UNIVERSITÉ JEAN MONNET
SAINT-ÉTIENNE
CIEREC

GRAME
CENTRE NATIONAL DE CRÉATION MUSICALE

Linux Audio Conference 2017

Conferences - Workshops- Concerts - Installations

May 18-21, 2017

Saint-Etienne University

Proceedings

Foreword

Welcome to Linux Audio Conference 2017 in Saint-Etienne!

The field of computer music and digital audio is rich of several well-known scientific conferences. But the Linux Audio Conference is very unique in this landscape! Its focus on Linux-based (but not only) free/open-source software development and its friendly atmosphere are perfect to speak code from breakfast to late at night, to demonstrate early prototypes of software that still crash, and more generally to exchange audio and music related technical ideas without fear. LAC offers also a unique opportunity for users and developers to meet, to discuss features, to provide feedback, to suggest improvements, etc.

LAC 2017 is the first edition to take place in France. It is co-organized by the University Jean Monnet (UJM) in Saint-Etienne and GRAME in Lyon.

GRAME is a National Center for Music Creation, an institution devoted to contemporary music and digital art, scientific research, and technological innovation. In 2016 the center hosted 26 guest composers and artists, produced 88 musical events and 25 exhibitions in 20 countries. GRAME is the organizer of the *Biennale Musiques en Scène* festival, one of France's largest international festival of contemporary and new music with guest artists ranging from Peter Eötvös, Kaija Saariaho, Michael Jarrell, Heiner Goebbels, Michel van der Aa, etc. GRAME develops research activities in the field of real-time systems, music representation, and programming languages. Since 1999 all software developed by GRAME are open source and in most cases multiplatform (Linux, macOS, Windows, Web, Android, iOS, ...).

UJM is part of the **University of Lyon**, a consortium of higher-education and research institutions located within the two neighboring cities of Lyon and Saint-Étienne. The University of Lyon-Saint-Etienne is the main French higher-education and scientific center outside the Paris metropolitan area, composed of 4 public universities, 7 high schools (grandes écoles) and the CNRS (the French National Center for Scientific Research), forming a group of 12 member institutions. The University of Lyon also assembles 19 associated institutions, offering specific disciplinary training programs. Altogether, the Université de Lyon regroups 137,600 students and 168 public laboratories.

The **Université Jean Monnet** (UJM), founded in 1969, is a comprehensive university enrolling some 20,000 students, about 15% international students from 111 countries. A high proportion of international students come from Africa and Asia.

The university is composed of 5 faculties (arts, letters and languages, humanities and social sciences, law, sciences and technology, and medicine), four institutes: Institut Supérieur d'Economie d'Administration et de Gestion, Telecom Saint-Etienne, and University Institutes of Technology (IUTs) in Saint-Etienne and Roanne.

The **CIEREC** is a research center devoted to the field of contemporary expression, which brings together professors, researchers and PhD students in aesthetics and sciences of art, plastics arts, design, digital arts, literature, linguistics and musicology. Its main field is the arts and literature of the twentieth and twenty-first centuries.

The **Music Department** offers graduate, post-graduate and doctoral training in music and musicology. In the field of technologies, it proposes, since 2011, a **Professional Master's Degree in Computer Music (RIM)** that is unique in France, in collaboration with GRAME, with the main audio production studio of Saint-Etienne (Le FIL) and with the National Superior Conservatory of Music in Lyon.

In 2016, we have created another **Professional Master for Digital Arts** (RAN). The Professional Masters of RIM & RAN are aimed at developing students' applied knowledge and understanding of electronic and digital technologies for the creation and they prepare to the professions of "Producer in Computer Music (RIM - Réalisateur en Informatique Musicale) and in Digital Arts (RAN - Réalisateur en Arts Numériques). These producers are direct actors in musical and artistic productions, and they are at the interface between software developers, applied computer scientists, composers, artists ... and all people likely to integrate video, image and sound in their activities. **Most courses are available in English** (see http://musinf.univ-st-etienne.fr/indexGB.html).

Thanks to all the contributors who submitted papers and proposed workshops, installations and music, we will have a very interesting and varied program at the conference. We are pleased to welcome, over a period of 4 days, some twenty conferences on various subjects with speakers from different backgrounds and countries.

We'd like to thank all those who contribute to the realization of this edition, and especially Albert Gräf, Philippe Ezequel, Jean-François Minjard, Stéphane Letz, Lionel Rascle, Thomas Cipierre, Sébastien Clara, Landrivon Philippe, David-Olivier Lartigaud, Jean-Jacques Girardot, Martine Patsalis, Nadine Leveque-Lair and all the reviewers and members of the scientific and artistic committees.

Thanks to our partners who helped to finance this conference : the CIEREC, GRAME, Masters RIM & RAN, the UJM Music and Arts Departments, The Arts, Lettres, Langues Faculty, Random-Lab at ESADSE (Art School of Saint-Etienne), Le Son des Choses, Electro-M, IDjeune, the Commission Sociale et Vie Etudiante at l'UJM.

LAC 2017 has been also partially funded by the FEEVER project [ANR-13-BS02-0008] supported by the Agence Nationale pour la Recherche.

We hope that you will enjoy the conference and have a pleasant stay in Saint-Etienne!

Vincent Ciciliato, Yann Orlarey et *Laurent Pottier*

LAC 2017 Teams

Organizers

- CIEREC (Centre Interdisciplinaire d'Étude et de Recherche sur l'Expression Contemporaine), director: Danièle Méaux, directors of the Electronic Team: Laurent Pottier & Vincent Ciciliato
- Music Department of Jean Monnet University (UJM), director: Anne Damon-Guillot
- GRAME (National Center for Musical Creation), director: James Giroudon, scientific director: Yann Orlarey
- Random-Lab, Center for Open Researches in Art, Design and New Media at ESADSE (Art School of Saint-Etienne), director: David-Olivier Lartigaud
- Association « Le son des choses » (Acousmatic Music)
- Association « Electro-M » (Masters RIM RAN students)

Organizing committee

- Vincent Ciciliato, Lecturer (Digital Arts) at CIEREC (UJM)
- Thomas Cipierre, PhD student (Musicology) at CIEREC (UJM)
- Sébastien Clara, PhD student (Musicology) at CIEREC (UJM)
- Philippe Ezequel, Lecturer (Computer Sciences) at CIEREC (UJM)
- Jean-Jacques Girardot, Programmer (Computer Sciences) at Le son des Choses
- Stéphane Letz, Researcher at GRAME
- Philippe Landrivon, Audiovisual technician, ALL faculty (UJM)
- David-Olivier Lartigaud, Director of Random-Lab (ESADSE)
- Jean-François Minjard, Composer at Le Son des Choses
- Yann Orlarey, Scientific director of GRAME
- Laurent Pottier, Lecturer (Musicology) at CIEREC (UJM)
- Lionel Rascle, Professor (Musical School – St Chamond & Rive de Giers)

Scientific committee

- Fons Adriaensen
- Marije Baalman
- Tim Blechmann
- Alain Bonardi
- Ivica Ico Bukvic
- Guilherme Carvalho
- Vincent Ciciliato
- Thierry Coduys
- Myriam Desainte–Catherine
- Goetz Dipper
- Catinca Dumitrascu
- Philippe Ezequel

- John Ffitch
- Dominique Fober
- Robin Gareus
- Albert Gräf
- Marc Groenewegen
- Florian Hollerweger
- Madeline Huberth
- Jeremy Jongepier
- Pierre Jouvelot
- David-Olivier Lartigaud
- Victor Lazzarini
- Stéphane Letz
- Fernando Lopez-Lezcano
- Kjetil Matheussen
- Romain Michon
- Frank Neumann
- Yann Orlarey
- Dave Phillips
- Peter Plessas
- Laurent Pottier
- Miller Puckette
- Elodie Rabibisoa
- Lionel Rascle
- David Robillard
- Martin Rumori
- Bruno Ruviaro
- Funs Seelen
- Julius Smith
- Pieter Suurmond
- Harry Van Haaren
- Steven Yi
- Johannes Zmölnig

SUMMARY

Special Guests

Conferences

Posters / Speed-Geeking

Workshops

Concerts

Installations

Special Guests

PAUL DAVIS

Paul Davis is the lead developer of the open source Ardour digital audio workstation, as well as the JACK Audio Connection Kit. Before his18 years involvement with audio software, Paul moved between academia and the corporate computing worlds, including 4-1/2 years at the University of Washington's Computer Science & Engineering department, and then becoming the 2nd employee at Amazon.com. In 2008/2009 he taught at the Technische Universitat, Berlin as the Edgar Varese Visiting Professor. Paul normally lives near Philadelphia, PA, but can also be found living and working in a solar-powered van. To his regret, Paul does not play any musical instruments.

Talk:

20 years of Open Source Audio: Success, Failure and The In-Between

I will talk about the 20-year history of open source audio development (focused on Linux but including other platforms when appropriate). It is a story that includes successes, failures and a lot of more ambiguous elements. I will discuss the way that the open source model does and does not help with software development, and also the sometimes surprising ways that "open source" might be pushing audio and music technology in the near future.

Thierry Codyus

Artist, musician, new technology expert, Thierry Coduys specializes in collaborative and multidisciplinary projects where interactivity meets the contemporary arts. Since 1986, he has worked closely with the avant-garde of contemporary music (e.g. Karlheinz Stockhausen, Steve Reich, ...) to realize electroacoustic and computer systems for live performance. After a few years spent at the IRCAM in Paris, he becomes the assistant to Luciano Berio. Building on his experience of the contemporary art scene, he creates his own company in 1999: an artistic research and technology laboratory called ‘La kitchen’, where artists from various horizons (e.g. music, dance, theatre, video, network) came to develop projects in collaboration with the team and where artists were encouraged to use Open Source Software. Thierry, among others, has been for more than 15 years the project manager of ‘IanniX’ (GNU GPL3 application), an interactive software interface, inspired by the UPIC of Iannis Xenakis and senior consultant for the development of ‘Rekall' (GNU GPL3 application), a video-annotation software to document digital performances.

Talk:

Why could the open source software change the way of writing for contemporary creation?

I will try to explain how it is important to propose to the artists to use open source’s tools. For such a long time artists have been obliged to play the game of commerce and industries, and also to use dedicated platforms in research and creation centres, they are waiting now for new concepts and not only for new production’s tools. Open Source community attracts brilliant developers, very motivated and often not very well paid. Many reasons can explain this interest, like clean design, liability, easy maintenance in the respect of the rules and values shared by the community, but also and mostly its freedom without constrictions that leave space to new concepts. I will show some examples of creation in collaboration with artists in all phases of development and their impact on the functions of the software which are used in the creation process.

Conferences

OpenAV Ctlra: A Library for Tight Integration of Controllers

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

Abstract

Ctlra is a library designed to encourage integration of hardware and software. The library abstracts events from the hardware controller, emitting generic events which can be mapped to functionality exposed by the software.

The generic events provide a powerful method to allow developers and users integrate hardware and software, however a good development workflow is vital to users while tailoring mappings to their unique needs.

This paper proposes an implementation to enable a fast scripting-like development workflow utilizing on-the-fly recompilation of C code for integrating hardware and software in the Ctlra environment.

Keywords

Controllers, Hardware, Software, Integration.

1 Introduction

Ctlra aims to enable easy integration between DAWs and controllers. At OpenAV we believe that enabling hardware controllers to be 1st class citizens in controlling music software will provide the best on-stage workflow possible.

Ctlra has been developed due to lack of a simple C library that affords interacting with a range of controllers in a generic but direct way, that enables tight integration.

1.1 Existing Projects

Although many projects exist to enable hardware access, very few aim to provide a generic interface for applications to use.

Projects such as maschine.rs[Light, 2016], HDJD[Pickett, 2017], OpenKinect[OpenKinect-Community, 2017] and CWiid[Smith, 2007] all enable hardware access, however they each expose a a unique API to the application, resulting in the need to explicitly support each controller.

The o.io[Freed, 2014] project aims to unify communications for various types of interaction using an OSC API, which is similar to the generic events concept. Discoverability and familiarity with the implementation presented possible issues, so Ctlra is designed as a simple C API that will be instantly familiar to seasoned developers.

Hence, Ctlra is implemented as a C library that provides generic events to the application, regardless of the hardware in use.

1.2 Modern Controllers

Each year there are new, more powerful and complex hardware controllers, often with large numbers of input controls, and lots of feedback using LEDs etc. The latest generations have seen an uptake in high-resolution screens built into the hardware.

The capabilities of these devices require an equally powerful method to control the hardware, or risk not utilizing them to the full potential. As such, any library to interface with these controllers should afford handling these complex and powerful controller devices easily.

1.3 Why a Controller library?

Although every application could implement its own device-handling mechanism, there are significant downsides to this approach.

Firstly, a developer will not have access to all controllers that are available, so only a subset of the controllers will have tight integration with their software. As an end result, the users controller may not be directly supported by the application.

Secondly, duplication of effort is significant, both in the development and testing of the controller support. This is particularly true if a device supports multiple layers of controls.

Thirdly, advanced controller support features like hotplug and supporting multiple devices of the same type must also be tested - requiring both access to multiple hardware units and time.

The Ctlra library shares the effort required to develop support for these powerful devices, pro-

viding users and developers with an easy API to communicate with the hardware.

1.4 Tight integration

The terms "tight integration" or "deep integration" are often used to describe hardware and software that collaborate closely together, perhaps they are even specifically designed to suite one other.

Tight integration leads to better workflows for on-stage usage of software, as it allows operations from inside the software to be controlled by the hardware device and appropriate feedback returned to the user.

The advantage of tight integration is providing a more powerful way of integrating the physical device and the software. As an example, many DAWs support MIDI Control Change (CC) messages, and allow changing a parameter with it. Although such a 1:1 mapping is useful, most workflows require more flexibility. For example, each physical control could effect a number of parameters with weighting applied to provide a more dynamic performance.

1.5 Controller Mapping

The Ctlra library allows mappings to be created between physical controls and the target software. DAWs could expose this functionality for technical users - giving them full control over the software.

Given the variation in live-performances and on-stage workflows, there is no ideal mapping from a device to the application - it depends on the user. As a result, OpenAV is of the opinion that enabling users to create custom mappings from controllers to software using a generic event as a medium to do so is the best approach.

1.6 Scripting APIs

Various audio applications provide APIs to allow users script functionality for their controller. Enabling users to script themselves requires technical skill from the user, however it seems like there is no viable alternative.

The solution proposed in section 4 also proposes "crowd-sourcing" the effort in writing controller mappings to the users themselves, as they have access to the physical device and have knowledge of their ideal workflow.

Examples of audio applications that provide scripting APIs are Ardour[Davis, 2017], Mixxx[Mixxx, 2017] and Bitwig Studio[Bitwig, 2017]. Although Ableton Live[Ableton, 2017] doesn't officially expose a scripting API, the are members of the community that have investigated and successfully written scripts to control it[Petrov, 2017].

A brief review shows high-level scripting languages are favoured over compiled languages. Mixxx and Bitwig are both using JavaScript, while Ableton Live uses Python, and Ardour uses the Lua language.

These solutions are all valid and workable, however they do require that the application developer to exposes a binding API to glue the scripting API to the core of the application.

With the exception of Lua, none of the above scripting languages provide real-time safety unless very carefully programmed - which should not be expected of user's scripts.

OpenAV feels that providing controller support in the native language of the application ensures that all operations that the application is capable of are also mappable to a controller. Other advantages of having the controller mappings in the native language of the application is that they can be compiled into the application itself.

2 Ctlra Implementation

This section details the design decisions made during the implementation of the Ctlra library. The core concepts like the context, device and events are introduced.

2.1 Ctlra Context

The main part of the Ctlra library is the context, it contains all the state of that particular instance of the Ctlra library. This state is represented by a `ctlra_t` in the code. Using a state structure ensures that Ctlra is usable from inside a plugin, for example an LV2 plugin.

Devices and metadata used by Ctlra are stored internally in the `ctlra_t`. The end goal is to enable multiple `ctlra_t` instances to exist in the same process without interfering with one-another. This is more difficult than it sounds as not all backends provide support for context style usage.

2.2 Generic Events

Ctlra is built around the concept of a generic event. The generic event is a C structure `ctlra_event_t` which may contain any of the available event types. The available event types include all common hardware controller interaction types, such as `BUTTON`, `ENCODER`, `SLIDER` and `GRID`. The events are

prefixed by `CTLRA_EVENT_`, so `BUTTON` becomes `CTLRA_EVENT_BUTTON`.

Once the type of the event is established, the contents of the event can be decoded. The generic event has a `union` around all events, so an event must represent one and only one type of event. It is expected that the application will use a `switch()` statement to decode the event types, and process them further.

The power of generic events is shown by the `examples/daemon` sample application, which translates *any* Ctlra supported device into an ALSA MIDI transmitting device.

2.2.1 Button

The button event represents physical buttons on a hardware device. It contains two variables, `id` and `pressed`. The button id is guaranteed to be a unique identifier for this device, based from 0, and counting to the maximum number of buttons. The `pressed` variable is a boolean value set high when the button is pressed by the user.

2.2.2 Slider

The slider event represents physical controls that have a range of values, but the interaction is of limited range, eg: faders on a mixing desk. The slider has and `id` as a unique identifier for the slider, and floating-point `value` that represents the position of the control. The `value` variable range is normalized as a linear value from `0.f` to `1.f` to allow generic usage of the event.

2.2.3 Encoder

The encoder represents an endless rotary control on a hardware device. There are two types of encoders, which we will refer to as "stepped" and "continuous". Stepped controls have notches providing distinct steps of movement, while the continuous type is smooth and provides no physical feedback during rotation.

The stepped controls notify the application for each notch moved by setting the `ENCODER_FLAG_INT`, and the delta change is available from `delta`. Similarly the `ENCODER_FLAG_FLOAT` tells the application to read the `delta_float` value, and interpret the value as a continuous control.

2.2.4 Grid

The grid represents a set of controls that are logically grouped together, eg: the squares of the Push2 controller. The `grid` event type contains multiple variables: `id`, `flags`, `pos`, `pressure` and `pressed`.

The `id` identifies the grid number, allowing controllers with more than one grid to distinguish between them. The `flags` allows the event to identify which values are valid in this event. Currently two flags are defined, `BUTTON` and `PRESSURE`, there are 14 bits remaining for future expansion.

If the flag `GRID_FLAG_BUTTON` is set, the `pressed` variable is valid to read, and represents if the button is currently pressed or not. The `BUTTON` flag should only be set in the device backend if the state of the grid-square has changed, this eases handling events in the application. When `GRID_FLAG_PRESSURE` is set, the floating-point `pressure` variable may be read, The `pressure` value is normalized to the range `0.f` to `1.0f`.

2.3 Devices

In Ctlra, any physical controller is represented internally in by a `ctlra_dev_t`. Devices do not appear available to the application directly, but instead operations on the device are performed through the `ctlra_t` context. There is an abstracted representation of a device at the API level, which the application has access to in the `event_handle()` callback.

The reason that the device is not exposed to the application directly is that ownership and cleanup of resources becomes blurred when hotplug functionality is introduced. Using the `ctlra_t` context as a proxy for multiple devices not only simplifies the application handling of controllers, but actually helps define stronger memory ownership rules too. See section 2.4 for hotplug implementation details.

2.3.1 Device Backends

A device backend is how the software driver connects to the physical device.

The implementation of the driver calls a `read()` function, which indicates the driver wishes to receive data. The backend library will send an async read to the physical device, and return immediately. Upon completion of the transaction a callback in the driver is called which decodes the newly received data, and can emit events to the application if required. To write data to the device, a `write()` is provided.

Note that a single device driver may open multiple backends, or utilize multiple connections of the same backend in order to fully support the capabilities of the hardware. An example could be a USB controller that exposes both a USB interrupt endpoint for buttons *and*

a USB bulk endpoint for sending data to a high-resolution screen.

Note that more backends can be added to support more devices if it is required in future.

2.4 Hotplug Implementation

Implementing a hotplug feature is difficult; it requires handling device additions and removals in the library itself, as well as a method to communicate any changes of environment with the application.

As Ctlra is a new library built from the ground up, hotplug was a consideration from the start as a required feature. As such, the API has been influenced by and designed for hotplug capabilities. The concept of a `ctlra_t` context that contains devices was introduced to allow transparent adding of devices without blurring memory ownership rules.

Hotplug of USB devices is enabled by LibUSB, which provides a hotplug callback, when a hotplug callback is registered and hotplug is supported on the platform. The USB hotplug callback is utilized to call the `accept_device()` callback in the application, providing details of the controller. The info provided allows the application to present the user with a choice of accepting or rejecting the controller, and if accepted, it will be added to the `ctlra_t` context.

3 Application Usage of Ctlra

This section will introduce the reader to the steps required to integrate Ctlra into an application. Refer to the `examples/simple/simple.c` sample to see a minimal program in action.

The following steps summarize Ctlra usage:

1. `ctlra_create()`
2. `ctlra_probe()`
   - Accept controller in callback
3. `ctlra_iter()`
   - Handle events in callback
4. `ctlra_exit()`

This creates a single `ctlra_t` context, probes and accepts any supported controller. The accepted controllers are connected to the particular context that it was probed from.

Calling `ctlra_iter()` causes the event to be polled and the application is given a chance to send feedback to the device. Finally, `ctlra_exit()` releases any resources and gracefully closes the context.

3.1 Interaction

The main interaction between Ctlra and the application happens in two functions. Events from the device are handled in the application provided `event_handle()` function, while feedback can be sent to a device from the `feedback_func()`.

These functions are callback functions, and they are invoked for each device when the application calls `ctlra_iter()`.

To understand the events passed between the device and the application, please review the generic events (Section 2.2), and browse the `examples/` directory.

3.2 Controller's View of State

Each application has its own way of representing its state. Similarly, each controller has its own capabilities in terms of controls and feedback to the user. Given the specific application state and capabilities of the hardware, it is useful to create a struct specifically for storing the view that the controller has of the application.

Note that the controller view should be tracked *per instance* of the controller, as users may have multiple identical controllers. This controller's instance of the struct is very useful for remapping the controls to provide an alternate map when a "shift" key is held down. As the struct depends on the application and device, this problem can not be solved elegantly at the library layer.

Ctlra provides a userdata pointer for each instance which can be purposed for to point to the state struct. If the application's state must be accessed from the state-struct, a "back-pointer" to the application elegantly provides that.

The memory for the state struct can be allocated in the `accept_device()` callback from Ctlra, and the memory can be released in when the device is disconnected using the `remove_device()` callback.

4 Device Scripting in C

This section describes a solution to providing a fast and interactive development workflow for scripting mappings between software and device using the C language.

C is typically a compiled and static language, not one that comes to mind when discussing dynamic and scripting type workflows. Although generally accurate, C can be used as a dynamic language with certain compromises. The following section details how applications can imple-

ment a C scripting workflow for users to quickly develop "Ctlra scripts".

4.1 Dynamic Compilation

Dynamically compiling C at runtime can be achieved by bundling a small, lightweight C compiler with your application. This may sound a little crazy, but there are very small and lightweight C compilers available designed for this type of usage. The "Tiny C Compiler", or TCC[Bellard, 2017] project is used to enable compiling C code at runtime of the application.

Please note that the security of dynamically compiling code is not being considered here as the goal is to enable user-scripted controller mappings for musical performance. If security is a concern, the reader is encouraged to find a different solution.

4.2 TCC and Function Pointers

The TCC API has various functions to create a compilation context, set includes, and add files for compilation. Once initialized, TCC takes an ordinary `.c` source file, and compiles it.

When compilations completes successfully, TCC allows requesting functions from the script by name, returning a function pointer.

The returned function pointer may be called by the host application, forming the method of communicating with the compiled script.

4.3 The Illusion of Scripting

To provide the illusion that the code is a script, the application can check the modified time of a script file, and recompile the file if needed. By swapping in the new function pointers, the update code runs. The old program can then be freed, cleaning up the resources that were consumed by the now outdated script.

The `examples/tcc_scripting/` directory contains a minimal example showing how the event handling for any Ctlra supported device can be dynamically scripted.

Providing this workflow requires some extra integration from the application, however the time pays off easily in developer time saved when time save in scripting support for each controller is considered.

4.4 C and C++ APIs

Note that TCC is a C compiler only - explicitly not a C++ compiler. This has some impact on how scripts can interact with applications, as many large open-source audio projects are written in C++. The solution is to provide wrapper functions to C, if the hosts language is C++.

Often real-time software uses message-passing in plain C structs through ringbuffers. This is a good way to communicate between dynamically compiled scripts and the host, as it provides a native C API, as well as a method to achieve thread-safe message passing.

5 Case Study: Ctlra and Mixxx

This section briefly describes the work performed to integrate Ctlra with the open-source Mixxx DJ software. It is presented here to showcase how to integrate the Ctlra library in an existing project.

5.1 Implementation

This section details the steps taken to integrate the Ctlra library in Mixxx to test Ctlra in the real-world.

5.1.1 Class Structure

Mixxx has a very object oriented design, utilizing C++ classes to abstract behaviour of control devices and managers of those control devices. The `ControllerManager` class aggregates the different types of `ControllerEnumerator` classes, which in turn add `Controller` class instances to the list of active controllers. Ctlra has been integrated as a `ControllerEnumerator` sub-class for this proof-of-concept implementation, really it should be integrated at the `ControllerManager` level.

5.1.2 Threading in the Mixxx Engine

The Mixxx engine currently creates many threads. This design is supported by the use of an "atomic database" of values (see next Section 5.1.3). Given this design, the Ctlra integration is done by spawning a Ctlra handling thread, which performs any polling and interacting with Ctlra supported devices.

5.1.3 Communicating with the Engine

The Mixxx engine is composed of values, which can be controlled from any thread anywhere in the code. These values are represented in the code by `ControlObject` and `ControlProxy` classes. A `ControlObject` is the equivalent to owning a value, while the `ControlProxy` allows atomic access to update the value. Lookup of these values is performed using "group" and "key" strings. The strings are constant allowing Ctlra and the Mixxx engine to understand the meaning of each value represented by a particular `ControlProxy`.

5.1.4 Mixxx's C++ API

An issue arises due to Mixxx having a ControlProxy being a C++ class which is not possible to access from a TCC compiled script (refer to C and C++ APIs, Section 4.4).

The solution is to create a C wrapper function, which simply provides a C API to the desired C++ function to be called on a `ControlProxy` instance. This provides the power of the Mixxx engine to the dynamically compiled script code:

```
void mixxx_config_key_set(
        consl char *group,
        const char *key,
        float value);
```

5.2 Mixxx and Hotplug

Since Ctlra hides the hotplug functionality from the application due to the design of the `accept_device()` callback, Mixxx supports on-the-fly plug-in and plug-out transparently.

This is achieved by the Ctlra library having its own thread to poll events (see Section 5.1.2), and handling the connect or disconnect events. The Mixxx application code did not have to be modified to support hotplugging of controllers in any way (beyond adding basic Cltra support).

5.3 Scripting Controller Support

With the Ctlra library integrated in Mixxx, users are now able to script the tight integration of the Ctlra supported hardware and Mixxx. The next sections demonstrate simple mappings from a device to Mixxx and vice-versa.

5.3.1 Event Input to Mixxx

When a user presses a physical control on a device, the action is presented to the application as an event. The user can map these events to the application in a variety of ways, in order to suit their own requirements on how they wish to control the software application.

For example, the following snippet shows how we can bind slider ID 10 to channel 1 volume in Mixxx (note the usage of the C function from Section 5.1.4):

```
case CTLRA_EVENT_SLIDER:
  switch(e->slider.id) {
  case 10:
    mixxx_config_key_set(
        ''[Channel1]'',
        ''volume'',
        e->slider.value);
    break;
```

5.3.2 Mixxx Feedback to Device

The reverse of the previous paragraph is to send Mixxx state to the physical device, providing feedback to the user. Each parameter that Mixxx exposes via the `ControlProxy` is available for reading as well as writing. The allows the script to query the state of a particular variable from Mixxx, and update the state of an LED on the device, using the Ctlra encoding for colour and brightness:

```
int play;
play = mixxx_config_key_get(
          ''[Channel1]'',
          ''play_indicator'');

led = play > 0 ? 0xffffffff : 0;

ctlra_dev_light_set(dev,
          DEVICE_LED_PLAY,
          led);
```

6 Future Work

To make Ctlra a ubiquitous library for event I/O is a huge task, however the benefit to all applications if such a library did exist would be huge too.

Imagine easily scripting your DIY controller to easily control any aspect of any software - huge potential for customized powerful user-experience. OpenAV intends to use the Ctlra library and integrate it with any projects that would benefit from a powerful customizable workflow.

6.1 Device Support

At time of writing, the Ctlra library supports 6 advanced USB HID devices, one USB DMX device, a generic MIDI backend, and plans are in place to support a common bluetooth console controller - but more must be added to make the Ctlra library really useful!

An interesting angle may be so that DIY platforms like Arduino can be used to build controllers that use a generic Ctlra backend, allowing controllers to be auto-supported.

The previously mentioned hardware enabling projects that provide access to specific hardware devices could be integrated with Ctlra, transparently benefiting applications that use Ctlra.

The number of supported hardware devices is paramount to the success of the Ctlra library, so OpenAV welcomes patches or pull-requests that add support for a device.

6.2 Software Environments

From the software point-of-view there is huge potential for integrating into existing software.

For example mapping Ctlra events to LV2 Atoms would expose the Ctlra backends to any LV2 Atom capable host.

Integration with DSP languages like FAUST or PD may prove interesting and allow for faster prototyping and more powerful control over performance using those tools.

Hardware platforms like the MOD Duo[MOD, 2017] could use the Ctlra library to enable musicians to use a wider variety of controllers in thier on-stage setups in conjunction with the DSP on the DUO.

7 Conclusion

This paper presents Ctlra, a library that allows an application to interface with a range of controllers in a powerful and customizable way.

It shows how applications and devices can interact by using generic events. A case study showcases integrating Ctlra with the open-source Mixxx project as a proof of concept.

To enable a fast development workflow for creating mappings between applications and devices, a method to dynamically compile C code is introduced. This enables developers and users to write mappings between devices and applications as if C was a scripting language, but provides native access to the applications data structures.

Ctlra is available from github here[OpenAV, 2017], please run the sample programs in the `examples/` directory of the source to experience the power of Ctlra yourself.

8 Acknowledgements

OpenAV would like to acknowledge the linux-audio community and open-source ecosystem as a whole for providing novel solutions to various problems and being a great place to collaborate and innovate. For the work on Ctlra certain people and projects provided lots of inspiration and support, thanks!

Thanks to the TCC project, which allows dynamically compiling Ctlra scipts, it is awesome to script in C!

Thanks to William Light for writing `maschine.rs`, David Robillard for the creation of PUGL[Robillard, 2017], the Mixxx project devs (particular shout outs to `be_`, `Pegasus_RPG` and `rryan` on `#mixxx` on irc.freenode.net.

References

Ableton. 2017. Music production with live and push — ableton. https://www.ableton.com.

Fabrice Bellard. 2017. Tcc : Tiny c compiler. https://http://www.bellard.org/tcc/.

Bitwig. 2017. Bitwig music productiona nd performance system for windows, macos and linux. https://www.bitwig.com.

Paul Davis. 2017. Ardour: Record, edit, and mix on linux, os x and windows. http://ardour.org/.

Adrian Freed. 2014. o.io: a unified communications framework for music, intermedia and cloud interaction. *International Computer Music Conference (ICMC) 2014*.

William Light. 2016. Maschine.rs, open-source ni maschine device handling. https://github.com/wrl/maschine.rs.

Mixxx. 2017. Mixxx dj software, dj your way. for free. https://mixxx.org/.

MOD. 2017. Mod duo, the definitive stompbox. https://moddevices.com/pages/mod-duo.

OpenAV. 2017. Ctlra is a library providing support for controllers, designed to integrate hardware and software. https://github.com/openAVproductions/openAV-Ctlra.

OpenKinect-Community. 2017. Open source libraries that will enable the kinect to be used with windows, linux, and mac. https://openkinect.org.

Hanz Petrov. 2017. Introduction to the ableton framework classes. http://remotescripts.blogspot.com/2010/03/introduction-to-framework-classes.html.

Neale Pickett. 2017. Hercules dj controller driver for linux. https://github.com/nealey/hdjd.

David Robillard. 2017. Pugl is a minimal portable api for opengl guis. https://drobilla.net/software/pugl.

Donnie Smith. 2007. A collection of linux tools written in c for interfacing to the nintendo wiimote. http://abstrakraft.org/cwiid/.

Binaural Floss – Exploring Media, Immersion, Technology

Martin RUMORI
Institute of Electronic Music and Acoustics (IEM)
University of Music and Performing Arts Graz
Inffeldgasse 10/3, 8010 Graz, Austria
rumori@iem.at

Abstract

Technology for binaural audio, that is, relating two audio signals to the psychophysical properties of the human hearing apparatus, is capable of recording, synthesising and reproducing the spatial information of an auditory environment comprising an immersive quality. While current scholarly research on binaural rendering and reproduction techniques for personal, mobile and interactive audio augmented environments is well advanced, their grounding with respect to the aesthetic experience in an integral listening act is not. Based on the case study of an intermedia installation, *Parisflâneur*, an attempt towards the exploration and reflection of binaural media properties is made. Here, a special emphasis is put on the role of FLOSS tools in an arts-based research context.

Keywords

binaural audio, immersion, floss tools, intermedia art, field recordings

1 Introduction

Binaural audio means to relate a pair of audio signals to the psychophysical properties of the human hearing apparatus, that is, the signals are regarded as so called *ear signals*. Binaural audio is among the earliest attempts of recording, reproducing and synthesising the spatial information of an auditory scene by dummy head microphones, appropriate signal processing and by presenting the binaural signal pair isolated from each other to the left and right ear, respectively, usually via headphones. Nowadays, in the view of ubiquitous headphone use and the advent of widespread three-dimensional video projection, binaural technology constantly gains significance, and so does research on the optimal rendering and projection of personal, mobile and interactive audio augmented environments.

When it comes to the creation of such environments, optimisation targets become much less clear. Questions of immersion, perception and cognition arise as components of an integral aesthetic experience. Methods in scholarly research usually segment complex processes such that, for instance, certain psychoacoustic parameters are isolated for separate investigation. The results of listening tests according to such methods often cannot be generalised for regarding a complex listening process that involves musical or anecdotal aspects of the sound material, cognitive contribution or previous experience by the listeners, to name just a few factors.

Obviously, this paper cannot provide solutions or answers. What I am going to present is a personal attempt of approaching theoretical, aesthetic and engineering reflections along the development of an artistic case study, *Parisflâneur*, which is work in progress.

In the next section, I will describe the case study from a phenomenological point of view, that is, how it appears to the visitor of an imagined exhibition. The description will be followed by a detailed discussion of technical implementation decisions in close relation to aesthetic reflections on conditions of the media involved. A special emphasis will be put on the role of Free and Libre Open Source Software (FLOSS) in the described process.

2 *Parisflâneur*: visitor's experience

Parisflâneur is a sound installation that explores the relation of binaural recording and binaural rendering of a virtual scene by providing a reactive, playful environment.

From the outside, the appearance of *Parisflâneur* is quite reduced: it does not consist of much more than a pair of headphones and an empty area in space of about twenty to fourty square meters. The visitor is invited to put on the headphones and explore the installation solely by listening and freely moving in the area whose boundaries are usually marked on the floor.

Both the position and orientation of the headphones are tracked, which to date requires an optical multi-camera tracking, given the required latency limits and the relatively large tracking volume. That means that a tracking target, a rigid body of four or five reflective balls, is a quite noticeable part mounted on top of the headphones.[1] Additionally, in most practical installations of *Parisflâneur* the headphones are cabled as no satisfying wireless solution with respect to transmission quality and robustness, low latency and signal dynamics (i. e., no audio compression) was available so far. This fact is mentioned as it potentially interferes with the visitor's mobility (see [Rumori, 2017]).

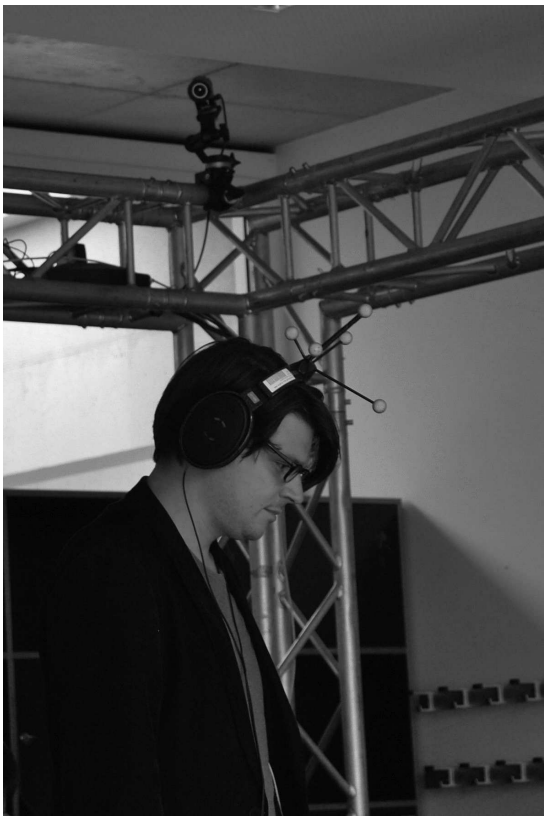


Figure 1: Visitor exploring *Parisflâneur*.

When the listener enters the installation, he is presented a virtual auditory scene, which can be navigated. Urban and rural situations such as a street, pedestrian area, or park are recognisable by typical sounds like cars, footsteps, voices, crickets, an aeroplane or rain. They appear to come from different directions around the listener. When walking around guided by listening it turns out that each of the sound situations is fixed at a certain location in space. Their positions may be found by bodily movement, approaching, turning towards and away from the sounds. They react by loudness attenuation and filtering on increasing distance and directional changes relative to the listener's head, compensating his movements and thus resulting in a perceived steady configuration inscribed into the surrounding space. When the listener reaches exactly the same location as a sound situation, it appears to reside inside his head. This auditory effect is a common experience when listening to speaker-based stereophonic signals on headphones. In total, there are seven of such sound spots representing different everyday situations in *Parisflâneur*.

When the location of a sound situation was found, the listener may "enter" it by performing a ducking gesture, that is, by bending down such that the head goes well below the usual standing or walking height and subsequently raising the head again at the found location. This procedure is communicated to the visitors beforehand using the metaphor of tracing "sonic hats" in space which can be "put on" and "taken off."

Entering a sound situation yields a substantial change in the audio listened to. The virtual sound scenery composed of multiple anecdotal situations gradually disappears except of the single sound being entered. The remaining one is no longer represented by a single spot but opens towards a rich, expanded auditory scene on its own that immerses the listener. Technically, the rendered binaural signal is replaced by a static binaural recording, which also serves as a basis for the sound sources in the virtual scene. As the recording is static, it does not any longer respond to the listener's movements but is attached to his head, as known from the common listening experience with headphones. In terms of the above-mentioned metaphor, the "sonic hat" that has been "put on" is now "carried around."

The entered sound situation may be left by performing the ducking gesture once more: by bending down and coming up again from underneath the sound spot, thus "taking off" the "sonic hat" and leaving it in space. The binaural

[1] A promising alternative is presented by the *Lighthouse* system developed for the *HTC Vive* goggles and to be released soon as an independent tracking solution. It shall provide a nearly comparable performance to camera-based systems by *OptiTrack* or *Vicon* at a much lower cost and setup complexity, cf. `http://www.roadtovr.com/valve-sell-base-stations-directly-lower-barrier-steamvr-tracking-development/` (last retrieved February 27, 2017).

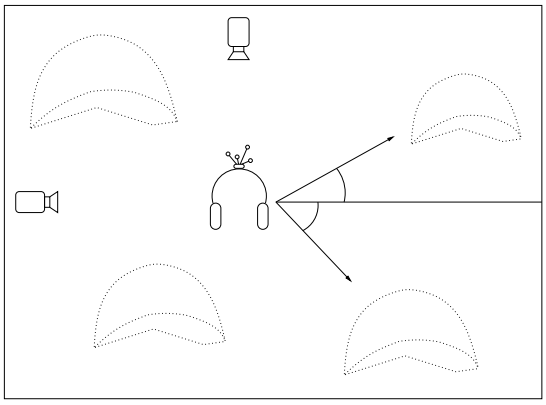


Figure 2: Schematic setup of *Parisflâneur*.

recording crossfades into the virtual scene again comprising all of the seven sound sitations, each represented by a single point in space.

When a sound situation was left, it remains at the location in space where it was dropped. That means, when the listener moves with a "sonic hat" currently "put on" the spatial configuration of the virtual scene is rearranged. There is no immediate audible feedback hinting at this change as in this moment solely the entered situation is presented in its non-reactive form. Only after having re-entered the virtual scene, the reconfiguration becomes audible.

Tracing, entering, leaving and rearranging everyday sound situations shall allow for a playful exploration of the sound material and an associative recombination of narratives in the sense of anecdotal music as coined by Luc Ferrari. Along with the perceptual differences of rendered and reactive audio at one hand and recorded and static material on the other, *Parisflâneur* is at the same time a study of the properties and conditions of so called immersive spatial media.

3 Media, software, technology

Several kinds of media technology are involved in the realisation of *Parisflâneur*, among them audio recording and reproduction over headphones, optical tracking of position and rotation, the application logic that evaluates the tracking data and finally controls different levels of signal processing for creating the presented output. The focus in this paper is on the last-mentioned building blocks that are represented by software, which form a major part of the artistic development.

I am going to discuss the implementation with a special emphasis on the application of FLOSS tools and their relation to the artistic creation process and aesthetic aims. Like in most other intermedia artefacts, the general purpose computer acts as a kind of meta-medium that, owing to universal digital data representation, allows for the actualisation of more specific media machines by means of software [Manovich, 2013]. It shall be stressed though that all the other media involved, including and especially non-digital ones, have an equally significant influence on the aesthetics of the work (for a discussion, see [Rumori, 2017]).

In the following, I will present technical considerations in stretto with reflections on the artistic process and on the properties of media.

3.1 Software involved

Parisflâneur is realised by combining a few software building blocks, all of them being FLOSS. The processing of the tracking data, most of the signal processing and the application logic is implemented in *Supercollider*[2]. Supercollider allows for constructing modular multichannel realtime signal processing networks controlled by a general-purpose object-oriented language. Details on the binaural rendering will be presented in the following sections, which will also make clear why an open and flexible framework like Supercollider is necessary for developing this installation, rather than a monolithic, optimised software package (cf. [Magnusson, 2008]).

Most binaural synthesis techniques involve a matrix of realtime convolutions, sometimes using room impulse responses of several seconds duration. In earlier versions, *Parisflâneur* uses 24 binaural room impulse responses (BRIR) of 64 k samples each, in a later version 12 of those BRIRs plus 36 free-field two-channel responses of 512 samples. For performing the convolution, *Jconvolver* by Fons Adriaensen is used [Adriaensen, 2006b]. It provides very efficient, low-latency, multi-threaded convolution while matrices of any layout and of large sizes may be configured. Supercollider and Jconvolver are connected via the *Jack Audio Connection Kit*[3].

The binaural room impulse responses (but not the free-field ones) used for the convolution were measured in the Cube laboratory at *Institute of Electronic Music and Acoustics Graz* (IEM) [Rumori et al., 2010]. For the measure-

[2] `http://supercollider.github.io` (last retrieved February 28, 2017)

[3] `http://www.jackaudio.org` (last retrieved February 28, 2017)

ments, a customised version of *Aliki*, again by Fons Adriaensen, was used [Adriaensen, 2006a]. Customisations include a higher number of supported channels and some automation facilities, which were used in conjunction with Supercollider [Hollerweger and Rumori, 2013].

Editing and processing of the field recordings was performed using Ardour[4].

3.2 Binaural rendering

At first glance, the rendering of the virtual auditory scene in *Parisflâneur* seems to be a standard engineering problem of moderate complexity, which is a correct assumption to a large extent. There are seven monaural point sources, not too many, each with the same trivial, that is, omni-directional radiation pattern, to be rendered in a so far not further specified virtual space, probably not requiring a too complex underlying model. The scene should be rendered for one dynamically moving listener according to tracking data input. The sound sources are not dynamically moving, and if so, their movements are not audible at the same time, which might allow for non-realtime optimisations. Furthermore, only one source is moving at a time.

Despite its moderate technical demands, *Parisflâneur* is not about developing or using an "optimal" binaural rendering technique. In fact, the artistic reflection is targeted at the question of what "optimal" could actually mean in this context. Does it mean to model as accurately as possible the physical sound propagation starting from the emitters, the contribution of the surrounding space to the radiated sound waves, their arrival at the human head, finally the effect of a two-channel, spaced and individually filtered pressure receiver array, our hearing apparatus? In other words: does it mean to capture the physics of an existing or imagined real-world situation and simulate it?

Obviously, there is no corresponding real-world situation to seven spatial field recordings reduced to monaural signals and put into a navigable virtual space. Potentially, an installation of seven loudspeakers distributed in space and each playing back one of the recordings could come close physically but the mere thought experiment makes evident that the artistic point would be entirely missed. Although the navigation aspect may be retained in principle, each of the sound spots would be represented by a physical object, being both an obstacle for moving in space and a hindrance for the orientation by listening due to its visual presence. Apart from that, such an installation would lack the reactive capability of entering one of the recordings.

I wrote that the envisioned "hardware" replication of the virtual scene "*could* come close physically" and "*may* be retained *in principle*" to indicate that the rendered scene and its physical counterpart have nothing in common in terms of sound propagation properties and reactive behaviour. There is no evidence whatsoever why the virtual scene should be designed such that its acoustical properties match those of reality. Rather its perception and cognition, that is, its integral aesthetic experience, shall provoke an imagination that supports the further engagement with the artwork. Aesthetic experience depends on previously made experience. In the case of navigating an auditory environment it relates to our spatial awareness which to date is mostly trained by orientation in reality. Again, this does not mean that matching physical stimuli are sufficient or the right way at all to evoke matching auditory impression. In *Parisflâneur*, probably among many other examples, it is not even desired [Rumori, 2016].

> This basic assumption [...], that a subject will always hear the same sound when exposed to identical sound signals, is obviously not true [...]. Yet [...], authentic reproduction is rarely required. [... S]ound material on the radio and on disk is processed in such a way as to achieve the optimal auditory effect, for instance, from an artistic point of view. [Blauert, 1997, 374]

Blauert does not elaborate on how "the optimal auditory effect" would be approached and when it is reached. For a reason: processed sound material is only one part of an integral aesthetical experience; individual perception, various levels of familiarity with certain technologies, subjective cognitive contribution, cultural differences are others. From the "artistic point of view" there is no clear optimum either: artworks open perceptual spaces for individual exploration and offer a multitude of strands for interpretation. Of course, a kind of "aesthetic nucleus" can be assumed that is central to both the artist's and the recipient's reflection. There may be more or less appropriate ways of grasping and conveying it using media, but a single optimal one is unlikely to exist.

