springer Edition, New York.

Victor Lazzarini and Joseph Timoney. 2010. Theory and practice of modified frequency modulation synthesis. *J. Audio Eng. Soc*, 58(6):459–471.

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marcelo Pimenta. 2012. The Mobile Csound Platform. In *Proceedings of ICMC 2012*.

Victor Lazzarini, Edward Costello, Steven Yi, and John ffitch. 2014. Csound on the Web. In *Linux Audio Conference*, pages 77–84, Karlsruhe, Germany, May.

Joseph Timoney, Victor Lazzarini, Jari Kleimola, and Vesa Valimaki. 2014. Examining the Oscillator Waveform Animation Effect. In *Proceedings of DAFx 2014*.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Linux Audio Conference*, volume 6.

[6] `https://www-ssl.intel.com/content/www/us/en/do-it-yourself/edison.html`

MobileFaust: a Set of Tools to Make Musical Mobile Applications with the Faust Programming Language

Romain Michon
CCRMA
Stanford University
CA 94305-8180
USA
rmichon@ccrma.stanford.edu

Julius O. III. Smith
CCRMA
Stanford University
CA 94305-8180
USA
jos@ccrma.stanford.edu

Yann Orlarey
GRAME
Lyon
France
orlarey@grame.fr

Abstract

This work presents a series of tools to turn FAUST code into various elements ranging from fully functional applications to multi-platform libraries for real time audio signal processing on iOS and Android. Technical details about their use and function are provided along with audio latency and performance comparisons, and examples of applications.

Keywords

Faust, iOS, Android, DSP

1 Introduction

Mobile platforms offer a great opportunity to the world of open source audio to make sound synthesis and processing accessible to a wider audience [Yi and Lazzarini, 2012; Brinkmann et al., 2011]. The use of smartphones and tablets as musical instruments is now accepted by a large number of musicians. Not only are mobile devices widespread and owned by many, they offer a higher level user interface paradigm than computers, which often makes them more stable and simpler to use. In particular, Android devices, which are more open than iPhones and iPads (§3) offer a good compromise between open-source, stability, and ease of use.

FAUST[1] [Orlarey et al., 2002] is a functional programming language for real-time digital signal processing (DSP) that generates highly efficient DSP code in a variety of languages (C, C++, LLVM, asmjs, and more) that can be compiled into a variety of forms using a system of wrappers. These wrappers, called *architecture files*, describe how to adapt the DSP computation to the external world [Fober et al., 2011]. Therefore, it is easy to go from FAUST to standalone applications for different kinds of platforms, Web applications, audio plug-ins, externals for music programming languages, and so on.

This paper presents a series of tools that can turn FAUST code into various elements ranging from fully functional applications to multi-platform libraries for real-time audio signal processing on iOS and Android. Technical details about their use and function are provided, along with audio latency and performance comparisons, and examples of applications.

2 Faust2api

The main idea of `faust2api` is to provide iOS and Android developers with a system that generates custom high-level APIs for real-time audio signal processing. Even though the APIs work quite differently "under the hood" on iOS than on Android, they are accessed similarly on the two platforms.

The `faust2api` script operates as a command-line tool in a shell. A FAUST source file is provided as an argument, along with the option `-ios` or `-android` specifying the desired architecture, and one or more source files are created as output (a single C++ header file for iOS, and a directory containing both Java and C++ source files for Android). The library takes care of starting the audio engine and instantiating the DSP code, as well as connecting them together. At the API level, this is all done by the C++ method `init(sr,bs)` which takes the desired sampling-rate and audio buffer-size as arguments. Computing of the audio process is launched by a `start()` method. Finally, the audio engine can be closed and the memory freed by simply calling `stop()`.[2]

On both iOS and Android, the audio process runs in its own high-priority thread. The various parameters of the FAUST object can be accessed and written via `getParam(path)` and `setParam(path)` where the parameter `path` is

[1] `http://faust.grame.fr`

[2] Detailed documentation of the API can be found here: `https://ccrma.stanford.edu/~rmichon/mobileFaust/#ref`

the parameter's path in the user-interface tree defined in the FAUST code (as discussed further below in §3.3 on OSC and MIDI support).

If the FAUST object provided to `faust2api` has no inputs, and has `freq`, `gain`, and `gate` parameters defined, it is considered as an instrument and automatically made polyphonic. The different voices (eight by default, but this can be changed) can be triggered using a `keyOn()` method that takes a MIDI note number and MIDI velocity as an argument. This method is linked to the `freq`, `gain`, and `gate` parameters (§3.1) and allocates a new voice every time it is called. The `keyOff()` method sets the `gate` parameter of the voice to zero and waits until the level of the voice falls below -60 dB to deallocate it.

2.1 iOS

The command "`faust2api -ios faustCode.dsp`" will generate a single C++ header file that can be included in any iOS app project. The API relies on the `AVAudioSession`[3] framework to connect to the audio engine.

"Touch to sound" and "round-trip" latency measurements for iOS audio applications generated by `faust2api` were carried out on an iPad2 and an iPhone5 (Fig. 1).

Device	Touch to Sound	Round-Trip
iPhone5	36 ms	13 ms
iPad2	45 ms	15 ms

Figure 1: Audio Latency for Different iOS Devices Using the FAUST Library.

"Round-trip" latency was measured by creating a simple app that just plays back any sound that comes to its audio input (in our case, the audio input jack) and by comparing how long it takes for the iOS device to play back an impulse sent to this app.

"Touch to sound" latency was measured by creating a simple test app where a button on the screen is used to trigger an impulse. The audio output jack of the iOS device was connected to an ADC[4] in order to be able to record the impulse on a computer. A microphone connected to the same ADC on a different channel was placed at the top of the screen of the device. The latency measurements were carried out by measuring the time difference between the "acoustic" impulse detected by the microphone and the synthesized one (Fig. 2).

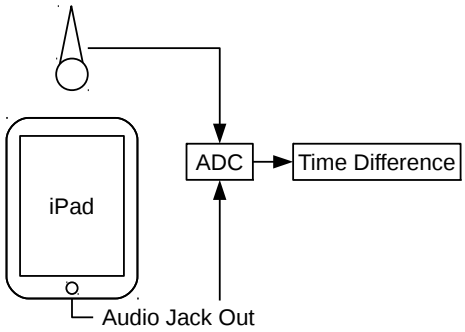


Figure 2: Touch to Sound Audio Latency Measurement Set-Up.

2.2 Android

`faust2api` is slightly more complex to use on Android than iOS. Indeed, Android apps are primarily programmed in Java. However, this language is not very well suited for real time DSP so most of the library generated by `faust2api` is written in C++ with a Java interface.

The audio engine is accessed, controlled and connected to the DSP code generated by FAUST on the "native" side of the library where everything is computed in a high-priority thread. This allows the signal processing part of the app to be fully independent from the Java side.

The native portion of the library is compiled as a shared library using the Android NDK[5] and can be controlled in Java using a JNI[6] interface generated by SWIG.[7] More details about the way this system works can be found in [Michon, 2013].

In practice, `faust2api` will generate the Android API by using the `-android` option instead of `-ios` (cf. previous section) and will provide a set of Java and C++ files to be copied in the Android app project.[8]

[3] `https://developer.apple.com/library/ios/-documentation/AVFoundation/Reference/-AVAudioSession_ClassReference/index.html#//-apple_ref/occ/cl/AVAudioSession`

[4] *Analog to Digital Converter*

[5] *Native Development Toolkit*: `https://developer.android.com/tools/sdk/ndk/`

[6] *Java Native Interface*

[7] *Simplified Wrapper and Interface Generator*: `http://www.swig.org/`

[8] A tutorial on how to do this can be found here: `https://ccrma.stanford.edu/~rmichon/mobileFaust/#f2apAndroid`

Latency measurements using the same techniques presented in the previous section were carried out on a Samsung Galaxy S5, a Google Nexus 5, and a Google Nexus 7 that were all running on Android 5.0 (Lollipop). It is difficult to make a complete comparison here in the same way as for iOS because latency varies greatly between devices and manufacturers. The main observation that can be made though is that audio latency is much larger on Android than iOS.

Device	Touch to Sound	Round-Trip
Samsung Galaxy S5	72 ms	78 ms
Google Nexus 5	90 ms	92 ms
Google Nexus 7	130 ms	130 ms

Figure 3: Audio Latency of Different Android Devices Using the FAUST Library.

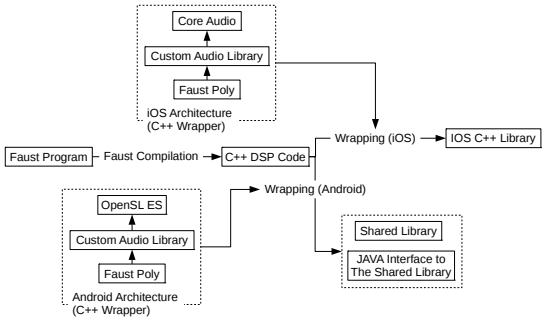


Figure 4: `faust2api` Overview.

3 Faust2android

While a preliminary version of `faust2android` was presented in [Michon, 2013] it has been totally rewritten since then and offers a large number of new functionalities.

`faust2android` is built on top of `faust2api` (Fig. 9). Its user interface is constructed using the JSON description provided by the shared library generated by `faust2api`. All the standard FAUST UI elements are available: horizontal and vertical groups, horizontal and vertical sliders, numerical entries, knobs, checkboxes, buttons, drop-down menus, radio buttons, bargraphs, etc. Some examples are shown in figure 5. The values of the parameters of the audio process running on the native side are changed using the `setParam()` function of the API.

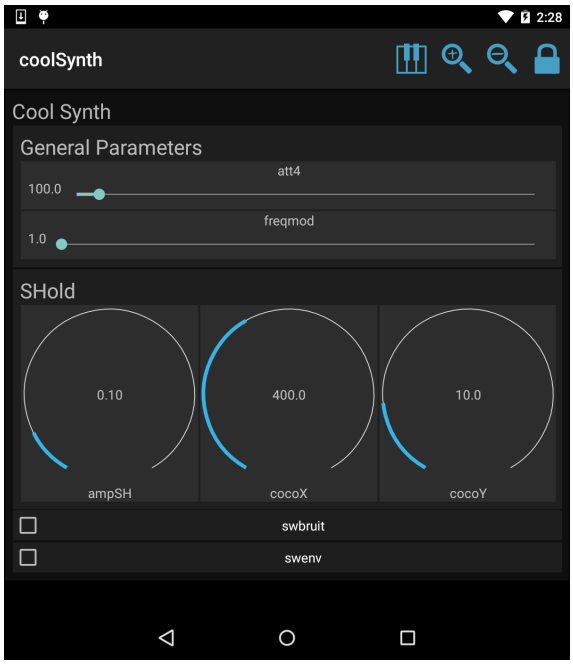


Figure 5: Example of interface generated by `faust2android` containing groups, sliders, knobs and checkboxes.

3.1 Keyboard and Multitouch Interface

`faust2android` allows assignment of more interactive interfaces to the FAUST process. For that, three different metadata items can be added to the top-level group of a FAUST program. In FAUST, a metadata item consists of a key:value pair, specified between square brackets within a title string, i.e., `"Some Title [key:value]..."`.

The `[style:keyboard]` metadata item specifies that the `freq`, `gain`, and `gate` parameters in the FAUST code should be assigned to a piano keyboard that can be opened by touching the "keyboard icon" in the top right corner of the app. Also, these three parameters will be automatically removed from the main interface for controlling the other parameters.

The following example program illustrates a simple usage:

```
import("music.lib");
s = button("gate");
g = hslider("gain",0.1,0,1,0.001);
f = hslider("freq",100,20,10000,1);
process = vgroup("[style:keyboard]",
    s*g*osc(f));
```

This interface uses the polyphonic capabilities of `faust2api`. Touching a key on the key-

board determines the reference pitch of the note but sliding the finger across the X axis of the screen allows the user to continuously control it. The Y axis determines the gain of the note. If a MIDI keyboard is plugged into the Android device, it will be able to control the keyboard interface (§3.3).

The `[style:multi]` metadata item will create a simple interface in which parameters are represented by moveable dots on the screen. Each dot can have two parameters assigned to it, corresponding to x and y screen coordinates. This interface can also be opened by touching the keyboard icon on the top right corner of the screen. Parameters are linked to the interface via `[multi:x]` metadata where `x` is the ID of the parameter in the interface. For example, the FAUST program

```
import("music.lib");
freq = hslider("freq[multi:0]",
    440,50,2000,0.1);
process = hgroup("[style:multi]",
    osc(freq));
```

creates an app in which the frequency parameter of a sine oscillator is controlled by the X axis of the dot in the multitouch interface. Parameters that have the accelerometer assigned to them (cf. §3.2) will continue to be driven by the accelerometer in the interface.

Finally, the `[style:multikeyboard]` metadata combines the keyboard and multitouch interface into one (Fig. 6).

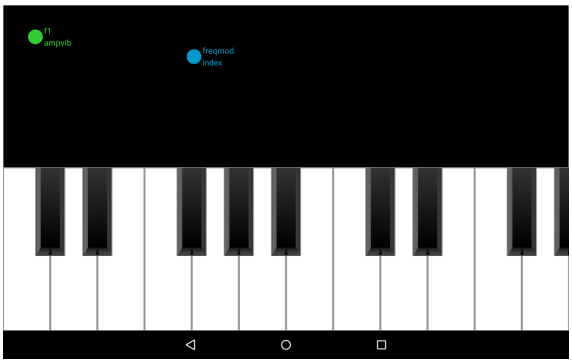


Figure 6: Example of "Multi-Keyboard" Interface of an Application Generated by `faust2android`.

3.2 Using the Built-In Accelerometer

The Accelerometer can be used to control some elements of the user interface. Assignments are made in the *Accelerometer Parameters* panel that can be opened by holding the label of a parameter for more than one second (Fig. 7). From here, the mapping of an accelerometer to a parameter can be configured precisely to create complex linear and non-linear behaviors. For instance, the user can choose which axis will control the parameter (x, y, or z), its motion orientation, and sensitivity.

Three different modes can be used to control the orientation of the accelerometer, *normal*, *inverted*, and *bell*. In *bell* mode, the maximum value of the accelerometer is output when it is in center position and the minimum value when it is fully inclined to the left or right.

Sensitivity can be configured with three different parameters, *min*, *max*, and *center*, that are all expressed in $m/s^2 \times 10^{-1}$. As an example, setting *min* to -1, *max* to 1, and *center* to 0 will create a linear behavior where the minimum value of the parameter being controlled is given at position -90 degrees and the maximum value at position +90 degrees. Any acceleration beyond these limits will be clipped.

All these parameters can be configured from the FAUST code using metadata by specifying `[acc: a b c d e]`, where `a` is the axis (0 for x, 1 for y, 2 for z), `b` the orientation (0 for *normal*, 1 for *inverted*, 2 for *bell*), `c` the minimum, `d` the maximum and `e` the center.

Raw data from the accelerometers are passed directly to the FAUST audio process. Filtering can be carried out in FAUST which is better suited for that kind of task than Java.

Finally, the accelerometer parameters window is only accessible if the app is unlocked by touching the "lock" icon on the top right corner of the screen (Fig. 5). Apps can be locked to prevent users from opening a configuration window or rotating the screen during a performance.

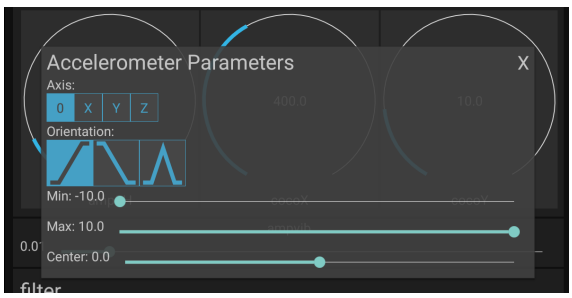


Figure 7: Accelerometer Configuration Panel of an Application Generated by `faust2android`.

3.3 OSC and MIDI Support

OSC support is enabled by default for all the parameters of applications generated by `faust2android`. The OSC address of a parameter corresponds to the path to this parameter in the FAUST code. For example, the OSC address of the `freq` parameter of the FAUST code

```
freq = hslider("freq",
    440,50,2000,0.1);
process = hgroup("Main",osc(freq));
```

will be `/Main/freq`.

MIDI support is also enabled by default in apps generated by `faust2android`. MIDI Key Number is automatically mapped to the `freq` parameter by converting it to frequency in Hz, and similarly MIDI velocity → `gain`. Note on/off events control the `gate` parameter, just like the `keyOn()` and `keyOff()` functions of `faust2api`. Synthesizer apps generated with `faust2android` all have eight voices of polyphony.

MIDI control numbers can be assigned to specific parameters from the FAUST code using the `[midictl:x]` metadata where `x` is the MIDI control number.

3.4 Audio IO Configuration

Android applications generated with `faust2android` automatically choose the best sampling rate and buffer size as a function of the device that is running them (for *Nexus*[9] devices only). Indeed, it was explained during the *Google I/O 2013 conference on High Performance Audio*[10] that Android phones and tablets achieve better audio latency performance if they run with a specific buffer size and sampling rate (see Fig. 8). Users may override these default values in the settings menu of the app.

Device	Sampling Rate	Buffer Size
Nexus S	44100	880
Galaxy Nexus	44100	144
Nexus 4	44100	240
Nexus 7	44100	512
Nexus 10	44100	256
Others	44100	512

Figure 8: Preferred Buffer Sizes and Sampling Rates for Various Android Devices.

[9] `http://www.google.com/nexus/`

[10] `http://youtu.be/d3kfEeMZ65c`

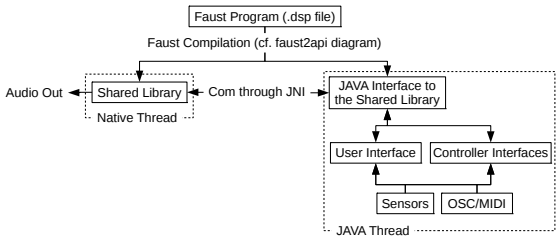


Figure 9: `faust2android` Overview.

3.5 Easy App Generation

While it is relatively simple to use `faust2android`, it requires the programmer to have an important number of dependencies installed (Android SDK and NDK, etc.). However, FAUSTLIVE [Denoux et al., 2014] and the FAUST Online Compiler [Michon and Orlarey, 2012] make the process of turning FAUST code into an Android application very simple. Indeed, when the user chooses to compile a FAUST program as an Android app, a QR code pointing to the generated app package is displayed that can be scanned by the device where the user want the app to be installed.

4 Applications

The FAUST distribution contains a collection of libraries that implement a large number of common and less-common audio effects, filters, and synthesizers. With `faust2api`, iOS and Android programmers who don't know signal processing or who never worked with real-time audio can easily integrate any of the pre-written FAUST modules into their project without having to write a single line of DSP code. On the other hand, this tool also gives the opportunity to FAUST developers to have their work used by more people. A concrete use of this tool was made this year in the *Music 256b* class[11] *"Mobile Music"* offered at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA)[12] where students were given the opportunity to use `faust2api` in their final projects.