[4] `http://ardour.org` (last retrieved April 3, 2017)

Due to the absence of compulsory realisation schemes in an artistic context, the rendering techniques adapted for *Parisflâneur*, the sound propagation laws modelled in and the rules of reactive behaviour applied to the virtual environment are found by experimentation and intuition. The reference are not ear signals in reality but the conditions and implications of media. Here, this includes limitations of space and tracking capability, computational power and implementation feasibility, and, most importantly, the cultural technique of headphone listening and its heritage (for a discussion on the latter, see [Rumori, 2017]).

3.2.1 Virtual Ambisonics

Earlier implementations of *Parisflâneur* use a modified virtual Ambisonics approach for rendering the binaural scene [Noisternig et al., 2003]. Instead of synthesised room acoustics and free-field (i. e., anechoic) impulse responses, measured binaural room impulse responses (BRIR) are used. This way, the virtual scene is embedded in captured real-room acoustic properties rather than a simplified model. Furthermore, the BRIRs were measured in the location of the work's first presentation, the Cube laboratory at IEM Graz, such that the virtual acoustics presented via headphones matched that of the surrounding real space. The idea was to provoke the notion of an overlay inscribed into the existing aural space rather than replacing it by a different one.

One disadvantage of combining the virtual Ambisonics approach with BRIRs is that the proposed rendering optimisations cannot be applied unless the measured room acoustics is assumed to be fully symmetric (cf. [Noisternig et al., 2003]). More significantly, the implementation is "incorrect" in terms of communications engineering: As the BRIRs were only measured for a single orientation of the dummy head, rotation in the Ambisonics domain upon tracking input results in the room acoustics being turned along with the listener while the relative source positions are correctly adjusted. The resulting misleading spatial cues may degrade localisation accuracy and externalisation (cf. [Rumori, 2017]).

The virtual Ambisonics approach has been incorporated in *Parisflâneur* using modified classes of the *AmbIEM* Supercollider quark[5].

[5] `https://github.com/supercollider-quarks/AmbIEM` (last retrieved February 28, 2017)

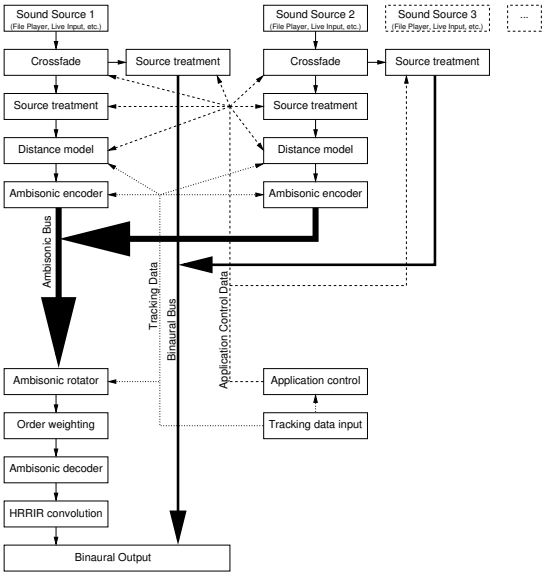


Figure 3: Block diagram of *Parisflâneur* using the virtual Ambisonics approach.

3.2.2 Distance model

Classical Ambisonics does not encode distance information of sound sources, only directions, that is, sources are plane waves. Extensions exist to take into account the near field effect of the projection system by appropriate filters [Daniel, 2003] or to use an additional Ambisonics channel for encoding source distances [Penha, 2008]. Still, these do not include models for translating a distance vector into processing parameters like amplitude attenuation, low-pass filtering, or the ratio of direct signal to reverb energy.

Scholarly research shows that auditory distance estimation is highly dependent on the source material and cannot be reliably performed even in reality [Zahorik, 2002]. In the light of the reflection above (see section 3.2), modelling the source distance in a rendered scene is not a means of referring to reality but to the aesthetic framework of the installation.

Amplitude attenuation in *Parisflâneur* is much stronger than in reality as described by the inverse squared law. Otherwise, the seven sound situations would not be distinguishable at all by approaching one or the other as their levels would differ too little, given the limited tracking volume and the relatively low maximum distances of sources. Similarly, low-pass filtering by air absorption would be hardly noticeable at such short distances, while in *Parisflâneur* it is used as an acoustical "magnifier" for the closer surrounding of the listener.

In implementations of *Parisflâneur* using the virtual Ambisonics approach (see section 3.2.1), the ratio of direct and reverb signal energy is fixed by the impulse responses. It could be made variable by an implementation using two Ambisonics domains, a "dry" and a "wet" one.

3.2.3 Circular panning

More recent implementations of *Parisflâneur* drop the virtual Ambisonics approach as most of its advantages do not apply here. Different to the three-dimensional Ambisonics approach, the rendering was additionally reduced to two dimensions. Listeners in *Parisflâneur* mostly move in a plane only, while the third dimension has no orienting function. The sound spots are meant to be at ear level all the time, independent of the height of the listener. When applying the "sonic hat" metaphor (see section 2), elevation information could have a certain value, but this interaction scheme was also replaced by a different one in later versions of the installation (see [Rumori, 2017]).

Currently, *Parisflâneur* incorporates two domains of simple circular panning, implemented using Supercollider's `PanAz` unit generator. One domain uses 12 channels of a measured circular loudspeaker array in a fairly reverberant room, while the second one has 36 output channels representing a ten-degree resolution of anechoic impulse responses taken from the SoundScapeRenderer project[6]. Sources in the far field are projected using the first panning domain while the energy contribution is gradually shifted towards the second domain for closer sources. Obviously, the latter represents a stronger direct portion of the source signal.

For sources very close to the listener's head, the binaural domain in a classic understanding is left, that is, no head-related impulse responses are involved anymore. Instead, the usually undesired effects of intensity panning on headphones are exploited for provoking near-field and in-head experiences (see section 3.3.3).

3.3 Applications of binaural recordings

What does it mean to represent a spatial, head-related field recording by a monaural single-point object in a virtual auditory scene? Similar to sound stored on tape, a vinyl record or a compact disc, the recording becomes an object in terms of the environment, be it a physical carrier medium or a sound source rendered in virtual space. This is different from simply playing it back, which rarely focuses the recording media itself, rather, its properties shall be hidden behind the recorded. In *Parisflâneur*, the relation of the recording in its head-related form and its appearance as a virtual object is a central point of reflection, plus the anecdotal, that is, musical relation of several of such objects to each other by providing them for rearrangement by the listener.

While the recordings are left widely unprocessed for their binaural presentation when a sound situation is "entered," their monaural counterparts as objects in the virtual scene have to be derived from the recordings with some treatment.

3.3.1 Monaural representation

An important point is to achieve some degree of monaural compatibility in order to reduce comb filter effects especially in the lower frequencies when mixing both channels of a binaural recording to a single one.

A simple monaural representation would only use one channel of the binaural recording, and in fact that has been done in preliminary versions of *Parisflâneur*. Of course this results in an unbalanced spatial interpretation of the signal, as the higher frequency portions at the far side that are attenuated by the head are omitted. Nevertheless, for providing an overall impression of a field recording and its recognition in a virtual scene this solution may suffice.

A more advanced approach to monaural compatibility would be to turn the phase differences in the low frequencies into level differences. This is exactly the purpose of the so called *Blumlein Shuffler* [Gerzon, 1994], "the greatest forgotten invention in audio engineering"[7]. It was patented by Alan Blumlein in 1933 for the loudspeaker reproduction of time-of-arrival stereophonic signals.

In *Parisflâneur*, the *Blumlein Shuffler* implementation *bls1* by Fons Adriaensen is used[8]. It provides one of the few accessible implementations, the most advanced one due to its use of carefully designed FIR filters and, to my knowledge, the only free and libre implementation.

[6] `http://spatialaudio.net/ssr/` (last retrieved February 27, 2017)

[7] `http://www.pspatialaudio.com/blumlein_delta.htm` (last retrieved February 27, 2017)

[8] `http://kokkinizita.linuxaudio.org/linuxaudio/zita-bls1-doc/quickguide.html` (last retrieved February 27, 2017)

3.3.2 Frequency response

A virtual source's spectrum is likely to be distorted by rendering compared to that of the underlying binaural recording. As both instances are related to each other in the installation, the signals used as virtual sources are filtered according to experimental exploration of different spatial constellations, that is, different rendered directions and distances from the listener.

3.3.3 Transition design

The moment of transition from the virtual scene to the binaural recording and back is one of the central aesthetic experiences in *Parisflâneur*, hence the importance of its design. In the course of refining the work, transition design evolved from a simple cross-fade between the two domains, nevertheless using special overlapping curves, towards a more complex multi-stage process.

In the phenomenological description (see section 2) I stated that in-head localisation in the virtual scene is desired in order to indicate the exact position of a sound spot. Early implementations of *Parisflâneur* used the virtual Ambisonics approach for the binaural rendering of the scene (see section 3.2.1). For closer sources, the energy contribution of higher Ambisonics orders is gradually reduced after encoding, which achieves a spatial widening until only the zeroth order remains when the position of the source is reached. This corresponds to an omnidirectional receiver pattern at the listener's position, hence the source's signal is projected equally from all directions in the virtual Ambisonics speaker setup. In a certain understanding, this might represent the notion of being "inside" a sound source, especially in the case of real loudspeaker reproduction and when the source is attributed a certain extension, for instance, that of the reproduction space.

For the binaural projection of *Parisflâneur* and its narrative, another approach to conveying the "inside" notion appears to be much more appropriate: the often undesired in-head localisation of loudspeaker-based stereophony or monaural signals presented on headphones. Its application means leaving the integrity of both binaural playback and binaural rendering in a strict sense of communications engineering. Rather, signals usually not considered binaural are interpreted as ear signals in order to exploit the resulting, yet uniquely binaural effect. For this reason, I do not attribute the quality "binaural" to a signal pair because of its technical properties such as the presence of interaural time or level differences but rather due to its *intentional* interpretation as ear signals. Furthermore, this example is a strong indication why open software systems are a precondition for pursuing the artistically motivated approach to binaural technology as described here. Most monolithic implementations, even if advanced and optimised with respect to latest research, do not allow for modelling and accessing the signal path at every level.

When experimenting with the abovementioned *Blumlein Shuffler* (see section 3.3.1), I noticed that its output provides a perceptual bridge between monaural in-head localisation and binaural externalisation. Some features are retained from the originating binaural signal allowing for a partial externalisation, while others, due to their monaural compatibility, enable panpot-like processing for achieving a variable in-head stereo width. In later implementations of *Parisflâneur*, such a Blumlein shuffled stereo signal is used as an intermediate transition phase for gradually opening the monaural in-head spot, until the listener's head is "left" by fading into the immersive binaural recording.

4 Conclusion

In this paper, I presented an intermedia installation of mine called *Parisflâneur*. It takes place in auditory space which is presented binaurally via headphones. The work incorporates seven urban and rural sound situations arranged in a virtual scene that is navigable by bodily motion and orientation by listening. Upon interaction, each of the sound situations can be entered, that is, the virtual scene can be left in favour of the original static, binaural recording of that situation. Subsequent movements do not allow for a further navigation within the situation, instead, the virtual scene will be rearranged, which becomes audible only after having left again the static recording.

I described in detail the visitor's experience of the installation and realisation alternatives using FLOSS tools. By doing so, I tried to relate technical implementation details to both common approaches as suggested by scholarly research and to alternative findings driven by aesthetic reflection and artistic experimentation. One of my central arguments is that the design of virtual audio environments always has to reference the aesthetic experience and the condi-

tions of their reception rather than explicit or implicit real-world situations. The presentation of spaces by transforming media such as binaural audio technology is not a real-world experience in the sense of sound propagation directly and solely through air.

I aimed at pointing out that FLOSS tools are a precondition for *artistic engineering* as performed in the presented project. As any given approach or process is subject to critical reflection and potential modification, the implementations involved have to be accessible anywhere in the signal path and at any level that turns out to be appropriate. Neither would it be possible for me (and probably for any artist) to implement all the building blocks myself that require a deep access to their inner mechanisms, nor would monolithic and closed software allow for entangling artistic quest, aesthetic reflection and engineering ambition as attempted to exemplify in this paper.

5 Acknowledgements

Parisflâneur was initally conceived in 2008 during two short-term scientific missions at *Institute of Electronic Music and Acoustics Graz* (IEM), funded by the *Sonic Interaction Design* European COST action (COST IC 0601, [Rocchesso, 2011]). The installation has been further developed within the *Klangräume* research project (2013–2015), supported by *Zukunftsfonds Steiermark* (funds for the future development of the region of Styria, Austria) as part of the programme *Exciting Science and Social Innovations.*

References

Fons Adriaensen. 2006a. Acoustical impulse response measurement with ALIKI. In *Proceedings of Linux Audio Conference*, pages 9–14. ZKM.

Fons Adriaensen. 2006b. Design of a convolution engine optimised for reverb. In *Proceedings of Linux Audio Conference*, pages 49–53. ZKM.

Jens Blauert. 1997. *Spatial Hearing.* MIT Press.

Jérôme Daniel. 2003. Spatial sound encoding including near field effect: Introducing distance coding filter and a viable, new ambisonic format. In *Proceedings of 23rd AES International Conference.*

Michael Gerzon. 1994. Applications of blumlein shuffling to stereo microphone techniques. *Journal of the Audio Engineering Society*, 42(6):435–453.

Florian Hollerweger and Martin Rumori. 2013. Production and application of room impulse responses for multichannel setups using FLOSS tools. In *Proceedings of Linux Audio Conference*, pages 125–132. IEM.

Thor Magnusson. 2008. Expression and time: The question of strata and time management in creative practices using technology. In Aymeric Mansoux and Marloes de Valk, editors, *FLOSS + Art*, pages 232–247. GOTO10/OpenMute.

Lev Manovich. 2013. *Software Takes Command.* Number 5 in International Texts in Critical Media Aesthetics. Bloomsbury.

Markus Noisternig, Thomas Musil, Alois Sontacchi, and Robert Höldrich. 2003. 3 D binaural sound reproduction using a virtual Ambisonic approach. In *IEEE International Symposium on Virtual Environments*, pages 174–178.

Rui Penha. 2008. Distance encoding in Ambisonics using three angular coordinates. In *Proceedings of Sound and Music Computing Conference.*

Davide Rocchesso. 2011. *Explorations in Sonic Interaction Design.* Logos.

Martin Rumori, Florian Hollerweger, and Andrés Cabrera. 2010. Binaural room impulse responses for composition, documentation, virtual acoustics and audio augmented environments. In *Proceedings of 26th VDT International Convention*, pages 670–679. VDT.

Martin Rumori. 2016. Konstruierte Räume – ästhetische Implikationen von Verfahren und Werkzeugen der Binauraltechnik. In *Proceedings of 29th VDT International Convention*, pages 210–216. VDT. [german].

Martin Rumori. 2017. Space and body in sound art: Artistic explorations in binaural audio augmented environments. In Clemens Wöllner, editor, *Body, Sound and Space in Music and Beyond: Multimodal Explorations*, pages 235–256. Routledge. Forthcoming.

Pavel Zahorik. 2002. Auditory display of sound source distance. In *Proceedings of International Conference on Auditory Display.* ICAD.

A versatile workstation for the diffusion, mixing, and post-production of spatial audio

Thibaut CARPENTIER
UMR 9912 STMS IRCAM–CNRS–UPMC
1, place Igor Stravinsky,
75004 Paris, France,
thibaut.carpentier@ircam.fr

Abstract

This paper presents a versatile workstation for the diffusion, mixing, and post-production of spatial sound. Designed as a virtual console, the tool provides a comprehensive environment for combining channel–, scene–, and object–based audio. The incoming streams are mixed in a flexible bus architecture which tightly couples sound spatialization with reverberation effects. The application supports a broad range of rendering techniques (VBAP, HOA, binaural, etc.) and it is remotely controllable via the Open Sound Control protocol.

Keywords

sound spatialization, mixing, post-production, object-based audio, Ambisonic

1 Introduction

This paper presents a port of the *panoramix* workstation to Linux. First, we give a brief presentation of *panoramix* and typical use-cases of this environment. Then, we present some recently added features and discuss the challenges involved with porting the application to the Linux OS.

Panoramix is an audio workstation that was primarily designed for the post-production of 3D audio materials. The needs and motivations for such tool have been discussed in previous publications [Carpentier, 2016; Carpentier and Cornuau, 2016]: *panoramix* typically addresses the post-production of mixed music concerts[1] where the sound recording involves a large set of heterogeneous elements (close microphones, ambient miking, surround or Ambisonic microphone arrays, electronic tracks, etc). During the post-production stage, the sound engineers need tools for spatializing sonic sources (e.g., spot microphones or electronic tracks), encoding and decoding Ambisonic materials, adding artificial reverberation, combining and mixing the heterogeneous sound layers, as well as rendering, monitoring and exporting the final mix in multiple formats. *Panoramix* provides a unified framework covering all the required operations, and it allows to seamlessly integrate all spatialization paradigms: channel-based, scene-based, and object-based audio.
Besides post-production purposes, *panoramix* is also suitable for the diffusion of sound in live events since the audio engine operates in realtime and without latency.[2] Indeed, it has recently been used by sound engineers and computer musicians in order to control the sound spatialization for live productions at Ircam.

2 Architecture

The general architecture of the workstation has been presented in previous work [Carpentier, 2016]. In a nutshell, the *panoramix* signal flow consists of input tracks which are sent to busses dedicated to spatialization and reverberation effects. All busses are ultimately collected into the Master strip, which delivers the signals to the output audio driver. Each channel strip in the workstation comes with a set of specific DSP features.
One major improvement of the new version herein presented is the introduction of "parallel bussing". Namely, this means that each track can be sent to multiple busses in parallel.[3] The benefit of such parallel bussing architecture is

[1] The practical use of the software in such a context has also been demonstrated in the above-mentioned publications, through the case study of an electro-acoustic piece by composer Olga Neuwirth.

[2] Only a few specific DSP treatments may induce a latency, e.g., the encoding of Eigenmike signals (discussed later in this paper). Also there is the irreducible latency of the audio I/O device.

[3] The number of parallel sends is currently restricted to three busses, referred to as A/B/C. In practical mixing situations, it appeared useless to provide more than three sends although there is no technical constraint to increase this limit.

twofold; it allows:
1) to simultaneously produce a mix in multiple formats: tracks can for instance be sent to a VBAP bus and to an Ambisonic bus; both busses are rendered in parallel, with shared settings, and it is fast and easy to switch from one to another e.g., for A/B comparison.
2) to "hybridize" spatialization techniques: for instance, when producing binaural mixes, it is sometimes useful to combine "true" binaural synthesis (or recordings) with conventional stereophony. Adjusting the level of the two parallel busses, the sound engineer can balance between the 3D layer (with well-known binaural artifacts such as timbral coloration, front-back confusions, in-head localization, etc.) and the stereo layer (often considered as more robust and spectrally transparent). Such hybridization appeared especially useful and convincing when producing content intended for non-individual HRTF listening conditions.

Figures 1 and 2 present the signal-flow graph of the tracks and busses respectively. They also exhibit how the signal processing blocks relate to the controllers exposed in the user interface (see also Figure 4 for a general view of this interface).
When parallel bussing is involved, some elements of the depicted audio graph are replicated and run concurrently.

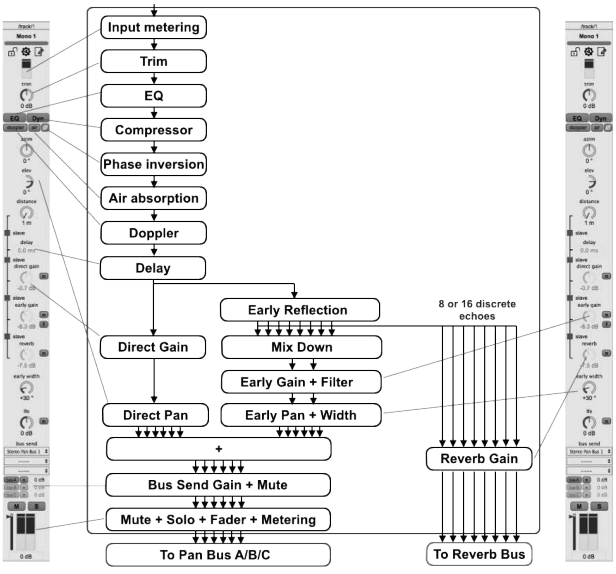


Figure 1: Anatomy of a track: a track is essentially used for pre-processing the incoming audio source (compression, equalization, delay, etc.) and for generating a set of early reflections that will later 1) feed the late reverb FDN and 2) be spatialized.

The overall processing architecture is inspired from the Spat design [Jot and Warusfel, 1995; Jot, 1999; Carpentier et al., 2015] which tightly combines an artificial reverberation engine with a panning module. This framework relies on a simplified space-time-frequency model of room acoustics wherein the generated room effect is divided in four temporal segments (direct sound, early reflections, late reflections, and reverb tail); each segment is individually filtered and then spatialized (direct sound and early reflections are localized as point sources while the late segments are spatially diffuse).
In the first release of *panoramix*, only the filtering of direct sound was proposed. In the presented version, we have introduced additional filters for the early and late reflection sections, therefore extending the range of possible effects.

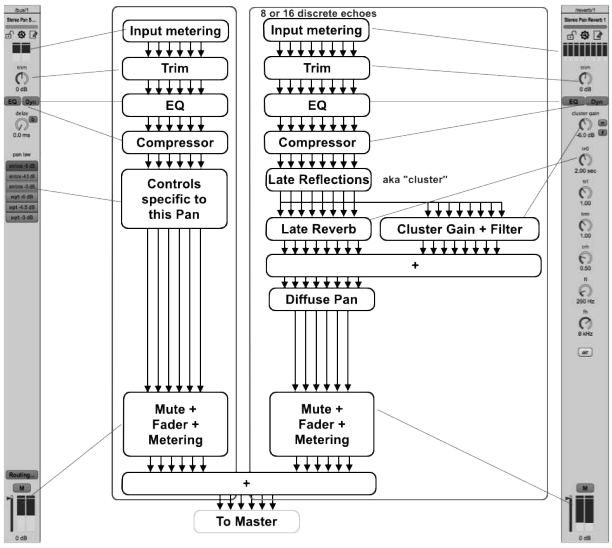


Figure 2: Anatomy of a bus: the purpose of a bus is twofold: it generates a late/diffuse reverberation tail (shared amongst multiple tracks for efficiency) and it provides control over the spatialization rendering. The lefthand side (violet frame) depicts the panning bus; the righthand side (red frame) represents the late reverb bus.

Note that the number of tracks, busses and channel per strip is unlimited, only restricted by the available computing power.

3 Main features

This section presents the main functionalities of the software, with an emphasis on newly added features. The interested reader may also refer to [Carpentier, 2016].

3.1 2D panpot

The first version of *panoramix* was focusing exclusively on 3D rendering approaches, namely VBAP [Pulkki, 1997], Higher Order Ambisonics (HOA) [Daniel, 2001], and binaural [Møller, 1992]. It rapidly appeared convenient to also integrate 2D techniques, as it is common practice to add horizontal-only layers even when mixing for 3D formats. A number of traditional 2D techniques have therefore been implemented (time and/or intensity panning laws such as 2D-VBAP or VBIP [Pernaux et al., 1998], etc). The workstation now offers a broad range of algorithms, being able to address arbitrary loudspeaker layouts.

3.2 Ambisonic processing

Higher Order Ambisonic (HOA) is a recording and reproduction technique that can be used to create spatial audio for circular or spherical loudspeaker arrangements. It has been supported in the workstation since its origin, and further improvements have been made, especially in the encoding and transformation modules.

3.2.1 HOA encoding

Compact spherical microphone arrays such as the Eigenmike[4] are sometimes used for music recordings as they are able to capture natural sound fields with high spatial resolution. The signals captured by such pickup systems do not directly correspond to HOA components; an encoding stage is required. Such encoding usually necessitates to regularize the modal radial filters as they are ill-conditioned for certain frequencies. Various equalization approaches have been proposed in the literature, in particular: Tikhonov regularization [Moreau, 2006; Daniel and Moreau, 2004], soft-limiting [Bernschütz et al., 2011], filter bank applied in the modal domain [Baumgartner et al., 2011]. There is yet no consensus about which method is the most appropriate; consequently they have all been implemented in *panoramix*. An adjustable maximum amplification factor is also controllable by the user.

Besides HOA recordings, it is also possible to synthesize Ambisonic virtual sources and there is no restriction on the maximum encoding order.

Note finally that *panoramix* supports all usual HOA normalization (N3D, N2D, SN3D, SN2D, FuMa, MaxN) and sorting (ACN, SID, Furse-Malham) schemes.

[4] `http://www.mhacoustics.com`

3.2.2 HOA manipulations

One benefit of the Ambisonic formalism is that a HOA stream can be flexibly manipulated so as to alter the spatial properties of the sound field. In addition to 3D rotations of the sound field [Daniel, 2001; Daniel, 2009], two new transformation operators have been recently integrated to the workstation:

1) a directional loudness processor [Kronlachner and Zotter, 2014] which allows to spatially emphasize certain regions of the sound field (Figure 3) and

2) a spatial blur effect [Carpentier, 2017] which reduces the resolution of an Ambisonic stream, indeed simulating fractional order representation and varying the "bluriness" of the spatial image.

These transformation operators are achieved by applying a (time and frequency independent) transformation matrix in the Ambisonic domain. The implementation is therefore very efficient, making them suitable for realtime automation.

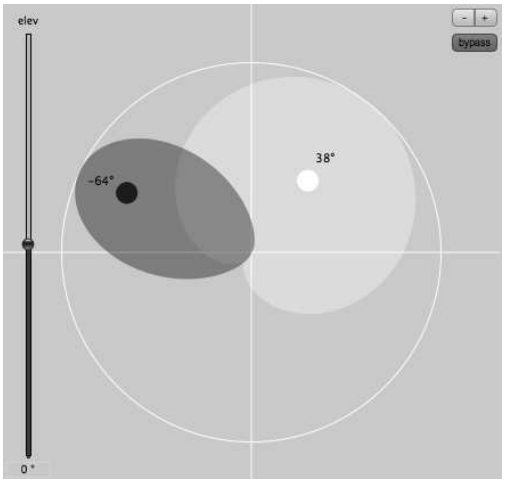


Figure 3: HOA focalization interface: the simple user interface allows to steer one or multiple virtual beams in space; the radial axis is used to control the "selectivity" of the virtual beam (from omnidirectional to highly directional). This is especially useful in post-production contexts, either to emphasize the sound from certain directions (e.g., instruments) or to attenuate undesired regions.

3.2.3 HOA decoding

A HOA bus serves as a decoder (with respect to a given loudspeaker layout) and it comes with a comprehensive set of decoding flavors including: sampling Ambisonic decoder [Daniel,

2001], mode-matching [Daniel, 2001], energy-preserving [Zotter et al., 2012], and all-round decoding [Zotter and Frank, 2012]. In addition, dual-band decoding is possible, with adjustable crossover frequency, and in-phase or max-re optimizations can be applied in each band [Daniel, 2001].

3.3 Binaural rendering

Panoramix implements binaural synthesis for 3D rendering over headphones. It is possible to load HRTF in the SOFA/AES-69 format [Majdak et al., 2013]. Two SOFA conventions are currently supported: "SimpleFreeFieldHRIR" for convolution with HRIR, and "SimpleFreeFieldSOS" for filtering with HRTF represented as second-order sections and interaural time delay.[5]
SOFA data can be either loaded from a local file or remotely accessed through the OpenDAP protocol [Carpentier, 2015a; Carpentier et al., 2014a]. The binaural bus features a user interface for rapid navigation/search through the available SOFA files (Figure 5).

Figure 5: UI for loading or downloading SOFA files. ① Filters for quick search. ② Text search field. ③ Results matching query.

3.4 Reverberation

As mentioned in previous sections, *panoramix* embeds a reverberation engine that allows to generate artificial room effects during the mixing process. The reverb processor currently used is a feedback delay network (FDN) originally designed by [Jot and Chaigne, 1991]. This FDN is particularly flexible and scalable; in typical use-cases, it involves eight feedback channels and provides decay control in three frequency bands.
In addition to that, there is an on-going work to further integrate convolution-based or hybrid reverberators [Carpentier et al., 2014b] in the bus architecture.

[5] see `www.sofaconventions.org` for further details on SOFA conventions.

3.5 OSC communication

All parameters of the *panoramix* application can be remotely accessed via the Open Sound Control (OSC) protocol [Wright, 2005]. This fosters easy and efficient communication with other applications (e.g., Pd) or external devices (e.g., head-tracker for realtime binaural rendering).
OSC communication may also be used for remote automation with a digital audio workstation (DAW) through the ToscA plugin [Carpentier, 2015b]. Note, however, that the latter has not yet been ported to the Linux platform.

A dedicated window allows to monitor the current OSC state of the *panoramix* engine (see ⑦ in Figure 4). Also, the mixing session itself is stored to disk as a "stringified" OSC bundle (human readable and editable).

3.6 Enhanced productivity

A number of other features have been added for enhanced productivity, compared to previous versions. This includes: a large set of keyboard shortcuts (the key mapping can further be customized and stored – see ⑧ in Figure 4) for handling most common tasks (create new tracks, enable/disable groups, etc.), tooltip pop-up that present inline help tips, the possibility to split the console window in multiple windows (especially useful when using multiple screens and dealing with a high number of tracks), etc.

4 Software aspects and Linux port

Panoramix was originally developed as a set of two Max/MSP[6] externals (`panoramix~` for the DSP rendering and `panoramix` for the GUI controller) and released in the form of a Max standalone application for macOS and Windows.
The DSP code is written is `C++`. It is OS-independent, host-independent (i.e. it does not rely on Max/MSP) and highly optimized, extensively using vectorized SIMD instructions and high performance functions from the Intel® Integrated Performance Primitives.[7] The application can easily handle dozens or even hundreds of tracks on a modern computer. The GUI component, also written in `C++`, is

[6] `http://www.cycling74.com`
[7] `http://software.intel.com`

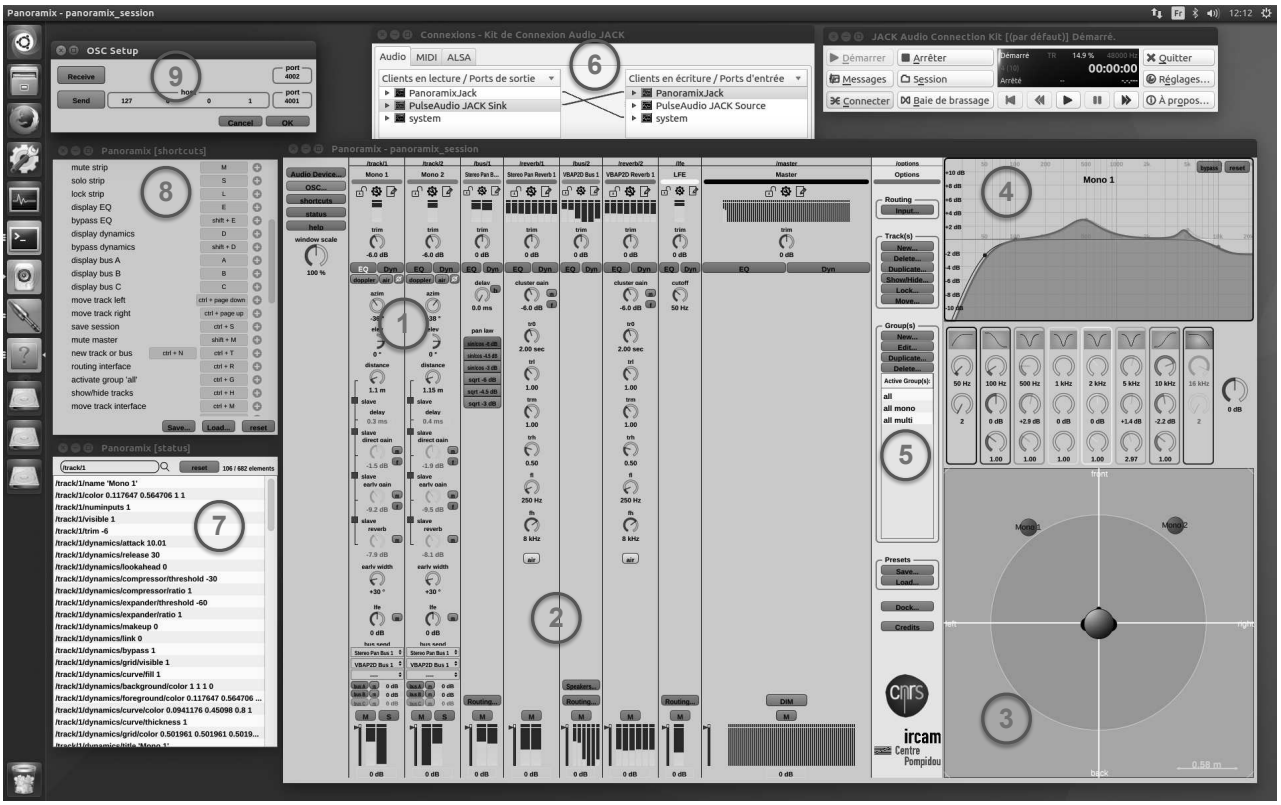


Figure 4: *panoramix* running in Ubuntu with Jack server (QJackctl) as audio pilot. ① Input tracks in the mixing console. ② Spatialization and reverberation busses. ③ Geometrical representation of the sound scene. ④ Parametric equalizer. ⑤ Group management. ⑥ Jack server and inter-application connections. ⑦ Status window: allows to inspect the current state of the engine and all parameters exposed to OSC messaging. ⑧ Shortcut window: allows to edit the key mappings. ⑨ OSC setup window: configure input and output port for remote communication.

built with the Juce[8] framework which facilitates cross-platform development and provides a large set of useful widgets.

For the Linux environment, it was first envisioned to port the Max externals to Pure Data (Pd) [Puckette, 1997]. Porting the DSP engine is straightforward as the Max and Pd APIs are relatively similar in this regard. Porting the GUI object, however, was problematic: Pd uses Tcl/Tk as its windowing system, and to the best of the author's knowledge there is no easy way to embed GUI components developed with other frameworks (such as Juce or Qt) in the Tk engine. As an alternative, it was decided to create an autonomous application, handling both the GUI and the audio engine (i.e. an "AudioAppComponent" in Juce's dialect). The application thus operates independently of any host engine (Pd or Max) and it processes the audio directly to/from the audio devices. Furthermore, it is compatible with the Jack Audio Connection Kit,[9] which makes it pluggable with potentially any audio application. In typical use-cases, a digital audio workstation such as Ardour[10] is used to send audio streams to the *panoramix* processor. The processed buffers may be re-routed to the DAW, e.g., for bouncing, or directly played back through the output device (see Figure 6).

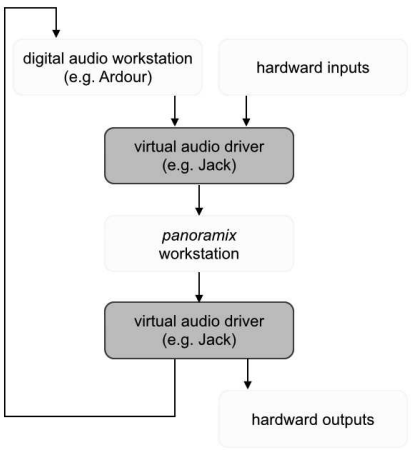


Figure 6: Typical workflow.

[8] http://juce.com/
[9] http://www.jackaudio.org
[10] http://www.ardour.org

5 Conclusions

This paper discussed the Linux port of an audio engine designed for the diffusion, mixing, and post-production of spatial audio. We highlighted several new features that extend the possibilities of the tool and improve productivity and user experience. Future work will mainly focus on the integration of convolution-based reverberation into the framework herein presented.

6 Acknowledgements

The author is very grateful to Clément Cornuau, Olivier Warusfel, and the sound engineering team at Ircam for their invaluable help in the conception of this tool. Thanks also to Anders Vinjar for his insightful advices and beta testing on the Linux platform.

References

Robert Baumgartner, Hannes Pomberger, and Matthias Frank. 2011. Practical implementation of radial filters for ambisonic recordings. In *Proc. of the 1st International Conference on Spatial Audio (ICSA)*, Detmold, Germany, Nov.

Benjamin Bernschütz, Christoph Pörschmann, Sascha Spors, and Stefan Weinzierl. 2011. Soft-limiting der modalen Amplitudenverstärkung bei sphärischen Mikrofonarrays im Plane Wave Decomposition Verfahren. In *Proc. of 37th German Annual Convention on Acoustics (DAGA)*, Düsseldorf, Germany, March.

Thibaut Carpentier and Clément Cornuau. 2016. panoramix: station de mixage et post-production 3D. In *Proc. of the Journées d'Informatique Musicale (JIM)*, pages 162 – 169, Albi, France, April.

Thibaut Carpentier, Hélène Bahu, Markus Noisternig, and Olivier Warusfel. 2014a. Measurement of a head-related transfer function database with high spatial resolution. In *Proc. of the 7th EAA Forum Acusticum*, Kraków, Poland, Sept.

Thibaut Carpentier, Markus Noisternig, and Olivier Warusfel. 2014b. Hybrid Reverberation Processor with Perceptual Control. In *Proc. of the 17th Int. Conference on Digital Audio Effects (DAFx)*, pages 93 – 100, Erlangen, Germany, Sept.

Thibaut Carpentier, Markus Noisternig, and Olivier Warusfel. 2015. Twenty Years of Ircam Spat: Looking Back, Looking Forward. In *Proc. of the 41st International Computer Music Conference (ICMC)*, pages 270 – 277, Denton, TX, USA, Sept.

Thibaut Carpentier. 2015a. Binaural synthesis with the Web Audio API. In *Proc. of the 1st Web Audio Conference (WAC)*, Paris, France, Jan.

Thibaut Carpentier. 2015b. ToscA: An OSC Communication Plugin for Object-Oriented Spatialization Authoring. In *Proc. of the 41st International Computer Music Conference*, pages 368 – 371, Denton, TX, USA, Sept.

Thibaut Carpentier. 2016. Panoramix: 3D mixing and post-production workstation. In *Proc. of the 42nd International Computer Music Conference (ICMC)*, pages 122 – 127, Utrecht, Netherlands, Sept.

Thibaut Carpentier. 2017. Ambisonic spatial blur. In *Proc. of the 142nd Convention of the Audio Engineering Society (AES)*, Berlin, Germany, May.

Jérôme Daniel and Sébastien Moreau. 2004. Further Study of Sound Field Coding with Higher Order Ambisonics. In *Proc. of the 116th Convention of the Audio Engineering Society (AES)*, Berlin, Germany, May.

Jérôme Daniel. 2001. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia*. Ph.D. thesis, Université de Paris VI.

Jérôme Daniel. 2009. Evolving Views on HOA : From Technological To Pragmatic Concerns. In *Proc. of the 1st Ambisonics Symposium*, Graz, Austria, June.

Jean-Marc Jot and Antoine Chaigne. 1991. Digital delay networks for designing artificial reverberators. In *Proc. of the 90th Convention of the Audio Engineering Society (AES)*, Paris, France, Feb.

Jean-Marc Jot and Olivier Warusfel. 1995. A Real-Time Spatial Sound Processor for Music and Virtual Reality Applications. In *Proc. of the of the International Computer Music Conference (ICMC)*, Banff, Canada.

Jean-Marc Jot. 1999. Real-time spatial processing of sounds for music, multimedia and

interactive human-computer interfaces. *ACM Multimedia Systems Journal (Special issue on Audio and Multimedia)*, 7(1):55 – 69.

Matthias Kronlachner and Franz Zotter. 2014. Spatial transformations for the enhancement of Ambisonic recordings. In *Proc. of the 2nd International Conference on Spatial Audio (ICSA)*, Erlangen, Germany, Feb.

Piotr Majdak, Yukio Iwaya, Thibaut Carpentier, Rozenn Nicol, Matthieu Parmentier, Agnieszka Roginska, Yôiti Suzuki, Kanji Watanabe, Hagen Wierstorf, Harald Ziegelwanger, and Markus Noisternig. 2013. Spatially Oriented Format for Acoustics: A Data Exchange Format Representing Head-Related Transfer Functions. In *Proc. of the 134th Convention of the Audio Engineering Society (AES)*, Roma, Italy, May 4-7.

Henrik Møller. 1992. Fundamentals of binaural technology. *Applied Acoustics*, 36:171 – 218.

Sébastien Moreau. 2006. *Étude et réalisation d'outils avancés d'encodage spatial pour la technique de spatialisation sonore Higher Order Ambisonics : microphone 3D et contrôle de distance*. Ph.D. thesis, Université du Maine.

Jean-Marie Pernaux, Patrick Boussard, and Jean-Marc Jot. 1998. Virtual Sound Source Positioning and Mixing in 5.1 Implementation on the Real-Time System Genesis. In *Proc. of the Digital Audio Effects Conference (DAFx)*, Barcelona, Spain, Nov.

Miller Puckette. 1997. Pure Data. In *Proc. of the International Computer Music Conference (ICMC)*, pages 224 – 227, Thessaloniki, Greece.

Ville Pulkki. 1997. Virtual Sound Source Positioning Using Vector Base Amplitude Panning. *Journal of the Audio Engineering Society*, 45(6):456 – 466, June.

Matthew Wright. 2005. Open Sound Control: an enabling technology for musical networking. *Organised Sound*, 10(3):193 – 200, Dec.

Franz Zotter and Matthias Frank. 2012. All-round ambisonic panning and decoding. *Journal of the Audio Engineering Society*, 60(10):807 – 820.

Franz Zotter, Hannes Pomberger, and Markus Noisternig. 2012. Energy-preserving ambisonic decoding. *Acta Acustica united with Acustica*, 98:37 – 47.

Teaching Sound Synthesis in C/C++ on the Raspberry Pi

Henrik von Coler and **David Runge**
Audio Communication Group, TU Berlin
voncoler@tu-berlin.de
david.runge@campus.tu-berlin.de

Abstract

For a sound synthesis programming class in C/C++, a Raspberry Pi 3 was used as runtime and development system. The embedded system was equipped with an Arch Linux ARM, a collection of libraries for sound processing and interfacing, as well as with basic examples. All material used in and created for the class is freely available in public git repositories. After a unit of theory on sound synthesis and Linux system basics, students worked on projects in groups. This paper is a progress report, pointing out benefits and drawbacks after the first run of the seminar. The concept delivered a system with acceptable capabilities and latencies at a low price. Usability and robustness, however, need to be improved in future attempts.

Keywords

Jack, Raspberry Pi, C/C++ Programming, Education, Sound Synthesis, MIDI, OSC, Arch Linux

1 Introduction

The goal of the seminar outlined in this paper was to enable students with different backgrounds and programming skills the development of standalone real-time sound synthesis projects. Such a class on the programming of sound synthesis algorithms is, among other things, defined by the desired level of depth in signal processing. It may convey an application-oriented overview or a closer look at algorithms on a sample-wise signal processing level, as in this case. Based on this decision, the choice of tools, respectively the programming environment should be made.

Script languages like Matlab or Python are widely used among students and offer a comfortable environment, especially for students without a background in computer science. They are well suited for teaching fundamentals and theoretical aspects of signal processing due to advanced possibilities of debugging and visualisation. Although real-time capabilities can be added, they are not considered for exploring applied sound synthesis algorithms in this class.

In a more application-based context, graphical programming environments like Pure Data (Pd), MAX MSP or others would be the first choice. They allow rapid progress and intermediate results. However, they are not the best platform to enhance the knowledge of sound synthesis algorithms on a sample-wise level by nature.

C/C++ delivers a reasonable compromise between low level access and the comfort of using available libraries for easy interfacing with hardware. For using C/C++ in the development of sound synthesis software, a software development kit (SDK) or application programming interface (API) is needed in order to offer access to the audio hardware.

Digital audio workstations (DAW) use plugins programmed with SDKs and APIs like Steinberg's VST, Apple's Audio Units, Digidesign's RTAS, the Linux Audio Developer's Simple Plugin API (LADSPA) and its successor LV2, which offer quick access to developing audio synthesis and processing units. A plugin-host is necessary to run the resulting programs and the structure is predefined by the chosen platform. The JUCE framework [34] offers the possibility of developing cross platform applications with many builtin features. Build targets can be audio-plugins for different systems and standalone applications, including Jack clients, which would present an alternative to the chosen approach.

Another possibility is the programming of Pd externals in the C programming language with the advantage of quick merging of the self-programmed components with existing Pd internals. FAUST [6] also provides means for creating various types of audio plugins and standalone applications. Due to a lack in functional programming background it was not chosen.

For various reasons we settled for the Jack API [18] to develop command line programs on a Linux system.

- Jack clients are used in the research on binaural synthesis and sound field synthesis, for example by the WONDER interface [15] or the SoundScape Renderer [14]. Results of the projects might thus be potentially integrated into existing contexts.
- Jack clients offer quick connection to other clients, making them as modular as audio-plugins in a DAW.
- The Jack API is easy to use, even for beginners. Once the main processing function is understood, students can immediately start inserting their own code.
- The omission of a graphical user interface for the application leaves more space for focusing on the audio-related problems.
- The proposed environment is independent of proprietary components.

A main objective of the seminar was to equip the students with completely identical systems for development. This avoids troubles in handling different operating systems and hardware configurations. Since no suitable computer pool was available, we aimed at providing a set of machines as cheap as possible for developing, compiling and running the applications. The students were thus provided with a Raspberry Pi 3 in groups of two. Besides being one of the cheapest development systems, it offers the advantage of resulting in a highly portable, quasi embedded synthesizer for the actual use in live applications.

The remainder of this paper is organized as follows: Section 2 introduces the used hard- and software, as well as the infrastructure. Section 3 presents the concept of the seminar. Section 4 briefly summarizes the experiences and evaluates the concept.

2 Technical Outline

2.1 Hardware

A Raspberry Pi 3 was used as development and runtime system. The most recent version at that time was equipped with 1.2GHz 64-bit quad-core ARMv8 CPU, 802.11n Wireless LAN, a Bluetooth adapter, 1GB RAM, 4 USB ports 40 GPIO pins and various other features [11].

Unfortunately the on-board audio interface of the Raspberry Pi could not be configured for real-time audio applications. It does not feature an input and the Jack server could only be started with high latencies. After trying several interfaces, a Renkforce USB-Audio-Adapter was chosen, since it delivered an acceptable performance at a price of 10 €. The Jack server did perform better with other interfaces, yet at a higher price and with a larger housing.

Students were equipped with MIDI interfaces from the stock of the research group and private devices. The complete cost for one system, including the Raspberry Pi 3 with housing, SD card, power adapter and the audio interface, were thus kept at about 70 €(vs. the integrated low-latency platform bela[5], that still ranks at around 120 €per unit).

2.2 Operating System

First tests on the embedded system involved Raspbian [12], as it is highly integrated and has a graphical environment preinstalled, that eases the use for beginners. Integration with the libraries used for the course proved to be more complicated however, as they needed to be added to the software repository for the examples.