Another use of applications generated by `faust2android` and `faust2ios` is the *SmartFaust*[13] project led by Xavier Garcia and Christophe Lebreton at GRAME. The idea was

[11] `https://ccrma.stanford.edu/courses/256b-winter-2015/`

[12] `http://ccrma.stanford.edu`

[13] `http://www.grame.fr/anything_slides/concert-smartfaust`

to make a series of concerts where the music is made by the audience with their mobile phones. Several applications were put on the *Apple Store* and the *Google Play Store* that people could download prior to the concert. This project led to more metadata for controlling the user interfaces; for example, it is possible to choose to not integrate a UI element to the interface. This enables the FAUST programmer to control some specific parameters with the accelerometer (using metadata too) without displaying them in the interface. `faust2android` can also generate "concert apps", where the user can switch between different FAUST objects within the same application.

5 Conclusions

Several tools that use FAUST to help design or make ready-to-use Android and iOS applications were presented. We believe that they make the development of musical applications on mobile platforms easier and that they will contribute to making the use of FAUST objects more accessible to musicians and performers.

While iOS real-time audio applications provide much better (smaller) audio latency than Android, the various restrictions imposed by Apple on their deployment makes them less accessible which is a big issue for the use that we make of them with FAUST. Therefore, we hope that Google will resolve the audio latency issues for Android applications in the near future.

FAUSTLIVE and the Online Compiler provide easy ways to use the tools presented in this paper. However, we think that enhancing them with an online platform where FAUST developers can easily share their work with others in order to create a repository of FAUST resources would be a great addition.

6 Acknowledgments

Part of this work has been implemented under the FEEVER project [ANR-13-BS02-0008] supported by the Agence Nationale pour la Recherche.

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*. Bauhaus U., Weimar (Germany).

Sarah Denoux, Stephane Letz, Yann Orlarey, and Dominique Fober. 2014. FaustLive: just-in-time Faust compiler and much more. In *Proceedings of the Linux Audio Conference (LAC-14)*, pages 102–107. ZKM, Karlsruhe (Germany), May.

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 Yann Orlarey. 2012. The Faust online compiler: a web-based IDE for the Faust programming language. In *Proceedings of the Linux Audio Conference (LAC-12)*, pages 111–116. CCRMA, Stanford University (USA).

Romain Michon. 2013. Faust2android: a Faust architecture for Android. In *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-2013)*, pages 98–102. National University of Ireland (Maynooth), September.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2002. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMA)*, pages 542–547. Gothenburg, Sweden.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Proceedings of the Linux Audio Conference (LAC-12)*, pages 233–239. CCRMA, Stanford University (USA).

A Taste of Sound Reasoning in Faust

Emilio Jesús Gallego Arias, Olivier Hermant, Pierre Jouvelot

MINES ParisTech, PSL Research University, France
{emilio.gallego_arias, olivier.hermant, pierre.jouvelot}@mines-paristech.fr

Abstract

We address the question of what software verification can do for the audio community by showcasing some preliminary design ideas and tools for a new framework dedicated to the formal reasoning about Faust programs. We use as a foundation one of the strongest current proof assistants, namely Coq combined with SSReflect. We illustrate the practical impact of our approach via a use case, namely the proof that the implementation of a simple low-pass filter written in the Faust audio programming language indeed meets one of its specification properties.

The paper thus serves three purposes: (1) to provide a gentle introduction to the use of formal tools to the audio community, (2) to put forward programming and formal reasoning paradigms we think are well suited to the audio domain and (3) to illustrate this approach on a simple yet practical audio signal processing example, a low-pass filter.

Keywords

DSP; audio; program verification; theorem proving

1 Introduction

Formal program verification is gaining strong support in the computer programming world with projects such as the CompCert certified C compiler [Leroy, 2009], with more and more tools such as the Coq[1] proof assistant striving to ease the development of correctness proofs for hopefully the every-day programmer.

While there has been some work in the mathematical correctness of DSP — see for instance [Krishnaswami, 2013; Brunel et al., 2014] for type-based techniques, and [Souari et al., 2014; Ghafari et al., 2011] for approaches using theorem proving — formal verification is still largely absent in the DSP and Computer Music (CM) communities. Yet, users and musicians are always striving for ever better sound experience and audio realism. Thus, ensuring the adequacy between an intended audio specification, for instance some sort of a limited-bandwidth filtering, and its actual implementation, along with other key properties such as robustness [Chaudhuri et al., 2012] or Bounded-Input Bounded-Output (BIBO) stability, is warranted. Too often, the only correctness test performed is to check that "it sounds about right", which is a methodology that clearly deserves some improvements.

The overall goal of our paper is to provide a case for the introduction of tools and best practices dedicated to the formal mathematical reasoning about audio/sound application programs. We illustrate this vision via the use of Coq/SSReflect for the Faust audio language[2]. We introduce a new framework for mathematically reasoning about Faust audio programs. We show how simple properties can be already proven for some Faust filtering applications using such an approach, thus paving the way to the future introduction of dedicated proof techniques for audio processing systems.

This paper is structured as follows. In Section 2, we introduce the Coq proof assistant and its SSReflect extension, which we use all along. Section 3 describes the Faust language core, using a low-pass filter as an example, while Section 4 provides a Coq implementation. We introduce in Section 5 a specific logic that will help reasoning about Faust programs. We finally put these tools into good use in Section 6, where we show how the low-pass filter can be proven equivalent to its specification. We discuss future work and conclude in Section 7.

2 A Coq/SSReflect Tour

Coq [Bertot and Castéran, 2004] is a software development environment in which programmers can write functional programs and prove properties about them. Coq's programming language Gallina is very similar to other functional programming languages like Ocaml, but with some added restrictions (in particular, all Coq pro-

[1] `coq.inria.fr`

[2] `http://faust.grame.fr`

grams must terminate) and a more advanced type system. Coq is a *strongly-typed* language; in particular, a type is understood as a property P, and an expression e of type P, written `e : P`, is the proof for P.

SSReflect [Gonthier et al., 2008] is a proof language and extensive library built on top of Coq. It promotes a structured style of programming and facilitates proof development by profiting from the fact that many properties of interest can be expressed as programs of type boolean.

Recursive Definitions We will illustrate some basics of the SSReflect's proof language by proving a toy property over a toy programming language. We first load some required libraries:

```
Require Import ssrfun ssrbool eqtype ssrnat.
```

The Abstract Syntax Tree (AST) of our toy programming language is defined in Coq with the `Inductive` command:

```
Inductive exp : Type :=
| cst : nat → exp
| mul : exp → exp → exp.
Notation "' n" := (cst n).
Notation "a ×b" := (mul a b).
```

which declares the recursive (so-called inductive) type `exp` with two constructors, `cst` for embedding natural integer constants, and `mul` for the multiplication of two expressions. Coq provides support to declare convenient notations such as × for the multiplication operation.

Recursive functions are declared in Coq using the `Fixpoint` command. Then, the value of an expression in our toy language is defined by recursion over its structure:

```
Fixpoint eval (e : exp) : nat :=
  match e with
    | 'n => n
    | e1 × e2 => eval e1 * eval e2
end.
```

We can run our program with the `Eval` command:

```
Eval compute in eval ('2 ×('0 × '3)).
```

which will display, as expected, `0`. Going one step further, Ocaml code can be generated automatically from Coq programs, providing reasonably efficient and provably-correct implementations of the specified algorithms.

Proving Properties Following upon our previous simple test, assume we want now to *prove* the following, 0-absorbing property, named `eval0eB`: all expressions `e` that contain a `'0` subexpression evaluate to 0. We can write a boolean function `mem_exp` that checks if `'0` occurs in an expression `e` easily[3]:

```
Fixpoint mem_exp m e :=
  match e with
    | 'n => n == m
    | e1 × e2 => mem_exp m e1 || mem_exp m e2
  end.
```

Theorem `eval0eB` is rephrased now as: for all expressions `e`, if `mem_exp 0 e` is `true`, then `eval e` is `0`. Let's begin by proving an easier property for constant expressions; it is stated here in Coq as Lemma `eval0cB`, followed by its proof:

```
Lemma eval0cB :
  forall n, mem_exp 0 ('n) → eval ('n) == 0.
Proof. by move=> n /=; exact. Qed.
```

In addition to programming in Gallina, we can also use automated program building procedures called *tactics*. We can use an interactive proof editor such as Proof General of CoqIDE to step from `Proof` to `Qed`; after each tactic (terminated by a `.`), the current proof state is displayed. The proof state consists of a set of hypotheses $e : t$ (the "context") and a goal g which should be seen as a stack of properties p_0->....->p_n->g. Most SSReflect tactics `tac` can perform context manipulation operations before and after running. `tac`: `x` will remove a named hypotheses `x` : p from the context, pushing p to the goal before `tac` is run; `tac` => `x` will pop the top of the goal after `tac` is run, naming it `x`. Thus, if p_0->A is the current goal, `move`=> `x` will add `x` : p_0 to the context. Plenty of additional operations can be performed in addition to moving.

The previous proof started with the documentation-only `Proof` command, getting the goal:

```
forall n, mem_exp 0 ('n) → eval ('n) == 0.
```

with an empty context. The first step is to move n to the context with `move`=> `n` /=. The /= switch asks Coq to perform partial evaluation of the goal, resulting in a new goal `n` == 0 → `n` == 0, which the `exact` tactic solves. Finally, `Qed` checks that the proof is correct, and, as for all previous function and type definitions, `eval0cB` is added to the global context for further use.