A more holistic approach, integrating the operating system, was aimed at, in order to leave as few dependencies on the students' side as possible and not having to deal with the hosting of a custom repository of packages for Raspbian. Due to previous experience with low latency setups using Arch Linux [7], Arch Linux ARM [2] was chosen. With its package manager pacman [9] and the Arch Build System [1] an easy system-wide integration of all used libraries and software was achieved by providing them as preinstalled packages (with them either being available in the standard repositories, or the Arch User Repository[3]). Using Arch Linux ARM, it was also possible to guarantee a systemd [13] based startup of the needed components, which is further described in Section 2.5. At the time of preparation for the course, the 64bit variant (AArch64) - using the mainline kernel - was not available yet. At the time of writing it is still considered experimental, as some vendor libraries are not yet available for it. Instead the ARMv7 variant of the installation image was used, which is the default for Raspberry Pi 2.

2.3 Libraries

All libraries installed on the system are listed in Tab. 1. For communicating with the audio hardware, the Jack API was installed. The jackcpp framework adds a C++ interface to Jack. Additional libraries allowed the handling of audio file formats, ALSA-MIDI interfacing, Open Sound Control, configuration files and Fast Fourier Transforms.

Table 1: Libraries installed on the development system

Library	Ref.	Purpose
jack2	[18]	Jack audio API
jackcpp	[26]	C++ wrapper for jack2
sndfile	[19]	Read and write audio files
rtmidi	[32]	Connect to MIDI devices
liblo	[23]	OSC support
yaml	[20]	Configuration files
fftw3	[22]	Fourier transform
boost	[4]	Various applications

2.4 Image

The image for the system is available for download[1] and can be asked for by mailing to the authors in case of future unavailability. Installation of the image follows the standard procedure of an Arch Linux ARM installation for Raspberry Pi 3, using the blockwise copy tool dd, which is documented in the course's git repository [31].

2.5 System Settings

The most important goal was to achieve a round-trip latency below 10 ms. This would not be sufficient for real-time audio applications in general, but for teaching purposes. With the hardware described in Section 2.1, stable Jack server command line options were evaluated, leading to a round-trip latency of 2.9 ms:

```
/usr/bin/jackd -R \K
        -p 512 \
        -d alsa \
                -d hw:Device \
                -n 2 \
                -p 64 \
                -r 44100
```

[1] https://www2.ak.tu-berlin.de/~drunge/klangsynthese

As these settings were not realizable with the internal audio card, the *snd-bcm2835* module - the driver in use for it - was blacklisted using */etc/modprobe.d/**, to not use the sound device at all.

For automatic start of the low-latency audio server and its clients, systemd user services were introduced, that follow a user session based setup. The session of the main system user is started automatically as per systemd's user@.service. This is achieved by enabling the linger status of said user with the help of loginctl [8], which activates its session during boot and starts its enabled services.
A specialized systemd user service [30] allows for Jack's startup with an elevated CPU scheduler without the use of dbus [21]. The students' projects could be automatically started as user services, that rely on the audio server being started by enabling services such as this example:

```
[Unit]
Description=Example project
After=jack@rpi-usb-44100. service
[Service]
ExecStart=/path/to/executable \
    parameter1 \
    paramter2
Restart=on-failure
[Install]
WantedBy=default.target
```

2.6 Infrastructure

For allowing all students in the class the access via SSH, a WIFI was set up, providing fixed IP addresses for all Raspberry Pis. Network performance showed to be insufficient to handle all participants simultaneously, though. Thus, additional parallel networks were installed.

For home use, students were instructed to provide a local network with their laptops, or using their home network over WiFi or cable. Depending on their operating system, this was more or less complicated.

3 The Seminar

The seminar was addressed to graduate students with different backgrounds, such as computer science, electrical engineering, acoustics and others. One teacher and one teaching assistant were involved in the planning, preparation and execution of the classes.

The course was divided into a theoretical and a practical part. In the beginning, theory and basics where taught in mixed sessions. The fi-

nal third of the semester was dedicated to supervised project work.

3.1 System Introduction

Students were introduced to the system by giving an overview over the tools needed and presenting the libraries with their interfaces. Users of Linux, Mac and Windows operating systems took part in the class. Since many students lacked basic Linux skills, first steps included using Secure Shell (SSH) to access the devices, which proved to be hard for attendees without any knowledge on the use of the command-line interface. Windows users were aided to install and use PuTTY [10] for the purpose of connecting, as there is no native SSH client. After two sessions, each group was able to reach the Raspberry Pi in class, as well as at home.

3.2 Sound Synthesis Theory

Sound synthesis approaches were introduced from an algorithmic point of view, showing examples of commercially available implementations, also regarding their impact on music production and popular culture. Students were provided with ready-to-run examples from the course repository for some synthesis approaches, as well as with tasks for extending these.

Important fundamental literature was provided by Zölzer [35], Pirkle [27] and Roads [29]. The taxonomy of synthesis algorithms proposed by Smith [33] was used to structure the outline of the class, as follows.

A section on *Processed Recording* dealt with sampling and sample-based approaches, like wave-table synthesis, granular synthesis, vector synthesis and concatenative synthesis.

Subtractive synthesis and analog modeling were treated as the combination of the basic units *oscillators*, *filters* and *envelopes*. Filters were studied more closely, considering IIR and FIR filters and design methods like bilinear transform. A ready to run biquad example was included and a first order low-pass was programmed in class.

Additive Synthesis and *Spectral Modeling* were introduced by an analysis-resynthesis process of a violin tone in the SMS model [25], considering only the harmonic part.

Physical Modeling was treated for plucked strings, starting with the Karplus-Strong algorithm [17], advancing to bidirectional [24].

FM Synthesis [16] was treated as a representative of abstract algorithms in the class. The concept was mainly taught by a closer look at the architecture and programming of the Yamaha DX7 synthesizer.

3.3 Projects

Out of the 35 students who appeared to the first meetings, 18 worked on projects throughout the whole semester. It should be noted that (with one exception) only Linux and MAC users continued.

No restrictions were made regarding the choice of the topic, exept that it should result in an executable program on the Raspberry PI. The student projects included:

- A vector synthesis engine, allowing the mixture of different waveforms with a succeeding filter section
- A subtractive modeling synth with modular capabilities
- A physical string model, based on the Karplus-Strong Extended with dispersion filter
- A sine-wave Speech Synthesis [28] effect, which includes a real-time FFT
- A guitar-controlled subtractive synthesizer, using zero-crossing rate for pitch detection
- A wave-digital-filter implementation with sensor input from the GPIOs

In order to provide a more suitable platform for running embedded audio applications, one group used buildroot[2] to create a custom operating system.

4 Conclusions

The use of the Raspberry Pi 3 for the programming of Jack audio applications showed to be a promising approach. A system with acceptable capabilities and latencies could be provided at a low price. Accessibility and stability, however, need to be improved in future versions: A considerable amount of time was spent working on these issues in class and the progress in the projects was therefore delayed considerably. The overhead in handling Linux showed to be a major problem for some students and probably caused some people to drop out. A possible step would be to provide a set with monitor, keyboard and mouse, as this would increase accessibility. The stability of the Jack server needs to be worked on, as sometimes the hardware would

[2] `https://buildroot.org`

not be started properly, leading to crashing Jack clients.

In future seminars, which are likely to be conducted after these principally positive experiences, most of the issues should be worked out and the image, as well as the repository will have been improved.

References

[1] Arch Build System - ArchWiki. https://wiki.archlinux.org/index.php/Arch_Build_System.

[2] Arch Linux ARM homepage. https://www.archlinuxarm.org/.

[3] Arch User Repository. https://aur.archlinux.org/.

[4] Boost C++ Libraries - Homepage. http://www.boost.org.

[5] buildroot homepage. https://bela.io.

[6] FAUST - Homepage. http://faust.grame.fr/.

[7] Linux Audio Conference 2015 - Workshop: Arch Linux as a lightweight audio platform - Slides. http://lac.linuxaudio.org/2015/download/lac2015_arch_slides.pdf.

[8] loginctl man page. https://www.freedesktop.org/software/systemd/man/loginctl.

[9] Pacman homepage. https://www.archlinux.org/pacman/.

[10] PuTTY Homepage. http://www.putty.org/.

[11] Raspberry PI Homepage. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/.

[12] Raspbian homepage. https://www.raspbian.org/.

[13] systemd homepage. https://www.freedesktop.org/wiki/Software/systemd/.

[14] Jens Ahrens, Matthias Geier, and Sascha Spors. The soundscape renderer: A unified spatial audio reproduction framework for arbitrary rendering methods. In *Audio Engineering Society Convention 124*. Audio Engineering Society, 2008.

[15] Marije AJ Baalman. Updates of the wonder software interface for using wave field synthesis. *LAC2005 Proceedings*, page 69, 2005.

[16] John M Chowning. The synthesis of complex audio spectra by means of frequency modulation. *Journal of the audio engineering society*, 21(7):526–534, 1973.

[17] Julius O. Smith David A. Jaffe. Extensions of the Karplus-Strong Plucked-String Algorithm. *Computer Music Journal*, 7(2):56–69, 1983.

[18] Paul Davies. JACK API. http://www.jackaudio.org/.

[19] Erik de Castro Lopo. Libsndfile. http://www.mega-nerd.com/libsndfile/.

[20] Clark C. Evans. YAML: YAML Ain't Markup Language. http://yaml.org/.

[21] Free Desktop Foundation. dbus homepage. https://wiki.freedesktop.org/www/Software/dbus/.

[22] Matteo Frigo and Steven G. Johnson. FFTW Fastest Fourier Transform in the West. http://www.fftw.org/.

[23] Steve Harris and Stephen Sinclair. liblo Homepage: Lightweight OSC implementation. http://liblo.sourceforge.net/.

[24] Matti Karjalainen, Vesa Välimäki, and Tero Tolonen. Plucked-string models: From the Karplus-Strong algorithm to digital waveguides and beyond. *Computer Music Journal*, 22(3):17–32, 1998.

[25] Scott N. Levine and Julius O. Smith. A Sines+Transients+Noise Audio Representation for Data Compression and Time/Pitch Scale Modi cations. *Proceedings of the 105th Audio Engineering Society Convention*, 1998.

[26] Alex Norman. JackCpp. http://www.x37v.info/projects/jackcpp/.

[27] Will Pirkle. *Designing Software Synthesizer Plug-Ins in C++*. Focal Press, 2014.

[28] Robert E Remez, Philip E Rubin, David B Pisoni, Thomas D Carrell, et al. Speech perception without traditional speech cues. *Science*, 212(4497):947–949, 1981.

[29] Curtis Roads. *The computer music tutorial*. MIT press, 1996.

[30] David Runge. uenv homepage. https://git.sleepmap.de/software/uenv.git/about/.

[31] David Runge and Henrik von Coler. AK-Klangsynthese Repository. https://gitlab.tubit.tu-berlin.de/henrikvoncoler/Klangsynthese_PI.

[32] Gary P. Scavone. RtMidi. http://www.music.mcgill.ca/~gary/rtmidi/.

[33] Julius O. Smith. Viewpoints on the History of Digital Synthesis. In *Proceedings of the International Computer Music Conference*, pages 1–10, 1991.

[34] Jules Storer. JUCE. https://www.juce.com/.

[35] Udo Zoelzer, editor. *DAFX: Digital Audio Effects* . John Wiley & Sons, Inc., New York, NY, USA, 2 edition, 2011.

Open signal processing software platform for hearing aid research (openMHA)

Tobias Herzke[1] and **Hendrik Kayser**[2] and **Frasher Loshaj**[1] and **Giso Grimm**[1,2] and **Volker Hohmann**[1,2]
[1] HörTech gGmbH and Cluster of Excellence "Hearing4all",
Marie-Curie-Str. 2, D-26129 Oldenburg, Germany
[2] Medizinische Physik and Cluster of Excellence "Hearing4all"
Universität Oldenburg, D-26111 Oldenburg, Germany
info@openmha.org

Abstract

Hearing aids help hearing impaired users participate in the communication society. Development and improvement of hearing aid signal processing algorithms takes place in the industry and in academic research. With openMHA, we present a development and evaluation platform that is able to execute hearing aid signal processing in real-time on standard computing hardware with a low delay between sound input and output. We lay out the application specific requirements and present how openMHA meets these and will be helpful in future research in the field of signal processing for hearing aids.

Keywords

Hearing aids, audio signal processing, plugin host

1 Introduction

Development of hearing aid signal processing is widely conducted by hearing aid manufacturers on proprietary systems that are not accessible to the research community and that underlie commercial constraints. Providing open tools to the hearing aid research community lowers barriers, accelerates studies with novel acoustic processing algorithms and facilitates translation of these advances into widespread use with hearing aids, cochlear implants, and consumer electronics devices for sub-clinical hearing support. A software platform for the development and evaluation of hearing aid algorithms should

- offer a complete set of hearing aid signal processing reference algorithms that can be combined with newly developed algorithms to form a complete hearing aid signal processing chain,
- enable researchers to perform offline-processing as well as real-time signal processing with a reliable low delay between acoustic input and output of less than 10 milliseconds, even when algorithms need significant processing power,
- provide a library for common signal processing tasks and commonly needed services in hearing aid signal processing, like support for acoustic calibration and filterbanks,
- be able to run on a wide range of hardware, from high-performance PCs to execute bleeding-edge algorithms in real-time, to portable, power-efficient, headless, battery-powered devices for improved testing capabilities in realistic usage scenarios and field tests.

Several open-source tools for audio signal processing, that can also be used in hearing aid research, exist:

Octave. Octave is actively used in hearing aid research for the development of signal processing algorithms for hearing aids. It is a suitable tool to quickly develop, change and evolve isolated algorithms as long as no real-time audio processing is required. However, Octave is unsuitable for executing hearing aid algorithms in real-time with live input and output sound signals with low delay.[Eaton et al., 2015]

NumPy/SciPy. Scientific Computing Tools for Python enable researchers to develop signal processing algorithms. Technically, this software platform is equivalent to octave, but it is to our knowledge currently not actively used in hearing aid research.[Jones et al., 2001]

Pure Data. Pd is a real-time signal processing platform. It features a graphical programming interface. Pd is actively used mainly by artists to perform signal processing of music and other data. Pd can achieve a low delay in real-time processing. In principle it would be possible to develop hearing aid signal processing algorithms on Pd, and have these algorithms process audio signal in real-time. We are not aware of any hearing aid research being performed on the Pd platform and would consider it too la-

borious to implement modern hearing aid algorithms in the graphical programming environment. Pd can be extended with C, therefore, hearing aid algorithms could be implemented for Pd in C or C++.[Puckette, 1996]

Plugin hosts. Various plugin hosts for different plugin architectures (VST, LADSPA, LV2) exist, that can load and combine algorithms in plugins to form complex signal processing chains. Most hosts can achieve a low delay in real-time audio processing. Plugins can be written in C or C++ using the plugin-architecture specific SDK. (Using the VST SDK requires signing a license agreement.) Plugin hosts are mainly used by sound engineers and also by artists to process recorded or live music and other sounds.

Signal processing toolboxes and languages. A signal processing toolbox like the Synthesis ToolKit (STK) [Cook and Scavone, 1999] and domain-specific languages (DSL) like SuperCollider [McCartney, 2002] and Faust [Orlarey et al., 2009] provide useful signal processing primitives to ease development of audio signal processing algorithms. We are not aware of any hearing aid research being performed using these toolboxes and DSLs.

While the dynamic programming languages Octave and Python are suitable to develop algorithms and execute them offline, their runtime environment is not suitable for real-time processing when low delay is required at high processing loads. Octave and Python do not give algorithm implementers the necessary control to prevent heap memory allocation in the signal processing path, which can cause unpredictable interruptions in the real-time processing due to priority inversion situations. Pd and plugin hosts are real-time safe themselves and allow algorithms to be implemented in C or C++. The C and C++ programming languages allow developers sufficient control to implement algorithms in a real-time safe way. However, Pd and plugin hosts do not provide commonly needed services to hearing aid signal processing developers like calibration or an existing set of hearing aid algorithms.

The HörTech Master Hearing Aid (MHA) [Grimm et al., 2006; Grimm et al., 2009a] is an existing software platform for hearing aid algorithm development and evaluation that meets all the requirements and has been used by the hearing aid industry as well as in academic research. Until recently, it was only available as a closed-source commercial product. To enable and facilitate collaborative research efforts and comparative studies in the research community, an open-source version of the MHA software platform for real-time audio signal processing is now being developed and made available: the open Master Hearing Aid (openMHA). In February 2017, a pre-release of the openMHA has been published on GitHub under an open-source license (AGPL3) by [HörTech gGmbH and Universität Oldenburg, 2017]. This pre-release features an initial set of reference algorithms for hearing aid processing, which will be expanded in subsequent releases. Thereby, openMHA provides a growing benchmark for the development and investigation of novel algorithms on this platform in the future. With the openMHA we provide an open-source tool that is tailored to the needs of hearing aid algorithm research which was not available before as a specialized tool in the open-source domain.

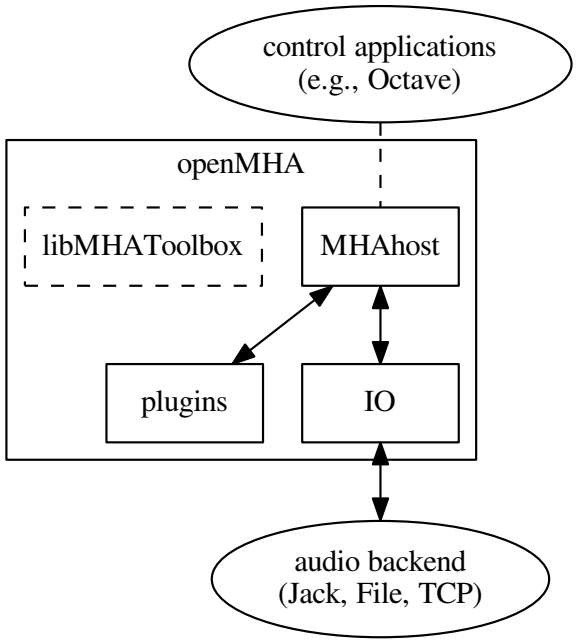


Figure 1: Structure of the openMHA. The openMHA contains a toolbox library "libMHAToolbox", a command line host application, which acts as an openMHA plugin host and provides the configuration interface, and openMHA plugins.

2 Structure

The openMHA can be split into four major components (see Figure 1 for an overview):

1. The openMHA command line application
2. Signal processing plugins
3. Audio input-output (IO) plugins
4. The openMHA toolbox library

The openMHA command line application acts as a plugin host. It can load signal processing plugins as well as audio input-output (IO) plugins. Additionally, it provides the command line configuration interface and a TCP/IP based configuration interface. Several IO plugins exist: For real-time signal processing, commonly the "MHAIOJack" plugin is used, which provides an interface to the Jack Audio Connection Kit (JACK) [Davis, 2003]. Other IO plugins provide audio file access or TCP/IP-based processing.

openMHA plugins provide the audio signal processing capabilities and audio signal handling. Typically, one openMHA plugin implements one specific algorithm. The complete virtual hearing aid signal processing can be achieved by a combination of several openMHA plugins.

The openMHA toolbox library "libMHAToolbox" provides reusable data structures and signal processing classes. Examples are class templates for the implementation of openMHA plugins, and container classes for audio data. Furthermore, several filter classes in temporal or spectral domain, filter banks, and hearing aid specific classes are provided in this library.

3 openMHA Platform Services and Conventions

The openMHA platform offers some services and conventions to algorithms implemented in plugins, that make it especially well suited to develop hearing aid algorithms, while still supporting general-purpose signal processing.

3.1 Audio Signal Domains

As in most other plugin hosts, the audio signal in the openMHA is processed in audio chunks. However, plugins are not restricted to propagate audio signal as blocks of audio samples in the time domain – another option is to propagate the audio signal in the short time Fourier transform (STFT) domain, i.e. as spectra of blocks of audio signal, so that not every plugin has to perform its own STFT analysis and synthesis. Since STFT analysis and re-synthesis of acceptable audio quality always introduces an algorithmic delay, sharing STFT data is a necessity for a hearing aid signal processing platform, because the overall delay of the complete processing has to be as short as possible.

Similar to some other platforms, the openMHA allows also arbitrary data to be exchanged between plugins through a mechanism called "algorithm communication variables" or short "AC vars". This mechanism is commonly used to share data such as filter coefficients or filter states.

3.2 Real-Time Safe Complex Configuration Changes

Hearing aid algorithms in the openMHA can export configuration settings that may be changed by the user at run time. To ensure real-time safe signal processing, the audio processing will normally be done in a signal processing thread with real-time priority, while user interaction with configuration parameters would be performed in a configuration thread with normal priority, so that the audio processing does not get interrupted by configuration tasks. Two types of problems may occur when the user is changing parameters in such a setup:

1. The change of a simple parameter exposed to the user may cause an involved recalculation of internal runtime parameters that the algorithm actually uses in processing. The duration required to perform this recalculation may be a significant portion of (or take even longer than) the time available to process one block of audio signal. In hearing aid usage, it is not acceptable to halt audio processing for the duration that the recalculation may require.

2. If the user needs to change multiple parameters to reach a desired configuration state of an algorithm from the original configuration state, then it may not be acceptable that processing is performed while some of the parameters have already been changed while others still retain their original values. It is also not acceptable to interrupt signal processing until all pending configuration changes have been performed.

The openMHA provides a mechanism in its toolbox library to enable real-time safe configuration changes in openMHA plugins: Basically, existing runtime configurations are used in the processing thread until the work of creating an

updated runtime configuration has been completed in the configuration thread. In hearing aids, it is more acceptable to continue to use an outdated configuration for a few more milliseconds than blocking all processing. The openMHA toolbox library provides an easy-to-use mechanism to integrate real-time safe runtime configuration updates into every plugin.

3.3 Plugins can Themselves Host Other Plugins

An openMHA plugin can itself act as a plugin host. This allows to combine analysis and resynthesis methods in a single plugin. We call plugins that can themselves load other plugins "bridge plugins" in the openMHA. When such a bridge plugin is then called by the openMHA to process one block of signal, it will first perform its analysis, then invoke (as a function call) the signal processing in the loaded plugin to process the block of signal in the analysis domain, wait to receive a processed block of signal in the analysis domain back from the loaded plugin when the signal processing function call to that plugin returns, then perform the re-synthesis transform, and finally return the block of processed signal in the original domain back to the caller of the bridge plugin.

3.4 Central Calibration

The purpose of hearing aid signal processing is to enhance the sound for hearing impaired listeners. Hearing impairment generally means that people suffering from it have increased hearing thresholds, i.e. soft sounds that are audible for normal hearing listeners may be imperceptible for hearing impaired listeners. To provide accurate signal enhancement for hearing impaired people, hearing aid signal processing algorithms have to be able to determine the absolute physical sound pressure level corresponding to a digital signal given to any openMHA plugin for processing. Inside the openMHA, we achieve this with the following convention: The single-precision floating point time-domain sound signal samples, that are processed inside the openMHA plugins in blocks of short durations, have the physical pressure unit Pascal ($1\mathrm{Pa} = 1\mathrm{N/m}^2$). With this convention in place, all plugins can determine the absolute physical sound pressure level from the sound samples that they process. A derived convention is employed in the spectral domain for STFT signals. Due to the dependency of the calibration on the hardware used, it is the responsibility of the user of the openMHA to perform calibration measurements and adapt the openMHA settings to make sure that this calibration convention is met. We provide the plugin *transducers* (cf. section 4.1) which can be configured to perform the necessary signal adjustments in most situations.

4 February 2017 Pre-Release

In February 2017, HörTech and Universität Oldenburg published a pre-release of the openMHA on GitHub under an open-source license (AGPL3). This pre-release contains the openMHA command line application, the toolbox library "libMHAToolbox", an initial set of openMHA plugins and openMHA sound input/output (IO) libraries, and example configurations. The initial set of plugins and sound IO libraries was selected so that a basic research hearing aid configuration can be realized with the contained plugins, and users could process both, live sounds via JACK as well as sound from and to files. The basic hearing aid algorithms present in the pre-release include

- an adaptive differential microphone algorithm that suppresses interfering noise from the rear hemisphere (cf. section 4.3),
- a binaural coherence filter that provides feedback suppression and dereverberation (cf. section 4.5), and
- a multi-band dynamic range compression algorithm that restores audibility of sounds for the hearing impaired user (cf. section 4.7).

Apart from the plugins that implement just these algorithms, additional supporting plugins are contained in the pre-release that are required to form a complete hearing aid implementation. The contained plugins are briefly described in the following subsections.

For real-time hearing aid processing, an input-output delay below 10 ms is required. This ensures that

- the hearing-impaired user is not confused by asynchrony between lip movements of a conversation partner and the perceived sound,
- no echo-effects are audible if the direct sound can also be perceived by the hearing aid user, and

- fewer frequencies are available for possibly annoying acoustic feedback loops [Grimm et al., 2009a].

The example configuration that combines all three example algorithms mentioned here shows an algorithmic delay of 4.4 ms. On top of this algorithmic delay, input and output of the sound through a sound card causes additional delay in the range of two to three block durations depending on the hardware in use. The example configuration uses a block size of 64 samples at 44100 Hz sampling rate. We have found, that e.g. with the RME Multiface II sound card and the snd-hdsp alsa driver used by JACK, this will add 4.4 ms delay between acoustic input and output on a Linux system with a low-latency kernel and real-time priorities set up for JACK and the alsa sound driver.

This results in an overall delay of 8.8 ms of the example configuration containing the plugins described in the following in the order of their processing.

4.1 The *transducers* Plugin

A device-dependent calibration is required for plugins to be able to deduce the physical signal level that is present at the hearing aid input. When connecting a microphone to a sound card and using that sound card to feed sound samples to the openMHA, these sound samples do not automatically follow the openMHA level convention outlined in section 3.4. The same is true when using sound files instead of sound cards for input and output. Different microphones have different sensitivities. Sound cards have adjustable amplification settings. Sound files may have been normalized before they have been saved to disk. To be able to implement the openMHA level convention, i.e., that the numeric value of time-domain sound samples in the openMHA should reflect their sound pressure amplitude in Pascal, we need to be able to adjust for arbitrary physical level to digital level mappings in the openMHA. This is done with the help of the plugin *transducers*, which is the only plugin that must not rely on this convention, because it is the one plugin that has to make sure that all other plugins can rely on this convention. For this reason, *transducers* is usually loaded as the first plugin into the openMHA, and will itself (i.e. as a bridge plugin, cf. section 3.3) load another openMHA plugin into the openMHA process. This other plugin receives the calibrated input signal from *transducers*, and it sends its processed but still calibrated output signal back to the *transducers* plugin to adjust for the physical outputs. *transducers* provides filters and gain adjustments to ensure calibration of inputs and outputs. Typical output calibration values are in the order of 110 dB SPL of a full-scale signal.

4.2 The *mhachain* Plugin

An *mhachain* plugin can itself load several other plugins in a configurable order, where each plugin processes the output signal of the previous plugin.

4.3 The Adaptive Differential Microphone (*adm*) Plugin

Reduced audibility of soft sounds is not the only problem that hearing impaired listeners face when communicating. Another commonly experienced problem is a reduced intelligibility of speech in noisy environments, even if the speech is loud enough to be perceived. Hearing aids therefore regularly employ signal processing algorithms to enhance the signal-to-noise ratio of speech in noisy environments. In this context assumptions about target and noise sources play an important role as well as robustness and generalization capabilities of the method used. Adaptive differential microphones (ADM, [Elko and Pong, 1995]) aim at the preservation of a target signal while suppressing background noise. For this purpose, two general assumptions are made: the target is assumed to be present in the frontal hemisphere of a listener, while noise occurs in the rear hemisphere. ADMs work for pairs of omnidirectional microphones separated by a small distance, and combine a two-channel input to a single-channel output signal by adding up delayed and weighted versions of the input as shown in Figure 2. In a binaural setting two independent, bilateral ADMs are realized, each using a two-microphone pair located in the a hearing aid device on one ear.

4.4 The *overlapadd* Plugin

overlapadd is one of the openMHA plugins that perform conversion between time domain and spectral domain as a service for algorithms that process a series of short time Fourier transform (STFT) signals. Thereby, not every openMHA plugin that processes spectral signal has to perform its own spectral analysis.

overlapadd is a bridge plugin (cf. section 3.3) and performs both, the forward and

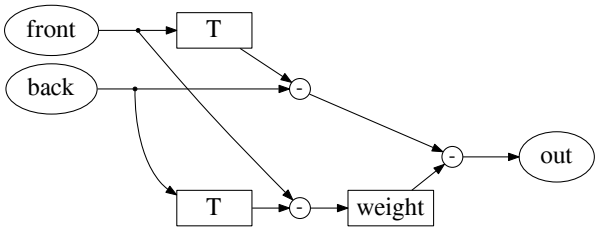


Figure 2: Adaptive differential microphone signal flowchart. The input of the *front* and *back* microphone is combined to a single-channel output after applying a delay T and a weighting.

the backward transform, and can load another openMHA plugin which analyses and modifies the signal while in the spectral domain. The plugin performs the standard process of collecting the input signal, windowing, zero-padding, fast Fourier transform, inverse fast Fourier transform, additional windowing, and overlap-add time signal output. It can be used in standard overlap-add (OLA) and weighted overlap-add (WOLA) contexts.

4.5 The Binaural *coherence* Filter Plugin

An important issue in hearing aid processing is the reduction of feedback that can occur between the hearing aid receivers (outputs) and the closely located inputs (microphones). At high output levels a sound loop can emerge, causing annoying, self-sustaining beep tones.

Binaural coherence filtering, i.e., coherence-based gain control is applied to reduce this effect and enable higher gain levels of the hearing device [Grimm et al., 2009b].

Figure 3 shows that the binaural coherence is measured between the left and the right input signals to the hearing aids and used to derive frequency-dependent gains.

Coherence filtering also contributes to noise and reverberation reduction, as diffuse, incoherent background sounds are also reduced. A combination the binaural coherence filtering with preceding bilateral ADMs was shown to be beneficial, i.e., increased speech intelligibility with a binaural hearing aid setup [Baumgärtel et al., 2015].

4.6 The *fftfilterbank* Plugin

In the hearing impaired, the hearing loss generally varies with frequency. To restore audibility in hearing impaired listeners with amplification and compression in hearing aid signal processing, it is therefore common practice to amplify

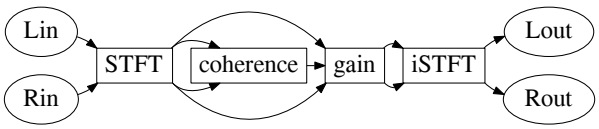


Figure 3: Coherence filter signal flowchart. Binaural coherence-based gain control is applied to the left and the right input channel in different frequency bands in the STFT domain.

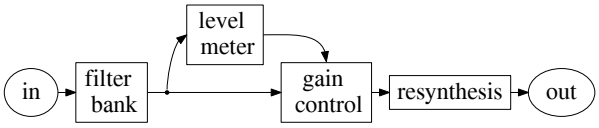


Figure 4: Dynamic compression signal flowchart. The input is split into frequency bands by a *filter-bank*. Before *re-synthesis*, an input-level dependent gain rule is applied.

and compress the signal differently in different frequency bands, and let the time-varying input level in the different frequency bands control the gain selection. The *fftfilterbank* plugin receives broadband spectra for each audio channel and divides the incoming spectra into multiple narrower frequency bands for processing by the following openMHA algorithms. The *fftfilterbank* provides flexibility for filter-bank design. The output frequency bands may overlap or not, with variable degrees of overlap, with customizable filter shapes and different frequency scales to specify the edge or center frequencies of the filters.

4.7 Hearing Loss Compensation (*dc*)

The *dc* plugin applies Multi-band dynamic range compression [Grimm et al., 2015] to the signal. This operation serves two important aspects in a hearing aid: The hearing loss is compensated by defining gain rules between input and output level. Specific gain rules are also used to compensate recruitment effects that often comes along with a hearing loss, i.e., a decreased range between the percept of a soft sound and the loudest sound with a still comfortable level. To compensate for this effect, soft input sounds are usually amplified with higher gains than loud sounds. The *dc* plugin allows to specify a gain-matrix with different gains for different frequencies and input sound levels. Input sound levels in hearing aid frequency bands are commonly measured with attack-release level filters, the time constants of which can be freely configured in the *dc* plugin. Figure 4 shows the

signal flow for dynamic compression with the *dc* plugin. The *dc* plugin also allows to configure binaural and inter-frequency interactions of gain derivation.

4.8 The *combinechannels* Plugin

Because the *fftfilterbank* splits broadband signals into frequency bands for processing by the *dc* plugin, these frequency bands have to be recombined to broadband channels again, after *dc* has processed them. This is done in the *combinechannels* plugin. Of course, the *fftfilterbank* and *combinechannels* plugins could be combined into a single bridge plugin (cf. section 3.3). This would generally be a better implementation choice. It is not done here to showcase the flexibility of the openMHA platform: It is also possible to have analysis and resynthesis of some transform as separate plugins, and to propagate the signal from one plugin to the next inside a single *mhachain* plugin while the domain changes from one plugin to the next (here: few broadband channels vs many narrowband channels).

5 Software

openMHA is a command line application with no graphical user interface (GUI) of its own. openMHA can be configured with command line parameters, configuration files, interactively over a network connection, or by a combination of all three methods. The same text-based configuration language is used in all three methods. Special-purpose GUIs can be produced to control the openMHA over the network connection. Such GUIs can be produced in any programming language or framework that is able to connect to the openMHA over a TCP network connection. Some special-purpose GUIs exist for the closed-source MHA that also work with the openMHA, but are not yet part of the first open-source pre-release. GUIs will be added in later releases of the openMHA.

5.1 Configuration Interface

The openMHA application itself and also its plugins are controlled through a simple, text-based configuration language. The language allows hierarchical configuration similar to the concept of Octave and Matlab structures. The configuration language enables variable assignments, queries, and loading and saving of configuration files. Variables of different types (integers, floating point and complex numbers, strings) and dimensions (scalars, vectors, matrices) are supported. For more details, please refer to [Grimm et al., 2006].

5.2 Plugin Development

New plugins can be developed for the openMHA by implementing a C++ class derived from a generic base class, implementing the methods and compiling it to a shared object. Together with other helper classes provided by the MHA-Toolbox library, out-of-the box support for exporting variables to the configuration interface (cf. section 5.1) and for thread safe configuration updates (cf. section 3.2) is available.

Simple plugins will usually output the signal in the same domain (spectrum or waveform) as the input domain. It is also possible to implement domain transformations (from the time domain to spectrum or vice versa) inside a plugin, as well as change the number of audio channels, and even the number of audio samples per block and the sampling rate (e.g. for resampling).

A detailed manual for plugin development and implementation will be provided with a near-future release.

6 Conclusions

The openMHA provides the means for sustainable research on and development of hearing aid processing algorithms and assistive hearing systems. The software is further developed in the project "Open community platform for hearing aid algorithm research", additionally, updates based on the feedback of the research community will be conducted. Future work will extend the openMHA in several directions: The set of reference algorithms will be expanded and experimental algorithms will be included. Additional hardware and operation systems will be included, i.e., real-time runtime support for Beaglebone Black ARM and similar platforms, as well as support for Windows operations systems. Increased usability on different user levels is achieved by the preparation of a GUI for the pure application of the openMHA, e.g., in the context of audiological measurements, availability of reference manuals for the configuration as well as the implementation of plugins for realization and implementation of own algorithms and methods and their evaluation.

The openMHA is intended to serve as a platform for extensive research and evaluations by the community. A pre-release of the software in

its current version including example configuration files as described here can be downloaded via `http://www.openmha.org`.

7 Acknowledgments

The project "Open community platform for hearing aid algorithm research" is funded by the National Institutes of Health (NIH Grant 1R01DC015429-01).

References

Regina M. Baumgärtel, Martin Krawczyk-Becker, Daniel Marquardt, Christoph Völker, Hongmei Hu, Tobias Herzke, Graham Coleman, Kamil Adiloglu, Stephan M. A. Ernst, Timo Gerkmann, Simon Doclo, Birger Kollmeier, Volker Hohmann, and Mathias Dietz. 2015. Comparing Binaural Preprocessing Strategies I: Instrumental Evaluation. *Trends in Hearing*, 19:article No. 2331216515617916.

Perry R Cook and Gary P Scavone. 1999. The synthesis toolkit (stk). In *ICMC*.

Paul Davis. 2003. Jack audio connection kit. http://jackaudio.org/.

John W. Eaton, David Bateman, Søren Hauberg, and Rik Wehbring. 2015. GNU Octave version 4.0.0 manual: a high-level interactive language for numerical computations. `http://www.gnu.org/software/octave/doc/interpreter`.

G. W. Elko and Anh-Tho Nguyen Pong. 1995. A Simple Adaptive First-order Differential Microphone. In *Proceedings of 1995 Workshop on Applications of Signal Processing to Audio and Accoustics*, pages 169–172.

Giso Grimm, Tobias Herzke, Daniel Berg, and Volker Hohmann. 2006. The Master Hearing Aid: a PC-based Platform for Algorithm Development and Evaluation. *Acta acustica united with Acustica*, 92:618–628.

Giso Grimm, Tobias Herzke, and Volker Hohmann. 2009a. Application of Linux Audio in Hearing Aid Research. In *Linux Audio Conference 2009*.

Giso Grimm, Volker Hohmann, and Birger Kollmeier. 2009b. Increase and Subjective Evaluation of Feedback Stability in Hearing Aids by a Binaural Coherence-based Noise Reduction Scheme. *IEEE Transactions on Audio, Speech, and Language Processing*, 17(7):1408–1419.

Giso Grimm, Tobias Herzke, Stephan Ewert, and Volker Hohmann. 2015. Implementation and Evaluation of an Experimental Hearing Aid Dynamic Range Compressor Gain Prescription. In *DAGA 2015*, pages 996–999.

HörTech gGmbH and Universität Oldenburg. 2017. openMHA web site on GitHub. `http://www.openmha.org/`.

Eric Jones, Travis Oliphant, Pearu Peterson, et al. 2001–. SciPy: Open source scientific tools for Python. `http://www.scipy.org/`.

James McCartney. 2002. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68.

Yann Orlarey, Dominique Fober, and Stéphane Letz. 2009. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 290.

Miller Puckette. 1996–. Pure data. `https://puredata.info/`.

Towards dynamic and animated music notation using INScore

Dominique Fober, Yann Orlarey and **Stéphane Letz**
GRAME - Centre national de création musicale
11 cours de Verdun Gensoul
69002 Lyon
France,
{fober, orlarey, letz}@grame.fr

Abstract

INScore is an environment for the design of augmented interactive music scores opened to conventional and non-conventional use of the music notation. The system has been presented at LAC 2012 and has significantly evolved since, with improvements turned to dynamic and animated notation. This paper presents the latest features and notably the dynamic time model, the events system, the scripting language, the symbolic scores composition engine, the network and Web extensions, the interaction processes representation system and the set of sensor objects.

Keywords

INScore, music score, dynamic score, interaction.

1 Introduction

Contemporary music creation poses numerous challenges to the music notation. Spatialized music, new instruments, gesture based interactions, real-time and interactive scores, are among the new domains that are now commonly explored by the artists. Common music notation doesn't cover the needs of these new musical forms and numerous research and approaches have recently emerged, testifying to the maturity of the music notation domain, in the light of computer tools for music notation and representation. Issues like writing spatialized music [Ellberger et al., 2015], addressing new instruments [Mays and Faber, 2014] or new interfaces [Enström et al., 2015] (to cite just a few), are now subject of active research and proposals.

Interactive music and real-time scores are also representative of an expanding domain in the music creation field. The advent of the digital score and the maturation of the computer tools for music notation and representation constitute the basement for the development of this musical form, which is often grounded on non-traditional music representation [Smith, 2015] [Hope et al., 2015] but may also use the common music notation [Hoadley, 2012; Hoadley, 2014].

In order to address the notation challenges mentioned above, INScore [Fober et al., 2010] has been designed as an environment opened to non-conventional music representation (although it supports symbolic notation), and turned to real-time and interactive use [Fober et al., 2013]. It is clearly focused on music representation only and in this way, differs from tools integrated into programming environments like Bach [Agostini and Ghisi, 2012] or MaxScore [Didkovsky and Hajdu, 2008].

INScore has been already presented at LAC 2012 [Fober et al., 2012a]. It has significantly evolved since and this paper introduces the set of issues that have been more recently addressed. After a brief recall of the system and of the programming environment, we'll present the scripting language extensions and the symbolic scores composition engine that provides high level operations to describe real-time and interactive symbolic scores composition. Next we'll describe how interaction processes representations can be integrated into the music score and how remote access is supported using the network and/or Web extensions. Tablet and smartphone support have led to integrate gestural interaction with a set of *sensor* objects that will be presented. Finally, the time model, recently extended, will be described.

2 The INScore environment

INScore is an environment to design interactive augmented music scores. It extends the music representation to arbitrary graphic objects (symbolic notation but also images, text, vectorial graphics, video, signals representation) and provides an homogeneous approach to manipulate the score components both in the graphic and time spaces.

It supports *time synchronization in the*

graphic space, which refers to the graphic representation of the temporal relations between components of a score - via a synchronization mechanism and using *mappings* that express relations between time and graphic space segmentations (Fig. 1).

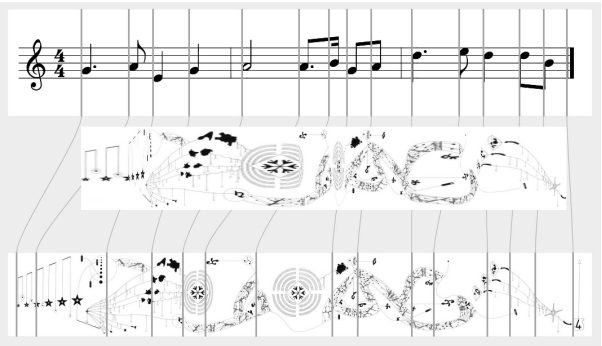


Figure 1: A graphic score (Mark Applebaum's graphic score *Metaphysics of Notation*) is synchronized to a symbolic score. The picture in the middle is the result of the synchronization. The vertical lines express the graphic to graphic relationship, that have been computed by composing the objects common relations with the time space.

INScore has been primarily designed to be controlled via OSC[1] messages. The format of the messages consists in an OSC address followed by a message string and 0 to n parameters (Fig. 2).

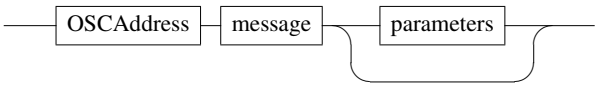


Figure 2: INScore messages general format.

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

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

may be viewed as the following method call:

```
ITL[scene[score]]->color(255 128 40 150)
```

The system provides a set of messages for the graphic space control (`x, y, color, space,` etc.), for the time space control (`date, duration,` etc.), and to manage the environment. It includes two special messages:

- the `set` message that operates like a constructor and that takes the object type as parameter, followed by type specific parameters (Fig. 3).
- the `get` message provided to query the system state (Fig. 4).

```
/ITL/scene/obj set txt "Hello world!";
```

Figure 3: A message that creates a textual object, which type is `txt`, with a text as specific data.

```
/ITL/scene/obj get;
  -> /ITL/scene/obj set txt "Hello
world!";

/ITL/scene/obj get x y;
  -> /ITL/scene/obj x 0;
  -> /ITL/scene/obj y 0.5;
```

Figure 4: Querying an object with a `get` message gives messages on output (prefixed with `->`). These messages can be used to restore the corresponding object state.

The address space is dynamic and not limited in depth. It is hierarchically organized, the first level `/ITL` is used to address the application, the second one `/ITL/`*`scene`* to address the score and the next ones to address the components of a score (note that *scene* is a default name that can be user defined). Arbitrary hierarchy of objects is supported.

3 The scripting language

The OSC messages described above have been turned into a textual version to constitue the INScore scripting language. This language has been rapidly extended to support :

- variables, that may be used to share parameters between messages (Fig. 5).
- message based variables and/or parameters that consists in querying an object to retrieve one of it's attributes value (Fig. 6).
- an extended OSC addressing scheme that allows to send OSC messages to an external application for initialization of control purposes (Fig. 7).

[1] `http://opensoundcontrol.org/`

- JavaScript sections that can be evaluated at parsing and/or run time. A JavaScript call is expected to produce INScore messages as output (Fig. 8).
- mathematical expressions (`+ - / *`, `conditionals`, etc.) that can be used for arguments computation (Fig. 9).
- symbolic scores composition expressions that are described in section 4.

```
greylevel = 140;
color = $greylevel $greylevel $greylevel;
/ITL/scene/obj1 color $color;
/ITL/scene/obj2 color $color;
```

Figure 5: Variables may be used to share values between messages.

```
ox = $(/ITL/scene/obj get x);
/ITL/scene/obj2 x $(/ITL/scene/obj get x);
```

Figure 6: The output of `get` messages can be used by variables or as another message parameter.

```
/ITL/scene/obj set txt "Hello world!";
localhost:8000/start;
```

Figure 7: This script initialises a textual object and sends the `/start` message to an external application listening on UDP port 8000.

4 Symbolic scores composition

Rendering of symbolic music notation makes use of the Guido engine [Daudin et al., 2009]. Thus the primary music score description format is the Guido Music Notation format [Hoos et al., 1998] [GMN]. The MusicXML format [Good, 2001] is also supported via conversion to the GMN format.

The Guido engine provides a set of operators for scores level composition [Fober et al., 2012b]. These operators consistently take 2 scores as argument to produce a new score as output. They allow to put scores in sequence `(seq)`, in parallel `(par)`, to cut a score in the time dimension `(head, tail)`, in the polyphonic dimension `(top, bottom)`, to transpose `(transpose)`, to stretch `(duration)` a score and to apply the rhythm or the pitch of a score to another one `(rhythm, pitch)`.

```
<?javascript
  function randpos(address) {
    var x = (Math.random() * 2) - 1;
    return address + " x " + x + ";";
  }
?>
/ITL/scene/javascript run
        'randpos("/ITL/scene/obj")';
```

Figure 8: The JavaScript section defines a `randpos` function that computes an `x` message with a random value, addressed to the object given as parameter. This function may be next called at initialization or at any time using the static JavaScript node embedded into each score.

```
/ITL/scene/o x ($shift ?  $x + 0.5 :  $x);
```

Figure 9: A mathematical expression is used to compute the position of an object depending on 2 previously defined variables.

The INScore scripting language includes *score expressions*, a simple language providing score composition operations. The novelty of the proposed approach relies on the dynamic aspects of the operations, as well as on the persistence of the score expressions. A score may be composed as an arbitrary graph of score expressions and equipped with a fine control over the changes propagation.

4.1 Score expressions

A score expression is defined as an operator followed by two scores (Fig. 10). The leading `expr` token is present to disambiguate parenthesis in the context of INScore scripts.