Moving then to our full theorem, we state:

```
Theorem eval0eB :
  forall e, mem_exp 0 e → eval e == 0.
Proof.
  elim.
  – by apply: eval0cB.
  – move=> e1 H1 e2 H2 /=.
```

[3] Coq uses type inference to get the types of `m` and `e`.

```
    case/orP=> [/H1 | /H2] /eqP → .
    + by rewrite mul0n.
    + by rewrite muln0.
Qed.
```

Here, the proof starts by performing induction on the structure of expressions. The induction tactic `elim` operates on the top of the goal, which in this case is the expression e. The base and inductive subgoals are generated, displayed as:

```
subgoal 1 (ID 18) is:
forall n : nat, mem_exp 0 (' n) →
      eval (' n) == 0

subgoal 2 (ID 19) is:
forall e : exp,
 (mem_exp 0 e → eval e == 0) →
  forall e0 : exp,
   (mem_exp 0 e0 → eval e0 == 0) →
    mem_exp 0 (e ×e0) → eval (e ×e0) == 0
```

We must prove each goal separately (the `-` and `+` symbols indicate a new proof step). The base case is dealt with by first using Lemma `eval0cB` from the global context with the `apply` tactic (the `by` so-called "closing tactical" ensures that the current goal is finished). In the inductive case, the goal includes the facts that `eval0eB` is true on each subexpression, named here `e` and `e0` by Coq. We first move `e` and `e0`, renamed `e1` and `e2`, to the context, each followed by its induction hypotheses H_i, before performing a partial evaluation. After the move, we have the following proof state:

```
e1 : exp
H1 : mem_exp 0 e1 → eval e1 == 0
e2 : exp
H2 : mem_exp 0 e2 → eval e2 == 0
============================
mem_exp 0 e1 || mem_exp 0 e2 →
    eval e1 * eval e2 == 0
```

The top of the goal is a boolean disjunction, on which we can perform case analysis using the `case` tactic with the `orP` view. Each of the two resulting cases happen to be the premise of the induction hypotheses, `H1` and `H2`; thus the disjunctive pattern `[/H1|/H2]` will rewrite the goal to:

```
eval e1 == 0 → eval e1 * eval e2 == 0
```

and similarly for `e2`. The pattern `/eqP→` will rewrite `eval e1` to 0 in the goal obtaining a goal of $0 * \texttt{eval e2} == 0$.

This particular step is paradigmatic of the "proof by rewriting" technique: from a property `a = b`, we can replace all `a` terms occurring in the goal by `b`. The final step of the proof is again by rewriting, this time using the `mul0n` and `muln0` lemmas part of the `ssrnat` library, with type `mul0n : forall x, 0 * x = 0` and symmetrically.

Rewriting Magic SSReflect's emphasis on boolean expressions fosters proof by rewriting in arithmetic proofs. For instance, in the following lemma:

```
Lemma leq_2add :
  forall (x y : nat), x <= y → x + x <= y + y.
Proof. by move=> x y xy; rewrite leq_add. Qed.
```

the proof, where ";" combines proof steps, is done by rewriting with the `leq_add` lemma, provided by the `ssrnat` library, of type:

```
leq_add : forall (m1 m2 n1 n2 : nat),
   m1 <= n1 → m2 <= n2 → m1 + m2 <= n1 + n2
```

How could that happen? Equalities do not seem to occur in the conclusion of `leq_add`, but they have actually just been removed by the pretty printer. In fact, the exact conclusion is a boolean equality, namely `m1 + m2 <= n1 + n2 = true`, as the resulting type of the `<=` operator is boolean. Thus, `x + x <= y + y = true` can be easily matched and rewritten to true with proper bindings of `leq_add` variables, leading to the goal `true = true`.

Going Further It is obviously impossible to give a complete overview of SSReflect in two pages. In particular, SSReflect advocates particular conventions and coding styles that we did not fully follow for the sake of space and pedagogy. In particular, the idiomatic proof for Theorem `eval0eB` is:

```
by elim=> //= ? IH1 ? IH2 /orP
   [/ IH1|/IH2] /eqP → ; rewrite ?(mul0n, muln0).
```

The Mathematical Components library is a companion project to SSReflect, and includes extensive libraries about finite groups, algebra, number theory and more, which have been used to formally prove significant large theorems like the Four Color or Feit-Thompson theorems.

3 The Faust Audio Language

Faust— which stands for "Functional Audio Stream" — is a DSP language [Orlarey et al., 2004]. Faust's main focus is the fast development of efficient digital audio programs, and has been used in live performances and offline and online audio applications, such as FaustLive [Denoux et al., 2014] or moForte [4]. It has also been used in other signal processing contexts [Barkati et al., 2014].

[4] `www.moforte.com/`

Framework FAUST programs are structured in two layers. The high-level layer is a macro language corresponding to an untyped, full-fledged functional language. It is used to generate programs written in the *Core* FAUST syntax, where only a few primitives are available. In order to deal with the real-time constraints imposed by audio processing, FAUST programs are then optimized and compiled towards an efficient language; FAUST's main compiler generates `C++` code, while alternative backends can now generate other formats, such as `asm.js` code.

While the core language is unpractical for writing real applications, it contains enough primitives for the rest of this paper. Moreover, it enjoys a strong type system and operational semantics. For a more precise description of FAUST syntax and semantics, we refer the reader to [Jouvelot and Orlarey, 2011].

Semantics *Signal processors* are FAUST's key components. A signal is a (potentially infinite) stream of amplitude values. In the discrete case, we can represent signals as functions from (discrete) time to real numbers. Formally, we write $\mathbb{S}$ for the function space $\mathbb{N} \to \mathbb{R}$. We assume that signals have amplitude 0 when time is negative. Signal processors from i inputs to o outputs are functions in $\mathbb{S}^i \to \mathbb{S}^o$, and every valid FAUST program can be interpreted as a function of this type. We use "semantics brackets" to denote the function that maps a FAUST program to its mathematical interpretation. For instance, the *delay* processor, which delays its input signal by 1 audio sample, is denoted as:

$$[\![\texttt{delay}]\!](i)(n) = i(n-1).$$

The first argument i is the signal to be delayed, and n represents time. Then, `delay` will simply return the sample from i corresponding to the previous time value.

Core FAUST For the purposes of this paper, we focus on a minimal but functional subset of FAUST consisting of three particular signal processor-building blocks: primitives, composition, and feedback.

Examples of primitive signal processors are $+$, which takes as input two signals and outputs a signal with amplitude the sum of the input signal amplitudes, or $*(\texttt{c})$, that scales the amplitude by a constant factor c. Formally:

$$\begin{aligned}[\![+]\!](i_1, i_2)(n) &= i_1(n) + i_2(n)\\ [\![*(c)]\!](i_1)(n) &= c * i_1(n).\end{aligned}$$

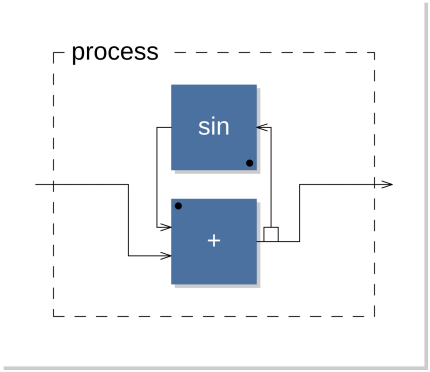


Figure 1: Simple Feedback in FAUST

We write $\texttt{f} : \texttt{g}$ for the composition of signal processors `f` and `g`:

$$[\![\texttt{f} : \texttt{g}]\!](i) = [\![\texttt{g}]\!]([\![\texttt{f}]\!](i)).$$

The interesting construction is feedback, written as $\texttt{f} \sim \texttt{g}$. In this case, `f` is assumed be a processor from two signals to one, and `g`, a unary signal processor. Then, $\texttt{f} \sim \texttt{g}$ represents the 1-delay feedback loop through `g`.

$$[\![\texttt{f} \sim \texttt{g}]\!](i) = [\![\texttt{f}]\!]([\![\texttt{g}]\!]([\![\texttt{f} \sim \texttt{g}]\!]([\![\texttt{delay}]\!](i))), i).$$

Note that the definition of the semantics of feedback is recursive. For example, the FAUST program $+ \sim \texttt{sin}$ is depicted in Figure 1.

A Simple Low-Pass Filter For the rest of the paper, we will work with a simple low-pass filter written in FAUST as:

$$\texttt{smooth(c)} = *(1 - \texttt{c}) :+ \sim *(\texttt{c}).$$

`smooth` is intended to be used with a coefficient c in the interval $[0, 1]$. If c is 0, then the filter has no effect, whereas as we increase c the cutoff frequency decreases, with a limit case of $c = 1$, that outputs a constant signal. The filter first multiplies the input amplitude by $1 - c$, then to perform 1-sample additive feedback with coefficient c. Its block diagram with $c = 0.9$ is drawn in Figure 2.

While this filter may not be very adequate for audio, due to its frequency response curve, it is useful for instance for smoothing control parameters, and for other applications with high-frequency components. A key property of filters is *stability*. That is to say, we expect `smooth`'s output amplitude to remain in bounds that depend on the input. An excessive amount of feedback could cause the filter to behave badly.