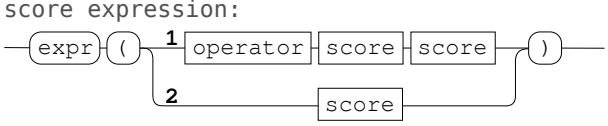


Figure 10: Score expressions syntax.

The score arguments may be:

- a literal score description string (GMN or MusicXML formats),
- a file (GMN or MusicXML formats),
- an existing score object,

- a score expression.

An example is presented in Fig. 11.

```
expr( par score.gmn (seq "[c]" score))
```

Figure 11: A score expression that puts a score file (`score.gmn`) in parallel with the sequence of a literal score and an existing object (`score`). Note that the leading `expr` token can be omitted inside an expression.

4.2 Dynamic score expression trees

The score expressions language is first transformed into an internal tree representation. In a second step, this representation is evaluated to produce GMN strings as output, that are finally passed to the INScore object as specific data.

Basically, the tree is reduced using a depth first post-order traversal and the result is stored in a cache. However, the score expressions language provides a mechanism to make arbitrary parts of a tree variable using an ampersand (&) as prefix of an argument, preventing the corresponding nodes to be reduced at cache level (Fig. 12).

```
expr( par score.gmn (seq "[c]" &score))
```

Figure 12: A score expression that includes a reference to a `score` object. Successive evaluations of the expression may produce different results, provided that the `score` object has changed.

INScore events system (described in section 8.2) provides a way to automatically trigger the re-evaluation of an expression when one of it's variable parts has changed. These mechanisms open the door to dynamic scores composition within the INScore environment. More details about the score expressions language can be found in [Lepetit-Aimon et al., 2016].

5 Musical processes representation

INScore includes tools for the representation of musical processes within the music score. In the context of interactive music and/or when a computer is involved in a music performance, it may provide useful information regarding the state of the musical processes running on the computer. This feedback can notably be used to guide the interaction choices of the performer.

On INScore side, a process state is viewed as a signal. Signals are part of a score components and can be combined into *graphic signals* to become first order objects of a score. They may be notably used for the representation of a performance [Fober et al., 2012a].

Regarding musical processes representation, a signal can be connected to any attribute of an object (Fig. 13), which makes the signal variations visible (and thus the process activity) with the changes of the corresponding attributes.

```
/ITL/scene/signal/sig size 100;
/ITL/scene/obj set rect 0.5 0.5;
/ITL/scene/img set img 'file.png';
/ITL/scene/signal connect sig
    "obj:scale" "img:rotatez[0,360]";
```

Figure 13: A signal `sig` is connected to the `scale` attribute of an object and to the `rotatez` attribute of an image. Note that for the latter, the signal values (expected to be in [-1,1]) are scaled to the interval [0,360].

6 Network and Web dimensions

INScore supports the aggregation of distributed resources over Internet, as well as the publication of a score via the HTTP and/or WebSocket protocols. In addition, a score can also be used to control a set of remote scores on the local network using a *forwarding* mechanism.

6.1 Distributed score components

Most of the components of a score can be defined in a literal way or using a file. All the file based resources can be specified as a simple file path, using absolute or relative path, or as an HTTP url (Fig 14).

```
/ITL/scene/obj1 set img 'file.png';
/ITL/scene/obj2 set img
      'http://www.adomain.org/file.png';
```

Figure 14: File based resources can refer to local or to remote files.

When using a relative path, an absolute path is built using the current path of the score, that may be set to an arbitrary location using the `rootPath` attribute of the score (Fig 15).

The current rootpath can also be set to an arbitrary HTTP url, so that further use of a relative path will result in an url (Fig. 16).

```
/ITL/scene rootPath '/users/me/inscore';
/ITL/scene/obj set img 'file.png';;
```

Figure 15: The `rootPath` of a score is equivalent to the current directory in a shell. With this example, the system will look for the file at '/users/me/inscore/file.png'

```
/ITL/scene rootPath
             'http://www.adomain.org';
/ITL/scene/obj set img 'file.png';;
```

Figure 16: The `rootPath` supports urls. With this example, the system will look for the file at 'http://www.adomain.org/file.png'

This mechanism allows to mix local and remote resources in the same music score, but also to express local and remote scores in a similar way, just using a `rootPath` change.

6.2 HTTPd and WebSocket objects

A music score can be published on the Internet using the HTTP or the WebSocket protocols. Specific objects can be embedded in a score in order to make this score available to remote clients (Fig. 17).

```
/ITL/scene/http set httpd 8000;
/ITL/scene/ws set websocket 8100 200;
```

Figure 17: This example creates an httpd server listening on the port 8000 and a WebSocket server listening on the port 8100 with a maximum notification rate of 200 ms.

The WebSocket server allows bi-directional communication between the server and the client. It sends notifications of score changes each time the graphic appearance of the score is modified, provided that the notification rate is lower than the maximum rate set at server creation time.

The communication scheme between a client and an INScore Web server relies on a reduced set of messages. These messages are protocol independent and are equally supported over HTTP or WebSocket :

- `get`: requests an image of the score.
- `version`: requests the current version of the score. The server answers with an integer value that is increased each time the score is modified.
- `post`: intended to send an INScore script to the server.
- `click`: intended to allow remote mouse interaction with the score.

More details are available from [Fober et al., 2015].

6.3 Messages forwarding

Message forwarding is another mechanism provided to distribute scores over a network. It operates at application and/or score levels when the `forward` the message is send to the application (`/ITL`) or to a score (`/ITL/scene`). The message takes a list of destination hosts specified using a host name or an IP number, and suffixed with a port number. All the OSC messages may be forwarded, provided they are not filtered out (Fig. 18). The filtering strategy is based on OSC adresses and/or on INScore methods (i.e. messages addressing specific objects attributes).

```
/ITL forward 192.168.1.255:7000;
/ITL/filter reject
    '/ITL/scene/javascript';
```

Figure 18: The application is requested to forward all messages on INScore port (7000) to the local network using a broadcast address. Messages addressed to the JavaScript engine are filtered out in order to only forward the result of their evaluation.

7 The *sensor* objects

INScore runs on the major operating systems including Android and iOS. Tablet and smartphone support have led to integrate gestural interaction with a set of sensor (Table 1).

Sensors can be viewed as objects or as signals. When created as a signal node, a sensor behaves like any signal but may provide some additional features (like calibration). When created as a score element, a sensor has no graphical appearance but provides specific sensor events and features.

All the sensors won't likely be available on a given device. In case a sensor is not supported, an error message is generated at creation request and the creation process fails.

8 The time model

INScore time model has been recently extended to support *dynamic* time. Indeed and with the

name	values
accelerometer	x, y, z
ambient light	light level
compass	azimuth
gyroscope	x, y, z
light	a level in lux
magnetometer	x, y, z
orientation	device orientation
proximity	a boolean value
rotation	x, y, z
tilt	x, y

Table 1: The set of sensors and associated values

initial design, the time attributes of an object are fixed and don't change unless a time message `(date, duration)` is received, which can only be emitted from an external application or using the *events* mechanism. The latter (defined very early) introduced another notion of time: the *events* time, which takes place when an event occurs. The *events* system has also been extended for more flexibility.

8.1 The musical time

Regarding the time domain, any object of a score has a date and a duration. A new tempo attribute has been added, which has the effect of moving the object in the time dimension when non null, according to the tempo value and the absolute time flow. Let t_0 be the time of the last tempo change of an object, let v be the tempo value, the object date d_t at a time t is given by a time function f:

$$f(t) \rightarrow d_t = d_{t0} + (t - t_0) \times v \times k, \quad t \geqq t_0 \quad (1)$$

where d_i is the object date at time t_i and k a constant to convert absolute time in musical time. In fact, absolute time is expressed in milliseconds and the musical time unit is the whole note. Therefore, the value of k is $1/1000 \times 60 \times 4$.

Each object of a score has an independent tempo. The tempo value is a signed integer, which means that an object can move forward in time but backward as well.

From implementation viewpoint and when its tempo is not null, an object sends `ddate` (a relative displacement in time) to itself at periodic intervals (Fig. 19).

This design is consistent with the overall system design since it is entirely message based. It is thus compatible with all the INScore mechanisms such as the forwarding system.

```
/ITL/scene/obj tempo 60

-> /ITL/scene/obj ddate f(r_i)
-> /ITL/scene/obj ddate f(r_{i+1})
-> /ITL/scene/obj ddate f(r_{i+2})
-> ...
```

Figure 19: A sequence of messages that activate the time of an object `obj`. Messages prefixed by `->` are generated by the object itself. r_i is the value of the absolute time elapsed between the task i and $i-1$.

8.2 The events system

The event-driven approach of time in INScore preceded the musical time model and has been presented in [Fober et al., 2013]. The event-based interaction process relies on messages that are associated to events and that are sent when the corresponding event occurs. The general format of an interaction message is described in Fig. 20.

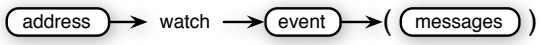


Figure 20: Format of an interaction message: the `watch` request installs a messages list associated to the event `event`.

Initially, the events typology was limited to classical user interface events (e.g. mouse events), extended in the time domain (see Table 2).

Graphic domain	Time domain
mouseDown	timeEnter
mouseUp	timeLeave
mouseEnter	durEnter
mouseLeave	durLeave
mouseMove	

Table 2: Main INScore events in the initial versions.

This typology has been significantly extended to include:

- touch events `(touchBegin, touchEnd, touchUpdate)`, available on touch screens and supporting multi-touch.
- any attribute of an object: modifying an object attribute may trigger the corresponding event, that carries the name of the attribute (e.g. `x, y date`, etc.).

- an object specific data i.e. defined with a `set` message. The event name is `newData` and has been introduced for the purpose of the symbolic score composition system.
- user defined events, that have to comply to a special naming scheme.

Any event can be triggered using the `event` message, followed by the event name and event's dependent parameters. The `event` message may be viewed as a function call that generates OSC messages on output. This approach is particularly consistent for user events that can take an arbitrary number of parameters, which are next available to the associated messages under the form of variables named \$1...\$$n$ (Fig. 21).

```
/ITL/scene/obj watch MYEVENT (
  /ITL/scene/t1 set txt $1,
  /ITL/scene/t2 set txt $2
);
/ITL/scene/obj event MYEVENT
              "This text is for t1"
              "This one is for t2";
```

Figure 21: Definition of a user event named `MYEVENT` that expects 2 arguments referenced as \$1 and \$2 in the body of the definition. This event is next triggered with 2 different strings as arguments.

The time dimension of the events system allows to put *functions* in the time space under the form of events that trigger messages that can modify the score state and/or be addressed to external applications using the extended OSC addressing scheme (Fig. 22).

Combined with the dynamic musical time, this events system allows to describe autonomous animated score. The example in Fig. 23 shows how to describe a cursor that moves forward and backward over a score by watching the time intervals that precedes and follows a symbolic score and by inverting the tempo value.

9 Conclusion

INScore[2] is an ongoing open source project that crystallizes a significant amount of research addressing the problematics of the music notation and representation in regard of the contemporary music creation. It is used in artistic

[2] `http://inscore.sf.net`)

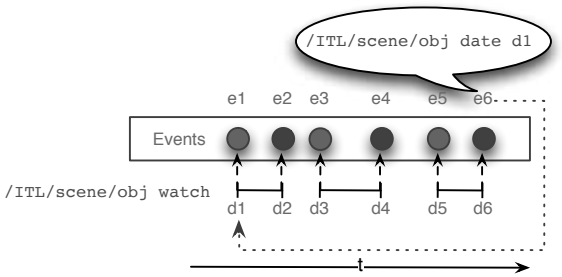


Figure 22: Exemple of events placed in the time space. These events are associated to time intervals (`timeEnter` and `timeLeave`) and are triggered when entering (in red) of leaving (in blue) these intervals. The last event (`e6`) emits a `date` message that creates a loop by putting the object back at the beginning of the first interval.

```
# first clear the scene
/ITL/scene/* del;

# add a simple symbolic score
/ITL/scene/score set gmn '[c d e f g a h
c2 ]';

# add a cursor synchronized to the score
/ITL/scene/cursor set ellipse 0.1 0.1;
/ITL/scene/cursor color 0 0 250;
/ITL/scene/sync cursor score syncTop;

# watch different time zones
/ITL/scene/cursor watch timeEnter 2 3
  ( /ITL/scene/cursor tempo -60 );

/ITL/scene/cursor watch timeEnter -1 0
  ( /ITL/scene/cursor tempo 60 );

# and finally start the cursor time
/ITL/scene/cursor tempo 60;
```

Figure 23: A cursor that moves forward and backward over a symbolic score.

projects and many of the concrete experiences raised new issues that are reflected into some of the system extensions. The domain is quite recent and there are still a lot of open questions that we plan to address in future work and in particular:

- turning the scripting language into a *real* programming language would provide a more powerful approach to music score description. The embedded JavaScript en-

gine may already be used for an algorithmic description of a score, but switching from one environment (INScore script) to another one (JavaScript) proved to be a bit tedious.

- extending the score components to give a time dimension to any of their attributes could open a set of new possibilities, including arbitrary representations of the passage of time.

Finally, migrating the INScore native environment to the Web is part of the current plans and should also open new perspectives, notably due to the intrinsic connectivity of Web applications.

References

Andrea Agostini and Daniele Ghisi. 2012. Bach: An environment for computer-aided composition in max. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 373–378.

C. Daudin, Dominique Fober, Stephane Letz, and Yann Orlarey. 2009. The guido engine – a toolbox for music scores rendering. In LAC, editor, *Proceedings of Linux Audio Conference 2009*, pages 105–111.

Nick Didkovsky and Georg Hajdu. 2008. Maxscore: Music notation in max/msp. In ICMA, editor, *Proceedings of International Computer Music Conference*.

Emile Ellberger, Germán Toro-Perez, Johannes Schuett, Linda Cavaliero, and Giorgio Zoia. 2015. A paradigm for scoring spatialization notation. In Marc Battier, Jean Bresson, Pierre Couprie, Cécile Davy-Rigaux, Dominique Fober, Yann Geslin, Hugues Genevois, François Picard, and Alice Tacaille, editors, *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*, pages 98–102, Paris, France. Institut de Recherche en Musicologie.

Warren Enström, Josh Dennis, Brian Lynch, and Kevin Schlei. 2015. Musical notation for multi-touch interfaces. In Edgar Berdahl and Jesse Allison, editors, *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 83–86, Baton Rouge, Louisiana, USA, May 31 – June 3. Louisiana State University.

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

Dominique Fober, Yann Orlarey, and Stephane Letz. 2012a. Inscore – an environment for the design of live music scores. In *Proceedings of the Linux Audio Conference – LAC 2012*, pages 47–54.

Dominique Fober, Yann Orlarey, and Stéphane Letz. 2012b. Scores level composition based on the guido music notation. In ICMA, editor, *Proceedings of the International Computer Music Conference*, pages 383–386.

Dominique Fober, Stéphane Letz, Yann Orlarey, and Frederic Bevilacqua. 2013. Programming interactive music scores with inscore. In *Proceedings of the Sound and Music Computing conference – SMC'13*, pages 185–190.

Dominique Fober, Guillaume Gouilloux, Yann Orlarey, and Stéphane Letz. 2015. Distributing music scores to mobile platforms and to the internet using inscore. In *Proceedings of the Sound and Music Computing conference – SMC'15*, pages 229–233.

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

Richard Hoadley. 2012. Calder's violin: Realtime notation and performance through musically expressive algorithms. In ICMA, editor, *Proceedings of International Computer Music Conference*, pages 188–193.

Richard Hoadley. 2014. December variation (on a theme by earle brown). In *Proceedings of the ICMC/SMC 2014*, pages 115–120.

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

Cat Hope, Lindsay Vickery, Aaron Wyatt, and Stuart James. 2015. The decibel scoreplayer - a digital tool for reading graphic notation. In Marc Battier, Jean

Bresson, Pierre Couprie, Cécile Davy-Rigaux, Dominique Fober, Yann Geslin, Hugues Genevois, François Picard, and Alice Tacaille, editors, *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*, pages 58–69, Paris, France. Institut de Recherche en Musicologie.

Gabriel Lepetit-Aimon, Dominique Fober, Yann Orlarey, and Stéphane Letz. 2016. Inscore expressions to compose symbolic scores. In Richard Hoadley, Chris Nash, and Dominique Fober, editors, *Proceedings of the International Conference on Technologies for Music Notation and Representation - TENOR2016*, pages 137–143, Cambridge, UK. Anglia Ruskin University.

Tom Mays and Francis Faber. 2014. A notation system for the karlax controller. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 553–556, London, United Kingdom, June. Goldsmiths, University of London.

Ryan Ross Smith. 2015. An atomic approach to animated music notation. In Marc Battier, Jean Bresson, Pierre Couprie, Cécile Davy-Rigaux, Dominique Fober, Yann Geslin, Hugues Genevois, François Picard, and Alice Tacaille, editors, *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*, pages 39–47, Paris, France. Institut de Recherche en Musicologie.

PlayGuru, a music tutor

Marc Groenewegen
Hogeschool voor de Kunsten Utrecht
Ina Boudier-Bakkerlaan 50
3582 VA Utrecht
The Netherlands
marc.groenewegen@hku.nl

Abstract

PlayGuru is a practice tool being developed for beginning and intermediate musicians. With exercises that adapt to the musician, the intention is to help a music student to develop several playing skills and motivate them to practice in-between classes. Because all exercise interaction is entirely based on sound, the author believes PlayGuru is particularly useful for blind and visually impaired musicians. Research currently focuses on monophonic exercises. This paper is a report of the current status and ultimate goals.

Keywords

Computer-assisted music education, DSP, machine learning

1 Project objectives

The main reason for writing this paper is to bring the project to the attention of others so they can use, improve and benefit from the ideas, technology and the intended end product.

Even though few user experiences can be reported at the time of writing, some intermediate results and plans for the near future are given towards the end of this paper.

PlayGuru is a music tutor that operates exclusively in the sound domain. Because the focus is only on music, the author believes it is a very useful tool for beginning and intermediate musicians and particularly useful for blind and visually impaired musicians.

1.1 Practice motivation

How does an amateur musician find the motivation to pick up their instrument and play? What motivates children to practice?

These questions probably have a large spectrum of answers. Let us focus on two motivational forces: personal growth and affirmation.

PlayGuru is a set of music exercises based on an example and response approach. Dialogs usually start with an example being played and base the next action upon the response of the musician. The example and response can also be played simultaneously, creating a sense of playing together. Those synchronised exercises give room to improvisation and exploration.

The way in which PlayGuru aims to keep the user motivated is based on affirmation when the exercise is performed according to predefined objectives.

It will never flag a "wrong note", as this is considered highly demotivating. Instead, it responds by making the exercise slightly easier when it finds you are struggling with the current level, until it gets to a level from which you can continue growing again.

The most important affirmative motivators used are:

- increasing playing speed
- extending the phrase
- increasing complexity of the exercise

The current version contains a very basic user model, which is a starting point for a module that monitors a user's achievements and keep track of their progress, thus supporting their personal growth.

To perform user tests supporting the research, several exercises are being implemented. At the time of writing, a versatile sound-domain guitar tuner with arbitrary tuning is available, as is an exercise for remembering a melodic phrase and a riff trainer for practicing licks at high speed.

Most exercises are developed for the guitar. A great source of inspiration for guitar practice is the book by Scott Tennant: [Tennant, 1995] with a focus on motor skills and automation.

Because all exercises are based on pitch- and onset-detection in sound signals, adaptation for other instruments with discrete pitch and a clear onset should be reasonably straightforward. For instruments with arbitrary pitch

ranges and smooth transitions, as well as human voice, some additional provisions may be necessary.

1.2 Origin of the project

The PlayGuru project started as part of the author's Master's course. While trying to find ways to improve the effectiveness of practice routines for the guitar, some research was done into existing solutions.

Several systems for computer assisted music training were found and some have been put to the test. A shortlist is included at the end. One thing all the encountered solutions have in common is that they rely heavily on visual interaction. In many cases this implies written score or tablature, in other cases a game-like environment in which the user has to act upon events happening in a graphic scene.

In several cases the author found the visual information distracting from the music. Thus the idea arose for a practice tool exclusively working with sound.

Shortly after that, the foundation Connect2Music [1] came into view. Connect2Music, founded in 2013, provides information with respect to music practice by visually impaired musicians.

According to [Mak, 2015], the facilities for blind music students in The Netherlands are limited. Even though the situation is improving, a practice tool which focuses only on the music itself would be a much wanted addition.

Thus a project was born: to find ways to improve the learning path for beginning and intermediate musicians with music as the key element and primarily addressing blind and visually impaired people.

The prototype being developed for performing this research is called PlayGuru. The envisioned end product aims to help and encourage a music student to perform certain exercises inbetween music classes and is meant to complement rather than replace regular classes from a human teacher.

Through the contacts of Connect2Music with the community, several blind and visually impaired musicians and software developers in The Netherlands and Belgium expressed their interest in this project and offered help to assess and assist.

[1] `https://www.connect2music.nl`

2 Research

To support the research with experiences of end users, some application prototypes are being developed. The adaptive exercises used in these prototypes will briefly be introduced separately.

This chapters discusses the software and the chosen methods for interacting with the user.

2.1 Software

The framework and all exercises are currently implemented in C++11. For audio signal analysis, the Aubio [2] library is used. Exercises are composed in real time according to musical rules or taken from existing material like MIDI files and guitar tablature.

2.2 Dependencies

Development is done on Linux and Raspbian. Porting to Apple OSX should be relatively easy but has not been done yet. The most prominent dependencies, as in libraries, are jackd, aubio, fftw3 and portmidi. For generating sound, Fluidsynth is used. Stand-alone versions use a Python script to connect the hardware user-interface to the exercises.

2.3 Practice companion

When playing along with a song on the radio you will need to adjust to the music you hear, as it will not wait for you. Playing with other musicians has entirely different dynamics. People influence each other, try to synchronise, tune in and reach a common goal: to make the music sound nice and feel good about it.

When practicing music with a tool like PlayGuru it would be nice to have a dialog with the tool, instead of just obeying to its rules. This is exactly what makes PlayGuru interact so nicely. It listens to you and adapts, thus behaving like a practice companion.

How this is achieved is shown with reference to the software architecture and indications which parts have been realised and which are being developed.

2.4 Architecture

The modular design of PlayGuru is shown in figure 1, with the Exercise Governor as the module from which every exercise starts.

The Exercise Governor reads a configuration file containing information about the user, the type of exercise, parameters defining the course of the exercise, various composition settings and possibly other sources like MIDI files.

[2] `https://aubio.org`

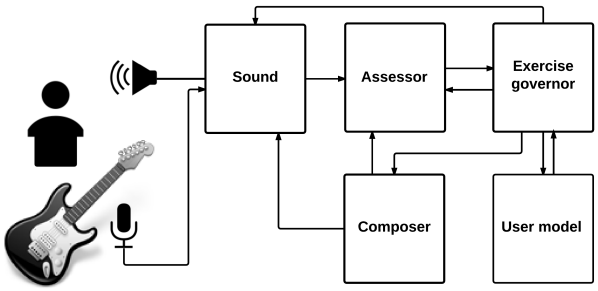


Figure 1: PlayGuru's software architecture

When the exercise starts, the Composer will generate a MIDI track, or read it from the specified file. During the exercise, this generating may take place again, depending on the type of exercise and the progress of the musician.

The MIDI track is played by the sound module, which also captures the sound that comes back from the musician or their instrument.

The Assessor contains all the sound processing and assessment logic and reports back to the Exercise Governor, which calls in the help of the User Model to decide how to interpret the data and what to do next.

2.5 Playback and analysis

Playback and analysis run in separate threads, but share the same time base for relating the output to the input.

Incoming audio is analysed in real time to detect pitch(es) and onsets, which are used to assess the musician's play in relation to the given stimuli. Pitch and onset detection are done using the Aubio library.

2.6 The Exercise Governor

Every exercise type is currently implemented as a separate program. The Exercise Governor is essentially a descriptive name for the main program of each exercise, which uses those parts from the other modules that it needs for a certain exercise. Currently these are compiled and linked into the program. With these building blocks it determines the nature of the exercise.

As an example: to let the musician work on accurate reproduction of a pre-composed phrase, the Exercise Governor will ask the Composer to read a MIDI file, call the MIDI play routine from the Sound module, then let the Assessor assess the user's response and consult the User Model, given the Assessor's data, for determining the next step.

For a play-along exercise using generated melodies, the Exercise Governor uses routines

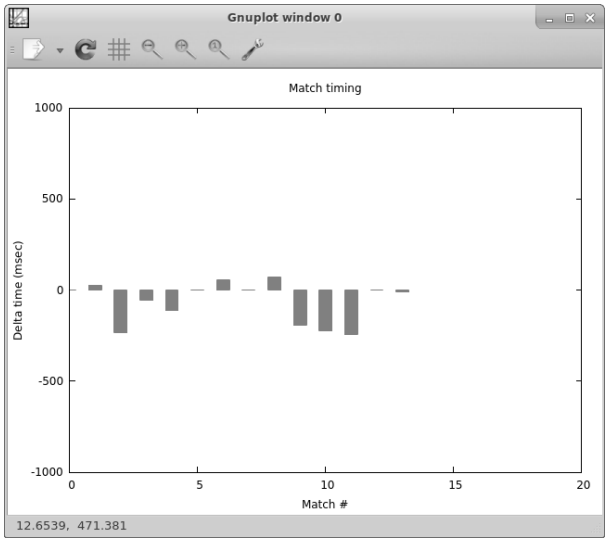


Figure 2: Note onset absolute time difference

from the same modules, but with a different intention. In this case it would let the Composer create a new phrase when needed, ask the Assessor to run a different type of analysis and perform concurrent scheduling of playback and analysis.

2.7 The Assessor

PlayGuru's exercises generally consist of a play- and evaluation loop. For some exercises, the evaluation process is run after the playback iteration, while for others they run simultaneously.

Figure 2 shows the measured timing of a musician playing along with an example melody. The time of each matched note is compared to the time when that note was played in the example. In this chart we see that the musician played slightly "before the beat".

This absolute timing indicates whether the musician is able to exactly copy the example, which can be seen clearly for a melody constisting of only equidistant notes.

More interesting however is relative timing. This indicates whether the musician keeps the timing structure of the example intact. In this case we calculate the differences in spacing of the onsets of successive notes, either numerically or as an indication of "smaller/equal/larger" and compare the result to the structure of the example. In figure 3 this is shown. Here we can see that the musician started out with confidence and needed more time to find the last notes of the phrase. The example consisted of equidistant notes, which would result in a chart of zeros and is therefore

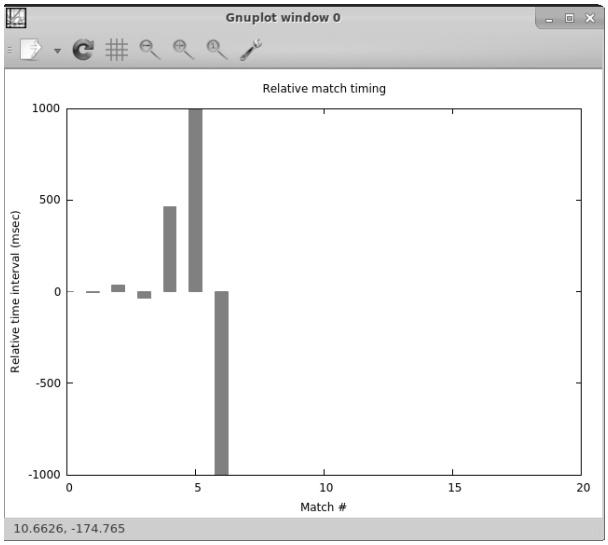


Figure 3: Note onset relative time difference

omitted.

2.8 The Composer

The Composer generates musical phrases based on given rules. The current implementation uses melodic intervals in a specified range, with an allowed set of intervals and within a given scale. The scale is listed as a combination of tones (T) and semitones (S), as in the examples in table 1 and can be specified as needed.

T,T,S,T,T,T,S	major
T,S,T,T,T,S,T	minor
S,S,S,S,S,S,S,S,S,S,S	12-tone

Table 1: Scale examples

2.9 The User Model

At the time of writing, the User Model is partly implemented.

The part that has been implemented and is currently tested by end users is the mapping from analysed properties of the user's playing to parameters that are musically meaningful or significant for the user's ambitions.

In general this mapping is a linear combination of those properties. For example, the user's melodic accuracy can be expressed as a combination of hitting the correct notes and the lack of spurious or unwanted notes.

The weight factors are empirically determined, as are several parameters in the exercises, such as the number of repetitions before moving on or the required proficiency for increasing the level of an exercise.

It is here where the assistance of a Machine Learning algorithm is wanted: to learn which weight factors and other parameters contribute to the user's goals and to optimise these. This has not been implemented yet and is currently being studied.

2.10 Melodic similarity

There are several ways to find out the similarity between the given example and the musician's response. In the current research, only the onset (i.e. start) and pitch of notes are taken into account. Although timbre, loudness and various other features are extremely useful, these are ignored for the time being.

In the exercises where the user is asked to memorise and copy an example melody, the accomplishment of this task is purely based on hitting the correct notes in the correct order. The similarity however is also reflected in the timing. The extent to which the musician keeps the rhythm of the example intact is a property that is measured and evaluated.

In the exercises where the musician plays along with a piece of music, we have much more freedom in the assessment. In this case, playing the exact same notes as in the example is not always necessary. For some exercises it would suffice to improvise within the scale or play certain melodic intervals.

In these situations, similarity measures also allow for more freedom.

A method that is used in an exercise called "riff trainer", focused on automation of and creating variations on a looped phrase, observes notes in the proximity of example notes and draws conclusions based on the objectives of the exercise. This allows for both very strict adherence to the original melody as well as melodic interval-based variations, depending on the assumed objectives.

Another method compares the Markov chain of the example with that of the musician's response.

Some inspiration is gained from this book about melodic similarity: [Walther B. Hewlett, 1998]

3 Personal objectives

Figure 4 shows that an exercise is 'composed' and played. The response of the musician is assessed and mapped to temporary skills. Long-term skills are accumulated in a user model, which tries to construct an accurate profile of the musician and their personal objectives.

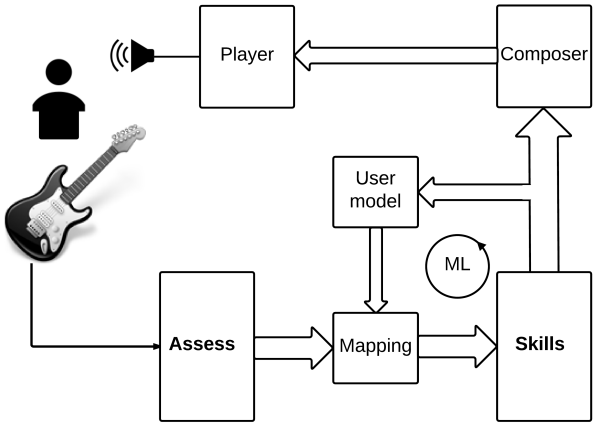


Figure 4: Machine assisted learning supported by machine learning

Examples of these objectives are to strive for faster playing, memorise long melodies or improve timing accuracy. These need to be expressed as quantifyable properties. Faster playing can be expressed as playing a phrase faster than before or playing it faster than the given example, which can be measured.

Likewise, memorising a melody involves the length of the melody that can accurately be played at reasonable speed. Obviously, the concepts 'accurately' and 'reasonable' have to be quantified.

An indication of accuracy in playing is obtained by measuring the number of spurious notes, missed notes and timing.

Some approaches for machine learning (ML) will be investigated. The term "machine learning" is used here to express that the machine itself is learning and does not refer to the "machine assisted learning", which is the main topic of this paper. A machine learning algorithm is thought to be able to achieve the user's objectives by adjusting the mapping parameters that translate measured quantities to short-time skills and several properties of the exercises.

Depending on the exercise, various factors are measured, such as matched notes, missed notes, spurious notes, adherence to the scale, speed and timing accuracy.

These quantities are mapped to short-term skills according to table 2.

Apart from these measured data, the exercises also contain configuration parameters that can be optimised for each user. These are found in composer settings and the curves used to control the playing speed and exercise complexity.

Measured quantity	Mapped to
timing deltas	timing accuracy
missed notes	melodic accuracy
spurious notes	clutter

Table 2: Mapping measured data to skills

4 Hardware

The starting point for this project is to assess the sound of an unmodified instrument. In this section, the current choice of hardware is discussed.

A brief side-project was undertaken to equip an acoustic guitar with resistive sensors for detecting the point where strings are pressed against the fretboard, but because this doesn't look and feel natural, would imply that all users would need to install a similar modification and would exclude all instruments other than guitar, this was discarded.

Because nylon-string acoustic guitar is the primary instrument for the author as well as for lots of beginning music students, the decision was to analyse the sound of the instrument with a microphone or some kind of transducer.

Using a microphone raises the problem that the sound produced by PlayGuru interferes with the sound of the instrument. Source separation techniques are not considered viable for this project due to the added complexity and because we want to be able to play and listen simultaneously, often to the exact same notes. This would justify a study of itself.

Requiring the musician to use a headset is also considered undesirable. So the only option left seems to use a transducer attached to the instrument.

After some experimenting with various combinations of guitar pickups, sensors, preamps and audio interfaces, it turned out that a combination of a simple piezo pickup and a cheap USB audio interface does the job very well.

Successful measurements were done with the piezo pickup attached to the far end of the neck of a guitar. It is advised to embed the pickup into a protective cover to prevent the element itself and the cable from being exposed to mechanical strain and mount it with the piezo's metal surface touching the wood of the guitar using a rubber padded clamp from the DIY store.

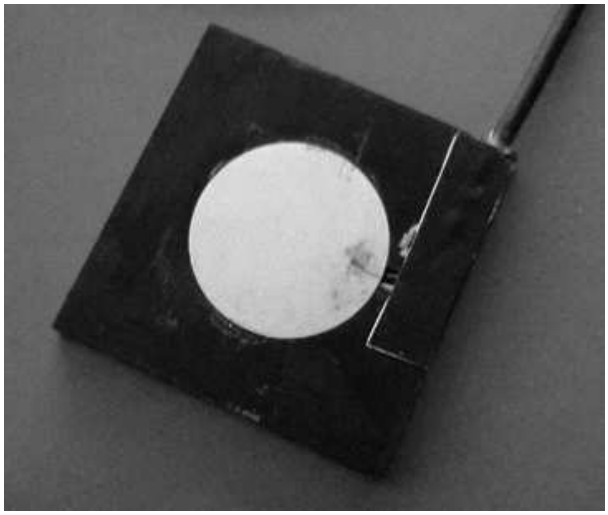


Figure 5: Embedded piezo pickup

5 Dissemination

User's experiences and feedback are of crucial importance for the development of this tool. While a browser application or mobile app seem obvious ways to reach thousands of musicians, development is currently done on Linux and Raspbian. This is a deliberate choice, partly inspired by the author's lack of experience with Webaudio intricacies and the acclaimed large round-trip audio latency of Android devices, for which a separate study may be justified.

For a large part however, this choice is supported by the wish to have an inexpensive stand-alone, self-reliant, single-purpose device.

A series of stand-alone PlayGuru test devices are being developed, based on a Raspberry Pi with a tactile interface meant to be intuitive to blind people. The idea is to attach a piezo pickup to the instrument, plug in, select the exercise and start practicing.

6 Conclusions

Several beginners, intermediate guitar players and some people with no previous experience have used PlayGuru's exercises in various stages of development. The two most mature exercises used are copying a melody and playing along with a melody. In most cases the melodies were generated in real time based on the aformentioned interval-based rules. In some cases a precomposed MIDI file was used.

From the start it was clear that users enjoy the fact that PlayGuru listens to them and rewards "well played" responses with an increase in speed or making the assignment slightly more challenging.

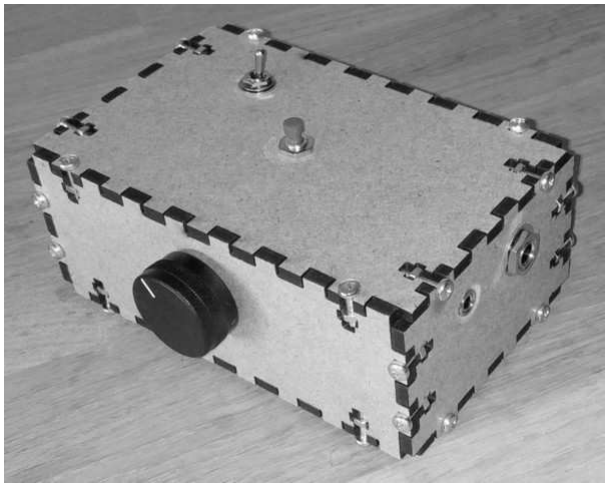


Figure 6: Stand-alone device for user tests

Several users mentioned a heightened focus, meaning that they were very concentrated for a longer time to keep the interaction going and the level rising. Mistakes bring down the speed or level in a suble way and do lead to a slight disappointment, which in many cases proved to be an incentive to get back into the 'flow' of the exercise.

The project is work in progress. For a well-founded opinion on the practical use, a lot more user tests need to be done but the author's conclusion, based on results so far, is that the approach is promising.

7 Future Work

At the time of writing, research concentrates on monophonic exercises with a basic machine learning algorithm.

For the near future the author has plans to perform a study with a larger group of users who can use the system by themselves for a longer time. This requires creating test setups and/or porting the software to other platforms.

Monophonic exercises may be the preferred way to develop several skills, but being able to play, or play along with, your favourite music is much more motivating, particularly for children. An approach resembling Band-in-a-Box® is taken, using a multi-channel MIDI file as input, with the possibility of indicating which channels will be heard and which channel will be 'observed'. This requires both polyphonic play and analysis, which are largely implemented but currently belong in the Future Work section. On the analysis side, a technique based on chroma vectors [Tzanetakis, 2003] is being tested.

Machine learning strategies are being studied but have not yet been implemented. The assistance of co-developers would be much appreciated.

8 Acknowledgements

I want to thank the HKU for supporting and facilitating my work and in particular Gerard van Wolferen, Ciska Vriezenga and Meindert Mak for their insights and ideas. Gerard van Wolferen and Pieter Suurmond were of great help proof-reading and correcting this paper. Lastly I would like to thank the LAC review committee for their excellent observations.

9 Other solutions for computer-assisted music education

- i-maestro
- Bart's Virtual Music School
- Rocksmith ™
- yousician.com
- bestmusicteacher.com
- onlinemuziekschool.nl
- gitaartabs.nl

References

Meindert Mak. 2015. Connecting music in the key of life. 1.2

Scott Tennant. 1995. *Pumping Nylon, The Classical Guitarist's Technique Handbook*. Alfred Music. 1.1

Ning Hu; Roger B. Dannenberg; George Tzanetakis. 2003. Polyphonic audio matching and alignment for music retrieval. *2003 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*. 7

Eleanor Selfridge-Field Walther B. Hewlett. 1998. *Melodic Similarity*. The MIT Press. 2.10

Faust audio DSP language for JUCE

Adrien ALBOUY and **Stéphane Letz**
GRAME
11, cours de Verdun (GENSOUL)
69002 LYON,
FRANCE,
{adrien.albouy, letz}@grame.fr

Abstract

FAUST [Functional Audio Stream] is a functional programming language specifically designed for real-time signal processing and synthesis [1]. It consists of a compiler that translates a FAUST program into an equivalent C++ program, taking care of generating the most efficient code. JUCE is an open-source cross-platform C++ application framework developed since 2004, and bought by ROLI[1] in November 2014, used for the development of desktop and mobile applications. A new feature to the FAUST environnement is the addition of architectures files to provide the glue between the FAUST C++ output and the JUCE framework. This article presents the overall design of the architecture files for JUCE.

Keywords

JUCE, FAUST, Domain Specific Language, DSP, real-time, audio

1 Introduction

From a technical point of view FAUST[2] (*Functional Audio Stream*) is a functional, synchronous, domain specific language designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing languages like Max, PD, Supercollider, etc., is that programs are not interpreted, but fully compiled.

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 provide automatically the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The compiler offers various options to control the generated code, including options to do fully automatic parallelization and take advantage of multicore machines.

The generated code can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers. It works at the sample level, it is therefore suited to implement low-level DSP functions like recursive filters up to fullscale audio applications. It can be easily embedded as it is selfcontained and does not depend of any DSP library or runtime system. Moreover it has a very deterministic behavior and a constant memory footprint.

Being a specification language the FAUST code says 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 [2]. 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 applications, JACK applications, etc.), and JUCE is now supported.

The aim of JUCE[3] is to allow software to be written such that the same source code will compile and run identically on Windows, Mac OS X, Linux platforms for the desktop devices, and on Android and iOS for the mobile ones. A notable feature of JUCE when compared to other similar frameworks is its large set of audio functionality. Those services, the user-interface possibilities and the multi-platform exportability position JUCE as a great framework for FAUST to get exported on, to have in the future less code to maintain up-to-date, and simpler utilization.

In section 2, the idea and the use of the `GUI` architecture file will be introduced. In section 3, the JUCE `Component` hierarchy will be presented without going into many details. Section 4 is the main one, explaining in detail the graphical architecture file for JUCE. MIDI and OSC architecture files are introduced in Section 5. Section 6 will treat of the "glue" between JUCE audio layers and FAUST ones. Section 7

[1] `https://roli.com/`
[2] `http://faust.grame.fr`

presents the `faust2juce` script. Section 8 is a quick tutorial on how to use JUCE for FAUST.

2 FAUST GUI architecture files

A FAUST UI architecture is a glue between a host control layer and a FAUST module. It is responsible to associate a FAUST module parameter to a user interface element and to update the parameter value according to the user actions. This association is triggered by the `dsp::buildUserInterface` call, where the `DSP` asks a `UI` object to build the module controllers.

Since the interface is basically graphic oriented, the main concepts are *widget* based: a UI architecture is semantically oriented to handle active widgets, passive widgets and widgets layout.

A FAUST `UI` architecture derives a `UI` class, containing active widgets, passive widgets, layout widgets, and metadata.

2.1 Active widgets

Active widgets are graphical elements that control a parameter value. They are initialized with the widget name and a pointer to the linked value. The widget currently considered are `Button`, `ToggleButton`, `CheckButton`, `RadioButton`, `Menu`, `VerticalSlider`, `HorizontalSlider`, `Knob` and `NumEntry`.

A `UI` architecture must implement a method `addxxx (const char* name, float* zone, ...)` for each active widget. Additional parameters are available to `Slider`, `Knob`, `NumEntry`, `RadioButton` and `Menu`: the init value, the min and max values and the step (`RadioButton`, `Menu` and `Knob` being special kind of `Sliders`, cf. subsection 2.4, Metadata).

2.2 Passive widget

Passive widgets are graphical elements that reflect values. Similarly to active widgets, they are initialized with the widget name and a pointer to the linked value. The widget currently considered are `NumDisplay`, `Led`, `HorizontalBarGraph` and `VerticalBarGraph`. A `UI` architecture must implement a method `addxxx (const char* name, float* zone, ...)` for each passive widget. Additional parameters are available, depending on the passive widget type. (`NumDisplay` and `Led` are a special kind of `BarGraph`, cf. Subsection 2.4).

2.3 Widget layout

Generally, a `UI` is hierarchically organized into boxes and/or tab boxes. A `UI` architecture must support the following methods to setup this hierarchy:

```
openTabBox (const char* label)
openHorizontalBox (const char* label)
openVerticalBox (const char* label)
closeBox (const char* label)
```

Note that all the widgets are added to the current box.

2.4 Metadata

The FAUST language allows widget labels to contain metadata enclosed in square brackets. These metadata are handled at `UI` level by a `declare` method taking as argument, a pointer to the widget associated value, the metadata key and value: `declare(float*, const char*, const char*)`. Metadata can also declare a DSP as polyphonic, with a line looking like `declare nvoices "8"` for 8 voices. This will always output a polyphonic DSP, either you use the polyphonic option of the compiler or not. This number of voices can be changed with the compiler (cf. Section 7).

For instance, if the program needs a `Slider` to be a `Knob`, those lines are written:

```
declare(&fVslider0, "style", "knob");
addVerticalSlider("Vol", &fVslider0,...);
```

The `style` can be a `knob`, `menu`, etc... depending on the program.

Multiple aspects of the items can be described with the metadata, such as the type of the item just as seen before, the tooltip of the item, the unit, etc...

3 JUCE Component class

To implement a complete program, the graphical elements described in the previous section need to be combined with JUCE classes. In the JUCE Framework, the component class is the base-class for all JUCE user-interface objects. The following section explains the relationship between FAUST GUI architecture files, and the JUCE mechanics.

3.1 Parent and child mechanics

As most frameworks have, JUCE has a hierarchy of `Component` objects, organized in a tree structure. The common way to set a `Component` as child of another component is to do `parent->addAndMakeVisible(child);`.

This function sets the child component as visible too, because it's not by default. Multiple functionalities are accessible to run through this `Component` tree, with methods that give the child `Component` at index `i`, or give the parent. There's even a function allowing to get the parent of a `Component` with a specific type, this type being a derived class of `Juce::Component`. However, this function does not exist for the child, and imply that `dynamic_cast` has to be done if you want to get a child of a certain type.

3.2 Component setup mechanics

First of all, a `Component` is drawn if it's visible, and its parent too. If a `Component` is not visible, its child and all of its children, etc... will not be visible, but as `addAndMakeVisible` function is used most of the time, this should not be a problem. A `Component` has a `Rectangle<int> boundsRelativeToParent`, containing its x and y coordinates, and its width and height. As the variable name implies, the bounds of a `Component` is relative to its parent, and not absolute in the window ; it is very important in the architecture files for FAUST, as will be demonstrated in subsection 4.4.

3.3 Drawing mechanics

A `Component` has two virtual functions[3] that are the main tools to handle a dynamic layout, the `void resized()` and `void paint(Graphics& g)` functions. The `resized` one is called each time a `Component` bounds are changed, and the `paint` one when the `Component` flag indicates that it needs to be repainted. The mouse cursor being on top of it, a mouse click, the `Component` bounds being changed, or one or multiple of its child needing to be repainted indicates that it needs to be repainted for example.

There is a design class called `LookAndFeel` that allows customization of the interface. The `LookAndFeel` objects defines the appearance of all the JUCE widgets, and subclasses can be used to apply different 'skins' to the application.

There is obviously a lot more to the `Juce::Component` class, but that's the basics, or at least what the architecture files need.

4 JuceGUI architecture file

To summarize what has been seen before, the system of widgets and boxes of FAUST needs to be adapted to the `Juce::Component` mechanics in an architecture file called JuceGUI.h. The following section discusses annotated examples.

4.1 Two different kinds of objects

There are two kinds of object used in the adaptation:

- `uiComponent`, which are basically any items of the FAUST program, like sliders or buttons.
- `uiBox`, which is container component, and so can contain a `uiComponent` or some others `uiBox`.

Both are derived classes of a `uiBaseComponent` class, which is itself a derived class of `Juce::Component`.