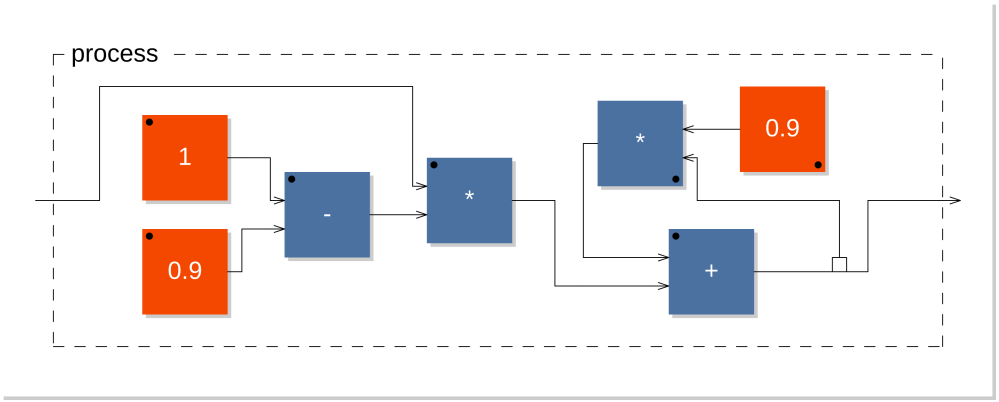


Figure 2: The `smooth` Low-Pass Filter

Stability also helps in compilation, as bounds known in the input signal can be propagated to the output, helping with buffer allocation and other issues.

4 Formalizing Faust in Coq

In the previous section, we described a core subset of Faust *on paper*. In this section, our goal will be to replicate this description inside a mechanized framework, Coq. This will enable us to later *mechanically reason* about Faust programs, avoiding a lot of potential error sources and getting strong confidence on the soundness of our reasoning.

Overview Mechanized reasoning about programs requires to encode their behavior or semantics in the particular theorem prover of choice. In a sense, this is equivalent to defining an *interpreter*; however the idiosyncrasy of theorem proving often makes the process quite different from writing a regular interpreter or compiler for our language.

Once the semantic representation is in place, we can wonder whether two programs have the same behavior (read: semantics), what happens when the input does not meet certain criteria, etc.

The concrete tasks we need to complete in order to start reasoning about Faust programs in Coq are: a) define a representation of Faust syntax in Coq; b) define a representation for signals, or streams, in Coq; c) write a function that takes a program's AST and returns a stream processor.

Once this is done, we can start proving! However, a key point in theorem proving is *how convenient* will it be to write proofs. We would have a hard time justifying formalized reasoning if we needed thousand of hours and lines of proof to perform trivial reasoning. We will address this point in Section 5, while devoting the rest of this one to explain how Faust is embedded into Coq. All the Coq code and examples can be downloaded from `https://github.com/ejgallego/mini-faust-coq/`.

Faust AST in Coq We will encode the *syntax* of the program using an *algebraic* or *inductive data type*. ADT (Abstract Data Types) are one of the most powerful features of Coq, allowing the user to define new richly-typed datatypes. In Section 2, we saw an example of an ADT for expressions. Here, we will make use of an extra feature known as *indexed* or dependently-typed ADT. Thus, we will encode Faust expressions using the `fterm` datatype, which carries additional information about the number of input and output signals of the program:

```
Inductive fterm : nat → nat → Type :=
| mult : num → 1 ⇝1
| plus : 2 ⇝ 1
| comp : 1 ⇝ 1 → 1 ⇝1 → 1 ⇝1
| feed : 2 ⇝ 1 → 1 ⇝1 → 1 ⇝1
where "i ⇝o" := (fterm i o).

Notation "'+" := plus.
Notation "'*(c)" := (mult c).
Notation "f : g" := (comp f g).
Notation "f ~ g" := (feed f g).
```

Thus, a program of type `i ⇝ o` will exactly go from i to o channels. Note that the constructors that correspond to the primitives enforce this requirement. In particular two signal processors can be composed only if they have the right types; thus no ill-typed Faust program can be built. We also define some notations to make display nicer.

Now, we can define our simple low-pass filter as follows:

```
Def smooth c : 1 ⇝1 := '*(1 - c) : '+ ~ '*(c).
```

Streams in Coq Once we have defined the syntax of our Faust programs, the next step is to define their semantics. We will encode signals as finite-length sequences of reals. We could have used several other representations, but it is beyond the scope of this paper to discuss the advantages of this particular definition.

We will index signals by their length, using the SSReflect type `n.-tuple R`, the type of sequences of exactly n reals. We write `'S_n` for `n.-tuple R` to shorten notation. Signal processors are encoded using regular Coq functions:

```
Notation "'S_n" := (n.-tuple R).
Notation "'SP(i,o)" := (∀ n, 'S_n^i → 'S_n^o).
```

For instance, the type for signals with three samples is `'S_3`. A signal processor (of type `"'SP(i,o)"`) must be able to process signals of arbitrary length; thus we quantify the second definition on all lengths n. We write `'S_n^k` for `k` copies of `'S_n`, e.g., `'S_n^2` is (`'S_n * 'S_n`).

Interpretation Function With both syntax and semantics in place, we can define a function linking the two worlds. In our case, a Faust expression of type `i ⇝ o` will be interpreted by a Coq function of type `∀ n, 'S_n^i → 'S_n^o`. Our interpreter I will thus have type:

```
I : i ⇝ o → 'SP(i, o)
```

Given a program `f`, the resulting function `I f` is *effective*, that is to say, given input signals, it computes the corresponding output ones. In particular, `I f n` formally corresponds to the semantic brackets introduced in Section 3, restricted to the first n samples of the signal. We write $[\![f]\!]_n$ for this truncated semantics.

How is I defined? Definition for primitives and composition is straightforward; the most interesting case is the feedback:

```
Definition I_feedback f g n i :=
  iter n (fun fb => f (g (x0 :: fb), i)) [::].
```

where `x0` is the initial value for the feedback loop, usually 0, and `iter n f x` is the function that applies `f n` times to `x`.

Note that this function outputs a signal of size n when the input is of size n, and does so by computing the feedback for n steps. The reader familiar with signal processing will notice that this implementation is extremely inefficient, as it may take quadratic time even for simple programs! Indeed, the goal of our interpretation is not to achieve efficiency, but to have a convenient representation of the semantics in order to reason about it. Usually, efficient implementations are not very well-suited for reasoning and vice-versa. The usual remedy when we care about efficiency inside Coq is to define two implementations, and prove them equivalent [Dénès et al., 2012].

First Steps in Proving With those ingredients, we can start reasoning about programs. For instance, a proof of the fusion property of the multiplication stream processors is:

```
(* Fusion of mult *)
Lemma multF : I ('*(a) : '*(b)) = I ('*(a*b)).
Proof.
  apply: val_inj; case: i => s /= _.
  by elim: s => //= x s → ; rewrite mulrCA mulrA.
Qed.
```

The proof is straightforward, by induction, associativity, and commutativity of the multiplication operator of the real numbers. However, some amount of boilerplate is necessary to set up the induction, task that can get tricky with more complex programs. Indeed, this inductive proof method is common to most proofs; thus we will identify common patterns and will define higher-level reasoning principles that allow us to prove things with less effort in the next section.

5 Structured Reasoning: A Sample Logic

As we just saw in Section 4, the Coq semantics allow us to state — and attempt to prove — almost any property imaginable about Faust programs. However, in most cases, reasoning can be repetitive, long and error-prone. That is the price we have to pay for accessing such a power.

A key observation is that proofs of certain classes of properties share common parts, while only a minor part of the proof actually depends on the property. As we saw in the previous fusion case, the relevant part of the proof is less than 10% of its total code.

Imagine a property φ, supposed to hold for all samples of a signal. Then, it is enough to define φ as a predicate over one sample, and we can "automatically" lift the predicate over signals, checking that φ holds for all time.

Indeed, to illustrate the principles of high-level structured reasoning over programs, we will focus on such "sample-level" properties in this section. While we will sacrifice quite a bit of expressivity, by limiting our language to one-sample statements, this will still be enough to carry out proofs of stability and will significantly

facilitate our proofs, allowing us to proceed in a short and structured way.

Sample-level Properties For our purposes, a predicate over a sample is a function from reals to booleans, $\varphi, \psi \in \mathbb{R} \to \mathbb{B}$, or, in Coq, `P, Q : R → bool`. Then[5], we say a property φ holds for a signal s if $\forall n.\varphi(s[n])$; that is, for all time moments n, the sample meets φ. In Coq, we can use the `all` function for sequences, thus writing `all P s`. For instance, the property that a signal is bound by the interval $[a, b]$ is defined as $\varphi(x) = x \in [a, b]$, or in Coq as `P := fun x => x \in `[a,b].

It makes sense to extend our properties to signal processors. In this case, we would like to relate properties over input signals with properties over the output signals. Given sample-level properties (φ, ψ) and input and output signals (i, o), a reasonable statement could be: "if the input signal i satisfies φ, then o should satisfy ψ."

If we think of our previous "being in an interval" property, its extension to signal processors allows us to capture stability. Indeed, we can precisely state now: "if the input signal is bounded, then the output signal will be too."

Judging the Sampling The previous relation between input and output signals and their properties constitutes an instance of a "high-level" reasoning principle. It is highly convenient thus to encode the fact that a signal processor satisfies the property as a "judgment." Our judgments will be of the form, $\{\varphi\}\ f\ \{\psi\}$, with intended interpretation $[\![\{\varphi\}\ f\ \{\psi\}]\!]$ such that, for all input signals with samples satisfying φ, all the output samples of f satisfy ψ. Formally:

$$[\![\{\varphi\}\ f\ \{\psi\}]\!] \iff \forall i.(\forall t.\varphi(i(t))) \implies (\forall t, \psi([\![f]\!])(i)(t))$$

the Coq version is expressed in a slightly different way, using `all`:

```
Definition '[[ { P } f { Q } ]] :=
  ∀ n (i : 'S_n), all i P ==> all (I f n) Q.
```

In the case of i input and o output signals, judgements are extended pointwise to use one predicate per signal: $[\![\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\varphi_1, \ldots, \varphi_o\}]\!]$.

Reasoning Rules Now, we'd like to have a system of rules to determine when a judgment is valid without resorting to analyzing its semantics.

[5]From now on, we will interchangeably use Coq and mathematical notation where no confusion can arise, omitting double definitions.

The standard way to achieve this goal is to introduce a "logic", or a set of rules to infer validity of judgments, and, by extension, of their intended properties. The form of a rule is

$$\frac{A_1 \qquad \ldots \qquad A_n}{B}$$

meaning that, if $A_1, \ldots, A_n$ are valid, then B also is. This way, we can hopefully reduce validity checking for B to smaller problems.

The rules of our particular system for sample-level reasoning are shown in Figure 3. Rule *Prim* is an example of a base rule, stating that a judgment about a primitive is valid if its semantics is. Rule *Comp* allows to reduce the verification of composition to the verification of its individual parts; a judgment about composition is valid if there are valid judgments about the individual signal processors such that the property of the output of f implies the required property for the input stream of g.

The *Feed* rule is quite similar to the composition rule: the internal state of the feedback should obey an invariant θ, and samples from the feedback output should be compatible with the requirements of g's input. We also require that the initial value x_0 satisfies ψ.

Now, all that remains is to check that the rules are sound, that is to say that validity of the premises implies the validity of the conclusion, and we can reason using the newly defined logic:

Theorem 5.1 (Soundness)**.** *For any program f of type $i \rightsquigarrow o$, if $\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\psi_i, \ldots, \psi_o\}$ is derivable, $[\![\{\varphi_1, \ldots, \varphi_i\}\ f\ \{\psi_i, \ldots, \psi_o\}]\!]$ is valid.*

Proof. We proceed by induction on the derivation. The base case is the *Prim* rule, and proof is immediate. For *Comp* soundness automatically follows by induction hypotheses. For *Feed*, we apply induction on the length of the input signal, plus induction hypotheses. □

6 Case Study: Filter Stability

As a case study, we will verify that the `smooth` filter of Section 3 is stable, that is, if the input amplitude is bounded, the output amplitude is.

Assume a well-formed interval $[a, b]$, including 0, and $c \in [0, 1]$. In Coq:

```
Hypothesis (Hab : a <= b).
Hypothesis (H0ab : 0 \in `[a, b]).
Hypothesis (Hrc : c \in `[0, 1]).
```

then, we will prove:

$$\{i \in [a, b]\}\ \texttt{smooth(c)}\ \{o \in [a, b]\}$$

$$\frac{\forall i_1, i_2, (\forall t. \varphi_1(i_1(t)) \wedge \varphi_1(i_2(t))) \implies (\forall t. \psi(i_1(t) + i_2(t)))}{\{\varphi_1, \varphi_2\} + \{\psi\}}\,Prim$$

$$\frac{\{\varphi\}\ f\ \{\theta\} \qquad \{\theta\}\ g\ \{\psi\}}{\{\varphi\}\ f : g\ \{\psi\}}\,Comp \qquad \frac{\models \psi(x_0) \qquad \{\theta, \varphi\}\ f\ \{\psi\} \qquad \{\psi\}\ g\ \{\theta\}}{\{\varphi\}\ f \sim g\ \{\psi\}}\,Feed$$

Figure 3: A simple logic for FAUST program verification

$$\frac{\dfrac{\square}{\{I_{ab}\} *(1-c)\ \{I_{ab\bar{c}}\}} \qquad \dfrac{\dfrac{\square}{\{I_{abc}, I_{ab\bar{c}}\} + \{I_{ab}\}} \qquad \dfrac{\square}{\{I_{ab}\} *(c)\ \{I_{abc}\}}}{\{I_{ab\bar{c}}\} + \sim *(c)\ \{I_{ab}\}}}{\{i \in [a,b]\} *(1-c) : + \sim *(c)\ \{o \in [a,b]\}}$$

with:

$$I_{ab}(x) \equiv x \in [a,b] \qquad I_{abc}(x) \equiv x \in [a*c, b*c] \qquad I_{ab\bar{c}}(x) \equiv x \in [a*(1-c), b*(1-c)]$$

Figure 4: Derivation for `smooth`

Let us recall the definition of smooth:

```
smooth(c) = *(1-c) : + ~*(c)
```

then, we should apply rule *Comp*, with $\theta(s) = s \in [a*(1-c), b*(1-c)]$. Using *Prim* gets us to a first obligation, shown in COQ as:

```
Hi : i \in '[ a, b]
============================
i * (1 - c) \in '[(a * (1 - c)), (b * (1 - c))]
```

which can be proved using the libraries by:

```
by rewrite ?itv_boundlr /= ?ler_wpmul2r
     ?ler_subr_addr ?add0r ?Hrc ?(itvP Hi).
```

The next step is to apply *Feed*, choosing $\theta(s) = s \in [a*c, b*c]$. Then, we apply *Prim* twice to get the obligations for + and *(c):

```
H1 : i1 \in '[( a * c), (b * c)]
H2 : i2 \in '[( a * (1 - c)), (b * (1 - c))]
============================
i1 + i2 \in '[a, b]
```

solved by:

```
have Ha: a = a * c + a * (1 - c)
    by rewrite -mulrDr addrC addrNK mulr1.
have Hb: b = b * c + b * (1 - c)
    by rewrite -mulrDr addrC addrNK mulr1.
by rewrite itv_boundlr /= Ha Hb
        !ler_add ?(itvP H1) ?(itvP H2).
```

where `have` introduces a local lemma, and

```
Hi : i \in '[ a, b]
============================
i * c \in '[( a * c), (b * c)]
```

solved by:

```
by rewrite itv_boundlr /=
        ?ler_wpmul2r ?(itvP Hi) ?Hrc.
```

We have chosen to prove the arithmetic obligations manually, but we should remark that there exists tools that can prove this kind of results automatically. The full derivation is in Figure 4.

7 Conclusion

We presented a case for the use of developer-assisted formal reasoning tools in the field of computer music and, more generally, audio DSP. We gave a quick tour of the COQ/SSREFLECT environment, which we believe can be particularly well fitted to reach such a vision. We applied our approach to the FAUST audio signal processing, providing a formal semantics for its core and detailing how a property of a filter can be proven using a specific logic designed for FAUST.

Future work will tackle other applications in the audio processing domain to assess our tool, together with the development of specific DSP mechanisms within COQ/SSREFLECT (tactics, tacticals, or even a dedicated DSP library). We would also be interested in seeing how our system can be of help to prove more foundational theorems such as the Shannon Sampling Theorem.

8 Acknowledgements

We want to thank Yann Orlarey and Arnaud Spiwack for their insightful comments. Partial funding for this research was provided by the ANR FEEVER Project.

References

Karim Barkati, Haisheng Wang, and Pierre Jouvelot. 2014. Faustine: A Vector Faust Interpreter Test Bed for Multimedia Signal Processing - System Description. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 69–85. Springer.

Yves Bertot and Pierre Castéran. 2004. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer.

Aloïs Brunel, Marco Gaboardi, Damiano Mazza, and Steve Zdancewic. 2014. A core quantitative coeffect calculus. In Zhong Shao, editor, *Programming Languages and Systems*, volume 8410 of *Lecture Notes in Computer Science*, pages 351–370. Springer Berlin Heidelberg.

Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. 2012. Continuity and Robustness of Programs. *Commun. ACM*, 55(8):107–115, August.

Maxime Dénès, Anders Mörtberg, and Vincent Siles. 2012. A refinement-based approach to computational algebra in COQ. In Lennart Beringer and Amy Felty, editors, *ITP - 3rd International Conference on Interactive Theorem Proving - 2012*, volume 7406 of *Lecture Notes In Computer Science*, pages 83–98, Princeton, USA. Springer, Springer.

Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober. 2014. FAUSTLIVE, Just-In-Time Faust Compiler... and much more. In *Linux Audio Conference*.

Naghmeh Ghafari, Ramana Kumar, Jeff Joyce, Bernd Dehning, and Christos Zamantzas. 2011. Formal Verification of Real-time Data Processing of the LHC Beam Loss Monitoring System: A Case Study. In *Proceedings of the 16th International Conference on Formal Methods for Industrial Critical Systems*, FMICS'11, pages 212–227, Berlin, Heidelberg. Springer-Verlag.

Georges Gonthier, Assia Mahboubi, and Enrico Tassi. 2008. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA.

Pierre Jouvelot and Yann Orlarey. 2011. Dependent vector types for data structuring in multirate Faust. *Computer Languages, Systems & Structures*, 37(3):113–131.

Neelakantan R. Krishnaswami. 2013. Higher-order functional reactive programming without spacetime leaks. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 221–232. ACM.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Comput.*, 8(9):623–632.

Anis Souari, Amjad Gawanmeh, Sofiène Tahar, and Mohamed Lassaad Ammari. 2014. Design and verification of a frequency domain equalizer. *Microelectronics Journal*, 45(2):167–178.

Music and Art Program

(Please make sure to check http://lac.linuxaudio.org/2015/program for changes in the program after the editorial deadline of this publication.)

Concert #1 Space / Landscape of Sound

Hochschule für Musik, Roter Saal, Thursday, April 9, 20:00

Topos Concrete

Clemens von Reusner

The territory (gr. topos) is a rough and harsh landscape with mountains, valleys, canyons and plains, sand and stones, though it appears evenly and smooth. The color is grey. The size is about 30 square-meters. It is the floor of a garage and it is made of concrete (engl.) / Beton (german).