The `uiBaseComponent` class regroups methods shared by both `uiBox` and `uiComponent`, like `void setRatio()`, `int getTotalWidth()`, etc.... This way, too many `dynamic_cast` in our code are avoided. Here's what the `uiBaseComponent` class contains:

```
float fHRatio, fVRatio;
int fTotalWidth, fTotalHeight;
int fDisplayRectHeight,
    fDisplayRectWidth;
String fName;

uiBaseComponent(int totWidth,
          int totHeight, String name);

int getTotalHeight() ;
int getTotalWidth();
virtual void setRatio();
float getHRatio();
float getVRatio();
String getName();
void setHRatio();
void setVRatio()
void setBaseComponentSize
                (Rectangle<int> r);
void mouseDoubleClick
  (const MouseEvent &event) override;

virtual void writeDebug() = 0;
virtual void setCompLookAndFeel
  (LookAndFeel* laf) = 0;
```

The `mouseDoubleClick` function is a JUCE overridable function, which is called every time a `Component` is double-clicked. Here it's used

[3] placeholder functions which programmer must implement

to call the `writeDebug` function, showing different characteristics of the double clicked `uiBox` or `uiComponent`.

The two pure virtual functions are defined to have their own behavior for both `uiBox` and `uiComponent`, not being the same obviously.

The `virtual void setRatio();` function is virtual because there is a special case with the `uiBox`, which is setting her own ratio, and need to be asking its child to set their ratios too, in a recursive way.

As said before, `uiComponent` inherits from those `uiBaseComponent` functions, and is itself a mother class for plenty of different widgets. Here's the inheritance diagram:

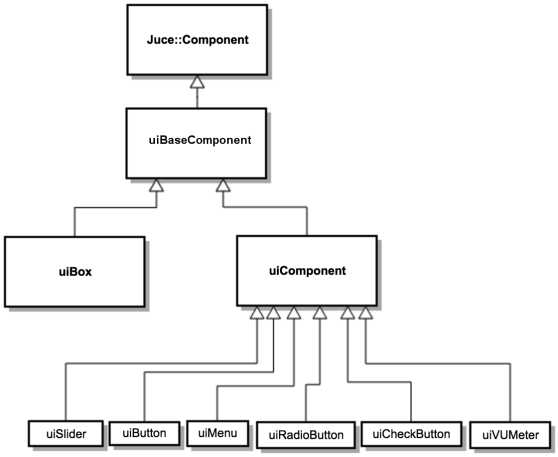


Figure 1: Inheritance diagram

A `uiComponent` subclasses can handle multiple "type" of items.

For instance, `uiSlider` groups every kind of sliders: `HorizontalSlider`, `VerticalSlider`, `NumEntry` and `Knob`.

4.2 The main window

The user interface cannot be shrunk infinitely in order to be always lisible and clear, so a minimal window size is defined. That implies that instead of a basic `Component` in a `DocumentWindow` (a resizable window with a title bar and maximise, minimise and close buttons), a `Viewport` in a `DocumentWindow` is used, which displays scrollbars when the window gets lower dimensions than the minimal size of the FAUST DSP program, allowing to have full access to the user interface even in the lower dimensions.

This `Viewport` can either contains a `uiBox` as presented before, or a `uiTabs` if the program requires tabs.

4.3 uiTabs class

The `uiTabs` class inherits of `Juce::TabbedComponent`, which is a `Juce::Component` with a `TabbedButtonBar` on one of its size. It just needs a `Juce::Component` for each tab, and a tab name, and it will display them.

A tab layout is needed when the `buildUserInterface` starts with a `openTabBox` call. In this, a boolean `tabLayout` is set to true, to know that it's a tab layout.

While parsing the `buildUserInterface`, a `uiBox` is given to the `uiTabs` every time the current tab is "closed". To do that, a variable called `order` keeps track of the "level" of the current box. The order starts at 0, is incremented when a new box is opened, and decremented when a box is closed. If the order is 0 in a `closeBox()` call, then a tab is being closed, and so the current box is added to the `uiTabs`, using the `TabbedComponent::addTab` function.

Once all the tabs are closed, the `tabBox` is closed too, the `order` is now at -1, and it triggers the initialization function of `uiTabs`, `uiTabs::init()`. It'll be described it in the next subsection.

4.4 Initialization of the layout

First of all, while parsing the `buildUserInterface` lines, which are listing the different boxes and items that need to be displayed, the tree is getting built. It's done using the `Juce::Component` mechanics of `addAndMakeVisible`. The different `uiBaseComponent` are added as child of different `uiBox`, and `uiBox` display rectangle size and total size are calculated every time a box is closed in the `buildUserInterface` (i.e. when `closeBox()` is called).

The `uiBox` ***display rectangle*** size is the sum of his child width and the maximum of his child height, and the contrary depending on its orientation. But margins are added to our display rectangle width and height, 4 pixels per child, for a margin of 2 pixels on the top, left, bottom and right, and the `uiBox` ***total size*** is obtained. This is to avoid an overlapping effect, having two items touching each other. Following the same spirit, 12 pixels are added to the height of the box if its name needs to be displayed, 12 pixels being the space needed to display its name.

Here's the `buildUserInterface` that display this program:

```
ui_interface->openHorizontalBox("TITLE1");
```

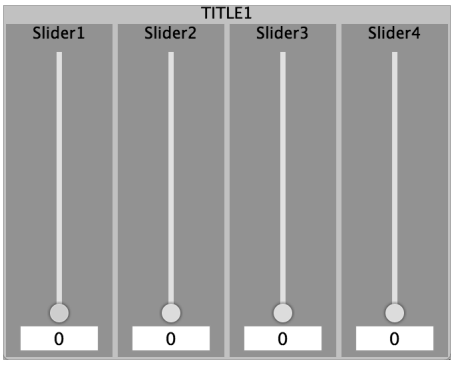


Figure 2: Representation of the display rectangle size and the total size of a box with four child

```
ui_interface->addVerticalSlider("Slider1",
      &fVslider0, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider2",
      &fVslider1, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider3",
      &fVslider2, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider4",
      &fVslider3, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->closeBox();
```

In Figure 2, the difference between the ***display rectangle size*** and the ***total size*** can be easily seen. The ***total size*** of the box here named "TITLE1" is the lighter gray, and the ***display rectangle size*** would be the four darker gray rectangle stick together. The layout is not aligned seamlessly because of the margin that is implemented to avoid the overlapping of the components.

The space left on the top of the box is for its title, and this margin is included in the ***total size***.

$$h = \sum_{i=0}^{n-1} (c_i.H) \tag{1}$$

$$w = \max_{i \in [0,n-1]} c_i.W \tag{2}$$

$$H = h + 4 * n \tag{3}$$

$$W = w + 4 \tag{4}$$

In those equations, H is the ***total height***, W the ***total width***, h the ***display rectangle height***, and w ***display rectangle width*** ; c_i being the nth child component of the current box.

H might get incremented by 12 pixels, depending on the need to display the box name.

4 pixels for each child component are added on a dimension to have margins between each of them, because they will be placed aside of each other in this dimension, and simply 4 pixels added to the other dimension to have 2 pixels separating parent and child box on each side.

Once `buildUserInterface` is done, the last box is closed, and the user interface initialized. This last box, that will be called the "main box" is initiated with ratios of 1 and 1, even if they are needed, because it'll take the window size. Here's how the UI is initialized:

- Setting the actual rendering size for the main box, because the total size is set here, but not the `Juce::Component` bounds. That's done through the `void setBaseComponentSize (Rectangle<int> r)` methods, which sets the size of the components, and especially position them right. Concretely, a 30 pixels offset is needed on the height for a tab layout, 30 pixels being the height taken by the tab bar. Only the main box needs to be set with an offset, because other boxes will be positioned depending on its parents coordinates.
- After that, the ratios are calculated for the whole tree, from root to leaves. The horizontal ratio is the component total width divided by its parent display rectangle width, same for the height. This way, it avoids to have the margins to mess with our ratios, and to have a sum of ratio equals to 1 instead of one approaching 1, but not being 1 exactly.
- Last step is to set the `LookAndFeel` for all `uiComponents`, which are for all of them the leaves of the trees. So the tree is fully parsed there, root to leaves.

The only possible change in the initialization of the program, is in a case of a tab layout. The `uiTabs::init()` method just calls the `uiBox::setRatio()` and the `uiBox::setCompLookAndFeel(LookAndFeel*)` for every of its tab component.

While going through all the tabs, the algorithm keeps track of the minimal size of the `uiTabs` component to be displayed. Its minimal dimensions being the maximum width and the maximum height of all its tabs.

There, the tree is built, the total size has been initiated, display rectangle size and the ratios for all components, all the `uiBox` and `uiComponent`.

4.5 Dynamic Layout

At that point, the user interface is displayed at his original size, but it needs to adapt to the potential resizing of the window. To do that, the `uiBoxes` are used to layout all the items. A `uiBox` item has a `void arrangeComponents(Rectangle<int> functionRect)` function, which is the main tool to organize the layout. It's called whenever the `resized()` function of the main box is called.

In this function, the initial rectangle given as argument, that is basically the window size, will propagate through all the child `uiBox` and `uiComponent`, in a recursive way [4].

At the beginning, it checks if the name needs to be displayed, and as no child components should be displayed there, it cuts 12 pixels from the top of the `functionRect`, given as argument.

After that, the margins are sets, so 2 pixels are cut on the left, top, right and bottom side. This way, overlapping components are avoided. Once it's done, it goes through all the child, to give them the right space to occupy and the right position of course.

The algorithm works that way: if the current box is vertical, then it needs to give its child a vertical part `functionRect`, and a horizontal one for a horizontal box of course.The amount of vertical or horizontal size of the child is calculated, still depending on the vertical nature of the current box. This size is the box current height or width, minus the margins, multiplied by the horizontal or vertical ratio. Concrete example: the current box is a horizontal display, and has 2 child components, one having a horizontal ratio of 0.7 and the other one of 0.3. The box display size is here 1000x500 pixels, and it's total size 1008x504 (2 items and it's a horizontal box, so $2 * (2 * margin) = 8$ on the width, and $2 * margin = 4$ on the height).

Let's say the size of the window almost doubled, and it's now 2008x1004 (arbitrary simple values). It will calculate that the first item get a $0.7*(2008-2*4) = 0.7*2000 = 1400$ pixels wide space and the second one $0.3*(2008-2*4) = 600$ pixels. First item bounds will be 1400x1000 and the second one 600x1000, height being kept the same, without the margins of course.

On top of that, to keep track of where to place our components, the `functionRect` get cut off little by little every time a `uiBaseComponent` is given a rectangle to be displayed in [5]. Basically, every rectangle that is given to child is removed from the original `functionRect`, and this allow us the keep track of the good `x` and `y` coordinates to give to the child component, with the margin added. It's done over and over again for each child component, cutting from the left or the top of the box`Rectangle<int> rect` depending on its orientation.

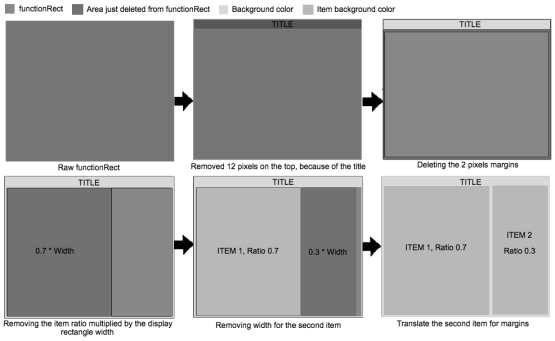


Figure 3: Representation of the layout algorithm

4.6 The MainContentComponent class

In the adapted `MainContentComponent` class, there is plenty of FAUST libraries, that are indispensable for the FAUST program. There are some optionals includes, for OSC, MIDI and polyphonic mode, that depends on the compilation options that the user sets.

The `MainContentComponent` class is the `Juce::Component` contained in our `Viewport`, and contains itself a `JuceGUI` object, that is a subclass of `Juce::Component`, FAUST `GUI` class and `MetaDataUI`. The minimal things to do is:

```
addAndMakeVisible(juceGUI);
fDSP = new mydsp();
fDSP->buildUserInterface(&juceGUI);
recommendedSize = juceGUI.getSize();
setSize (recommendedSize.getWidth(),
         recommendedSize.getHeight());
setAudioChannels (fDSP->getNumInputs(),
                  fDSP->getNumOutputs());
[...]

private:
   JuceGUI juceGUI;
```

A simple `buildUserInterface` call is needed, set the size of the `MainContentComponent`, and set the amount of audio channels. Following the

same spirit, there is optional code in case of a MIDI, OSC or polyphonic mode.

5 Other FAUST architecture files

Just a GUI architecture file isn't enough to run a FAUST program on JUCE, adaptations for different kind of control are also needed, such as OSC and MIDI.

5.1 OSC

OSC integration has been done by developing a new `JuceOSCUI` class, subclass of the base `UI` class. Two send and receive ports are defined. Input OSC messages are decoded by subclassing the JUCE `OSCReceiver` class, and implementing its `OSCReceiver::oscMessageReceived` method. Output OSC messages are sent by using the `OSCSender::send` method.

The special "hello" message allows to retrieve several parameters of the FAUST applications: its root OSC port, IP address, input and output port. The "get" message allows to retrieve the current, min and max values for a given parameter. Finally a float value received on a given path will allow to change the parameter value in real-time.

An application wanting to be controlled by OSC messages has to use an instance of the `JuceOSCUI` class, to be given to the DSP `buildUserInterface` method.

5.2 MIDI

MIDI messages handling is done by using the `MidiInput` and `MidiOutput` JUCE classes. A new `juce_midi` class subclassing the `MidiInputCallback` and implementing the required `MidiInputCallback::handleIncomingMidi Message` method has been defined. MIDI messages coming from the JUCE layer are decoded and sent to the corresponding application controllers. MIDI messages produced by the application controllers are encoded and sent using a `MidiOutput` object.

An application wanting to be controlled by MIDI messages has to use an instance of the `MidiUI` class, created with a `juce_midi` handler, to be given to the DSP `buildUserInterface` method.

6 Audio integration

To be connected to the external world, a given FAUST DSP has to be connected to an audio driver and a User Interface definition. JUCE framework already contains an abstract audio layer connected to a set of native audio drivers on all development platforms. JUCE developers can choose to deploy their code as standalone audio applications or audio plugins. A standalone application has to subclass the abstract `AudioAppComponent` class and implement the `prepareToPlay`, `getNextAudioBlock` and `releaseResources` methods:

- `prepareToPlay` is called just before audio processing starts with a sample rate parameter. The FAUST DSP is initiated with this sample rate value, and input/output channels number is possibly adapted to match the capabilities of the used native layer (that can a different number of input/output channels than the DSP).
- `getNextAudioBlock` is called every time the audio hardware needs a new block of audio data. Audio buffers presented as a `AudioSourceChannelInfo` data type are retrieved and adapted to be given to the FAUST DSP compute method.
- `releaseResources` is called when audio processing has finished. Nothing special has to be done at the FAUST level.

7 The faust2juce script

There are many scripts availiable in the FAUST ecosystem allowing to generate a ready to use binary, project file, or compiled file from a simple DSP file. They are labeled `faust2xxx`.

Following the same spirit, a `faust2juce` script has been implemented, that allows to create a JUCE project directory from a simple DSP file. The command is used as follow:

```
faust2juce [-options] dspFile.dsp
```

This will create a folder containing a .jucer file, and a "Source" folder containing the `Main.cpp` and the `MainComponent.h`. This folder is self contained, all needed FAUST includes are in the `MainComponent.h`, including the compiled DSP.

There are the options available at this moment for `faust2juce`:

- `-nvoices x`: produces a polyphonic self-contained DSP with x voices, ready to be used with MIDI events
- `-midi`: activates MIDI control
- `-osc`: activates OSC control

- `-help`: shows the different options available

As described in subsection 2.4, a number of voices can be hardcoded for a polyphonic DSP, but you can change it with the `nvoices` option. It has the priority over the metadata declaration. In the case of a non-hardcoded polyphonic DSP, it will just make it a polyphonic one with this compiler option. Some others options will be added later, it's still in development.

8 How to use JUCE architecture files

Using JUCE to export a FAUST DSP program file is easy: create the project folder with `faust2juce [-options] dspFile.dsp` and drag & drop the created folder named after the DSP to the "example" folder contained in the JUCE git folder.

Simply execute the .jucer file, and select "Save Project and Open in IDE...", the first time at least, to generate the JUCE header files, etc... And it's ready to execute your program on whatever export platform you chose.

9 Conclusion

The FAUST audio DSP language implementaion is now possible with JUCE, and can theoretically be exported to every platform that JUCE supports. It has been tested on OS X and iOS, both work correctly, and has a close performance to already available options, such as `faust2caqt` for OS X and `faust2ios`, for iOS.

MIDI control, polyphonic mode, and OSC control are implemented, more features are in progress of development, to permit a full compatibility with the whole FAUST library.

JUCE offers two types of "audio project", standalone applications or plug-in. Currently the FAUST architecture files are limited to describe standalone applications, but we are looking forward to adapt our code for plug-ins.

References

[1] Orlarey, Y., Fober, D., and Letz, S. (2009), "FAUST: an efficient functional approach to DSP programming." New Computational Paradigms for Computer Music, 290.

[2] D. Fober, Y. Orlarey, and S. Letz, "FAUST Architectures Design and OSC Support", *IRCAM*, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), pp. 231-216, 2011.

[3] JUCE online documentation `https://www.juce.com/doc/classes`

[4] JUCE "Tutorial: Advanced Rectangle techniques" `https://www.juce.com/doc/tutorial_rectangle_advanced`

[5] J. Storer "Developing Graphical User Interfaces with JUCE", JUCE Summit 2015 `https://www.youtube.com/watch?v=xsCZoE1s_uw`

Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files

Stéphane LETZ, Yann ORLAREY, Dominique FOBER
GRAME
Centre National de Création Musicale
11 Cours de Verdun (Gensoul)
69002, Lyon
France
{letz,orlarey,fober}@grame.fr

Romain MICHON
CCRMA
Stanford University
Stanford, CA 94305-8180
USA
rmichon@ccrma.stanford.edu

Abstract

The FAUST architecture files ecosystem is regularly enriched with new targets to deploy Digital Signal Processing (DSP) programs. This paper presents recently developed techniques to expand the standard *one DSP source, one program or plugin* model, and to better control parameter changes during the audio computation. Sample accurate control and polyphonic instruments definition have been introduced, and will be explained particularly in the context of MIDI control.

Keywords

FAUST, DSP programming, audio, MIDI

1 Introduction

FAUST is a functional programming language specifically designed for real-time signal processing and synthesis. From a high-level specification, its compiler typically generates the DSP computation as a C++ class[1] to be wrapped by so-called *architecture files* and connected to the external world.

1.1 Audio and UI Architecture Files

Native audio drivers are developed as subclasses of a base *audio* class, controllers as subclasses of a base *UI* class. Typical Graphical User Interface architectures are based on well established frameworks like QT[2] or JUCE[3], and allow to display a ready to use window with sliders, text zones and buttons. Audio and UI parts are finally combined with the actual DSP computation to produce the final audio application or plugin (see Figure 1).

Non graphical controllers can also be defined as subclasses of UI, simply by ignoring the layout description[4], and just keeping the actual controls definition (with their name, default value, value range etc.). OSCUI and httpdUI classes [1] typically follow this strategy.

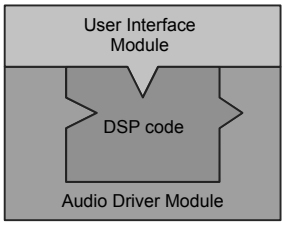


Figure 1: DSP code is generated by the compiler, audio and UI codes are added from the generic architecture files.

New architecture files have been regularly added to the already rich FAUST ecosystem, to expand the variety of possible targets for the DSP code.

1.2 Macro Construction of DSP Components

The FAUST program specification is usually entirely done in the language itself. But in some specific cases it may be useful to develop *separated DSP components* and *combine* them in a more complex setup.

Since taking advantage of the huge number of already available UI and audio architecture files is important, keeping the same `dsp` API is preferable[5], so that more complex DSP can be controlled and audio rendered the usual way:

```
class dsp {

    public:
        .....
        virtual int getNumInputs() {}
        virtual int getNumOutputs() {}
        virtual void buildUserInterface(UI* ui) {}
        virtual void init(int samplingRate) {}
```

[1] The faust2 development branch can also generate C, LLVM IR, WebAssemby etc. target languages.

[2] `http://doc.qt.io`

[3] `https://www.juce.com/doc/classes`

[4] Typically done using hgroup, vgroup or tgroup in the DSP source code.

[5] Only part of the complete DSP API is presented here.

```
        virtual void compute(int count,
                FAUSTFLOAT** inputs,
                FAUSTFLOAT** outputs) {}
        .....
};
```

Extended DSP classes will typically subclass the `dsp` root class and override part of its API.

This paper shows how this approach can be used to define new extended and combinable `dsp` classes. Section 2 describes tools to *combine* separately developed DSP. Section 3 explains how *sample accurate* parameter control of a given DSP can be done using the new `timed_dsp` class, and when it needs to be used.

Section 4 presents the model used to deploy polyphonic instruments, section 5 presents how the previously presented components can be used together in the context of MIDI control, and finally the conclusion tries to enlarge this work in a more general analysis of the FAUST compiler generated code.

2 Combining DSP

2.1 Dsp Decorator Pattern

A `dsp_decorator` class, subclass of the root `dsp` class has first been defined. Following the decorator design pattern[6], it allows behavior to be added to an individual object, either statically or dynamically.

The extended DSP class hierarchy is shown in Figure 2. As an example of the decorator pattern, the `timed_dsp` class allows to decorate a given DSP with sample accurate control capability as explained in section 3.

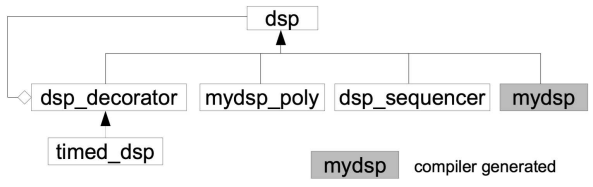


Figure 2: DSP classes diagram

2.2 Combining DSP Components

A few additional *macro construction* classes, subclasses of the root `dsp` class have been defined in the public *faust/dsp/dsp-combiner.h* header file:

- the `dsp_sequencer` class combines two DSP in sequence, assuming that the number of outputs of the first DSP equals the number of input of the second one. Its `buildUserInterface` method is overloaded to group the two DSP in a tabgroup, so that control parameters of both DSPs can be individually controlled[7]. Its `compute` method is overloaded to call each DSP compute in sequence, using an intermediate output buffer produced by first DSP as the input one given to the second DSP.

- the `dsp_parallelizer` class combines two DSP in parallel. Its `getNumInputs/getNumOutputs` methods are overloaded by correctly reflecting the input/output of the resulting DSP as the sum of the two combined ones. Its `buildUserInterface` method is overloaded to group the two DSP in a tabgroup, so that control parameters of both DSP can be individually controlled. Its `compute` method is overloaded to call each DSP compute, where each DSP consuming and producing its own number of input/output audio buffers taken from the method parameters.

3 Sample Accurate Control

DSP audio languages usually deal with several timing dimensions when treating control events and generating audio samples. For performance reasons, systems maintain separated audio rate for samples generation and control rate for asynchronous messages handling.

The audio stream is most often computed by blocks, and control is updated between blocks. To smooth control parameter changes, some language chose to interpolate parameter values [7] between blocks.

In some cases control may be more finely interleaved with audio rendering [8], and some languages [9] simply choose to interleave control and sample computation at sample level.

Although the FAUST language permits the description of sample level algorithms (like recursive filters etc.), FAUST generated DSP are usually computed by blocks. Underlying audio architectures usually give a fixed size buffer over and over to the DSP `compute` method which consumes and produces audio samples.

[6] `https://en.wikipedia.org/wiki/Decorator_pattern`

[7] Typically using any UI object.

3.1 Control to DSP Link

In the current version of the FAUST generated code, the primary connection point between the control interface and the DSP code is simply a memory zone. For control inputs, the architecture layer continuously write values in this zone, which is then *sampled* by the DSP code at the beginning of the `compute` method, and used with the same values during the entire call. Because of this simple control/DSP connexion mechanism, the *most recent value* is seen by the DSP code.

Similarly for control outputs[8], the DSP code inside the `compute` method possibly write several values at the same memory zone, and the *last value* only will be seen by the control architecture layer when the method finishes.

Although this behaviour is satisfactory for most use-cases, some specific usages need to handle the *complete* stream of control values with *sample accurate* timing. For instance keeping all control messages and handling them at their exact position in time is critical for proper MIDI clock synchronisation.

3.2 Time-Stamped Control

The first step consists in extending the architecture control mechanism to deal with *time-stamped* control events. Note that this requires the underlying event control layer to support this capability. The native MIDI API for instance is usually able to deliver time-stamped MIDI messages.

The next step is to keep all time-stamped events in a *time ordered* data structure to be continuously written by the control side, and read by the audio side.

Finally the sample computation has to take account of all queued control events, and correctly change the DSP control state at successive points in time.

3.3 Slices Based DSP Computation

With time-stamped control messages, changing control values at precise sample indexes on the audio stream becomes possible. A generic *slices based* DSP rendering strategy has been implemented in the `timed_dsp` class.

A ring-buffer is used to transmit the stream of time-stamped events from the control layer to the DSP one. In the case of MIDI control case for instance, the ring-buffer is written with a pair containing the time-stamp expressed in samples and the actual MIDI message each time one is received. In the DSP `compute` method, the ring-buffer will be read to handle all messages received during the previous audio block.

Since control values can change several times inside the same audio block, the DSP `compute` cannot be called only once with the total number of frames and the complete inputs/outputs audio buffers. The following strategy has to be used:

- several slices are defined with control values changing between consecutive slices.
- all control values having the same time-stamp are handled together, and change the DSP control internal state. The slice is computed up to the next control parameters time-stamp until the end of the given audio block is reached.
- in the Figure 3 example, four slices with the sequence of c1, c2, c3, c4 frames are successively given to the DSP `compute` method, with the appropriate part of the audio input/output buffers. Control values (appearing here as [v1,v2,v3], then [v1,v3], then [v1], then [v1,v2,v3] sets) are changed between slices.

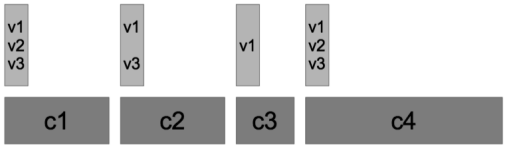


Figure 3: Audio block slice-based computation

Since time-stamped control messages from the previous audio block are used in the current block, control messages are aways handled with one audio buffer latency.

4 Polyphonic Instruments

Directly programing polyphonic instruments in FAUST is perfectly possible. It is also needed if very complex signal interaction between the different voices have to be described[9].

But since all voices would always be computed, this approach could be too CPU costly for simpler or more limited needs. In this case

[8] Using *bargraph* kind of UI elements.

[9] Like sympathetic strings resonance in a physical model of a piano for instance.

describing a single voice in a FAUST DSP program and externally combining several of them with a special *polyphonic instrument aware* architecture file is a better solution. Moreover, this special architecture file takes care of dynamic voice allocations and control MIDI messages decoding and mapping.

4.1 Polyphonic Ready DSP Code

By convention FAUST architecture files with polyphonic capabilities expect to find control parameters named *freq*, *gain* and *gate*. The metadata `declare nvoices "8";` kind of line with a desired value of voices can be added in the source code.

In the case of MIDI control, the *freq* parameter (which should be a frequency) will be automatically computed from MIDI note numbers, *gain* (which should be a value between 0 and 1) from velocity and *gate* from *keyon/keyoff* events. Thus, gate can be used as a trigger signal for any envelope generator, etc.

4.2 Using the mydsp_poly class

The single voice has to be described by a FAUST DSP program, the `mydsp_poly` class is then used to combine several voices and create a polyphonic ready DSP:

- the *faust/dsp/poly-dsp.h* file contains the definition of the `mydsp_poly` class used to wrap the DSP voice into the polyphonic architecture. This class maintains an array of `dsp` type of objects, manage dynamic voice allocations, control MIDI messages decoding and mapping, mixing of all running voices, and stopping a voice when its output level decreases below a given threshold.
- as a sub-class of DSP, the `mydsp_poly` class redefines the `buildUserInterface` method. By convention all allocated voices are grouped in a global *Polyphonic* tab-group. The first tab contains a *Voices* group, a master like component used to change parameters on all voices at the same time, with a *Panic* button to be used to stop running voices[10], followed by one tab for each voice. Graphical User Interface components will then reflect the multi-voices structure of the new polyphonic DSP (Figure 4).

[10] An internal control grouping mechanism has been defined to automatically dispatch a user interface action done on the master component on all linked voices.

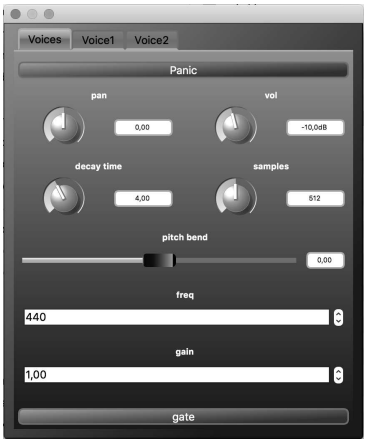


Figure 4: Extended multi-voices GUI interface

The resulting polyphonic DSP object can be used as usual, connected with the needed audio driver, and possibly other UI control objects like OSCUI, httpdUI, etc. Having this new UI hierarchical view allows complete OSC control of each single voice and their control parameters, but also all voices using the master component.

The following OSC messages reflect the same DSP code either compiled normally, or in polyphonic mode (only part of the OSC hierarchies are displayed here):

```
// Mono mode

/0x00/0x00/vol f -10.0
/0x00/0x00/pan f 0.0

// Polyphonic mode

/Polyphonic/Voices/0x00/0x00/pan f 0.0
/Polyphonic/Voices/0x00/0x00/vol f -10.0
...
/Polyphonic/Voice1/0x00/0x00/vol f -10.0
/Polyphonic/Voice1/0x00/0x00/pan f 0.0
...
/Polyphonic/Voice2/0x00/0x00/vol f -10.0
/Polyphonic/Voice2/0x00/0x00/pan f 0.0
...
```

The polyphonic instrument allocation takes the DSP to be used for one voice[11], the desired number of voices, the *dynamic voice allocation* state[12], and the *group* state which controls if separated voices are displayed or not (Figure 4):

```
DSP = new mydsp_poly(dsp, 2, true, true);
```

[11] The DSP object will be automatically cloned in the mydsp_poly class to create all needed voices.

[12] Voices may be always running, or dynamically started/stopped in case of MIDI control.

With the following code, note that a polyphonic instrument may be used outside of a MIDI control context, so that all voices will be always running and possibly controlled with OSC messages for instance:

```
DSP = new mydsp_poly(dsp, 8, false, true);
```

4.3 Controlling the Polyphonic Instrument

The `mydsp_poly` class is also ready for MIDI control and can react to *keyon/keyoff* and *pitchwheel* messages. Other MIDI control parameters can directly be added in the DSP source code.

4.4 Deploying the Polyphonic Instrument

Several architecture files and associated scripts have been updated to handle polyphonic instruments:

As an example on OSX, the script `faust2caqt foo.dsp` can be used to create a polyphonic CoreAudio/QT application. The desired number of voices is either declared in a `nvoices` metadata or changed with the `-nvoices num` additional parameter[13]. MIDI control is activated using the `-midi` parameter.

The number of allocated voices can possibly be changed at runtime using the `-nvoices` parameter to change the default value (so using `./foo -nvoices 16` for instance).

Several other scripts have been adapted using the same conventions.

4.5 Polyphonic Instrument with a Global Output Effect

Polyphonic instruments may be used with an output effect. Putting that effect in the main FAUST code is not a good idea since it would be instantiated for each voice which would be very inefficient. This is a typical use case for the `dsp_sequencer` class previously presented with the polyphonic DSP connected in sequence with a unique global effect (Figure 5).

`faustcaqt inst.dsp -effect effect.dsp` with inst.dsp and effect.dsp in the same folder, and the number of outputs of the instrument matching the number of inputs of the effect, has to be used. A `dsp_sequencer` object will be created to combine the polyphonic instrument in sequence with the single output effect.

[13]-nvoices parameter takes precedence over the metadata value.

Polyphonic ready *faust2xx* scripts will then compile the polyphonic instrument and the effect, combine them in sequence, and create a ready to use DSP.

Figure 5: Polyphonic instrument with output effect GUI interface: left tab window shows the polyphonic instrument with its *Voices* group only, right tab window shows the output effect.

5 MIDI Control

MIDI control connects DSP parameters with MIDI messages (in both directions), and can be used to trigger polyphonic instruments.

5.1 MIDI Messages Description in the DSP Source Code

MIDI control messages are described as metadata in UI elements. They are decoded by a new `MidiUI` class, subclass of UI, which parses incoming MIDI messages and updates the appropriate control parameters, or sends MIDI messages when the UI elements (sliders, buttons...) are moved.

5.2 Defined Standard MIDI messages

A special `[midi:xxx yyy...]` metadata needs to be added in the UI element. Here is the description of three common MIDI messages:

- `[midi:keyon pitch]` in a slider or bargraph will map the UI element value to keyon velocity in the (0, 127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0,
- `[midi:keyoff pitch]` in a slider or bargraph will map the UI element value to keyoff velocity in the (0,127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0,

- `[midi:ctrl num]` in a slider or bargraph will map the UI element value to (or from) (0, 127) range. When used with a button or checkbox, 1 will be mapped to 127, 0 will be mapped to 0.

The full description of supported MIDI messages is now part of the FAUST documentation.

5.3 MIDI Clock Synchronization

MIDI clock based synchronization can be used to slave a given FAUST program, using the sample accurate control mechanism described in section 3. The following three messages have to be used:

- `[midi:start]` in a button or checkbox will trigger a value of 1 when a start MIDI message is received
- `[midi:stop]` in a button or checkbox will trigger a value of 0 when a stop MIDI message is received
- `[midi:clock]` in a button or checkbox will deliver a sequence of successive 1 and 0 values each time a clock MIDI message is received, seen by FAUST code as a square command signal, to be used to compute higher level information.

A typical FAUST program will then use the MIDI clock command signal to possibly compute the Beat Per Minutes (BPM) information, or for any synchronization need it may have.

Here is a simple example of a sinusoid generated which a frequency controlled by the MIDI clock stream[14], and starting/stopping when receiving the MIDI start/stop messages:

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

// square signal (1/0), changing state
// at each received clock
clocker = checkbox("MIDI clock[midi:clock]");

// ON/OFF button controlled
// with MIDI start/stop messages
play = checkbox("On/Off [midi:start][midi:stop]");

// detect front
front(x) = (x-x') != 0.0;

// count number of peaks during one second
freq(x) = (x-x@ma.SR) : + ~ _;

process = os.osc(8*freq(front(clocker))) * play;
```

[14] Using an external MIDI clock generator and changing its tempo allow to precisely control the sinusoid frequency.

Note that the described sample accurate MIDI clock synchronization model can currently only be used at input level. Because of the simple memory zone based connection point between the control interface and the DSP code, output controls (like bargraph) cannot generate a stream of control values. Thus a reliable MIDI clock generator cannot be implemented with the current approach.

5.4 MIDI Classes

A `midi` base class defining MIDI messages decoding/encoding methods has been developed. A `midi_hander` subclass implements actual decoding. Several concrete implementations based on native API have been written (Figure 6) and can be found in the *faust/midi* folder.

Depending on the used native MIDI API, event time-stamps are either expressed in absolute time or in frames. They are converted to offsets expressed in samples relative to the beginning of the audio buffer.

Connected with the new `MidiUI` class, subclass of UI, they allow a given DSP to be controlled with incoming MIDI messages or possibly send MIDI messages when its internal control state changes.

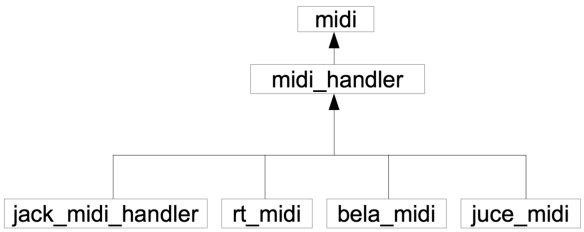


Figure 6: MIDI classes diagram

In the following piece of code, a `MidiUI` object is created and connected to a `rt_midi` [5] MIDI message handler, then given as parameter to the standard `buildUserInterface` to control the DSP parameters:

```
rt_midi midi_handler("MIDI");
MidiUI midiinterface(&midi_handler);
DSP->buildUserInterface(&midiinterface);
```

6 Deployment

The extended architecture files have been presented and used in the context of *statically generated and compiled DSP*, that is generating C++ code from FAUST, then compiling the resulting code in executable applications or plugins. They have been deployed in several *faust2xx*

scripts and especially in *faust2api* presented in [6].

Note that they can also be used with dynamically *libfaust* generated DSP[15] as in particular in FaustLive [3] standalone just-in-time FAUST compiler, or in *faustgen~* Max/MSP external object.

7 Conclusion

The sample accurate control model could easily be adapted to work with MIDI controllable plugins like LV2 instruments[16], so that MIDI clock synchronization could be used.

Expanding the polyphonic and sample accurate control model over the network in the *libfaustremote* [4] library is still in progress.

As a general concluding remark, a deeper rethinking of the control/DSP connection model in the FAUST compiled code will have to be done. As explained in section 3, control and DSP computation interaction is somewhat limited in the current model of the generated code.

The described solution stays at the architecture layer level with some limitations. Although sample accurate control for inputs can be done using the presented *slices based DSP computation*, this strategy does not help to properly retrieve the stream of control output values.

A cleaner approach would be to extend the model of control signals to be *a list of time-stamped values*, so that the `compute` would handle a slice of time-stamped input controls (kept from the previous block), and possibly produces a slice of time-stamped output controls. Having this more general strategy at the code generation level still has to be developed.

Acknowledgments

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

References

[1] D. Fober, Y. Orlarey, and S. Letz, "FAUST Architectures Design and OSC Support", *IRCAM*, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), pp. 231-216, 2011.

[2] Orlarey, Y., Fober, D., and Letz, S. (2009), "FAUST: an efficient functional approach to DSP programming." New Computational Paradigms for Computer Music, 290.

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

[4] S. Letz, S. Denoux and Y. Orlarey, "Audio Rendering/Processing and Control Ubiquity ? a Solution Built Using the Faust Dynamic Compiler and JACK/NetJack." ICMC/SMC, Athenes 2014.

[5] "RtMidi framework online documentation" `http://www.music.mcgill.ca/~gary/rtmidi/`

[6] R.Michon, J.Smith, C.Chafe, S. Letz and Y. Orlarey, "faust2api: a Comprehensive API Generator for Android and iOS." Linux Audio Conference, 2017.

[7] J.McCartney, "Rethinking the Computer Music Language: SuperCollider." Computer Music Journal, Winter 2002.

[8] P.Donat-Bouillud, JL.Giavitto, A.Cont, N.Schmidt and Y.Orlarey, "Embedding native audio-processing in a score following system with quasi sample accuracy." ICMC, Utrecht 2016.

[9] G. Wang, P. R. Cook, and S. Salazar, "Chuck: A strongly timed computer music language." Computer Music Journal, 2016.

[15] Dynamically libfaust generated DSP are objects of llvm_dsp or interpreter_dsp types, subclasses of the dsp root class with the same API.

[16] `http://lv2plug.in/doc/html/`

faust2api: a Comprehensive API Generator for Android and iOS

Romain Michon, Julius Smith, Chris Chafe
CCRMA
Stanford University
Stanford, CA 94305-8180
USA
{rmichon,jos,cc}@ccrma.stanford.edu

Stéphane Letz, Yann Orlarey
GRAME
Centre National de Création Musicale
11 Cours de Verdun (Gensoul)
69002, Lyon
France
{letz,orlarey}@grame.fr

Abstract

We introduce `faust2api`, a tool to generate custom DSP engines for Android and iOS using the FAUST programming language. FAUST DSP objects can easily be turned into MIDI-controllable polyphonic synthesizers or audio effects with built-in sensors support, etc. The various elements of the DSP engine can be accessed through a high-level API, made uniform across platforms and languages.

This paper provides technical details on the implementation of this system as well as an evaluation of its various features.

Keywords

FAUST, iOS, Android, Mobile Instruments

1 Introduction

Mobile devices (smart-phones, tablets, etc.) have been used as musical instruments for the past ten years, both in the industry (e.g., GarageBand[1] for iPad, Smule's apps,[2] moForte's GeoShred,[3] etc.), and in the academic community ([Tanaka, 2004], [Geiger, 2006], [Gaye et al., 2006], [Essl and Rohs, 2009] and [Wang, 2014]).

Implementing real-time Digital Signal Processing (DSP) engines from scratch on mobile platforms can be hard using standard audio APIs provided with common operating systems (we'll only cover iOS and Android here). Indeed, CoreAudio on iOS and OpenSL ES on Android are relatively low-level APIs offering customization possibilities not needed by most audio app developers. Fortunately, there exist several third party cross-platform APIs to work with real-time audio on mobile devices at a higher level (e.g., SuperPowered,[4] JUCE,[5] etc.). Additionally, several open-source tools allow to use objects written in common computer music languages such as PureData:[6] *libpd* [Brinkmann et al., 2011] and Csound:[7] *Mobile Csound Platform* (*MCP*) [Lazzarini et al., 2012] on mobile platforms.

Similarly, we introduced `faust2android` in a previous publication [Michon, 2013]: a tool allowing to turn FAUST[8] [Orlarey et al., 2009] code into a fully operational Android application. `faust2android` is based on `faust2api` [Michon et al., 2015]. It allows to turn a FAUST program into a cross-platform API usable on Android and iOS to carry out various kinds of real-time audio processing tasks.

In this paper, we present a completely redesigned version of `faust2api` offering the same features on Android and iOS:

- polyphony and MIDI support,
- audio effects chains,
- built-in sensors support,
- low latency audio,
- etc.

First, we'll give an overview of how `faust2api` works. Then, technical details on the implementation of this system will be provided. Finally, we'll evaluate it and present future directions for this project.

2 Overview

2.1 Basics

At its highest level, `faust2api` is a command line program taking a FAUST code as its main argument and generating a package containing a series of files implementing the DSP engine. Various flags can be used to customize the API. The only required flag is the target platform:

[1] `http://www.apple.com/ios/garageband`. All the URLs in this paper were verified on 01/26/17.
[2] `https://www.smule.com`
[3] `http://www.moforte.com/geoshredapp`
[4] `http://superpowered.com`
[5] `https://www.juce.com`
[6] `https://puredata.info`
[7] `http://www.csounds.com`
[8] `http://faust.grame.fr`

```
faust2api -ios myCode.dsp
```

will generate a DSP engine for iOS and

```
faust2api -android myCode.dsp
```

will generate a DSP engine for Android.

The content of each package is quite different between these two platforms (see §3), but the format of the API itself remains very similar (see Figure 1 at page 4). The iOS DSP engines generated with `faust2api` consist of a large `C++` object (`DspFaust`) accessible through a separate header file. This object can be conveniently instantiated and used in any `C++` or `Objective-C` code in an app project. A typical "life cycle" for a `DspFaust` object can be

```
DspFaust *dspFaust = new DspFaust(SR,
    blockSize); dspFaust->start();
dspFaust->stop(); delete dspFaust;
```

`start()` launches the computation of the audio blocks and `stop()` stops (pauses) the computation. These two methods can be repeated as many times as needed. The constructor allows to specify the sampling rate and the block size, and is used to instantiate the audio engine. While the configuration of the audio engine is very limited at the API level (only these two parameters can be configured through it), lots of flexibility is given to the programmer within the FAUST code. For example, if the FAUST object doesn't have any input, then no audio input will be instantiated in the audio engine, etc.

The value of the different parameters of a FAUST object can be easily modified once the `DspFaust` object is created and is running. For example, the `freq` parameter of the simple FAUST code

```
f = nentry("freq",440,50,1000,0.01);
    process = osc(f);
```

can be modified simply by calling

```
dspFaust->setParamValue("freq",440);
```

FAUST user-interface elements (`nentry` here) are ignored by `faust2api` and simply used as a way to declare parameters controllable in the API. API packages generated by `faust2api` also contain a markdown documentation providing information on how to use the API as well a list of all the parameters controllable with `setParamValue()`.

The structure of the DSP engine package is quite different for Android since it contains both `C++` and `JAVA` files (see §3). Otherwise, the same steps can be used to work with the `DspFaust` object.

2.2 MIDI Support

MIDI support can be easily added to a `DspFaust` object simply by providing the `-midi` flag when calling `faust2api`. MIDI support works the same way on Android and iOS: all MIDI devices connected to the mobile device before the app is launched can control the FAUST object, and any new device connected while the app is running will also be able to control it.

Standard FAUST MIDI meta-data[9] can be used to assign MIDI CCs to specific parameters. For example, the `freq` parameter of the previous code could be controlled by MIDI CC 52 simply by writing

```
f = nentry("freq[midi: ctrl
    52]",440,50,1000,0.01);
```

2.3 Polyphony

FAUST objects can be conveniently turned into polyphonic synthesizers simply by specifying the maximum number of voices of polyphony when calling `faust2api` using the `-nvoices` flag. In practice, only active voices are allocated and computed, so this number is just used as a safeguard.

As used for many years by the various tools for making FAUST synthesizers, such as `faust2pd`, compatibility with the `-nvoices` option requires the `freq`, `gain` and `gate` parameters to be defined. `faust2api` automatically takes care of converting MIDI note numbers to frequency values in Hz for `freq`, MIDI velocity to linear amplitude-gain for `gain`, and note-on (1) and note-off (0) for `gate`:

```
f = nentry("freq",440,50,1000,0.01); g
    = nentry("gain",1,0,1,0.01);
t = button("gate"); process = osc(f)*g*
    t;
```

Here, `t` could be used to trigger an envelope generator, for example. In such a case, the voice would stop being computed only after `t` is set to 0 and the tail-off amplitude becomes smaller than -60dB (configurable using macros in the application code).

A wide range of methods is accessible to work with voices. A "typical" life cycle for a MIDI note can be

```
long voiceAddress = dspFaust->keyOn(
    note,velocity);
dspFaust->setVoiceParamValue("param",
    voiceAddress,paramValue);
```

[9] http://faust.grame.fr/images/faust-quick-reference.pdf

```
dspFaust->keyOff(note);
```

`setVoiceParamValue()` can be used to change the value of a parameter for a specific voice.

Alternatively, voices can be allocated without specifying a note number and a velocity:

```
long voiceAddress = dspFaust->newVoice
    ();
dspFaust->setVoiceParamValue("param",
    voiceAddress,paramValue);
dspFaust->deleteVoice(voiceAddress);
```

For example, this can be very convenient to associate voices to specific fingers on a touch-screen.

When MIDI support is enabled in `faust2api`, MIDI events will automatically interact with voices. Thus, if a MIDI keyboard is connected to the mobile device, it will be able to control the FAUST object without additional configuration steps.

2.4 Adding Audio Effects

In most cases, effects don't need to be re-implemented for each voice of polyphony and can be placed at the end of the DSP chain. `faust2api` allows to provide a FAUST object implementing the effects chain to be connected to the output of the polyphonic synthesizer. This can be done simply by giving the `-effect` flag followed by a FAUST effects chain file name (e.g., `effect.dsp`) when calling `faust2api`:

```
faust2api -android -nvoices 12 -effect
    effect.dsp synth.dsp
```

The parameters of the effect automatically become available in the `DspFaust` object and can be controlled using the `setParamValue()` method.

2.5 Working With Sensors

The built-in accelerometer and gyroscope of a mobile device can be easily assigned to any of the parameters of a FAUST object using the `acc` or `gyr` meta-data:

```
g = nentry("gain[acc: 0 0 -10 0
    10]",1,0,1,0.01);
```

Complex mappings can be implemented using this system. This feature is not documented here, but more information about it is available in [Michon, 2017]. This reference also provides a series of tutorials on how to use `faust2api`.

3 Implementation

`faust2api` takes advantage of the modularity on the FAUST architecture system to generate its custom DSP engines. [Letz et al., 2017] For example, turning a monophonic FAUST synthesizer into a polyphonic one can be done in a simple generic way. Both on Android and iOS, `faust2api` generates a large `C++` file implementing all the features used by the high level API. On iOS, this API is accessed through a `C++` header file that can be conveniently included in any `C++` or `Objective-C` code. On Android, a `JAVA` interface allows to interact with the native (`C++`) block. The DSP `C++` code is the same for all platforms (see Figure 2 at page 5) and is wrapped into an object implementing the polyphonic synthesizer followed by the effects chain (assuming that the `-mvoices` and `-poly2` options were used during compilation).

In this section, we provide more information on the architecture of DSP engines generated by `faust2api` for Android and iOS.

3.1 iOS

The global architecture of API packages generated by `faust2api` is relatively simple on iOS since `C++` code can be used directly in `Objective-C` (which is one of the two languages used to make iOS applications along with `swift`). The FAUST synthesizer object gets automatically connected to the audio engine implemented using `CoreAudio`. As explained in the previous section, the sampling rate and the buffer length are defined by the programmer when the `DspFaust` object is created. The number of instantiated inputs and outputs is determined by the FAUST code. By default, the system deactivates gain correction on the input but this can be changed using a macro in the including source code.

MIDI support is implemented using `RtMidi` [Scavone and Cook, 2005], which is automatically added to the API if the `-midi` option was used for compilation. Alternatively, programmers might choose to use the `propagateMidi()` method to send raw MIDI events to the `DspFaust` object in case they would like to implement their own MIDI receiver.

The same approach can be used for built-in sensors using the `propagateAcc()` and `propagateGyr()` methods.

3.2 Android

Android applications are primarily written in `JAVA`. However, despite the fact that the FAUST compiler can generate `JAVA` code, it is not a

Basic Elements	**Parameters Control**
`DspFaust`: Constructor	`getParamsCount`: Get number of params
`~DspFaust`: Destructor	`setParamValue`: Set param value
`start`: Start audio processing	`getParamValue`: Get param value
`stop`: Stop audio processing	`getParamAddress`: Get param address
`isRunning`: True if processing is on	`getParamMin`: Get param min value
`getJSONUI`: Get UI JSON description	`getParamMax`: Get param max value
`getJSONMeta`: Get Metadata JSON	`getParamInit`: Get param init value
	`getParamTooltip`: Get param description
Polyphony	
`keyOn`: Start a new note	**Other Functions**
`keyOff`: Stop a note	`propagateMidi`: Propagate raw MIDI messages
`newVoice`: Start a new voice	`propagateAcc`: Propagate raw accel data
`deleteVoice`: Delete a voice	`setAccConverter`: Set accel mapping
`allNotesOff`: Terminate all active voices	`propagateGyr`: Propagate raw gyro data
`setVoiceParamValue`: Set param value for a specific voice	`setGyrConverter`: Set gyro mapping
`getVoiceParamValue`: Get param value for a specific voice	`getCPULoad`: Get CPU load

Figure 1: Overview of the API functions.

good choice for real-time audio signal processing [Michon, 2013]. Thus, DSP packages generated by `faust2api` contain elements implemented both in `JAVA` and `C++`.

The native portion of the package (`C++`) implements the DSP elements as well as the audio engine (see Figure 2) which is based on `OpenSL ES`.[10] The audio engine is configured to have the same behavior as on iOS. Native elements are wrapped into a shared library accessible in `JAVA` through a *Java Native Interface* (JNI) using the *Android Native Development Kit* (NDK).[11]

MIDI receivers can only be created in `JAVA` on Android (and only since Android API 23), thus MIDI support is implemented in the `JAVA` portion. Like on iOS, the `propagateMidi()` method can be used to implement custom MIDI receivers.

While raw sensor data can be retrieved in `C++` on Android, we decided to implement a system similar to the one used for MIDI, where raw sensor data are pushed from the `JAVA` layer to the native one.

[10] `https://www.khronos.org/opensles`

[11] `https://developer.android.com/ndk/index.html`

4 Evaluation

4.1 Use in Other Frameworks

`faust2api` is now used at the core of `faust2android` [Michon, 2013] and `faust2ios`. It is also used as the basis for our new `SmartKeyboard`[12] tool (currently under development), allowing to generate musical applications with advanced user interfaces on Android and iOS. Figure 3 presents *Nuance*, [Michon et al., 2016] a musical instrument based on `faust2api` and `SmartKeyboard`.

4.2 Audio Latency

We measured the "touch-to-sound" and the "round-trip" audio latency of apps based on `faust2api` for various devices using the techniques described by *Google* on their website.[13] The "touch-to-sound" latency is the time it takes to generate a sound after a touch event was registered on the touch screen of the device. The "round-trip" latency is the time it takes to process an analog signal recorded by the built-in microphone or acquired by the line input.

Latency performance hasn't improved on iOS (see Table 1) compared to our previous study [Michon et al., 2015], except for newer devices

[12] `https://ccrma.stanford.edu/~rmichon/smartKeyboard`

[13] `https://source.android.com/devices/audio/latency_measurements.html`

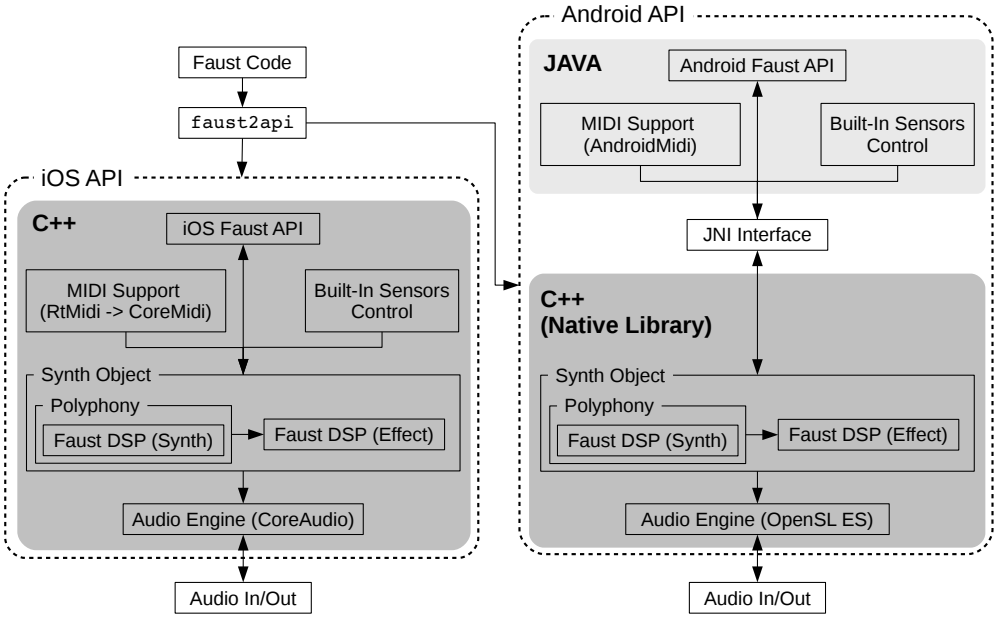


Figure 2: Overview of DSP engines generated with `faust2api`.

Figure 3: *Nuance*: a musical instrument using `faust2api`.

Device	Touch to Sound	Round Trip
iPhone6	30 ms	13 ms
iPhone5	36 ms	13 ms
iPodTouch	36 ms	13 ms
iPadPro	28 ms	12 ms
iPadAir2	35 ms	13 ms
iPad2	45 ms	15 ms

Table 1: Audio latency for different iOS devices using `faust2api`.

Device	Touch to Sound	Round Trip	OS
HTC Nexus 9	29 ms	15 ms	7.0
Huawei Nexus 6p	31 ms	17 ms	7.0
Asus Nexus 7	37 ms	48 ms	7.0
Samsung Gal. S5	37 ms	48 ms	5.0

Table 2: Audio latency for different Android devices using `faust2api`.

such as the *iPad Pro*. On the other hand, Android made huge progress (see Table 2), thanks to tremendous work carried out by *Google*, as well as our completely rewritten audio engine.

Table 2 shows that a "reasonable" latency can only be achieved with the latest version of `Android`, which confirms the measurements made by `Google`.[14] Unfortunately, such performances can only be attained on a few devices supported by *Google*, and configured with a specific sampling rate and buffer length.

5 Future Directions

We believe that `faust2api` has reached a mature and stable state. However, many elements can be improved:

First, while basic MIDI support is provided, we haven't tested it with complex MIDI interfaces such as the one using the Multidimensional Polyphonic Expression (MPE) standard (e.g. LinnStrument,[15] ROLI Seaboard,[16] etc.).

[14] `https://source.android.com/devices/audio/latency_measurements.html\#measurements`

[15] `http://www.rogerlinndesign.com/linnstrument.html`

[16] `https://roli.com/products/`

Currently, specific parameters of the various elements of the API (such as audio engine, MIDI behavior, etc.) can only be configured using source-code macros. We would like to provide a more systematic and in some cases dynamic way of controlling them.

Finally, we plan to add more targets to `faust2api` for various kinds of platforms to help design elements such as audio plug-ins, standalone applications, and embedded systems.

6 Conclusions

FAUST gives access to dozens of high quality open source sound processors and generators ranging from specialized types of filters, to virtual analog oscillators, etc. Thanks to `faust2api`, all these elements can be easily embedded and controlled in any Android or iOS app in a very simple manner.

One of the new experimental features of the FAUST compiler allows to select at run time the portions of a FAUST object that are computed. This makes it possible to create very large objects embedding multiple synthesizers and effects. We believe that this feature, in combination with `faust2api`, will allow to design complex FAUST-based DSP engines for a wide range of platforms.

References

Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. 2011. Embedding PureData with libpd. In *Proceedings of the Pure Data Convention*, Weinmar, Germany.

Georg Essl and Michael Rohs. 2009. Interactivity for mobile music-making. *Organised Sound*, 14(2):197–207.

Lalya Gaye, Lars Erik Holmquist, Frauke Behrendt, and Atau Tanaka. 2006. Mobile music technology: Report on an emerging community. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME-06)*, Paris, France, June.

Günter Geiger. 2006. Using the touch screen as a controller for portable computer music instruments. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME-06)*, Paris, France.

`seaboard-grand`

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. 2012. The mobile Csound platform. In *Proceedings of the International Conference on Computer Music (ICMC-12)*, Ljubljana, Slovenia, September.

Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon. 2017. Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files. In *Proceedings of Linux Audio Conference (LAC-17)*, Saint-Etienne, France.

Romain Michon, Julius Orion Smith, and Yann Orlarey. 2015. MobileFaust: a set of tools to make musical mobile applications with the Faust programming language. In *Proceedings of the Linux Audio Conference (LAC-15)*, Mainz, Germany, April.

Romain Michon, Julius O. Smith, Chris Chafe, Matthew Wright, and Ge Wang. 2016. Nuance: Adding multi-touch force detection to the iPad. In *Proceedings of the Sound and Music Computing Conference (SMC-16)*, Hamburg, Germany.

Romain Michon. 2013. faust2android: a Faust architecture for Android. In *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, September.

Romain Michon. 2017. Faust tutorials. Webpage. `https://ccrma.stanford.edu/~rmichon/faustTutorials`.

Yann Orlarey, Stéphane Letz, and Dominique Fober, 2009. *New Computational Paradigms for Computer Music*, chapter "Faust : an Efficient Functional Approach to DSP Programming". Delatour, Paris, France.

Gary Scavone and Perry Cook. 2005. RtMidi, RtAudio, and a synthesis toolkit (STK) update. In *Proceedings of the 2005 International Computer Music Conference*, Barcelona, Spain.

Atau Tanaka. 2004. Mobile music making. In *Proceedings of the 2004 conference on New interfaces for musical expression (NIME04)*, National University of Singapore.

Ge Wang. 2014. Ocarina: Designing the iPhone's Magic Flute. *Computer Music Journal*, 38(2):8–21, Summer.

New Signal Processing Libraries for Faust

Romain Michon, Julius Smith
CCRMA
Stanford University
Stanford, CA 94305-8180
USA
{rmichon,jos}@ccrma.stanford.edu

Yann Orlarey
GRAME
Centre National de Création Musicale
11 Cours de Verdun (Gensoul)
69002, Lyon
France
orlarey@grame.fr

Abstract

We present a completely re-organized set of signal processing libraries for the FAUST programming language. They aim at providing a clearer classification of the different FAUST DSP functions, as well as better documentation. After giving an overview of this new system, we provide technical details about its implementation. Finally, we evaluate it and give ideas for future directions.

Keywords

FAUST, Digital Signal Processing, Computer Music Programming Language

1 Introduction

FAUST is a functional programming language for real time Digital Signal Processing (DSP) targeting high-performance audio applications and plug-ins for a wide range of platforms and standards. [Orlarey et al., 2009]

One of FAUST's strength lies in its DSP libraries implementing a large collection of reference implementations ranging from filters to audio effects and sound generators, etc.

When FAUST was created, it had a limited number of DSP libraries that were organized in a "somewhat" coherent way: `math.lib` contained mathematical functions, and `music.lib` everything else (filters, effects, generators, etc.). Later, the libraries `filter.lib`, `oscillator.lib`, and `effect.lib` were developed [Smith, 2008], [Smith, 2012], which had significant overlap in scope with `music.lib`.

A year ago, we decided to fully reorganize the FAUST libraries to

- provide more clarity,
- organize functions by category,
- standardize function names,
- create a dynamic documentation of their content.

In this paper, we give an overview of the organization of the new FAUST libraries, as well as technical details about their implementation. We then evaluate them through the results of a workshop on FAUST that was taught at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University in 2016, and we provide ideas for future directions.

2 Global Organization and Standards

2.1 Overview

The new FAUST libraries[1] are organized in different files presented in Figure 1. Each file contains several subcategories allowing to easily find functions for specific uses. While some libraries host fewer functions than others, they were created to be easily updated with new elements. The content of the old (and now deprecated) FAUST libraries was spread across these new files, making backward compatibility a bit hard to implement (see §2.4).

More specifically, the old `music.lib` was removed since it contained much overlap in scope with `oscillator.lib`, `effect.lib`, and `filter.lib`.

`effect.lib` was divided into several "specialized" libraries: `compressors.lib`, `misceffects.lib`, `phaflangers.lib`, `reverbs.lib`, and `vaeffects.lib`. Similarly, the content of `oscillator.lib` is now spread between `noises.lib` and `oscillators.lib`. Finally, `demo.lib` hosts demo functions, typically adding user-interface elements with illustrative parameter defaults.

2.2 Prefixes

Each FAUST library has a recommended two-letter namespace prefix defined in the "meta library" `stdfaust.lib`. For example, `stdfaust.lib` contains the lines

[1] `http://faust.grame.fr/library.html`. All the URLs in this paper were verified on 01/30/17.

analyzer.lib
- Amplitude Tracking
- Spectrum-Analyzers
- Mth-Octave Spectral Level
- Arbitrary-Crossover Filter
- Banks and Spectrum Analyzers

basics.lib
- Conversion Tools
- Counters and Time/Tempo Tools
- Array Processing and Pattern Matching
- Selectors (Conditions)
- Other Misc Functions

compressors.lib
Compressors and limiters library.

delays.lib
- Basic Delay Functions
- Lagrange Interpolation
- Thiran Allpass Interpolation

demos.lib
- Analyzers
- Filters
- Effects
- Generators

envelopes.lib
Envelope generators library.

filters.lib
- Basic Filters
- Comb Filters
- Direct-Form Sections
- Direct-Form Second-Order
- Biquad Sections
- Ladder/Lattice
- Virtual Analog Filters
- Simple Resonator
- Butterworth Filters
- Elliptic (Cauer) Filters
- Filters for Parametric Equalizers (Shelf, Peaking)
- Arbitrary-Crossover Filter-Banks

maths.lib
- Constants
- Functions

misceffects.lib
- Dynamic
- Filtering
- Time Based
- Pitch Shifting
- Meshes

noises.lib
Noise generators library.

oscillators.lib
- Wave-Table-Based Oscillators
- LFOs
- Low Frequency Sawtooths
- Bandlimited Sawtooth
- Bandlimited Pulse, Square, and Impulse Trains
- Filter-Based Oscillators
- Waveguide-Resonators

phaflangers.lib
Phasers and flangers library

reverbs.lib
Reverbs library.

routes.lib
Signal routing library.

signals.lib
Misc signal tools library.

spats.lib
Spatialization tools library.

synths.lib
Misc synthesizers library.

vaeffects.lib
Virtual analog effects library.

Figure 1: Overview of the organization of the new FAUST libraries.

```
fi = library("filters.lib");
os = library("oscillators.lib");
```

so that functions from `oscillator.lib` can be invoked using the `os` prefix and functions from `filter.lib` through `fi`:

```
import("stdfaust.lib");
process = os.sawtooth(440) : fi.lowpass
    (2,2000);
```

It is of course possible to avoid prefixes using the `import` directive:

```
import("filters.lib");
import("oscillators.lib");
process = sawtooth(440) : lowpass
    (2,2000);
```

The libraries presently avoid name collisions, so it is possible to load all functions from all libraries into one giant namespace soup:

```
import("all.lib");
process = sawtooth(440) : lowpass
    (2,2000);
```

Alternatively, all FAUST-defined functions can be loaded into a single namespace separate from the user's namespace:

```
sf = library("all.lib"); // standard
    faust namespace
process = sf.sawtooth(440) : sf.lowpass
    (2,2000);
```

Further details can be found in the documentation for the libraries.[2]

2.3 Standard Functions

The FAUST libraries implement dozens of functions, and it can be hard for new users to find standard elements for basic uses. For example, `filter.lib` contains seven different lowpass filters, and it's probably not obvious to someone with little experience in signal processing which one should be used.

To address this problem, the new FAUST libraries declare "standard" functions (see Figure 2) that are automatically added to the library documentation.[3] Standard functions are organized by categories, independently from the library where they are declared (see §3). They should cover the needs of most users used to computer music programming environments such as PureData,[4] SuperCollider,[5] etc.

2.4 Backward Compatibility

With such major changes, providing a decent level of backward compatibility proved to be quite complicated. The old FAUST libraries (`effect.lib`, `filter.lib`, `math.lib`, `music.lib` and `oscillator.lib`) can still be used and will remain accessible for about one year.

In order to make this possible, we had to find a way to make them cohabit with the new libraries without creating conflicts. Thus, we decided to use plurals for the name of the new libraries, allowing to concurrently use our new `filters.lib` with the old `filter.lib`, for example.

If one of the old libraries is imported in a FAUST program, the FAUST compiler now throws a warning indicating the use of a deprecated library.

2.5 Other "Non-Standard" Libraries

A few "non-standard" libraries for very specific applications remain accessible but are not documented (see §3):

- `hoa.lib`: high order ambisonics library
- `instruments.lib`: library used by the FAUST-STK [Michon and Smith, 2011]
- `maxmsp.lib`: compatibility library for Max/MSP
- `tonestacks.lib`: tonestack emulation library used by Guitarix[6]
- `tubes.lib`: guitar tube emulation library used by Guitarix

3 Automatic Documentation

The new FAUST libraries use a new automatic documentation system based on the `faust2md` (FAUST to MarkDown) script which is now part of the FAUST distribution. It allows to easily write MarkDown comments within the code of the libraries by respecting the standards described below.

Library headers and descriptions can be created with

```
//##### Library Name ##### // Some
    Markdown text.
//#########################
```

Libraries can be organized into sections using the following syntax:

```
//===== Section Name ===== // Some
    Markdown text.
//=======================
```

Each function in a library should be documented as such:

```
//---- Function Name ---- // Some
    Markdown text.
//-----------------------
```

The libraries documentation can be conveniently generated by running:

```
make doclib
```

[2] `http://faust.grame.fr/library.html`

[3] `http://faust.grame.fr/library.html\#standard-functions`.

[4] `https://puredata.info`.

[5] `http://supercollider.github.io`.

[6] `http://guitarix.org`.

Analysis Tools	
an.amp_follower	Amplitude follower
an.mth_oct[...]	Octave analyzers
Basic Elements	
ba.beat	Pulse generator
si.block	Block a signal
ba.bpf	Break Point Function
si.bus	Bus of n signals
ba.bypass1	Bypass (mono)
ba.bypass2	Bypass (stereo)
ba.count	Counts in a list
ba.countdown	Samples count down
ba.countup	Samples count up
de.delay	Integer delay
de.fdelay	Fractional delay
ba.impulsify	Signal to impulse
ba.sAndH	Sample and hold
ro.cross	Cross n signals
si.smoo	Smoothing
si.smooth	Controllable smoothing
ba.take	Element from a list
ba.time	Timer
Conversion	
ba.db2linear	dB to linear
ba.linear2db	Linear to dB
ba.midikey2hz	MIDI key to Hz
ba.pole2tau	Pole to t60
ba.samp2sec	Samples to seconds
ba.sec2samp	Seconds to samples
ba.tau2pole	t60 to pole
Effects	
ve.autowah	Auto-wah
co.compressor	Compressor
ef.cubicnl	Distortion
ve.crybaby	Crybaby
ef.echo	Echo
pf.flanger	Flanger
ef.gate_mono	Signal gate
co.limiter	Limiter
pf.phaser2	Phaser
re.fdnrev0	Reverb (FDN)
re.freeverb	Reverb (Freeverb)
re.jcrev	Reverb (simple)
re.zita_rev1	Reverb (Zita)
sp.panner	Panner
ef.transpose	Pitch shift
sp.spat	Panner
ef.speakerbp	Speaker simulator
ef.stereo_width	Stereo width
ve.vocoder	Vocoder
ve.wah4	Wah

Envelopes	
en.adsr	ADSR envelope
en.ar	AR envelope
en.asr	ASR envelope
en.smoothEnv	Exponential envelope
Filters	
fi.bandpass	Bandpass (Butterworth)
fi.resonbp	Bandpass (resonant)
fi.bandstop	Bandstop (Butterworth)
fi.tf2	Biquad Filters
fi.allpass_fcomb	Comb (allpass)
fi.fb_fcomb	Comb (feedback)
fi.ff_fcomb	Comb (feedforward)
fi.dcblocker	DC blocker
fi.filterbank	Filterbank
fi.fir	FIR (arbitrary order)
fi.high_shelf	High shelf
fi.highpass	Highpass (Butterworth)
fi.resonhp	Highpass (resonant)
fi.iir	IIR (arbitrary order)
fi.levelfilter	Level filter
fi.low_shelf	Low shelf
fi.lowpass	Lowpass (Butterworth)
fi.resonlp	Lowpass (resonant)
fi.notchw	Notch filter
fi.peak_eq	Peak equalizer
Generators	
os.impulse	Impulse
os.imptrain	Impulse train
os.phasor	Phasor
no.pink_noise	Pink noise
os.pulsetrain	Pulse train
os.lf_imptrain	Low-freq pulse train
os.sawtooth	Sawtooth wave
os.lf_saw	Low-freq sawtooth
os.osc	Sine (filter-based)
os.oscsin	Sine (table-based)
os.square	square wave
os.lf_square	Low-freq square
os.triangle	Triangle
os.lf_triangle	Low-freq triangle
no.noise	White noise
Synths	
sy.additiveDrum	Additive drum
sy.dubDub	Filtered sawtooth
sy.combString	Comb string
sy.fm	FM
sy.sawTrombone	Lowpassed sawtooth
sy.popFiltPerc	Popping filter

Figure 2: Standard FAUST functions with their corresponding prefix when used with `stdfaust.lib`.

at the root of the FAUST distribution. This will generate an `html` and a `pdf` file in the `/documentation` folder using `pandoc`.[7]

4 Evaluation and Future Directions

The new FAUST libraries were beta tested during the *CCRMA Faust Summer Workshop* at Stanford University.[8] In previous editions of the workshop, students had to go through the library files to get the documentation of specific functions. During last year's workshop, thanks to the new libraries documentation, students were able to find information about functions simply by doing a search in the documentation file. Additionally, none of them encountered problems while using the new libraries which was very satisfying.

The FAUST libraries are meant to grow with time, and we hope that this new format will facilitate the integration of new contributions. Eventually, we plan to divide `filters.lib` into more subcategories, like we did for the old `oscillator.lib`. Finally, `physmodels.lib` which is a new library for physical modeling of musical instruments is currently under development.

5 Conclusions

The new FAUST libraries provide a platform to easily prototype DSP algorithms using the FAUST programming language. Their new organization, in combination with their automatically generated documentation, simplifies the search for specific elements covering a wide range of uses. New "standard functions" help to point new users to useful elements to implement various kind of synthesizers, audio effects, etc. Finally, we hope that this new format will encourage new contributions.

6 Acknowledgments

Thanks to Albert Gräf for his contributions to the design of the new libraries, and for single-handedly implementing a solid backward-compatibility scheme!

References

Romain Michon and Julius O. Smith. 2011. Faust-STK: a set of linear and nonlinear physical models for the Faust programming language. In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September.

Yann Orlarey, Stéphane Letz, and Dominique Fober, 2009. *New Computational Paradigms for Computer Music*, chapter "Faust : an Efficient Functional Approach to DSP Programming". Delatour, Paris, France.

Julius Orion Smith. 2008. Virtual electric guitars and effects using Faust and Octave. In *Proceedings of the Linux Audio Conference (LAC-08)*, pages 123–127, KHM, Cologne, Germany.

Julius O. Smith. 2012. Signal processing libraries for Faust. In *Proceedings of Linux Audio Conference (LAC-12)*, Stanford, USA, May.

[7] `http://pandoc.org`.

[8] `https://ccrma.stanford.edu/~rmichon/faustWorkshops/2016`.

Heterogeneous data orchestration
Interactive fantasy under SuperCollider

Sébastien Clara
CIEREC UJM – PhD student
35 rue du 11 novembre
Saint-Étienne, France, 42000
sebastien.clara@univ-st-etienne.fr

Abstract

For *L'Imaginaire* music ensemble, I composed a piece of interactive music involving acoustic instruments and surround electronic music. The interaction between live musicians and electronics is based on data collected in real time from acoustic instruments. This data is further used to adjust timbre and synchronize electronics with the rest of the music.

Keywords

SuperCollider, interactivity, music mixed

1 Introduction

For *L'Imaginaire*[1] music ensemble, I composed a piece of interactive music mixed. Mixed music is a term used in musicological literature to refer to a musical genre. It is defined by the alloy of instrumental music and electronic music. For this paper, I wish to focus on an interactive property of my piece.

Historically, the electronic part of mixed music is composed in a studio and fixed on a magnetic tape[2]. This technique makes impossible the interaction between the interpreters and the electronic sound. Feedback helps to improve the compositional process of the tape. The other technique is to perform audio processing of the acoustic instruments in real time[3]. In this case, we are talking about a device that increases the sonic possibilities of the acoustic instrument. Feedback is used to regulate electronic sound by a human or an automaton.

I use these two techniques of electronic sound accompaniment in my work, but the increase in computing power and the versatility of the tools have allowed a median way. In the next chapter, I present my problematic. Thereafter, I will show my issue with an example which could be extrapolated to other devices.

2 Problematic

According to Robert Rowe, "*Interactive computer music systems are what are the changes in response to musical input. Such responsiveness allows these systems to participate in live performances, of both notated and improvised music*" [1]. How can a system listen to a musician and make an appropriate decision to generate a sound response? In his book, Rowe analyzes systems that use the *MIDI* standard to communicate between instruments and computers. But how can we use traditional instruments?

The audio descriptors[4] correspond to parameters that describe an analyzed sound. A set of descriptors is used to construct a data set and create spaces for representing the sound. The parameters that are extracted can be described in different ways, depending on what is expected from the information conveyed by the parameter.

The sound acquisition sensor for the processing unit is a microphone. Therefore, the use of audio descriptors in interactive computer music systems allows the use of standard acoustic instruments.

[1]Musical ensemble composed by Keiko Murakami (flutes), Philippe Koerper (saxophones) and Maxime Springer (piano). http://www.limaginaire.org/.

[2]The first pieces of music using this technique: *Orphée 51* and *Orphée 53* by Pierre Schaeffer and Pierre Henry (1951 and 1953) and *Musica su due dimensioni* by Bruno Maderna (1952 and 1958).

[3]The first pieces of music using this technique: *Mixtur* (1964) and *Mikrophonie I* (1964) by Karlheinz Stockhausen.

[4]With *SuperCollider*, it is necessary to install its extension package to benefit from a wide choice of descriptors : https://github.com/supercollider/sc3-plugins.

To control electronic sound with the sound of an acoustic instrument, many descriptors can be used. To do this, it is necessary to match a sound characteristic with the values of the parameters that describe it. The preparation of this report will allow us to establish a particular threshold and once it is crossed, the system can trigger a response.

However, the interval between the extreme values of a parameter depends on its nature and on the analyzed sound source. To use this technique of interaction between an instrumentalist and electronic sound, a first difficulty is to negotiate with the heterogeneity of the data produced by the different audio descriptors.

For example, I want to use the nuance and pitch of a sound to build a particular accompaniment. The amplitude descriptor returns a number that can range from 0 to 1 and the pitch descriptor returns a frequency. The range of extreme values returned by the pitch descriptor depends on the ambit of the analyzed sound source. Determining a trigger threshold to activate a response of a system involves crossing values of different magnitudes sometimes from sound sources of different natures.

Moreover, this technique uses sound capture and this information is variable depending on its environment. Consequently, elaborate settings in a studio with a particular electro-acoustic chain would be less effective in another location with a different electro-acoustic chain. So, how can we treat the heterogeneity of the data produced by different audio descriptors, different sound sources, different electro-acoustic chains and different concert locations, to achieve homogeneous electronic music accompaniment for each interpretation of the same piece?

3 Creating a repository

The first step is to establish a reference. This will serve as a standard for a single piece, a specific electro-acoustic chain and a particular place. We will be obliged to renew it each time one of these three parameters changes. Furthermore, this repository will allow us to characterize our data and to determine thresholds for performing a particular electronic sound accompaniment.

In the following example, we use the amplitude descriptor (*Amplitude.kr*) and pitch (*Pitch.kr*). The values are transmitted by the *OSC* protocol (*SendReply.kr*) to the *SuperCollider* clients at the */dataTrigger* address. The data is sent whenever an onset is detected (*Onsets.kr*). When instantiating the synthesizer, two arguments are available. The first determines the number of inputs of the signal to be analyzed on our audio interface (*SoundIn.ar*). The second determines the onset detection threshold.

```
(
SynthDef(\dataTrigger, { arg in = 0, onsetsThres = 0.5;
    var signal = SoundIn.ar(in);

    var chain = FFT(LocalBuf(1024), signal);

    var trig = Onsets.kr(chain, onsetsThres);

    SendReply.kr(trig, '/dataTrigger',
        [Amplitude.kr(signal), Pitch.kr(signal)[0]]);
}).add;
)
```

Figure 1

Execution of figure 1 will only give the definition of your synthesizer to *SuperCollider* audio server[5]. Synthesizer is not instantiated, so it does not work and it does not ask for resources to your hardware. We will run[6] it only when we want to receive data (figure 2).

We now have to build a data collector to constitute our repository. To do this, we use the *OSCFunc* object. It is fast responder for incoming *OSC* messages. We configure it with the previously defined *OSC* address. When a new message arrives from the analyzer, it executes a function. In this case, this function saves the amplitude and pitch of the signal in array global variables.

```
(
var onsetsThres = -9.dbamp;

~dataTrigger = Dictionary.new;

// Run analytical synthesizer
~analyzerTrig = Synth(\dataTrigger,[\onsetsThres, onsetsThres]);


// Saving data from the current analysis
~oscTrigger = OSCFunc({ arg msg;
    ~dataTrigger[\amp] = ~dataTrigger[\amp].add(msg[3]);
    ~dataTrigger[\pitch] = ~dataTrigger[\pitch].add(msg[4]);
}, '/dataTrigger');
)


// At the end, free objects
~analyzerTrig.free; ~oscTrigger.free;
```

Figure 2

We set to -9 dB the onsets detection threshold. We can change this parameter for change the density of the data reception. The analyzer listens to the first input of our audio interface (default setting). In figure 2, running the first block starts the acquisition of the data. Running the last line kills the synthesizer and responder instances, frees memory and processor usage. The collected

[5] http://doc.sccode.org/Classes/SynthDef.html.

[6] http://doc.sccode.org/Classes/Synth.html.

information is stored in variable arrays. We can plot data in graph or histogram (figure 3).

```
~dataTrigger[\pitch].plot;
~dataTrigger[\pitch].plotHisto;
```

Figure 3

The graph shows the evolution of the parameter over time and allows us to make a correspondence between a sound characteristic and some values. The histogram provides a representation of the distribution of the values of the parsed parameter. This observation allows us to characterize the distribution produced by a descriptor.

4 Using the repository

In our system, we determine the value of a threshold to trigger a response to accomplish dynamic electronic music accompaniment. This choice may be arbitrary or be determined in response to a specificity of the analyzed sound source. Above a certain value, our program triggers a response for example. We can also choose this value according to its frequency of appearance in the distribution and according to the sound result produced by this choice, we can increase or decrease the density of our electronic accompaniment by modifying the value of our threshold. However, how do we handle values of different magnitudes?

We manipulate repository by the requested percentile. To use this method, it is necessary to install an additional library. For that, you can use the *SuperCollider* package manager[7] to install *MathLib*. This one gives us access to additional statistics methods for arrays.

The percentile rank corresponds to the proportion of the values of a distribution less than or equal to a determined value. We manipulate our data with float values from 0 to 1. For example, if we want to know the value equal to 90% of our data, we use 0.9.

```
~dataTrigger[\pitch].percentile(0.9);
```

Figure 4

During the adjustment phase of our system, we can tune several parameters of different magnitudes transparently with a single scale.

5 System Response

In order for our system to respond to certain stimuli, we must attribute to it a means of sound production. To do this, we define an arbitrary synthesizer (Figure 5).

```
(
SynthDef(\fmGrain, {arg out = 0, amp = 0.75, density = 20, carfreq
440, modfreq = 200, modIndex = 1, pos=0, dur = 3;

    var env = EnvGen.kr(Env.perc, levelScale: amp, timeScale:dur,
doneAction: 2);

    var signal = FMGrain.ar(Impulse.ar(density), 0.05, carfreq,
modfreq, env*modIndex, env);

    Out.ar(out, Pan2.ar(signal, Line.kr(pos, pos.neg, dur)))
}).add;
)
```

Figure 5

To implement a concrete example, we assume that our device listens to two types of percussion. One of the percussions emits sounds high-pitched than the other. We decide that our system will respond only to the percussion which emits the most high-pitched sounds and to the most loud sounds. With onsets detection threshold, our condition for triggering a response depends on two others parameters:

```
pitchThres = ~dataTrigger[\pitch].percentile(0.7);
ampThres = ~dataTrigger[\amp].percentile(0.5);
```

Figure 6

If the frequency of the answers does not suit us, we can return to the choice of the values of these variables to modify the sensitivity of our system.

```
(
var pitchThres, ampThres, onsetsThres;

// Interaction control
pitchThres = ~dataTrigger[\pitch].percentile(0.7);
ampThres = ~dataTrigger[\amp].percentile(0.5);
onsetsThres = -9.dbamp;


// Run analytical synthesizer
~analyzerTrig = Synth(\dataTrigger,[\onsetsThres, onsetsThres]);


~oscReply = OSCFunc({ arg msg;

    // Condition for reply density
    if((msg[4] > pitchThres).and(msg[3] > ampThres), {

        // Mapping control
        Synth(\fmGrain,[
            \density, msg[3].linlin(ampThres,1, 3,20),

            \carfreq, msg[4].linlin(
                pitchThres,
                ~dataTrigger[\pitch].maxItem,
                ~dataTrigger[\pitch].percentile(0.05),
                ~dataTrigger[\pitch].percentile(0.95)),

            \modfreq, msg[4] * msg[3].linlin(ampThres,1, 0.5,3),

            \modIndex, msg[3].explin(ampThres,1, 1,20),

            \dur, msg[3].linlin(ampThres,1, 2,10),

            \pos, 1.0.rand2
        ]);
    });
}, '/dataTrigger');
)
```

Figure 7

[7] http://doc.sccode.org/Classes/Quarks.html.

The implementation of a response for our system follows the same structure as Figure 2. A responder wait for incoming *OSC* messages from the analyzer. When a new message arrives, if it meets the previously formulated conditions (high-pitched and loud), a response is issued.

This response is customized according to the analysis data. This connection is made by a sonification process, "*technique of rendering sound in response to data and interactions*" [2]. I do not deal with the mapping technique in this paper, but its mastery is a source of variation and expressiveness for electronic music. To make this relation, we map the synthesizer parameter from an input range to an output range. We can set the input range with the repository information and adjust the output according to the desired sound quality. Figure 7 is the implementation of the response of our system.

At the end, we must free objects for frees memory and processor usage.