Concrete is a building material, a kind of unshaped dry powder made of sand, granulated stones and cement, dusty and chaotic. Mixed with water it becomes flexible and fluid and goes into a metamorphosis to become dry again, static and resistable and of any wanted shape. Aspects of working with native granularity, fluidness as well as stiffness and different kind of acoustic spaces were leading ideas of the composition. Software: Csound, Sox, SuperCollider. The csound 3rd-order ambisonic opcodes by Jan Jacob Hofmann were used for multichannel spatialization. – Premiere –

Clemens von Reusner (b. 1957) is a composer of electroacoustic music based in Germany. After studying musicology and music-education, drums with Abbey Rader and Peter Giger he has worked as a composer and a musician in different ensembles as well as a lecturer, music teacher and an author.

Klinga

Helene Hedsund

The sound material in Klinga is made up of recordings of an old saw blade being hit and scraped by various objects. The piece, and all sound processing is made in an instrument built by myself in SuperCollider.

Helene Hedsund is an electroacoustic composer, currently doing a Ph.D in Birmingham, UK. During the last 20 years she has been active at EMS (www.elektronmusikstudion.se) in Stockholm, Sweden. She has a background as a programmer and as a musician in non commercial rock bands.

Benjolin

Patrick Gunawan Hartono

Benjolin is an electroacoustic composition for eight-channel speakers, in which all the sound materials that have been used are the result of the Benjolin synthesizer designed by synth pioneer Rob Horddijk. The compositional structure is intuitive based on gradual and dramatic dynamic changes, which are represented by random pulses and square waves as offered by Benjolin. An open-source 3D sound algorithm was implemented for the spatialization structure of this piece, which has been realized in the SuperCollider environment.

Patrick Gunawan Hartono: Born in Makassar 1988, young Indonesian electroacoustic composer, intermedia artist, member of Awahita Nusantara, whose art and musical interest is to use technology and scientific approach as creativity tools. He also interested in 3-D sound spatialisation, analog/digital synthesis, psychoacoustics, and visual music.

Voce231114
Massimo Fragalà

All the sounds that form this composition derive from the elaboration of word splash that I recorded myself. Starting from this sample I tried to change the physical characteristics in order to generate a range of sounds more or less different compared to their original variety. This was possible using particular techniques of sound processing such as waveset distortion, brassage stretching, segmenting the sound and reassembling segments, reverberation, etc. This composition has been realized on Linux KXStudio.

Massimo Fragalà graduated in Electronic Music and in Classical Guitar. His music has been selected and performed in many festivals and conferences worldwide including ICMC 2003, ICMC 2005, Festival Zèppelin 05, EAR Sounds Electric 2005, LAC06, ICMC 2006, Festival Mùsica Viva 2008 (Sound Walk), NWEAMO 2008, Taukay FrammentAzioni 2008, Vox Novus 2008 (60x60 project), LAC 2011, Emufest 2011, 60x60 2012, PianoForte Mix, Csound Conference 2013, etc. One of his electroacoustic compositions has been published on CD by Electronic Music Foundation (EMF).

Concert #2 Time / Sound Machines
Hochschule für Musik, Roter Saal, Friday, April 10, 20:00

speaking clock
Mari Ohno

This work is an electroacoustic composition created with the recordings of speaking clocks in various sites around the world. A speaking clock is a tool of sonification of "time", a phenomenon people cannot hear. It has various expressions of time depending on the country or region. In this work, the music mixes various expressions of time, based on the concept of "the expression of time perception". Through this work, I attempt to give listeners curious and unique feelings through the same sound experience depending on their cultural background.

Mari Ohno is a sound artist, composer and sound designer, based in London and Tokyo. She was born in 1984 in Tokyo. She is studying at the MFA in Computational Studio Arts at Goldsmiths, University of London, after graduated from the MA in Creativity in Music and Sound at Tokyo University of the Arts. Her works are primarily focused in the areas of sound installation and electroacoustic composition, exploring various dimensions of human perception. In addition to her own work, she has also collaborated with other artists in composition and sound design for films.

Selva di varie intonazioni
Michele Del Prete

Csound, tape music, 8 channel, 9'.54", 2013
The piece is based on concrete sounds I recorded in the Frari church in Venice, a building that hosts two notable organs of the XVIII century. I have recorded their stops (as well as key noises) individually and in several combinations, thus already obtaining a very rich timbral material I have later worked on exclusively in Csound.

Michele Del Prete studied Philosophy in Berlin, Electronic Music in Venice (with Alvise Vidolin) and Composition in Graz (with Beat Furrer). He is currently working on spatialisation models rooted in the polychoral practice of the Renaissance Venetian school composing tape music and works with acoustical instruments and electronics. He is also founder of Pas-e, association for contemporary and electronic music in Venice.

The Hidden and Mysterious Machinery of Sound
Fernando Lopez-Lezcano

I had been thinking about a piece that used simple sine oscillators in complex ways for a long time. Going back to basics, in a sense. With that idea in mind I dived into the innards of Bill Schottstaedt's new Scheme-based version of the CLM synthesis language and its s7 interpreter, and stumbled into new ugens that allowed me to pile sinewaves in many different ways. The "imaginary machines" examples I found there also helped shape the first code fragments I experimented with. What remains of many lines of discarded Scheme code is the program that writes this piece. It creates fractal machines that manipulate clockwork mechanisms, big and small "virtual gears" that interlock, work without pause, and drive the basic sound synthesis instruments. This universe of miniature machines is spread over 3D space using Ambisonics, and the resulting soundscape is made of interlocking patterns of sound that drift through space. WARNING: the piece contains repetitive phrases of sound (also known as "rhythms"), and is only intended for immature audiences.

Fernando Lopez-Lezcano enjoys building things, fixing them when they don't work, and improving them even if they seem to work just fine. The scope of the word "things" is very wide, and includes computer hardware and software, controllers, music composition, performance and sound. His music blurs the line between technology and art, and is as much about form and sound synthesis and spatialization, as about algorithms and custom software he writes for each piece. He has been working in multichannel sound and diffusion techniques for a long time, and can hack Linux for a living. At CCRMA since 1993, he combines music, electronic engineering and programming with his love of teaching, composition and performance. He discovered the intimate workings of sound while building his own analog synthesizers a very long time ago, and even after more than 30 years, "El Dinosaurio" is still being used in live performances. He was the Edgar Varese Guest Professor at TU Berlin during the Summer of 2008.

Coloured Dots And The Voids In Between
Jan Jacob Hofmann

In the piece "Coloured Dots And The Voids In Between" spatial textures of dot-like sounds occur. The fields created by that expand and evolve in space and time. Important are not only the events of sounds themselves but also the spaces in between these, which expand in different dimensions spatially and temporally, overlap and thus create the actual space. All sounds have been generated using solely the "pluck"-opcode, which simulates the sound of a plucked string.

The piece is spatially encoded in 3rd order Ambisonic and has been created with the program "Csound" and "Cmask" along with Steven Yi's environment for composition "blue" using the self-conceived editor for spatial composition "Spatial Granulator".

Jan Jacob Hofmann: Born 1966 in Germany. Diploma, branch of architecture at the University Of Applied Sciences at Frankfurt am Main. Entered the class of Peter Cook and Enric Miralles at the Staedelschule Art-School Frankfurt am Main for conceptual design and architecture. Diploma at the Staedelschule in 1997. Since 1986 dealing with composition and electronic music. Since 2000 work on spatialisation of sound. Development and publication of Csound based tools for spatialisation via 3rd order Ambisonic. Latest development is an instrument for spatial granular synthesis via Ambisonic 3rd order. http://www.sonicarchitecture.de

Live Performance: Embedded Artist
Wolfgang Spahn, Malte Steiner
Philosophicum, Fakultätssaal, Friday, April 10, 21:30

Industrial noise and digital sound processing meets multi space projections: "Embedded Artist" is a media performance by Malte Steiner and Wolfgang Spahn. The performance combines four

different layers of visions that are merging into one visual Gesamtkunstwerk: 3D models, video scratching, live camera, and mechanical effects are likewise projected within the space. “Embedded Artist” is not only projecting all over the walls, but it is filming the audience and the space and re-projecting those images. Moreover, the light beam of each projector is fractionized by a prism and therefore sends broken images onto the walls. For the performance both artists developed a system for multiple embedded systems. To achieve this the following software and programs are used: Pure Data, Raspberry Pi, Raspian, and Python. As hardware components several Raspberry Pi's were combined with Paper-Duino-Pi's and remote controlled via OSC from the performers laptops.

Wolfgang Spahn is a visual artist based in Berlin. His work includes interactive installations, videos, projections, and miniature-slide-paintings. He studied mathematics and sociology in Regensburg and Berlin. He currently teaches at Medienwerkstatt BBK Berlin, and is associated lecturer at the University of Oldenburg, Institute of Art and Visual Culture.

Malte Steiner is media artist, sound designer and software developer. Started with electronic music and visual arts around 1983, developing his own vision of the interdisciplinary Gesamtkunstwerk. Concerts, exhibitions, workshops and residencies in Europe, Asia and America. Organizes twice a month the Pure Data patching circle at c-base Berlin. http://www.block4.com

Concert #3 Live / Sound at Play

Hochschule für Musik, Roter Saal, Saturday, April 11, 20:00

Spielzeug #1 - poco a poco accelerando al sinus - for two Wii-Remotes

Jonghyun Kim

The main concept behind this piece is changing the repetition speed. I have taken a sound file and cut it into sections. The excerpts are repeated at different playback rates. This affects the sound quality and pitch. When the repetition rate is extremely fast, the output changes dramatically. When each repetition is under 10 milliseconds in length, the original sound is no longer recognizable, and only a sine wave-like timbre remains. This process modifies the micro-structure of the sound.