```
~analyzerTrig.free; ~oscReply.free;
```

Figure 8

6 For further

For this paper, we concentrated our system to its simplest expression. In this chapter, we wish to develop it design. Some of these ideas were conceived during the development of our piece and others afterwards.

Robert Rowe divides his interactive computer music system (*Cypher*) into two sections [1]. The listener analyzes the data produced by a musician and the player delivers a musical response. The structure of *SuperCollider* source code implies this organization. Our analysis synthesizer is the listener and the function of the responder object for incoming *OSC* messages is the player. We keep these terms to locate the following points.

Our listener can also have an implicit function of time master. Indeed, we transmit the data of the analysis every time an onset is detected. But we can transmit them at a given frequency. The responses delivered by the system would then be have a beat.

We can use our listener to produce an automation (with *Env* and *EnvGen.kr* objects). An automation allows to control and to automate the variation of a parameter over a given time. In this way, we determine the evolution of any threshold or parameter.

We can parallelize other listeners who analyze other sound qualities (brightness, noise, dissonance, etc.) to achieve other triggers threshold and make complex electronic music accompaniment executed by our system.

For our player, we can define a maximum number of synthesizers executed in parallel in order to preserve the resources of the system and / or to control the acoustic density so as not to saturate our perception.

In addition, we can perform a certain musical process[8] or that feeds on our repository instead of running a simple synthesizer. The interactive system developed by Jean-Claude Risset for his duets for one pianist [3] is very interesting for this way. He uses *MIDI* data (pitch, velocity and duration) which he transforms according to traditional compositional operations: transposition, reversal, canon, etc.

7 Implementation

I control my audio processor by a graphical interface (Figure 9) and *MIDI* controller. The different audio tracks allow me to adjust the intensity of the electronic sound layers. The flute, sax and piano tracks deal with interactive electronic sound accompaniment. The synth track manages my non-real time composite electronic music. Finally, the live track amplifies the acoustic instruments.

The creation of the repository is realized directly in the interface and makes this action transparent. Graphical interface allows a sound engineer to play my work and this interface makes rehearsals and concerts easier.

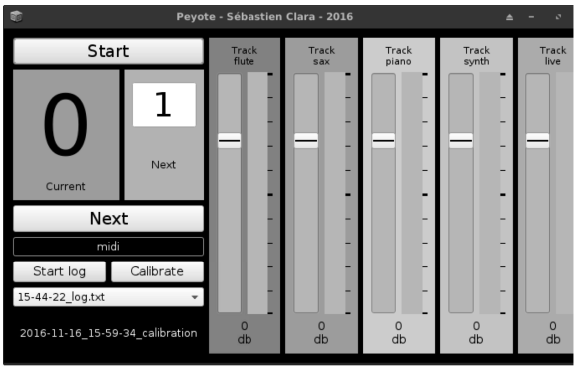


Figure 9

A *MIDI* pedal assigned to a performer manages the overall setting. Fourteen key moments

[8] http://doc.sccode.org/Tutorials/A-Practical-Guide/PG_01_Introduction.html.

articulate the electronics for a duration of fifteen minutes.

To create my electronic accompaniment, I use the descriptors of amplitude, pitch, centroid and noise. The operation of coupling between the data of the analysis and the parameters of the synthesizers [2] makes it possible to control the form of electronic music accompaniment by scaling, transposition or reversal. This relation can be fixed or variable.

Generally, the sound source of my electronic music accompaniment does not come from sound synthesis, but acoustic instruments. When the density of the responses of the system is not limited, the responses are superimposed to continuously transform the timbre of the electronic sound. When the density of the responses of the system is limited, the responses can arrive in successive waves and produce a dynamic accompaniment.

8 Epilogue

To design our system, our initial motivation was to simplify the use of different descriptors, to simplify our system settings, to customize the responses of the system according to the sensitivity of the interpreter and to produce a homogeneous electronic accompaniment to each interpretation of the same notated music under different conditions.

But in addition, we obtain an open interactive system that can be adapted from a specific model to the intuition of a musician. We identified four steps to explore in order to implement our practical solution and develop an interactive scenario [4].

The first step focuses on the sound of the instrumentalist. What particularities of sound do we want to relate to our system? What filter do we want to use to trigger an answer? In the example developed for this paper, we make our filter with the parameters of onset detection, pitch and amplitude. The thresholds established to constitute this filter allow us to play on the sensitivity or the particularity of the answers delivered by our system.

The second question to implement our solution is to choose the type of response to trigger. Should the answer be monophonic, polyphonic, contrapuntal, etc.? In other words, what organizational model should we use to develop our response? In our example, we produce one item per answer. This element is strongly correlated with the sound analyzed and the operations applied to determine the sound characteristics of our response are conceived during the last step of this practical solution.

The third step in using our system is to determine which synthesizer we want to assign to our system. Controls can be implemented in synthesizers. This possibility can give us solutions to build a previously chosen model. For example, a synthesizer can perform a glissando. In the example developed for this paper, we use a simple granular FM synthesis.

The final step in implementing our solution is to determine the type of relationship between the analysis data and the parameters of our synthesizer. How to get expressive sounds with sonification process [2] ? Should our relationship be static or dynamic? How should the plan of connections between these various elements be established? For the example developed in this paper, we have established a one-to-many and many-to-one static connection plan. The amplitude determined by the analysis is correlated with the granular density of the synthesis, the modulation index and the duration of the response. The pitch determined by the analysis is correlated with the pitch of our response. A transposition is performed by a scaling operation. Finally, the amplitude and the pitch analyzed serve to determine the modulation frequency of the FM synthesis of the response delivered by our system.

9 Conclusion

For this paper, we have implemented our open interactive system under *SuperCollider* - platform for audio synthesis and algorithmic composition. We could have implemented this system on other software. Moreover, use free software increases the durability of our work. Laurent Pottier recalls the history of the precariousness of technologies in electronic music [5] and free software answers to this problem.

An example concerns the portage of *Pluton* by Philippe Manoury from the *4X*[9] to *Max* [6]. The piece did not really sound exactly the same way on both platforms. After a thorough study of the *4X*, the engineers discovered that a *4X* hardware limitation influenced the sound result. This limitation was implemented in the *Max* patch to find an electronic music equivalent [7].

[9] *4X* is real time effect processors designed by Giuseppe di Guigno at IRCAM in the 1970s.

Free software is an important factor of durability and reproducibility in the digital art. The ubiquity [8] of free software allows more flexibility to imagine original devices [9]. In the end, researchers have no lock to study and increase the common good.

10 Acknowledgements

Thanks to Laurent Pottier for his encouragement and his pertinent advices.

References

[1] R. Rowe. 1993. *Interactive Music Systems: Machine Listening And Composing*. MIT Press, Cambridge.

[2] T. Hermann, A. Hunt, J. G. Neuhoff. 2011. *The Sonification Handbook*, Logos Verlag, Berlin.

[3] J.-C. Risset, S. Van Duyne. 1996. Real-Time Performance Interaction with a Computer-Controlled Acoustic Piano. In *Computer Music Journal 20/1*, pp. 62–75, MIT Press, Cambridge.

[4] B. Laurel. 2014, Second Edition. *Computers as Theatre*. Addison-Wesley, Crawfordsville.

[5] L. Pottier. 2015. L'évolution des outils technologiques pour la musique électronique, en rapport avec la pérennité des œuvres. Constat, propositions. A. Saemmer, editor, *E-Formes 3, Les frontières de l'œuvre numérique*, pp. 245-261, PUSE, Saint-Étienne.

[6] M. Puckette. 2002. Max at Seventeen. In *Computer Music Journal 26/4*, pp. 31-43, MIT Press, Cambridge.

[7] J. Szpirglas. 2012. *Composer, même avec trois bouts de ficelle… Entretien avec Philippe Manoury*. http://etincelle.ircam.fr/1077.10.html.

[8] S. Letz, S. Denoux, Y. Orlarey. 2014. Audio Rendering/Processing and Control Ubiquity? a Solution Built Using the Faust Dynamic Compiler and JACK/NetJack. In *Proceedings ICMC|SMC*, Athens.

[9] M. Lallement. 2015. *L'âge du faire. Hacking, travail, anarchie*. Le Seuil, Paris.

Higher Order Ambisonics for SuperCollider

Florian Grond
Input Device and
Music Interaction Laboratory, McGill
555, Sherbrooke W
H3A 1E3 Montreal CANDA,
floriangrond@gmail.com

Pierre Lecomte
LMSSC,
2 Rue Conté
75003 Paris,
FRANCE,
pierre.lecomte@gadz.org

Abstract

In this paper we present a library for 3D Higher Order Ambisonics (HOA) for the SuperCollider (SC) sound programming environment. The library contains plugins for all standard operations in a typical Ambisonics signal-flow: encoding, transforming and decoding up to the 5th order. Carefully designed PseudoUgens are the interface to those plugins to aim for the best possible code flexibility and code reusability. As a key feature, the implementation is designed to handle the higher order B-format as a channel array and to obey the channel expansion paradigm in order to take advantage of the powerful scripting possibilities of SC. The design of the library and its components is described in details. Moreover, some examples are given for how to built flexible HOA processing chains with the use of node proxies.

Keywords

SuperCollider, Higher Order Ambisonics.

1 Introduction

Ambisonics, i.e. the description of sound pressure fields through spherical harmonics decomposition, has been around for quite a while since its invention by [Gerzon, 1973]. Back in the days, the harmonics decomposition was up to the first order (First Order Ambisonics, FOA), using the 4-channel B-format[1] The playback of FOA audio content depended on special hardware and did not make it into mainstream audio in the first decade of its existence for various reasons, one of which being that FOA offers only limited spatial resolution.

Ambisonics research made significant advances in the 2000's through the work of Bamford [Bamford, 1995], Malham [Malham, 1999] and Daniel [Daniel, 2000], who extended the sound pressure field decomposition to higher orders hence the term (HOA). HOA increases the spatial resolution and thereby reduces the limitation of low spatial definition when compared with other spatialization techniques.

For streamlining and standardising content production, one hurdle that HOA was facing in the past was the coexistence of various channel ordering and normalization conventions. In order to address this issue, the *Ambix* standard was proposed by [Nachbar et al., 2011] and is ever since increasingly adopted by recent HOA implementations.

Today, processors can handle with ease multiple instances of multi-channel sound processes. Further, the rise of video games and Virtual Reality (VR) applications has elicited new interest in Ambisonics amongst audio researchers and content creators. This is mostly due to its inherent property to yield easy-to-manipulate isotropic 360 degree sound pressure fields, which can be rendered either through multi loudspeaker arrays or headphones. In the case of VR applications head-tracking is already available and the listener is always in the sweet spot. For the capturing of HOA 3D sound pressure fields, various microphone array prototypes have been developed some of them being available as commercial products like mhacoustic's Eigenmike® for instance. As far as multi loudspeaker reproduction is concerned, the number of loudspeaker domes with semi spherical configurations is growing and electroacoustic composers have also shown increasing interest in HOA as a spatialisation technique, notably amongst them for composition and [Barrett, 2010] and sonification [Barrett, 2016].

1.1 Ambisonics in various platforms

Over the last few years HOA has seen various implementations in diverse sound software environments, mostly as plugins in DAWs. The *Ambisonics Studio* plugins by Daniel Courville, for instance, have been around for some time[2].

[1] The term B-format is often used for FOA signals, in this paper we use the term also for higher orders.

[2] `http://www.radio.uqam.ca/ambisonic`

Another recent and very comprehensive example its the *Ambix plugin suite* from [Kronlachner, 2013] also see [Kronlachner, 2014a]. For Pure Data and MaxMSP, HOA libraries have been made available by the *Centre de recherche Informatique et Création Musicale*[3] an early implementation can also be found with the ICST Ambisonic tools [Schacher, 2010]. An early version for Pure Data can be found in the collection of abstractions called *CubeMixer* by [Musil et al., 2003]. Recently, the HOA library *Ambitools*[4] developed mainly in Faust has been made available [Lecomte and Gauthier, 2015].

1.2 SuperCollider

The audio synthesis environment SuperCollider (SC) by [McCartney, 2002] is particularly well suited for the creation of dynamic audio scenes. SC is split into two parts: The server *scsynth* for efficient sound synthesis and *sclang*, an object oriented programming language for the flexible configuration and re-patching of DSP trees on the server. Similar to most sound programming environments, synthesis is based in SC on unit generators called *Ugens*. Third party Ugens are collected separately in *SC3plugins*. Extensions to sclang are mananged through *Quarks*. Ugens can be assembled to more complex arrangements through synthesis definitions, known as *SynthDefs*, which are executable binaries for synthesis in scsynth. In sclang, *PseudoUgens* can be created, which is another way of handling complex arrangements of Ugens in sclang, which are compiled for scsynth, when needed. For a detailed introduction to SC see [Valle, 2016] and the SuperCollider book[5] by [Wilson et al., 2011].

1.3 Ambisonics in SuperCollider

In 2005, Frauenberger et al. implemented HOA in SC as the AmbIEM Quark[6]. This implementation goes up to the 3rd order, and follows the old Furse Malham channel ordering and normalization. All unit generators (Ugens) like encoding, rotation, and simple decoding are implement in sclang as PseudoUgens. AmbIEM comes with an simulation of early reflections in a virtual room but lacks functionality such as beamforming. The Ambisonics Toolkit (Atk) for SC by [Anderson and Parmenter, 2012] is a more recent and very comprehensive set of tools. The Atk includes for instance various transformations to manipulate the directivity of the sound field such as pushing and zooming. It is however only a first order implementation of Ambisonics.

1.4 Library design in SuperCollider

In this context the paper presents a modern HOA implementation for SC, which is modular and adopts all established standards in terms of channel ordering and normalizations. Inspired by the approach found in the Atk and typical for the general design of SC, computationally intensive parts like Ugens are split from PseudoUgens convenience wrapper classes of sclang. The HOA library comes hence in three parts, SC3plugins, PseudoUgens and audiocontent plus HRTFs for binaural rendering in a support directory.

1.4.1 SC3plugins

The first part of the library is a collection of Ugens, which is part of the SC3plugins collection. Each Ugen is compiled from C++ code. It consists of a SC language side representation of the Ugen as a *.sc* class file and a *.scx*, *.so* or *.dll* compiled dynamic link library, for the platforms (OSX, Linux or Windows) respectively. For each Ambisonics order (so far up to order 5), there are individual Ugens for the encoding, transforming and decoding processes in a typical Ambisonics signal flow. The C++ code for these Ugens is generated from the HOA library *Ambitools* [Lecomte and Gauthier, 2015] with the compilation tool *faust2supercollider*. This approach was taken for two reasons:

First, to leverage the work already accomplished in Faust. Indeed, the Faust compiler generates very efficient DSP code and the Faust code base allows to efficiently combine existing functionality. The meta approach through Faust will lead to future additions of functionality, which can then be easily integrated in the HOA library for SC.

Second, each Ambisonics order comes with a defined multichannel B-format, this in turn defines the amount of input and output arguments for the Ugens. For instance, a Ugen rotating an Ambisonic signal of order 3 has 16 input arguments plus the rotation angles and 16 output channels. While it is of interest to expose the Ambisonics order as an argument for the flexibility and reusability of code on the side of *sclang*, it is an argument unlikely to be changed while

[3] http://www.mshparisnord.fr/hoalibrary/en

[4] http://faust.grame.fr/news/2016/10/17/Faust-Awards-2016.html

[5] http://supercolliderbook.net/

[6] https://github.com/supercollider-quarks/AmbIEM

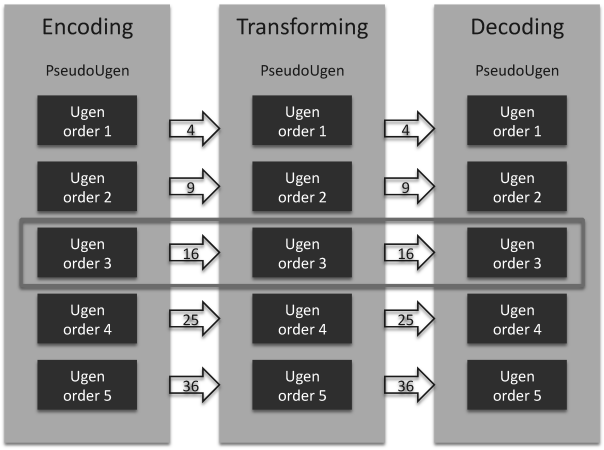


Figure 1: The Ambisonics processing chain in the HOA library for the selected order 3 with 16 channels.

an instance of the Ugen is running as a node in the DSP tree on *scsynth*. This is why for every order there is a unique Ugen for each function (encoding, transforming, decoding).

The Ugens follow these conventions, some of which are explained in subsequent sections):

- The Ambisonics channels are ordered according to the ACN convention.
- The default normalisation of the B-format is N3D.
- All azimuth and elevation arguments follow the spherical coordinates convention from SC.
- Operations of resource intensive Ugens can by bypassed.

Based on the implementation in Faust, the main functionalities of the HOA Ugens provided as the SC3plugins are so far:

- Encoding and decoding of planar waves and spherical waves using near field filters.
- Mirroring, Rotation (around azimuth and full 3D).
- Various Ugens for beamforming, returning mono as well as B-format signals.
- Various decoders in conjunction with Head-Related Impulse Responses (HRIRs) for binaural monitoring.

1.4.2 PseudoUgens

The second part of the library is available as the sclang extension HOA Quark. While the SC3plugins are designed for computational efficiency of the sound synthesis processes, The HOA Quark is conceived to unlock the flexibility of making sound in SC with respect to code reusability and the scaling of synthesis scripts. Each typical operation in Ambisonics (Encoding, Transforming, Decoding) is here provided as a PseudoUgen. Depending on the Ambisonics order provided as an argument, the PseudoUgen returns and instantiates the correct Ugen from the SC3plugins collection on the sound server. Since the Ambisonics order is an argument for the PseudoUgen, the number of channels in the B-format vary and so does the number of input arguments in the UGens. This is why the B-format is handled as a channel array. This makes the SC code flexible for experimentations with different orders depending on computational resources. All arguments of the PseudoUgen obey the channel expansion paradigm. This means that if any of the arguments is an array (or an array of arrays), the PseudoUgen returns an array (or an array of arrays) of Ugens.

Figure 1 shows the relation between the SC language side (PseudoUgens in light grey) and the SC3plugins (Ugens dark grey). If the Ambisonics order is set to 3 and passed as an argument to the PseudUgen, the corresponding Ugen with 16 channels is returned and a typical processing chain (Encoding Transforming Decoding, encircled in red) can be established. The main features of the design of the HOA library implementation on the language side are:

- B-format is handled as a channel array.
- All arguments obey the channel expansion paradigm.

This leads to the following advantage, when scripting HOA sound scenes in SC. Compared with graphical data flow programs like for instance Pure Data, changing the order means to reconnect all channels between objects interfacing with the B-format. In SC changing the Ambisonics order in a single global variable changes the order of the whole HOA processing chain.

1.4.3 Support directory

The third part of the library is a platform independent support directory for various HOA sound file recordings and convolutions kernels from HRIRs for the binaural rendering of HOA

sound scenes. The support directory approach is similar to the Atk implementation from which we adapted the corresponding class. The reason to keep these resources separate from the Quark directory is mostly due to the size of the included sound files. We provide some 4th order HOA sound files that have been recorded with the Eigenmike® with the support of Romain Dumoulin from CIRMMT. The HRIRs provided are either measured from a KU-100 dummy head [Bernschuetz, 2013] or computed from a 3D mesh scan of several people's face. The directions of the HRIRs follow a 50-node Lebedev grid, allowing an Ambisonic binaural rendering up to order 5 [Lecomte et al., 2016b].

2 Encoding

As the first step in an Ambisonics rendering chain, the library provides PseudoUgens for encoding into the B-format. One for the encoding of mono sound signals, one for microphone array prototypes and one for the commercially available Eigenmike® microphone array.

2.1 HOAEncoder

This PseudoUgen creates an HOA scene from mono inputs encoded as a (possibly moving) sound source in space. The source can be encoded 1) as a plane wave with azimuth and elevation (θ_p, δ_p) respectively 2) as a spherical wave with position $(r_s, \theta_s, \delta_s)$, where r_s is the distance to origin of the source. The spherical wave is encoded using near-field filters [Daniel, 2003]. In the current implementation, those filters are stabilized with near-field compensation filters. Thus, in this case, the radius of the loudspeaker layout r_{spk} used for decoding is needed. Note that if the spherical source radius is such as the source is focused inside the loudspeaker enclosure ($r_s \leq r_{spk}$), a "bass-boost" effect may occur with potential excessive loudspeaker gain. This effect increases as the source get closer to the origin [Daniel, 2003] [Lecomte and Gauthier, 2015].

```
HOAEncoder.ar(1,SinOsc.ar(f),a,e)
    ;
// returns
[OutputProxy,..,..,OutputProxy]

HOAEncoder.ar(1,
        SinOsc.ar([f1,f2]),
        a,e); // returns
[[OutputProxy,..,..,OutputProxy],
 [OutputProxy,..,..,OutputProxy]]

HOAEncoder.ar(1,
        SinOsc.ar([f1,f2]),
        a,e).sum;// returns
[OutputProxy,..,..,OutputProxy]
```

If an array of azimuth and elevation arguments, matching in size those of the source $SinOsc.ar([f1, f2])$, flexible and scalable code for multi source encoding can be created.

2.2 HOAEncLebedev06 / 26 / 50, HOAEncEigenmike

This collection of PseudoUgen offers at first the Discrete Spherical Fourier Transform (DSFT) for various spherical layout of rigid spherical microphone. In the current implementation the proposed geometries are 06- 26- or 50-node Lebedev grid [Lecomte et al., 2016b] and EigenMike grid[Elko et al., 2009]. The components of the DSFT are then filtered to take into account the diffraction by the rigid sphere and retrieve the Ambisonic components [Moreau et al., 2006] [Lecomte et al., 2015] The filters are applied by setting the filter flag to 1 as shown in the next code listing:

```
// Encode the signals from the
// Lebedev26 grid microphone
HOAEncLebedev26.loadRadialFilters
    (s);
{HOAEncLebedev26.ar(4, SoundIn.ar
    (0!26), filters: 1)}.play
```

3 Converting

In order to correctly reconstruct a sound field from the channels of the B-format, it is important to know about standard normalization methods for the spherical harmonic components, as well as channel ordering conventions. Two main channel ordering conventions exist: The original *Furse-Malham* (FuMa) [Malham, 1999] higher-order format, an extension of traditional first order B-format up to third order (16 channels). FuMa channel ordering comes with maxN normalization, which guarantee maximum amplitude of 1. The FuMa format has been widely used and is still in use but is increasingly replaced by the *Ambisonic Channel Number* (ACN) ordering [Nachbar et al., 2011]. ACN typically comes with (the full three-D normalisation) where all signals are orthonormal. SN3D (Semi-Normalized 3D) spherical harmonics. This normalization has the advantage that

none of the higher order signals exceeds the level of the first Ambisonic channel, W (ACN 0).

However, this normalization does not provide an orthonormal basis of spherical harmonics and this latter case is recommended for transformations which rely on the orthormality property of spherical harmonics. Therefore, the library uses internally the N3D (full 3D normalization) with ACN convention.

3.1 HOAConvert

The HOAConvert PseudoUgen accepts a B-format array as input and converts from and to ACN_N3D, ACN_SN3D, FuMa_MaxN. It is mostly meant to convert existing B-format recordings into ACN N3D for use within the library The other use case is to render B-format mixes to other conventions for other production contexts.

4 Transforming

In its current implementation, the HOA library provides 3 standard operations like rotation and mirroring to transform the B-format.

4.1 HOAAzimuthRotator

This PseudoUgen rotates the HOA scene around the z-axis, which is accomplished with a rotation matrix in x and y due to the symmetry in z of the spherical harmonics. For the matrix definition see [Kronlachner, 2014b]. In combination with horizontal head tracking, this transformation can stabilise horizontal auditory cues for left-right movements when the rendering is made over headphones in VR contexts.

4.2 HOAMirror

This PseudoUgen mirrors an HOA scene at the origin in the directions along the axes left-right (y), front-back (x), up-down(z). According to [Kronlachner, 2014b], this can be accomplished by changing a the sign of selected spherical harmonics.

4.3 HOARotatorXYZ

This PseudoUgen rotates a HOA scene around any given angle around x,y,z. The rotation matrix is computed in spherical harmonic domains using recurrence formulas [Ivanic and Ruedenberg, 1996].

5 Beamforming

5.1 HOAHCard2Mono

This PseudoUgen extract a mono signal from the HOA scene according to a beampatern. The channels from the B-format inputs are combined to produce a monophonic output as if a directional microphone was used to listen into a specific direction in the sound field. In the current implementation, the beampatern provided are regular hypercardioids up to order 5 see [Meyer and Elko, 2002]

5.2 HOAHCard2HOA

This PseudoUgen applies a hyper-cardioid beam-pattern to the HOA scene to enhance some directions and outputs a directional filtered HOA scene [Lecomte et al., 2016a]. The proposed beam-patterns are regular hypercardioids as described in [Meyer and Elko, 2002]. The selectivity of the directional filtering increases with the order of the beam-pattern. This transformation requires an order re-expansion such that the output HOA scene should be of the order of the input HOA scene plus the beampattern order [Lecomte et al., 2016a].

5.3 HOADirac2HOA

As in the previous section, this PseudoUgens performs a directional filtering on the HOA scene but this time the beam-pattern is a directional Dirac, that is to say a function which is zero everywhere except in the chosen direction. As a result the output HOA scene contains only the sound from the chosen direction. Thus, this tools helps to explore the HOA scene with a "laser beam". For more details see [Lecomte et al., 2016a].

6 Decoding

For the decoding of HOA signals two different ways of rendering the sound field are possible: First via headphones, or second through a setup of multiple loudspeakers.

For the headphone option the HOA signal is decoded to spherically distributed virtual speakers. For the best possible spatial resolution more speakers are needed then there are channels in the B-format. Each speaker signal is then convolved with HRTFs and the resulting left and right channels are summed respectively. For the distribution of the virtual speakers a regular distribution on the sphere is desirable, so that the decoding matrix is well behaved. This is why according to [Lecomte et al., 2015] and similar to the microphone array prototypes from above a Lebedev grid is chosen.

6.1 HOADecLebedev06 / 26 / 50

This collection of PseudoUgen decodes an Ambisonics signal up to 50 virtual speakers positioned as nodes on a Lebedev grid. The decoding on the 50-node Lebedev grid works up to order 5. This grids contains two several nested sub-grids which work up to lower order with less nodes[Lecomte et al., 2016b]. Therefore, the 6 first nodes are sufficient for first order and the 26 first nodes are sufficient up to the third order. If the HRTF filter flag is set to 1, the signals are convolved with the kernels and summed up to yield a left and right headphone speaker signal. Prior to this, the convolution kernels need to be loaded to the sound server as shown in the next code listing:

```
// load a HOA sound file
~file=Buffer.read(s,"hoa30.wav");
// prepare binaural filters
HOADecLebedev26.loadHrirFilters()
{HOADecLebedev26.ar(3,//order 3
   PlayBuf.ar(16,~file,1,loop:1),
   hrir_Filters:1)
}.play;
```

6.2 HOADec

For the case of decoding for speaker arrays [Heller et al., 2008] distinguish 3 cases:

1. regular polygons (square, octagon) and polyhedra (cube, octahedron)
2. semiregular arrays (non equidistant but opposing speakers, like in a shoebox)
3. general irregular arrays (e.g. ITU 5.1, 7.1 ... semispherical speaker domes)

For the cases 1 and 2, decoder matrices can be obtained by matrix inversion. If, depending on the positions of the speakers, the resulting decoder matrix has elements are of similar magnitudes, it is suitable for signal processing. For case 3, which are arguably the more realistic cases, a variety of state of the art techniques exists, see for instance [Zotter et al., 2012], [Zotter et al., 2010], and [Zotter and Frank, 2012]. An implememtation of these techniques exceeds the scope of this library. However, for the construction of decoders for specific irregular speaker arrays, we refer the user to the Ambisonic Decoder Toolbox by Aaron J. Heller[7]. This toolbox produces decoders as Faust files, which can be compiled online[8] as Ugens and in turn can then be integrated in the HOADec PseudoUgen class template.

7 The distance of sound sources

One novel aspect of the underlying Faust implmentation of Ambitools is the spherical encoding of sound sources using near field filters. For the correct reproduction of the HOA scene, the distance of the sound source and the radius of the reproducing (virtual) speaker array needs to be set. The correct near field filters are either applied by setting it in the encoding or in the decoding step.

```
// load the binaural filters
HOADecLebedev26.loadHrirFilters()
{ var src;
src=HOAEncoder.ar(
   3,//order
   PinkNoise.ar(0.1),//source
   az,//azimuth
   ele,//elevation
   plane_spherical:1,
   radius:2,
// set the speaker radius here
   speaker_radius:1)

HOADecLebedev26.ar(
   3,//order
   src,//source
// or set the speaker radius here
// speaker_radius:1,
   hrir_Filters:1)
}.play;
```

8 HOA and SynthDefs

The use of PseudoUgens leads to one important caveat when working with SynthDefs. The Ambisonics order is an argument pertaining to the PseudoUgen, it can hence not be an argument of a SynthDef. The reason is that at compile time the Ambisonics order would remain undefined and the PseudoUgen does hence not know which Ugen to return When working with SynthDefs code reusability can still be achieved as shown in the next code listing:

```
// set the max order:
~order = 5;
~order.do({|i| // iterate
 SynthDef( //create unique names
 "hoaSin"++(i+1).asString,
```

[7] https://bitbucket.org/ambidecodertoolbox/adt.git

[8] http://faust.grame.fr/onlinecompiler/

```
 {Out.ar(0,
        HOAEncoder.ar(
         i+1,//increase the order
         SinOsc.ar()))},
).add;
})
// play the Synths
Synth(\hoaSin1);
...
Synth(\hoaSin5);
```

9 HOA and Node Proxies

For the flexible creation of typical Ambisonics render chains, Node Proxies [Rohrhuber and de-Campo, 2011] provide an excellent tool in SC. Node Proxies autonomously handle audio busses and conveniently allow to crossfade between audio processes of a selected node, freeing silent process when the crossfade is completed. This allows to dynamically change sources in the encoding, transforming and decoding step in the rendering chain. The following code example shows a flexible scenario with changing seamlessly from an XYZ rotation to beamforming.

```
~o=3;
~chn=(~order+1).pow(2);
// load hoa sound file:
~bf=Buffer.read(s,"file.wav");

// b-format file player:
~player=NodeProxy(s,\audio,~chn);
~player.source=
          {PlayBuf.ar(~chn,~bf)};

// Node for xyz rotation:
~trans=NodeProxy(s,\audio,~chn);
~trans.source=
       {var in;in=\in.ar(0!16)
        HOATransRotateXYZ.ar(
                ~o,in,
                yaw,pitch,roll)};

// rotate the scene
~trans.set(\yaw, angle);

// decoding,
~dec=NodeProxy(s,\audio,~chn);
~dec.source=
       {var in;in=\in.ar(0!16)
       HOADec.ar(~o,in,)};

// chain the proxies together
~player <>> ~trans <>> ~dec;

// change rotation to beamforming

~trans.source=
       {var in;in=\in.ar(0!16)
        HOABeamDirac2Hoa.ar(
                ~o,in,
                az,ele)};

// direct the beam
~trans.set(\az, angle);
```

10 Conclusions

We have presented a HOA library for SC. The design of which resulted in great flexibility and makes it a valuable addition to experiment with HOA in various contexts. Due to the meta approach through Faust, future additions to the library are feasible and we look forward to experiment with it in the context of VR and video gaming platforms but also for the creation of sound material for electro acoustic compositions. We believe that the flexibility and live coding capacity of SC is particularly useful in the context of HOA, where repeated listening is essential to asses the perceptually complex mutual interdependence of temporal and spatial sound characteristics.

11 Acknowledgements

This work has been supported through the research-creation funding program of the Fonds de Recherche du Québec - Société et Culture (FRQSC).

References

J. Anderson and J. Parmenter. 2012. 3D sound with the Ambisonic Toolkit. In *Presented at the Audio Engineering Society 25th UK Conference / 4th International Symposium on Ambisonics and Spherical Acoustics*, York.

Jeffrey S. Bamford. 1995. *An Analysis of Ambisonic Sound Systems of First and Second Order*. Ph.D. thesis, University of Waterloo, Waterloo.

Natasha Barrett. 2010. Ambisonics and acousmatic space: a composer's framework for investigating spatial ontology. In *Proceedings of the Sixth Electroacoustic Music Studies Network Conference Shanghai*, 21-24 June.

Natasha Barrett. 2016. Interactive Spatial Sonification of Multidimensional Data for Composition and Auditory Display. *Computer Music Journal*, 40(2):47–69.

B. Bernschuetz. 2013. A spherical far field hrir/hrtf compilation of the neumann ku 100. In *in Proceedings of the 40th Italian (AIA) Annual Conference on Acoustics and the 39th German Annual Conference on Acoustics (DAGA)*, page 29.

J. Daniel. 2000. *Représentation de Champs Acoustiques, Application à la Transmission et à la Reproduction de Scènes Sonores Complexes dans un Contexte Multimédia, Ph.D. Thesis*. Ph.D. thesis, University of Paris 6, Paris, France.

Jerome Daniel. 2003. Spatial sound encoding including near field effect: Introducing distance coding filters and a viable, new ambisonic format. In *In Audio Engineering Society Conference: 23rd International Conference: Signal Processing in Audio Recording and Reproduction*, pages 1–15, Helsingor. AES.

G. Elko, R. A. Kubli, and J. Meyer. 2009. Audio system based on at least second-order eigenbeams.

Michael A. Gerzon. 1973. Periphony: With-height sound reproduction. *Journal of the Audio Engineering Society*, 21(1):2–10.

Aaron J. Heller, Richard Lee, and Eric M. Benjamin. 2008. Is My Decoder Ambisonic? In *the 125th Convention of the Audio Engineering Society*, San Francisco, oct. 1-5.

J. Ivanic and K. Ruedenberg. 1996. Rotation matrices for real spherical harmonics. Direct determination by recursion. *J. Phys. Chem.*, 100(15):6342–6347.

M. Kronlachner. 2013. Ambisonics plug-in suite for production and performance usage. In *LAC*, Graz, Austria, May.

M. Kronlachner. 2014a. Plug-in Suite for Mastering the Production and Playback in Surround Sound and Ambisonics. In *AES Student Design Competition (Gold Award)*, Berlin, April.

M Kronlachner. 2014b. Spatial transformations for the alteration of ambisonic recordings. *Graz University Of Technology, Austria.*

P. Lecomte and P.-A. Gauthier. 2015. Real-Time 3D Ambisonics using Faust, Processing, Pure Data, And OSC. In *In 15th International Conference on Digital Audio Effects (DAFx-15)*, Trondheim, Norway.

P. Lecomte, P.-A. Gauthier, C. Langrenne, A. Garcia, and A. Berry. 2015. On the use of a Lebedev grid for Ambisonics. In *in Audio Engineering Society Convention 139*.

P. Lecomte, P.-A. Gauthier, C. Langrenne, A. Berry, and A. Garcia. 2016a. Filtrage directionnel dans un scène sonore 3D par une utilisation conjointe de Beamforming et d'Ambisonie d'ordre elevés. *in CFA / VISHNO 2016*, pages 169–175.

Pierre Lecomte, Philippe-Aubert Gauthier, Christophe Langrenne, Alain Berry, and Alexandre Garcia. 2016b. A fifty-node lebedev grid and its applications to ambisonics. *Journal of the Audio Engineering Society*, 64(11):868–881.

D.G. Malham. 1999. Higher order ambisonic systems for the spatialisation of sound. In *ICMC99*, pages 1–4, Beijing. ICMC.

J. McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

J. Meyer and G. Elko. 2002. A highly scalable spherical microphone array based on an orthonormal decomposition of the soundfield. *in IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2:1781–1784.

S. Moreau, J. Daniel, and S. Bertet. 2006. 3d sound field recording with higher order ambisonics-objective measurements and validation of spherical microphone. In *in Audio Engineering Society Convention 120*, pages 1 – 24.

T. Musil, J. Zmoelnig, M. Noisternig, A. Sontacchi, and R. Hoeldrich. 2003. AMBISONIC 3D Beschallungssystem 5.Ordnung fuer PD. Report 15, Institute for Elektronic Music and Acoustics.

Christian Nachbar, Franz Zotter, Etienne Deleflie, and Alois Sontacchi. 2011. AMBIX - a suggested ambisonics format. In *Ambisonics Symposium*, Lexington, KY, June 2-3.

Julian Rohrhuber and Alberto deCampo, 2011. *The SuperCollider Book*, chapter Just

in Time Programming (ch 7) . MIT Press, MA: Cambridge.

Jan C. Schacher. 2010. Seven Years of ICST Ambisonics Tools for MAXMsp - a Brief Report. In *Proc. of the 2nd International Symposium on Ambisonics and Spherical Acoustics*, Paris, France, May 6 - 7.

Andrea Valle. 2016. *Introduction to SuperCollider*. Logos Publishing House.

S. Wilson, D. Cottle, and N. Collins, editors. 2011. *The SuperCollider Book*. MIT Press, MA: Cambridge.

F. Zotter and M. Frank. 2012. All-Round Ambisonic Panning and Decoding. *J. Audio Eng Soc*, 60(10):807–820, Nov.

F. Zotter, M. Frank, and A. Sontacchi. 2010. The virtual t-design ambisonics-rig using vbap. In *presented at the 1st EAA-EuoRegio 2010 Congress on Sound and Vibration*, pages 1 – 4, Ljubljana, Slovenia.

F. Zotter, H. Pomberger, and M. Noisternig. 2012. Energy-Preserving Ambisonic Decoding. *Acta Acoustica united with Acoustica*, 98:37 – 47.

STatic (LLVM) Object Analysis Tool: Stoat

Mark McCurry
Georgia Tech
United States of America
mark.d.mccurry@gmail.com

Abstract

Stoat is a tool which identifies realtime safety hazards. The primary use is to analyze programs which need to perform hard realtime operations in a portion of a mixed codebase. Stoat traverses the callgraph of a program to identify which functions can be called from a root set of functions which are expected to be realtime. If any unsafe function which could block for an unacceptable amount of time is found in the set of functions called by a realtime function, then an error is emitted to indicate where the improper behavior can be found and what backtrace is responsible for its call.

Keywords

Realtime safety, static analysis, LLVM

1 Motivation

When using low latency audio tools an all too common problem encountered by users is audio dropout caused by an excessive run time of the audio generation or processing routine. This artifact is also commonly known as an xrun. Xruns can be generated when there's simply too much to calculate during the allocated time, but it can also be easily generated by any function which takes an unreasonable amount of real time to execute[1]. The latter category of functions typically include operations involving dynamic memory, inter-process communication, file IO, and threading locks.

For low latency audio to reliably work, a frame of audio and midi data must be processed within a short fixed time window. Audio callbacks are then known as functions bound by a real time constraint, or *realtime* for short. A large portion of code can have it's total execution time bounded when the size of data is known in advance. Some code however cannot be simply bounded. As a simple example, consider prompting the user for synthesis parameters and waiting for a response. The user could enter a response quickly or they could never provide a response. The class of functions which aren't bounded by the real time constraint, that a realtime program operates with, are known as *non-realtime*.

To avoid xruns, realtime programs should be composed of functions with reasonable realtime bounds, and thus non-realtime functions are unsafe for a reliable program. Typically realtime system programmers acknowledge the timing constraint and design systems with this limitation in mind. Simple tests may be used to identify the typical execution times as well as variance, though it's easy for bugs to creep in. In particular, the C or C++ open source projects in Linux audio frequently have architectural issues making realtime use unreliable[2].

Maintaining a large codebase in C or C++ can make it very difficult to both know what a given function can end up calling or when a particular function could be called. Typically problems start with a mixed realtime/non-realtime system, such as UI and DSP sections of code; the segregation within one codebase may not be at all clear in implementation. This is further complicated by the opaqueness of some C++ techniques, such as virtual overloading, operator overloading, multiple inheritance, and implicit conversions. Overall these complications make manual verification of large scale realtime programs difficult.

Stoat offers a solution to identifying realtime hazards through an easy-to-use static analysis approach. Static analysis makes it possible to identify when functions claimed to be realtime can call unsafe non-realtime functions even when complex C or C++ call graphs are involved. The approach offered by Stoat makes it easy to identify these programming errors which can be used to greatly improve the reliability of

[1] as opposed to cpu time

[2] see appendix A

low latency tools.

1.1 Prior Art

Stoat isn't the first tool to address the problem of identifying these realtime safety hazards. Several years prior to the creation of Stoat, Arnout Engelen created jack-interposer which is a runtime realtime safety checker [Engelen, 2012]. Jack-interposer works by causing a program to abort if within the JACK process callback any known unsafe non-realtime function is called. The functions which jack-interposer identifies as unsafe include, IO functions (`vprintf()`, and `vfprintf()`), polling functions (`select()`, `poll()`), interprocess communication (`wait()`), dynamic memory functions (`malloc()`, `realloc()`, `free()`), threading functions (`pthread_mutex_lock()`, `pthread_join()`), and `sleep()`.

As a runtime analysis tool jack-interposer requires the program to be executed to identify errors and each error is reported as it's encountered. Individual errors are presented by a message without a backtrace or by halting the program and allowing a developer to use a debugger. Jack-interposer has the same issue as other runtime tools compared to static analysis. Namely, exhaustive testing requires the user or testing script to run the program through all states which involve different logic. Doing so is a difficult, error prone, and tedious task. Additionally, jack-interposer was designed to only be used with JACK clients, while Stoat works with any program, JACK based or not.

Stoat is based of an earlier attempt at creating another more general static analysis tool. The predecessor project, Static Function Property Verifier, or SFPV, attempted to address more general problem of tracking described properties through a programs feasible call graph [McCurry, 2014]. SFPV used the Clang compiler's API to record precise source level information [Lattner, 2008]. Unfortunately the Clang API was subject to rapid breaking changes, slow to compile, and vastly underdocumented, so SFPV was rewritten to create Stoat. Stoat in comparison uses a limited subset of the LLVM API without interfacing directly with Clang.

2 Examples

Both runtime and static analysis tools, including jack-interposer and Stoat, attempt to address the same overall problem. Both aim at detecting when a function which can be executed in a realtime thread can call a function which may block for an unacceptable amount of time. In C, an example of this is is shown in listing 1. `root_fn()` can call `malloc()` through two intermediate functions, `unannotated_fn()` and `unsafe_fn()`. When Stoat is provided with an out-of-source annotation on `root_fn()` it can then use the call graph to deduce that an unsafe function can be called.

Listing 1: Example C Program

```
void root_fn(void) {
    unannotated_fn();
}
void unannotated_fn(void) {
    unsafe_fn();
}
void unsafe_fn(void) {
    malloc(10);
}
```

For a C program many call graphs are relatively simple and no complex type information is needed. C++ call graphs however make extensive use of operator overloading, templates, and class based inheritance. Listing 2 shows an example of the `root_fn()` calling a method which may or may not be safe based upon which implementation of `method()` is called. As the class hierarchy is available to Stoat, the root function can be conservatively marked as unsafe as `method()` would call `malloc()` if Obj was an instance of the Unsafe class. Depending upon the workload of a particular program, this data dependency might be satisfied very rarely, so a purely runtime based approach may not identify the error.

Listing 2: Example C++ Program

```
void root_fn(Obj *o) {
    o->method();
}
class Unsafe: public Obj {
    virtual void method(void) {
        malloc(10);
    }
};
```

3 Stoat Implementation

Stoat consists of several components. First, there is a compiler shim to dump LLVM based metadata though LLVM IR files. Second, there is a series of LLVM compiler passes to extract

inline annotations and call graph information. Last, there is a ruby frontend to perform deductions on the extracted call graph and to produce diagnostic messages and diagrams.

Stoat uses information present in LLVM bitcode to capture the program's call structure. Generating bitcode for individual files can be difficult to integrate with complex software projects' build systems. A similar issue was presented by Clang's official static analysis tools [Kremenek, 2008]. Their solution was to have a stand-in which replaces the normal C/C++ compiler[3]. Stoat offers two compiler proxy binaries, `stoat-compile` and `stoat-compile++`, which provide a way to simplify generating LLVM bitcode similar to Clang's scan-build toolchain. For an autotool based project analysing source code is as simple as running `CC=stoat-compile CXX=stoat-compile++ ./configure && make` and then running `stoat -r ..`

For each LLVM bitcode file Stoat runs four custom LLVM passes. These passes respectively identify: the function calls, or call graph within the program; the C++ virtual methods associated in each class; the C++ class hierarchy; and in-source realtime safety annotations.

First the call graph is constructed. Within the LLVM IR the Call and Invoke operations call another function and they contain metadata about what function is being called. For C functions this is relatively simple. Consider the IR associated with `void foo(){bar()}` in listing 3.

Listing 3: LLVM IR For C Call

```
define void @foo() #0 {
entry:
  call void @bar()
  ret void
}
```

For C++, extracting the call graph is somewhat more complex due to the introduction of virtual methods. Virtual methods are a structured version of function pointers calls and they can be identified by the two-step process to obtain the function pointer. First, a class instance is converted to the virtual function table, or vtable. Then, the method's ID is used to extract the method from the vtable and the resulting function pointer is called. The LLVM IR for a virtual call is shown in listing 5 and it corresponds to the source shown in listing 4.

Listing 4: C++ Call

```
void foo(void) {
    Baz *baz;
    baz->bar();
}
```

Listing 5: LLVM IR For C++ Call

```
define void @_Z3foov() #0 {
entry:
  %baz = alloca %class.Baz*, align 4
  %0 = load %class.Baz** %baz,
       align 4
  %1 = bitcast %class.Baz* %0 to
       void (%class.Baz*)***
  %vtable =
    load void (%class.Baz*)*** %1
  %vfn = getelementptr inbounds
       void (%class.Baz*)** %vtable,
       i64 0
  %2 = load void (%class.Baz*)** %vfn
  call void %2(%class.Baz* %0)
  ret void
}
```

Next the vtable calls need to be mapped back to real functions. Vtables are stored as a global symbols and can be identified by the "_ZTV" prefix used in normal C++ symbol mangling procedures. The class hierarchy can be reconstructed by identifying chained constructors from class to class.

With the information presented by the normal call graph and the C++ virtual methods an augmented call graph can be constructed. First, any vtable methods are assumed to call any method implementation of the base class or any child class. Then, suppression file entries are used to remove edges from the augmented call graph to avoid false errors typically seen in error handling.

The last LLVM pass looks for in-source safety annotations in the form of `__attribute__((annotate("realtime")))` and `__attribute__((annotate("non-realtime")))`. These annotations can be added to the end of a function declaration to add metadata to the function within the C or C++ source. The annotations are augmented with out-of-source annotations in the form of whitelist and blacklist files.

Once the augmented call graph is constructed and a subset of the functions in the program

[3] `http://clang-analyzer.llvm.org/scan-build.html`

are annotated, a series of deductions can be made. Any function which is called, but never implemented is assumed to be non-realtime if not specified otherwise. Any function which is unannotated and called by a realtime function is assumed to be realtime. Any function which is realtime or assumed realtime that calls a non-realtime function produces an error with an associated deduction chain.

The errors can be presented in either a textual or graphical form. The current format includes the function that calls the unsafe function along with the deduced path. An example of an error flagged by Stoat is dynamic memory use within jalv when it is in a debug mode.

```
Error #514:
serd_stack_new
##The Deduction Chain:
 - serd_writer_new : Deduced Realtime
 - sratom_to_turtle : Deduced Realtime
 - jack_process_cb : Realtime (Annotation)
##The Contradiction Reasons:
 - malloc : NonRealtime (Blacklist)
```

Alternatively, figure 1 shows a partial view of a graphical representation of call graph nodes involved in errors. When dealing with a legacy codebase the graphical representation tends to be preferable as it visually shows which routines contain the most errors, and which errors are the most common. Additionally, for C++ codebases who's error involve long template expansions the graphical representation shortens the displayed names to result in a still large, but more manageable view on the software's architecture.

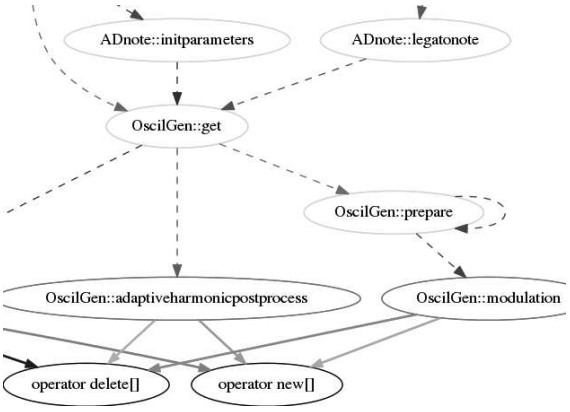


Figure 1: Partial output from Stoat applied to ZynAddSubFX 2.5.0[4]

[4] http://fundamental-code.com/2.5.0-realtime-issues.png

3.1 Limitations

Stoat offers a number of improvements over prior art, though Stoat does have its limitations. Namely, Stoat doesn't track data dependencies[5] on realtime safety. This task is one where runtime analysis tools, such as jack-interposer, can identify errors which Stoat isn't able to find or avoid false positives.

Two primary data dependent issues which produce misleading results include the use of unsafe function pointers and the use of unsafe error handling code. A short example of the former would be:

Listing 6: Function pointer call

```
void function(void (*fn)(void)) {
    fn()
}
```

If and only if `function()` is only passed realtime functions, then it is a realtime safe function, but the data passed into the function isn't analysed by Stoat, so function pointer calls are typically overlooked.

Debug and error handling code is a common source of false positives and the example error from jalv shows one such example. In listing 7, `function()` would be marked unsafe. The unsafe function should never be called in practical use and a runtime checker would not flag this case. A similar class of issues can occur if a function has different realtime safe behavior depending upon a flag passed to the function as may be the case with codebases which do not have separate functions for realtime and non-realtime tasks.

Listing 7: Example error handling