Technical Summary: This is a Wii-Remote live performance using granular synthesis. The granular-synth and algorithms was programmed in Pure Data. The performer uses Wii-Remotes in each hand and swings them in pitch and roll axis (gyroscope). The granular synthesis algorithm receives motion data from such movements of the controllers, and produces sound in realtime. The buttons on the Wii-Remote also trigger samples and changes performance mode.

Jonghyun Kim is composer and software developer. He studied composition, piano, and computer programming at Kyung Hee University in Seoul, Hochschule für Musik in Freiburg, HfM in Stuttgart (guest), and took part in several seminars including IRCAM Paris and the Darmstadt summer course. He is director of 'Open Source Art Forum', founder of 'Pure Data Korea' and 'Raspberry Pi Korea', developer of 'Good Metronome Pro' for iOS, a member of Seoul-based performance group 'Linux Computer Ensemble'. His pieces have been performed in ZKM Karlsruhe, HfM Freiburg, and SICMF (Seoul International Computer Music Festival) in Korea. Currently he is teaching computer music and composition at the Kyung Hee University in Seoul, and sound art at Kaywon University of Art & Design in Gyeonggi-do.

TBA

Matthias Grabenhorst, Jörn Nettingsmeier

Two improvisation sketches, played by Matthias and spatialized by Jörn. The plan is to do one free improvisation and one more song-oriented jazz tune, played on an electric guitar equipped

with a hex pickup to allow for individual spatialization of each string, both to give the guitarist an extra layer of freedom in the treatment of melody, and to allow us to "unwrap" complex voicings by spreading them out in space.

Dance I
Jaeseong You

The Dance Music series is now continuing with alphabetical index. More recent pieces tend to faithfully conform to IDM genre beyond simply borrowing its idioms and materials. In Dance I, the samples are initially put together in a random assemblage, and musical narratives and gestures are gradually extracted from such disorder. Some of the samples are created from SuperCollider in Linux (Ubuntu) and others are collected from both open source libraries such as freesound.org and commercial libraries.

Jaeseong You is a composer/researcher at Music & Audio Research Lab, Steinhardt, New York University, where You is currently serving as Editorial Manager at Journal SEMAUS and working under Dr. Tae Hong Park on Electro Acoustic Music Mine, Citygram, Urban Soundscape Event Classification, and Sound Beacon.

Linux Sound Night
Baron, Saturday, April 11, 22:00

Vagabundo Barbudo meets Listening Lights
Andres Perez-Lopez

Vagabundo Barbudo is an electro-experimental dance music project. Music is produced with free software tools, and distributed with copyleft licenses. Furthermore, "source" tracks are also distributed, in order to encourage modifications. Listening Lights is a project for automatic lighting of music. The core of the project is RTML, a graphical SuperCollider framework for Real-Time Music Information Retrieval.

In the performance, the music from Vagabundo Barbudo will be played, along with automatic reactive lighting provided by RTML.

Andrés Pérez-López (Valencia, 1987) is a music technologist and musician based on Barcelona. His scientific background is grounded on his Telecommunications Engineering (UPV) and Master in Sound and Music Computing (MTG, UPF) studies. Currently, he works as freelance technological developer for artistic performances; the SuperCollider libraries 3Dj (interactive 3D sound spatialization) and RTML (real-time music information retrieval) are examples of his recent work. Apart from that, Andrés is a modern music performer, and develops his interest into electronic music production and free culture in the Vagabundo Barbudo project.

Hallogenerator
Jakub Pisek

Performance sets up the mirror to producers, DJs and pop singers, who want to enhance their vocal digitally. But the performance brings to the stage the essence of entertainment and intellectual content in musical form at the same time. It also peacefully attacks the technological literacy of the nation, the jazz police, and computer reliability.

This live performance is running on open source software (Arch Linux, Radeon driver, Pure Data, Arduino...).

Superdirt² - cello & live linux electro
Vincent Rateau (FR), Daniel Fritzsche

Superdirt² - fascinating electro beats mixed in with virtuous performed cello sounds which give a result of a never achieved before dance ability! With Ras Tilo at the synthesizers and Käpt'n Dirt with the cello it provides a musical experience which is situated between drum'n'bass, house, dub, dubstep and even far beyond...

The Band: The two independent musicians knew each other through many musical projects and as flatmates, but are coming from totally different musical genres. While Ras Tilo (Vincent Rateau) got in place as a music producer and multi-instrumentalist, Käpt'n Dirt (Daniel Fritzsche) continued his classical cello studies with an open mind and ears to new musical genres. They are mixing musical genres, looping, synthesizing, improvising and never get tired of going beyond the thinkable knowledge of music. Their first full-length album is called "Algoriddims" - and licensed under a Creative Commons license.

visinin
William Light

Synthetic, electronic club music with an organic edge. Written and performed with Renoise, Monomes, and a variety of other software, both off-the-shelf and custom.

William Light has been writing music and software from an early age. He's big on Linux and big on audio, so, really, this is the perfect conference to catch him at! His musical style is influenced by everything from folky guitar to highly technical electronic music.

Pjotr & Bass
Pjotr Lasschuit, Bass Jansson (NL)

A Live performance where interaction between live-coded visuals and hybrid-trumpet are the key elements. The software used is Pure Data, Faust and OpenFrameworks.

Sound Installations and Demonstrations
Philosophicum, Fakultätssaal

Interactive virtual audio-visual concert simulation with TASCAR (software demonstration)
Giso Grimm, Joanna Luberadzka, Tobias Herzke, Volker Hohmann
Thursday, April 9 15:30

TASCAR is a toolbox for acoustic scene creation and rendering. In this demonstrator three concert stages can be interactively explored in a virtual audio-visual environment. On the first stage, a time varying physical feedback model with three moving microphones and three simulated loudspeakers is placed. This simulation without any external sounds results in a sound experience very similar to Steve Reich's "pendulum music". On the second stage a jazz band is playing. The third musical event is a contemporary piece with moving virtual sources, performed on a viola da gamba ensemble.

The interactive virtual acoustic environment is rendered in real-time and illustrates different aspects of the rendering software as well as the effect of binaural hearing. The simulation is rendered 3rd order horizontal Ambisonics. It will be reproduced in an 8-channel loudspeaker setup. This demonstration complements the TASCAR paper in the main track.

The Sound Of People (installation)
David Runge
Friday, April 10 10:00

Single-nucleotide polymorphisms (SNPs) are molecular DNA markers, that are actively being researched in general science all over the world right now. Companies like 23andMe (and others) offer sequencing parts of their customers genome (and afterwards sends that collection of data to them). To free these sets of data for use in open science, openSNP started its work on collecting, indexing and making available their users' uploads. The Sound of People was one of the first attempts of synthesizing sounds from them. It was written in the SuperCollider audio synthesis language and has been developed with the help of the Electronic Studio of TU Berlin. The software creates a unique audio experience for up to twelve speakers.

acoustic cluster (video presentation)
Mari Ohno
Friday, April/10 14:00

A number of pipes of different lengths suspended within a space each contain a microphone and are equipped with a freely movable speaker assembly beneath them. The distance between each speaker assembly and microphone is expressed in the "howling" acoustic response. Having divided the space with pipes, moving the speaker assemblies closer to the spaces within the pipes amplifies the otherwise insignificant howling in the space outside the pipes, producing a sound like that of a wind instrument. The pitch of these responses varies with the spatial properties of each pipe. This series of phenomena seeks to make audible the normally inaudible material of space. (This is a video presentation of the physical installation.)

Mari Ohno is a sound artist, composer and sound designer, based in London and Tokyo. She was born in 1984 in Tokyo. She is studying at the MFA in Computational Studio Arts at Goldsmiths, University of London, after graduated from the MA in Creativity in Music and Sound at Tokyo University of the Arts. Her works are primarily focused in the areas of sound installation and electroacoustic composition, exploring various dimensions of human perception. In addition to her own work, she has also collaborated with other artists in composition and sound design for films.

3Dj (software demonstration)
Andrés Perez-Lopez
Saturday, April 11 11:30

This is a demonstration of the 3Dj SuperCollider framework for real-time sound spatialization. In the demonstration, we will explore several metaphors for sound diffusion, as algorithmic spatialization, direct sound position control through orientation sensors, or spatialization based on Music Information Retrieval. Sonic material will be both existing fixed media and live produced electroacoustic sound.

Andrés Pérez-López (Valencia, 1987) is a music technologist and musician based on Barcelona. His scientific background is grounded on his Telecommunications Engineering (UPV) and Master in Sound and Music Computing (MTG, UPF) studies. Currently, he works as freelance technological developer for artistic performances; the SuperCollider libraries 3Dj (interactive 3D sound spatialization) and RTML (real-time music information retrieval) are examples of his recent work. Apart from that, Andrés is a modern music performer, and develops his interest into electronic music production and free culture in the Vagabundo Barbudo project.

C/K/P (installation)
Adam Neil
Saturday, April/11 14:00

In C/K/P, you will see three presentations of three video 'panels,' which change position in each presentation. The sounds associated with each panel are present throughout, but are brought forward in the mix when their panel takes the center position.

Adam Scott Neal (b. 1981, Atlanta) is a composer, video artist, and improviser. His theoretical research focuses on modes of practice in experimental music, music and philosophy, and popular music analysis. He earned a PhD at the University of Florida and previous degrees at Queen's University Belfast and Georgia State University. Adam has enjoyed over 100 performances of his music in 23 states, as well as the UK, Canada, China, Italy, Mexico, Slovenia, and Switzerland. He is Production Manager for the Charlotte New Music Festival and one of the Artistic Directors of Terminus Ensemble (Atlanta).