```
void function(void) {
    if(fatal_error)
        call_unsafe_function();
}
```

3.2 Discussion

Stoat and it's predecessor, SFPV, were originally created as a tool to assist with finding issues within the ZynAddSubFX synthesizer[6] and bringing it into compliance with realtime safety issues. While minor issues still exist, several users have reported improved reliability at lower latencies compared to earlier versions. Stoat has

[5] this includes any conditional code execution based upon constant or non-constant data

[6] http://zynaddsubfx.sf.net/

since been used as a verification tool in: librtosc[7], carla[8], ingen[9], and jalv[10]. Ideally it will be used on more projects within Linux Audio to identify realtime hazards in the future. The use of Stoat or jack-interposer would assist in correcting the poor user experience and possibly a negative reputation for stability that realtime hazards have created in a variety of realtime projects.

When Stoat doesn't understand regular structure within a program it is relatively easy to extend. ZynAddSubFX uses roughly 500 callbacks through librtosc. Stoat has already been extended to automatically annotate these callbacks As mentioned in the limitations, function pointers are difficult to reliably track with static analysis, librtosc callbacks however have per callback metadata which can be used to associate a statically known function pointer with information which can be used to identify which ones are expected to be executed in the realtime environment. This process was tested in ZynAddSubFX and used to resolve several bugs.

4 Conclusion

Stoat offers a new method to inspect existing software projects and direct attention towards code which may be responsible for realtime hazards. Addressing these realtime hazards can improve the experience within a variety of Linux audio applications and plugins. Through the use of automated tools such as Stoat realtime hazards can be identified and corrected quickly. Additionally, the static analysis approach of Stoat complements the prior art of runtime analysis that projects like jack-interposer provide. Stoat is available at `https://github.com/fundamental/stoat` under the GPLv3 license.

References

Arnout Engelen. 2012. Jack interposer. `https://github.com/raboof/jack_interposer`.

Ted Kremenek. 2008. Finding software bugs with the clang static analyzer. *Apple Inc.*

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar.

Chris Lattner. 2008. Llvm and clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2.

Mark McCurry. 2014. Static function property verification: sfpv. `https://github.com/fundamental/sfpv`.

A Brief Survey of Realtime Safety

In order to validate the claim that "projects in Linux audio frequently have architectural issues making realtime use unreliable", a survey was conducted on a sampling of Linux synthesizers. Each synthesizer as presented by `http://www.linuxsynths.com/` was given a brief manual code review (typically $<$ 15 minutes per project) looking for common realtime safety violations. If source code was not available or could not be located for a code review then the project was excluded. Projects marked with a '*' have had an in depth code review prior to the writing of this paper. The results shown in table 1 show that 18 of 40 projects (or 45%) have some easy to identify realtime safety issue.

Outside of LMMS and ZynAddSubFX the realtime hazards within each project has not received additional verification. Based upon experience working with projects not included in this list, additional realtime hazards are expected to be observed when tools like jack-interposer or Stoat are applied.

[7] `http://github.com/fundamental/rtosc`

[8] `http://kxstudio.linuxaudio.org/Applications:Carla`

[9] `https://drobilla.net/software/ingen`

[10] `http://drobilla.net/software/jalv`

Table 1: Linux Synthesizer Realtime Safety Observations

Software Name	Observed Status	Notes
6PM	likely unsafe	appears to launch threads within rt-thread
Add64	likely unsafe	blocking gui communication in rt-thread
Alsa Modular Synth	likely unsafe	unsafe data mutex
amSynth	likely unsafe	unsafe memory allocation in rt-thread
Borderlands	likely unsafe	unsafe locks/memory allocation in rt-thread
Bristol	likely safe	appears safe
Calf tools	likely safe	appears safe
Cellular Automaton Synth	likely safe	appears safe
Dexed	likely unsafe	unsafe memory allocations in rt-thread
DX-10	likely safe	appears safe
Helm	likely unsafe	memory allocation in rt-thread/addProcessor()
Hexter	likely safe	appears safe
JX-10	likely safe	appears safe
LB-302	likely unsafe	see LMMS
LMMS*	unsafe	unsafe locks in rt-thread, unsafe memory allocation in rt-thread, creation of threads in rt-thread, blocking communication to user interface in rt-thread, etc
Monstro	likely unsafe	see LMMS
Mr. Alias 2	likely safe	appears safe
Mx44	likely safe	appears safe
Nekobee	likely safe	appears safe
Newtonator	likely safe	appears safe
OBXD	likely safe	appears safe
Organic	likely unsafe	see LMMS
Oxe FM Synth	likely safe	appears safe
Peggy2000	likely safe	appears safe
Petri-Foo	likely safe	appears safe
Phasex	likely safe	appears safe
Samplev1	likely unsafe	possible memory allocation in rt-thread
SetBFree	likely safe	appears safe
Sineshaper	likely safe	appears safe
Sorcer	likely safe	appears safe
Synthv1	likely unsafe	possible memory allocation in rt-thread
Triceratops	likely safe	appears safe
Triple Oscillator	likely unsafe	see LMMS
Tunefish 4	likely safe	appears safe
Vex	likely safe	appears safe
Watsyn	likely unsafe	see LMMS
WhySynth	likely safe	appears safe
Wolpertinger	likely unsafe	unsafe memory allocation in setParameter()
Xsynth	likely unsafe	variety of mutexes used in the rt-thread
ZynAddSubFX*	unsafe	unsafe memory allocation in oscillator wavetable generation (the total number of realtime hazards was greatly decreased with the use of Stoat)

AVE Absurdum

Winfried Ritsch
Intitute for Electronic Music and Acoustics
Inffeldgasse 10/3
8010 Graz
Austria
ritsch@iem.at

Abstract

Auditory Virtual Environments (AVEs) are used to simulate audio environments in real spaces. As room in room reverberation system (RRR) they augment the acoustics in spaces, e.g. in concert halls and music theaters. Why not utilize them for theater music as acoustic stage design and therefore as a playable instrument ?

Even more, tune them to extreme configurations, so that absurd acoustic situations can be realized, absurd in the sense of not normal or possible in real physics and using distortions in time, space, frequency and signal domains.

This paper discusses the conceptualization and design of an artistic research project using AVEs for a theatre and some of the new aspects of these ideas are discussed. For the multi-space theater production "the Trial" from Franz Kafka for actors, singer, choir and stage design at the Art University in Graz networked AVEs have been realized, utilizing Ambisonics systems in concert halls and movable acoustics instruments on open spaces.

Keywords

Auditory Virtual Environment, acoustic, stage design, computer music, Ambisonics

1 Introduction

An auditory virtual environment (AVE) is a virtual environment (VE) that focuses on the auditory domain only. It sees itself independent from other modalities like vision. Nevertheless an AVE could also be combined with the visual domain. Depending on the application, the user may be either a passive receiver or be able to interact with the environment. Three different approaches for implementations of AVEs are listed in Blauert's book "Communication Acoustics"[Novo, 2005] from Novo:

1. Authentic reproduction of real existing environments.

   The virtual room should evoke in the listener the same percepts that would have been evoked by the corresponding real environment. He should have same spatial impression moving through and perceive his own movement inside the environment as well as the movements of sound sources.

2. Reproduction of plausible auditory events

   This approach tries to evoke auditory events which the listener perceives as having occurred in a real environment. Here only those features are implemented which are needed for a specific simulation situation.

3. Creation of non-authentic plausible auditory events or environments.

   The virtual room doesn't evoke percepts in the listener which are related to a real acoustic environment, evoking auditory events where no authenticity or plausibility restraints are imposed, targeting pure virtual environments like computer games.

Figure 1: physical adjustable acoustic for Beat Furer's music theater FAMA

1.1 the setting

For the music theater production based on the novel "Der Process" (engl. "The Trial"), written by Franz Kafka from 1914 to 1915 for actors, singer, choir and stage design, an experimental theater music composition should be done:

> The hopeless search of the main character "Josef K" for the reason of his arrest is the grandiose template for a inter-institutional project: a play with perception, a sensual journey through existential abysses and absurdities of the bureaucracy with students and lecturers of the institute stage design / acting / singing, song, oratorio / electronic music and acoustics, choir of the Kunstuniversität Graz.

The production was roughly spread in three parallel played scenes at three subsequent places with an collective intro at the foyer and collective finale at the concert hall:

Figure 2: Ligeti hall stage design with big moveable blocks

Gyorgi Ligeti hall $400m^2$ concert hall constructed for virtual acoustics.

theatre in the Palais (TIP) $200m^2$ theatre hall constructed traditional acoustics.

courtyard between these houses with a Peepshow construction as stage design.

After the "Intro" in the foyer, the audience was divided into 3 groups, each group attending a 30 minute performance in one of the places and have been guided from one place to the other within the intermissions.

1.2 the compositional approach

Additional constraints for the musical acoustic composition has been made to concentrate enforce the ideas of the piece:

One main idea was to use signal processing for the experimental theater music composition for the play "Der Process" on live signals only and do not use any pre-produced sound material. All material should be based on live recorded sound signals using different microphones at the places; to construct virtual soundscapes for different audio reproduction systems and virtual acoustics as a main instrument within these sceneries.

Another constraint that all places should be treated as networked AVEs.

Figure 3: TIP stage with big hole in the middle, traditional chariot-and-pole-system

1.3 the experimental approach

The a artistic research question have been: can a non-plausible non-authentic AVE, applied as a complex music instrument for theater music, produce a varying plausible acoustic sceneries.

As an extension, the AVE should use distortions in space, time, spectrum and signal domain and should therewith produce an distorted AVE, which is still perceived as acoustics, an absurd acoustics, Therefore the production was titled "AVE-Absurdum". With this concept the category of AVEs should be extended to a fourth category of AVE, let us name it "absurd AVE", which is non-authentic but plausible in an absurd way of reception and in respect to the visual domain. This AVE does evoke percepts in the listener which are related to a real acoustic environment and the live sound produced. by the actors and other real sound sources.

As a common audio 3D sound representation Ambisonics should be used, also to allow simulations of these AVEs at development phase prior the first rehearsals.

Ambisonics was chosen, not only because of already implemented Ambisonics system at the concert hall, which has been the very well tested in previous productions like "Pure Ambisonics", but for streaming the acoustical impact of one room to another. Therefore spatial recordings and mixes as 3D audio streams was used, so spatial information of the audio signals can be

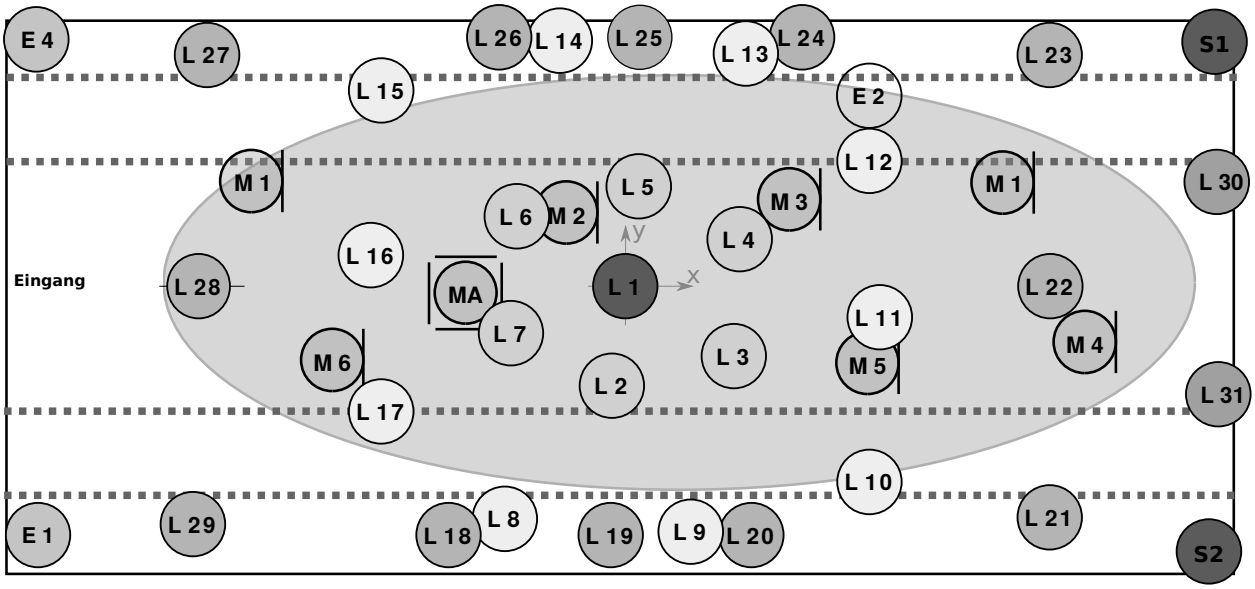


Figure 4: loudspeaker in Ligeti hall: Ambisonics L, subwoofer S, extra E, and microphones M. The heights of the speaker increases to the middle from 2m to 8m.

used in other spaces. Also Ambisonics can be used in directional speaker system, used for the move-able acoustics in between the spaces.

As a stage design using processes as backdrop, like an additional layer on the theater music itself, as an big invisible ensemble of signal processing algorithms, the generated sound environment represent a complex machine. So from another perspective these AVEs can be seen as part of the "theater machine" in the meaning of Gilles Deleuze concept of machines[Raunig, 2004; Deleuze and Guattari, 1977].

2 AVE Absurdi

For the three spaces, three different implementation of Ambisonics has been designed:

Ligeti-Saal Ambisoncis 4th order with 32 ambisonics speakers and 2 subwoofer, 7 directional microphones hanging from the ceiling, 2 headsets for main actors, 2 pickups on the floor and 2 on the blocks of the stage design.

Theater im Palais (TIP) ambisonics 2D ring on the ceiling, subwoofer, 2 Mikrofones for Reverberation, 2 microphones for enhancement of special plaxes, 1 headset for trigger only.

courtyard between two houses Movable spherical directional loudspeaker driven by embedded linux computers connected to multichannel amplifier and a directional microphone, powered by batteries and played by actors.

The order used in the different places is normally defined by the amplification system, but here we work also with Ambisonics streams and virtual microphones detecting different signals, the maximum is limited by the encoding system.

The Ambisonics system at the TIP, since the stage was designed as proscenium stage with the audience at one wall, was not satisfying and canceled by the director there, who wanted a purely stereo frontal speaker system. Anyway streams from other places has been used for the sound environment.

In the following the space in the Ligeti hall and the movable acoustic will be discussed.

2.1 AVE in Ligeti concert hall

Varying playable acoustics has been developed as an acoustician for Beat Furer's music theater FAMA. As a stage design a real room in room with rotate-able wall elements, one side absorbers one side reflectors, for 200 listeners was build like a huge machine. One restrain was to use no electro-acoustic element. Unlike this physical adjustable acoustics, electro-acoustic AVEs should be implemented.

Since the already installed Constellation Acoustic System from Meyer Sound[Sound, 2010] with circa eighty of small speakers and about twenty microphones in 5 meter heights as a closed system was not in any way flexible

enough to fulfill the requirements of the idea of an "AVE absurdum".

The speaker used for the 3D Ambisonics system are shown in figure4. 31 active Klingt&Freytag speakers have been used, where the first 29 of them are mounted on pantographs, which can adjust the heights and direction of each speaker individually as presets. The $L22$ speaker has to be adjusted higher, because of the blocks from the stage designe and has been second by two others $L30$ and $L31$ on stands on the floor to lower the acoustical horizon. Additional two sub-woofer left and right in the front corners for enhancing the Ambisonics sound and used for special subsonic effects have been placed.

The Hemisphere was slightly expanded as ellipse and stretched to the front to increase the "sweet spot". With this number of speaker a 5th order Ambisonics system could be realized. But since all the obstacles and additional movable blocks using a 3th order Ambisonics had smother results on moving sources, increased spatial continuity and avoided to spatial aliasing errors over the room which resulted in a bigger "sweet spot".

As an decoder the standalone decoder of the AmbiX plugin suite[Kronlachner, 2013] was used, for which Matthias Frank from the IEM calculated an suitable Allrad-decoder[Zotter and Frank, 2012] . For preproduction of effects and the development of the AVE in a studio or over headphones the binaural decoder with the special set of impulse-responses, measured in the Ligeti-Hall, was provided.

The decoder was fed with Ambisonics signals from applications within the Linux computer and over a MADI-Audio Interface input routed through the Lawo Mixing console from other computers, using "jackd". Therefore three computer musicians were able play in parallel using the same AVE system over one central decoder feeding the speaker. The sub-woofer management has been done in the Mixer, using the Ambisonics signals and an additional a subsonic effect channel for special effects.

The AVE-Absurdum has been implemented with Puredata[Puckette, 1996] running patches on different computers connected over MADI Audio Interfaces. The main computer implemented an Ambisonics Mixer with the room in room reverberation system (RRR), derived from the CUBEmixer[Ritsch et al., 2008] development of previous years and the "acre" Pd extension library with the therefore developed Ambisonics Toolbox module for Pd: "acre-amb"[Ritsch, 2016] using "iem-ambi" external library.

"acre-amb" is a collection of high level Pd abstraction, to implement Ambisonics functionality for Ambisonics mixing and processing of multichannel signals and controls to be used in compositions and effects. Also a goal was to easily integrate Ambisonics encoder, decoder with calibration and speaker distribution, providing also connection and processing targeting fast prototyping of new Ambisonics algorithms.

Figure 5: consoles (from left): Lawo Mixing Desk, Controller for effects, Computer Console with Pd Patch and AmbiX Decoder, Controller for time machine and memory player, Spectral Ambisoncis, with notebook as controller

2.1.1 Room in Room Reverberation for acoustics

The core of the RRR system is a multichannel reverberation system with 6 Inputs and 12 early reflections and 6 late reverb channels to be spatialized in the 3D space of the AVE.

It was not possible within this production time to mike a choir with 110 singer, especially because they move sometimes erratically in the room. Adjusting to limited rehearsal time, we had to find a solution where the actors and choir can play with different absurd acoustics, utilizing the conductor and movement-director to explore and fixate this effects within their re-

hearsals, starting within the very first rehearsals and eliminating the need to track the movement of the choir and actors for the composition.

The solution was chose was enabling "active zones", areas under microphones, where choir, actors and audience are enhanced and encounter different acoustics feedback. These effects can be switch or cross-faded for each scene. Also actors can be in an different acoustical spaces parallel to the choir or audience:

Therefore the RRR was driven directly by microphones in 3-4m heights, enabling the different playable acoustics from small garage reverb, long tunnels with scatter echoes to big halls, even further to echoes like from surrounding buildings, mountains, allowing $> 200ms$ early reflections. This allows to build none-plausible acoustics, like increasing energy on reflections and/or different acoustical rooms in one room: an absurd AVE.

Additional to limiting the output, especially decreasing feedbacks of the reverb, each microphone got an feedback suppression EQ for the 3 most resonant frequencies of the room.

2.1.2 Distortion in Space

Within the RRR spatializing early reflections from only one direction or placing all late reverb to the other site, the acoustical space can be shaped: eg. imaging a big room in one direction and a wall in the other. This can be done dynamically, with a sudden appearance of a late reverb from one site. Since a changing acoustics can be perceived better than a static one, since change of size of rooms, normally does not happen, are drawing more attention to listeners than static ones.

A overlap of one acoustic space over another seems to be more unnaturally, but since most singers and actors use reverb on stages, the audience is used to this effect.

2.1.3 Distortion in signal

Another effect was inserting processing of the microphone signal path. Within this project three types has been tested:

spectral Resonances and filters

dynamics Limiter, Compressor and Expander.

shaping Waveshaping: tubes, metal strings, noise (cut) and also string simulator, metal plate.

Spectral filter have same effects as different spectral properties of reflection material and is therefore only spectacular, if really applied strongly. Changing this dynamically changes the whole "sound color" of the scenes.

Dynamics have been mostly applied on singer and actors. Nowadays audience is widely familiarized with these effects for solo performers, but doing it extreme, which means silent passages become loud and loud voices decrease the volume is a strong effect, but is something singers do not like. Therefore it was used to increase the struggle of the actor against the environment, here acoustics. The drawback are that it was really hard to control without feedback at silent phases and can be perceived as an mistake in the performance very easily.

A really strong effect is the distortion especially of the early reflections in the reverb: A tube shape make the room a warm sound and using nonlinear shapes introduces noise. Additional metal distortion like ring-modulator with 2 inputs signals within a parallel dialog can produce really scary rooms. As drawback the feedback is again an issue, so mostly limiters has to be used.

Figure 6: Choir surrounding the audience and actors behind a transparent curtain

2.2 Distortion in Time

We called it "artistic time-stretching", which is an ongoing research project done by Manuel Planton on the IEM, where time-stretching should be applied in live situations. time-stretching and realtime is clearly a contradiction, since stretching leads in the past, which means the signal is not within the realtime constraints.

There has been three different phases in perception experienced:

- time-stretched signal within the early reflections delay $< 80ms$
- echos up to $300ms$

- a playback of a detached recording of the signal $> 1sec - inf$

Playing within this phases is the artistic approach, where voices has to be slowed down first and then speed up again. Doing this, the sound is amplified first, than scattered and becomes then a dialog with the live signal, which is a very thrilling effect, even more it seems like a replay like a "deja-vu" experience. It turned out to be a thrilling effect, which actors liked to play with. It was used on solo pieces on actors and repetition phases of the choir. Introducing feedback loops of the signal to the time-stretcher optionally combining with pitch shifts, expands the possibilities of this effect even further. So it was used solo for some scenes and the effect signal spatialized independently from the position of the source.

For the implementation a own Pd external was written, using the rubberband library and additional an overlap and add (OLA) algorithm. Considering the limited rehearsal time and the big parameter space, the time-stretcher has to be played interactively, observing the actors by an additional electronic musician. As an own instrument the Pd-patch was run on an separate computer with own controllers, mixed in via a Ambisoncis bus signal.

2.3 Spectrum processing

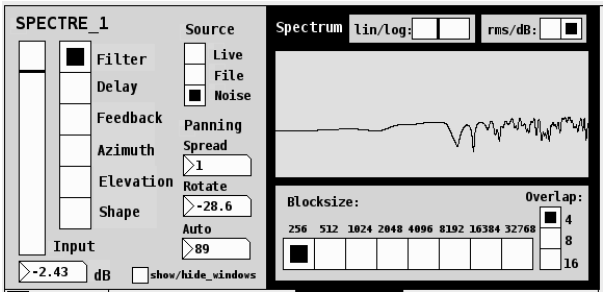


Figure 7: Spetcre Pd GUI

Another special effect has been the spectral distortion over space, we named it "spectre" developed and played by Christoph Ressi. With an special patch using small FFTs/IFFTs, the spectral information was split into several channels, which have been spatialized in the 3D space. High frequencies could be played from another direction than low ones and spread over the hemisphere. Drawing tables controls the movements and spreading. Also a feedback loop to within the effect was introduced, so it can do a kind of spectral freezing. This development was used to audio-process the choir input signals and distribute them in the space. The choir chant tends to be a acoustical environment with itself, especially if the choir is surrounding the audience. On transient signals with fast glissandi elements like shouting, clapping and stamping the effect is audible like a rapid movement of the sound in the room. On long notes especially accords, the rooms begin to feel like stretched and softened walls, because it is hard to hear any dimension, since reflections are masked by direct sound. The effect was used on a one scene as solo acoustic performance and frozen during conversions.

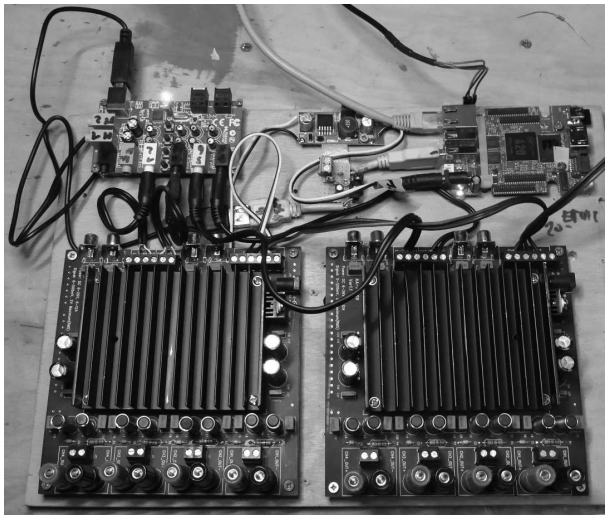


Figure 8: development prototype for linux player with 8x100W for 2 tetrahedron-speaker: decoupled USB 2/8channel, dc/dc, olimex-A20, 2x class-D amplifier to be powered by 12V battery

3 Movable Virtual Acoustics

Figure 9: agent with microphone playing 2 directional speaker

For virtual acoustics in the courtyard, where no speakers and fixed installation was available, a special concept of moving acoustics was conceptualized. Operated by electronic musicians as "agents" in the play, the movable acoustics instrument with AVE-function and stream rendering features, were integrated in the play by the director:

Using spherical loudspeaker arrays allows us to beam sound to many directions utilizing Ambisonics signals. With walls of buildings and rooms around in the courtyard, reflection can be induced, which triggers a kind of surrounding sound. The simplest of the spherical geometries the tetrahedron, which has been used before in a performance enhancing the room acoustics of a church[Robert lepenik, 2014] . The Tetrahedron loudspeaker have 4 wideband speaker mounted on each plane and can be placed on an portable stand. The electronics consists of a $4x100W$ class-D amplifier, supplied by an 12V12Ah rechargeable battery, driven by an "Olimex ARM-A20" embedded computer with a hacked multichannel USB audio interface, a phantom power microphone-preamp, speaker cable and an microphone over XLR cable. The agents can carry the whole electronics in their bags and hold the microphone and speaker.

A directional microphone has been chosen for interaction with the surrounding, so the agents can focus and play with the sound input of the environment using a kind of AVE-patch. Receiving the Ambisonics streams from the other spaces, using an addtional virtual microphone, they can select signals from other performances to be combined in the audio scene.

The whole signal processing was done by a Pd patch including different effects like feedback with reverb, pitch shifting, delays etc. to realize a movable AVE. This work was named "AVE-tetrahedron" and experimental explored before on the campus.

To play this instrument small controllers mounted to the arms have been used.

4 Ambisonics network

Streaming Ambisonics was developed for the COMEDIA project[Ritsch, 2010]. Using this technique, the 3D acoustic signal of an room can be delivered to other spaces, broadcasting eg. the 25 channel Ambisonics signal from Ligeti hall to others. The receiver can choose the AVE and place virtual microphones inside, using controllable Ambisonics decoder.

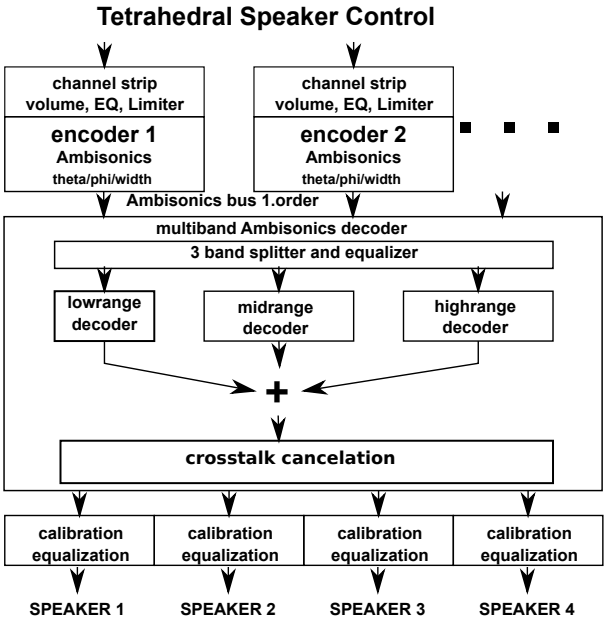


Figure 10: tetraeder drive for AVEs

For streaming scripts for "gstreamer" has been written as transmitter and receiver connected via "jackd" to Pd. This allows a adjustable and acceptable latency with a sufficient buffering for different situations.

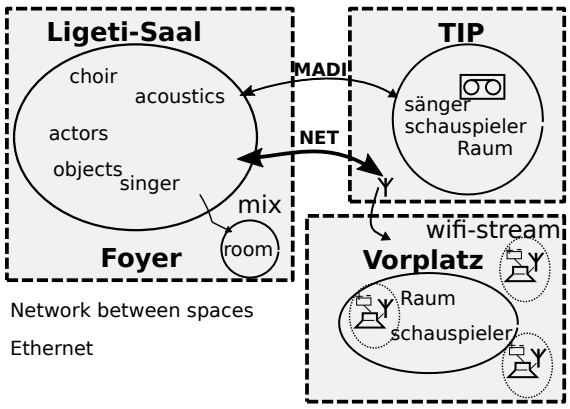


Figure 11: Network of AVEs

5 Conclusions

The whole production was a big success from the reaction of the audience and the participants. The AVE concept was accepted, after some persuasiveness, explaining the concept to all participants. Because of the limited time, there was not much space to criticize and overthrow the concept, so even it was very tight we tried to stay as close to the concept as possible or drop it, like within the TIP.

To focus on transformations and not so much

on sound-effects was a very wise decision, since effects brings to much additional parallel content and are not so invasive to support the idea of Kafka's absurd world.

The concept of AVE absurdum as a playable instrument has been proven in the Ligeti-hall. The movable acoustics instruments works nicely in small areas. Simple effects like distortion in time work surprisingly well. Imprinting other acoustics of one room in the other also works fine in most situations, but since listeners are used to it in media perception, are not spectacular.

6 Acknowledgements

Thanks go to all the participants of the play, more than 80 for support and being open minded to accept new concepts. Also to the university with greatly supported this production beyond the normal effort for workshops and lectures. Especially thanks to the Sabine Pinsker from the stage design department for organizing the whole project and the general director Horner for his encouragements to enter new path for theater music. For the development, Atelier Algorythmics which provides the Tetrahedron-player with embedded linux systems and other hardware.

Figure 12: Winfried Ritsch conduction pantographs in ligeti hall

References

G. Deleuze and F. Guattari, 1977. *Programmatische Bilanz für Wunschmaschinen*, page 489. Deleuze, Gilles: Kapitalismus und Schizophrenie. Suhrkamp. Die Neu-Erfindung des Maschinenbegriffs refereniert von Raunig Gerald.

M. Kronlachner. 2013. Plug-in suite for mastering the production and playback in surround sound and ambisonics. In *Linux Audio Conference 2013*, University of Music and Performing Arts, Graz, May. Institute for Electronic Music and Acoustics, Linux Audio Group.

Pedro Novo, 2005. *Auditory Virtual Environment*, chapter 11, pages 277++. Signals and Communication Technology. Springer.

M. Puckette. 1996. Pure Data. In *Proceedings, International Computer Music Conference.*, pages 224–227, San Francisco.

G. Raunig. 2004. Einige Fragmente über Maschinen. *Grundrisse 17.* Die Neu-Erfindung des Maschinenbegriffs.

W. Ritsch, J. Zmölnig, and T. Musil. 2008. he cubemixer a performance-, mixing- and masteringtool. In *Proceedings of the LAC 2008*, Cologne, Germany. ZKM, Linux Audio Users Group.

Winfried Ritsch. 2010. Nettrike. CO-ME-DI-A,EACEA Culture Project on Network Performance in Music.

Winfried Ritsch. 2016. Ambisonics toolbox for puredata. internet. Puredata abstraction library.

Winfried Ritsch Robert lepenik. 2014. Ex Machina Dei. commissioned work by Musikprotokoll Graz 2014 for robot organ player, robot piano player and 6 Tetrahedron-speaker at Church St. Andrei.

Meyer Sound. 2010. Constellationacoustic system. internet.

F. Zotter and M. Frank. 2012. All-round ambisonic panning and decoding. *Journal of the Audio Engineering Society. audio, acoustics, applications*, 60(10):807–820, 11.

Multi-user posture and gesture classification for 'subject-in-the-loop' applications

Giso GRIMM[1,2] and **Joanna LUBERADZKA**[2] and **Volker HOHMANN**[1,2]
[1] HörTech gGmbH, Marie-Curie-Str. 2, D-26129 Oldenburg, Germany
[2] Research group "digital hearing devices",
Department of Medical Physics and Acoustics,
Medizinische Physik und Cluster of Excellence Hearing4all
g.grimm@uni-oldenburg.de

Abstract

This study describes a posture classification method for a marker-free depth camera. The method consists of an object identification procedure, feature extraction, and a naïve Bayesian classification approach with a supervised training. Point clouds obtained from the depth camera are split into objects. For each object a set of features is extracted. A method of feature pre-processing is proposed and compared against a statistical orthogonalisation method. Using a manually labelled training data set, the probability distributions for the Bayesian classification are obtained. As a result of the classification, the most likely gesture is assigned to each object in real time. Classification performance was tested on a separate data set and reached about 80%.

Three different applications are described: Automatic estimation of user postures to estimate the influence of hearing devices on user behaviour in communication situations, the control of an interactive audio-visual art installation, and interactive light control on a dance-floor setup with multiple dancers. Classification performance in these applications was measured and discussed.

Keywords

gesture classification, behaviour analysis, hearing devices, interactive art, subject-in-the-loop

1 Introduction

With the development of assistive technologies, there is a growing need for robust automatic identification of human postures and gestures. Gesture recognition is used for improving the human-machine communication, e.g., in hand gesture-based device control [Freeman and Weissman, 1997; Richarz et al., 2008]. Another use case is the classification of gestures and postures that describe the subject's behaviour or provide information on the current state of the subject [Busso et al., 2008; Melo et al., 2015]. Automatic recognition of various postures has potential applications in research areas where the test subject's behaviour is analysed. As an example from the hearing research, in typical communication situations, leaning forward while listening is associated with a high listening effort, whereas sitting more relaxed indicates a lower effort [Paluch et al., 2015]. Manual labelling of user behaviour in similar tasks is usually time consuming and is not sufficient in case of the 'subject-in-the-loop' experiments, where the measurement is controlled by the responses of the test subject. Interaction between the subject reactions and the measurement procedure is desired when aiming at more realistic experimental conditions, but can also provide additional performance measures from the experimental feedback loop. 'Subject-in-the-loop' experiments require a real-time classification of gestures and postures. This differs from conventional behavioural experiments where a post-hoc analysis of the data is possible. Besides research applications, machine control functions based on natural postures are possible, e.g., a hearing device could increase the noise reduction efficiency when the user's change in posture indicates a higher listening effort. Such an application would require a body-worn motion tracking sensor, e.g., accelerometer and gyroscope embedded into a hearing device.

Gesture and posture recognition tools are also applied in music and arts [Ciglar, 2008; Donnarumma, 2011]. Typically, an artist controls music generation and modification tools with gestures, resulting in a mixture of dance and music performance. The classification system proposed here is designed to be useful for music and art applications with multiple users. One application is an audio-visual installation, where the postures of the audience influence the sound and vision. Another potential real-time application is the live interactive light and music control system for a dance-floor.

Real-time analysis of postures and gestures from depth images is commonly achieved via skeleton modelling [Shotton et al., 2013]. In the

applications of this study, such a high level posture model is not required, because only a limited number of posture and gesture classes need to be discriminated. Furthermore, these applications require a computationally fast method of classification. For this, a naïve Bayesian classifier as used in this study. This simple classification method can deal with a low-dimensional data and requires only a limited amount of training data [Ashari et al., 2013; Gupte et al., 2014]. For discriminating only a small set of classes, low level features describing the coarse point cloud distributions and the velocities of certain point cloud areas can be used. However, to fulfil the implicit statistical assumptions of the naïve Bayesian classifier, and to identify the most relevant application-specific feature sets, a pre-processing of features may be beneficial.

In this paper, methods of point cloud processing (sections 2.1 to 2.3), feature pre-processing (section 2.4) and classification (section 2.5) are described. In section 2.6, the training conditions in three different applications – posture classification for hearing research, multi-user control of an audio-visual art installation, and individualised light control for a dance-floor – are explained. Classification performance in the different applications with the proposed pre-processing methods are given in section 3 and discussed in section 4.

2 Methods and apparatus

For this study, one or more subjects were tracked using a Microsoft kinect depth camera. Although the final applications of this gesture and posture classification approach significantly differ, they have all the same structure, which is depicted in Fig. 1. First, the camera data was filtered for a more robust point cloud estimation and background removal. In a second step, the point cloud was split into multiple objects. For each object, a set of features was extracted, and based on this feature set, the posture or gesture of each object was classified. The point cloud processing and classification was implemented in the openMHA hearing device signal processing platform [Herzke et al., 2017; Grimm et al., 2009; Grimm et al., 2006]. Training and data analysis was implemented in Matlab. These processing blocks are described below.

2.1 Noise reduction and background removal

The Microsoft kinect depth camera is an optical sensor which measures the depth through the parallax of an infrared laser grid. It provides a depth value d for each pixel position (k, l). Invalid values (e.g., occlusion, absorption) get the depth value $d = 0$. In this study, the depth was scanned with a frame rate of 10 Hz.

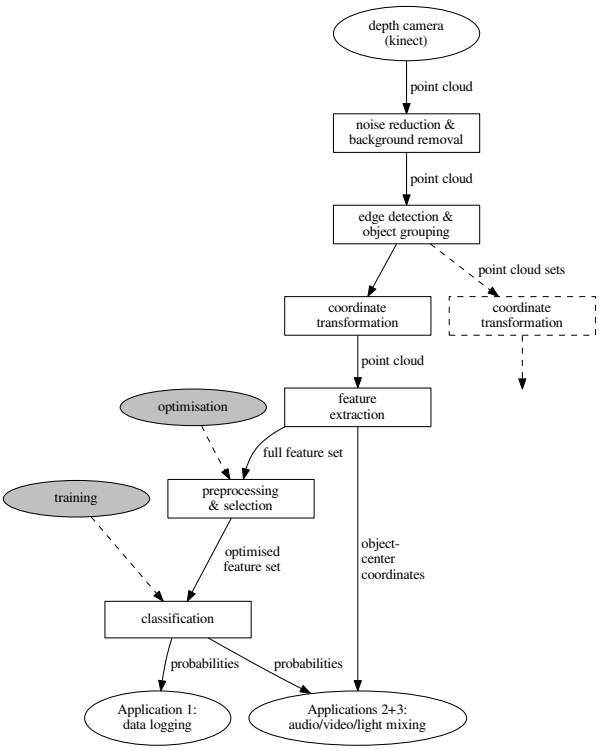


Figure 1: Structure of the proposed gesture and posture classification framework.

Absorbing objects and objects with a very uneven surface, e.g., curly hair, typically result in invalid data points for many frames. To increase robustness in such conditions, invalid values were replaced by the last available valid value, if a value was measured within the last second.

For the classification of objects it was essential to separate them from the background. Therefore, in an initial phase without subjects in the sensing area, the background depth was measured, and all depth values close to the background were removed. After this step, only those data points remain which were assumed to belong to a relevant object.

2.2 Edge detection and object grouping

An assumption for the object grouping was that all objects have a spatial separation, i.e., either the depth was not continuous or the objects were separated by background pixels. This allows to use a simple first-order gradient edge detection algorithm using the depth data. A

pixel (k, l) was an object boundary if the depth gradient was above a threshold d_t:

$$(d_{k,l} - d_{k+1,l})^2 + (d_{k,l} - d_{k,l+1})^2 > d_t^2 \quad (1)$$

To construct objects, a generic flood fill algorithm [Torbert, 2012] was applied to identify all pixels within a closed boundary. These pixels were marked on an object map with their object number. The set of pixels (k, l) belonging to one object was $\mathbb{P}$, which was then used for further object-specific processing if the number of elements of $\mathbb{P}$, p, has a sufficient size.

Coordinate transformation. At this stage the objects were defined by a set of pixels with a certain depth from the camera. For a robust feature extraction, these have to be transformed into a world coordinate system. In the first step, pixel data was transformed into a camera coordinate system $\mathbf{x_c} = (x_c, y_c, d)^T$, with the horizontal distance from the camera axis x_c, the vertical distance y_c and the distance from the camera d. These coordinates were linearly approximated by

$$(x_c, y_c) = \alpha(k - k_0, l - l_0)d_{k,l}. \quad (2)$$

(k_0, l_0) was the central pixel of the camera. World-coordinates $\mathbf{x} = (x, y, z)^T$ (x distance along camera axis, y to the left, z upwards) were calculated by rotation and translation of the camera-coordinates. These point clouds $\mathbb{P}$ were the basis of further feature extraction of each object. The object centre was $\overline{\mathbf{x}} = \langle \mathbf{x} \rangle_{\mathbb{P}}$, i.e., the mean of all points in the point cloud $\mathbb{P}$.

Temporal alignment of objects. At this point, the order of detected objects depends on the first object pixel position in the camera plane. This is not a robust measure, thus the object order may change from frame to frame. However, to allow for analysis of time related features, the objects were re-ordered based on a similarity measure of distance d and the object size ratio r between consecutive frames. The distance between the objects o and q at the time indices t and $t - 1$ was defined as $d_{o,q}(t) = ||\overline{\mathbf{x}}_o(t) - \overline{\mathbf{x}}_q(t-1)||$. The size ratio was $r_{o,q}(t) = e^{|\ln(p_o(t)) - \ln(p_q(t-1))|}$. Then the coherence matrix $\mathbf{C}(t)$ between two objects was defined by its elements

$$c_{o,q}(t) = r_{o,q}(t) e^{-\gamma d_{o,q}(t)} \quad (3)$$

with a weighting coefficient $\gamma = 10$. For a re-sorting of objects, the columns of $\mathbf{C}$ were ordered to maximise the elements on the diagonal, corresponding to a maximal temporal coherence.

2.3 Feature extraction

A list of all extracted features and their labels can be found in Table 1. Features corresponding to the object in the global coordinate system as well as features describing size and distribution of the point cloud $\mathbb{P}$ relative to its centre were extracted. The object rotation was estimated from the ratio of depth to width. Two methods of calculating point cloud distribution were tested: In the first method, weighted averages across $\mathbb{P}$ were calculated. For example, the average left bottom position was estimated by using a weight w with

$$w = \begin{cases} (z - z_{max})^2 + (y - \langle y \rangle)^2 & y > \langle y \rangle \\ 0 & \text{otherwise} \end{cases}. \quad (4)$$

To account for dynamic properties, which may be important for gesture classification, in addition to the above mentioned point cloud distribution related features, the absolute value of their temporal derivatives was calculated.

These features define the time-variant feature vector $\mathbf{f}(t)$ which was used as an input of feature pre-processing.

2.4 Feature pre-processing and optimization

Before the actual classification, the features $\mathbf{f}$ were pre-processed with a method $\mathcal{P}$ to maximise the classification performance,

$$\hat{\mathbf{f}}(t) = \mathcal{P}\{\mathbf{f}(t)\}. \quad (5)$$

The pre-processing method $\mathcal{P}$ was a combination of temporal low-pass filtering with the time constant τ, selection of optimal feature set $\mathbb{F}$, and PCA.

The pre-processing method $\mathcal{P}$ was iteratively optimised. In each iteration cycle m, the training of the classifier was done based on the pre-processed training data set, whereas the classification performance to which this pre-processing method $\mathcal{P}_m$ led, was computed using the test data set, pre-processed in the same way as the training data. The pre-processing method $\mathcal{P}_m$, which gave the best classification performance was chosen as the final pre-processing method for classification.

Orthogonalisation. The naïve Bayesian classifier used in the current work assumes

name	label
global coordinates:	
number of pixels p	`n.n`
mean position $\mathbf{x}$	`n.x`, `n.y`, `n.z`
median position	`n.xmed`, `n.ymed`, `n.zmed`
rotation	`n.rot`
local coordinates:	
size	`n.sx`, `n.sy`, `n.sz`
thickness	`n.r`
segment positions	`o.lx` `o.lz`, `o.rx`, `o.rz`, `o.lby`, `o.rby`
segment thickness	`n.r1`, `n.r2`, `n.r3`
z-quantiles	`n.z25`, `n.z50`, `n.z75`
velocities:	
object velocity	`o.vz`
size changes	`o.vsy`, `o.vsz`, `o.vsx`
vertical segment velocities	`o.vlz`, `o.vlz`, `n.vz1`, `n.vz2`, `n.vz3`
horizontal segment velocities	`n.vxy1`, `n.vxy2`, `n.vxy3`
angular velocity	`n.vrot`

Table 1: List of features per identified object. The features were calculated by two different implementations, as indicated by the prefix `o` and `n`.

conditional independence of all the features. This means, that adding features which are highly correlated with other features might degrade the performance of the classification. Therefore, an orthogonalisation of the feature space is required. In this study, two orthogonalisation methods were tested.

A principle component analysis (PCA) is a generic orthogonalisation method. A transformation matrix is estimated, which is then applied to the feature vector before classification. To avoid a dominance of large-scale features, all features were scaled to ensure a standard deviation of one before calculating the PCA coefficients.

As an alternative method, a feature selection method is proposed. First, the individual classification performance of each feature from the full feature set was computed, by training the classifier only on the given feature. Classification performance was then measured on the test data set. The features were then sorted by their individual classification performance. Starting with the best performing feature, features from the sorted feature set were added successively to the optimal feature set. This procedure was repeated until no further increase of classification performance was observed. Although this feature set is optimised for classification performance, it does not guarantee that it is orthogonal in a statistical sense.

Low-pass filtering. Low-pass filtering of the features across time results in a smaller feature variance within a class and thus a better class separation, which as a consequence leads to a better classification performance. On the other hand, with long time constants the classifier is not able to track transitions between the classes. The time constant τ can be adapted to the expected frequency of class transitions in the test data, or to increase classification performance and stability. The optimal τ was determined by a one-dimensional grid search, with and without PCA and feature selection.

2.5 Classification

To accomplish the gesture classification task, a Gaussian Naïve Bayesian Classifier was implemented. This approach assumes a set of conditionally independent and normally distributed features. Each class c_h, where $h = 1, ..., N_c$ is the class index, and N_c is the total number of classes, represented a different gesture or posture. $\hat{\mathbf{f}}$ is a data vector with extracted features $\hat{f}_j$, where $j = 1, ..., N_{\hat{f}}$ is the feature index, and $N_{\hat{f}}$ is the number of features. Considering the independence assumption, Bayes formula can be written in the following form:

$$p(c_h|\hat{\mathbf{f}}) = \frac{p(\hat{\mathbf{f}}|c_h)p(c_h)}{p(\hat{\mathbf{f}})} = \frac{\prod_{j=1}^{N_{\hat{f}}} p(\hat{f}_j|c_h)p(c_h)}{p(\hat{\mathbf{f}})}, \quad (6)$$

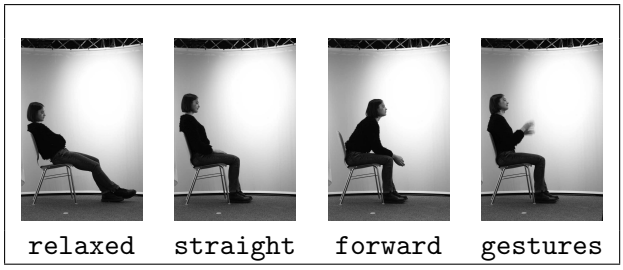


Figure 2: Labels of the project 1 (research application).

which means that the overall class conditional probability $p(\hat{\mathbf{f}}|c_h)$ can be computed by multiplying the conditional probabilities for each feature $p(\hat{f}_j|c_h)$.

Since the elements of $\hat{\mathbf{f}}$ are assumed to be normally distributed $p(\hat{f}_j|c_h) = N(\mu_{jh}, \sigma_{jh})$, the probability density function (PDF) of a feature j for a class h can be modelled by the mean μ_{jh} and standard deviation σ_{jh}. These parameters were estimated from the manually labelled training data. Also a flat prior probability was assumed, $p(c_h) = 1/N_c$.

In the current study, probabilities $p(c_h|\hat{\mathbf{f}}(t))$ were calculated for each object in each time frame. For estimating the classification performance, the confusion matrix was computed as an average posterior probability for each class. The classification performance was the geometric average across the diagonal of the confusion matrix.

2.6 Classification tasks and class labels.

The training was executed for three different classification tasks, corresponding to the use cases in hearing research, art and entertainment. In each training data set, data from nine test subjects (age from 23 to 44 years) was used. The recording of each gesture or posture lasted approximately 90 seconds for each subject.

In the first task ('project 1'), four classes with typical communication states were defined with an indention to track the subject's behaviour during the hearing experiment. There were three sitting postures with labels `relaxed`, `straight`, `forward`, and a class corresponding to gesticulation while talking, `gestures`.

The second task ('project 2') consisted of eight classes, either body movements or postures, which were used for controlling and mixing of sound and video art installation concerning different manifestations of water. The 'water' classes had the following labels: labels `lake`, `rain`, `ice`, `waves`, `ocean`, `boil`, `steam` and `thunder`.

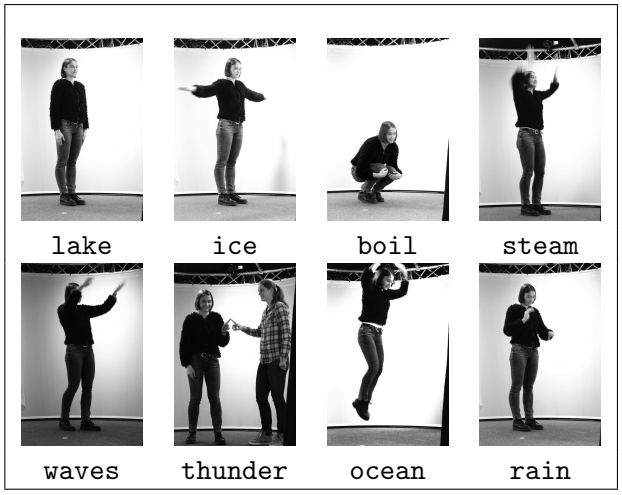


Figure 3: Labels of the project 2 (audio-visual art installation).

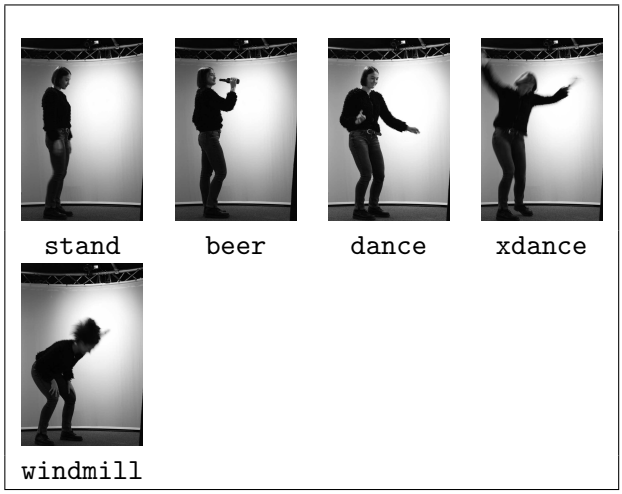


Figure 4: Labels of the project 3 (dance-floor light control).

The third classification task ('project 3') contained five classes related to typical actions on a dance-floor at parties, to control the light according to individual behaviour of the dancers. The labels `stand` (standing or slowly walking), `beer` (drinking from a bottle), `dance` (dancing), `xdance` (excessive dancing) and `windmill` (rotating head) were used.

Images from Figures 2, 3 and 4 present the selection of classes for each project.

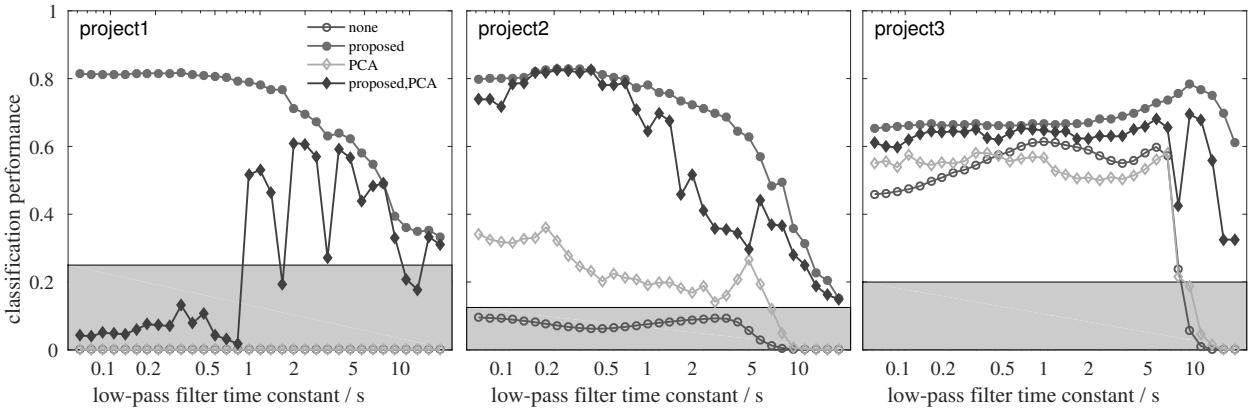


Figure 5: Classification performance as a function of feature low-pass filter time constant τ for the orthogonalisation methods 'none', 'proposed', 'PCA' and the combination of 'proposed' with 'PCA', in the three tested projects. The shaded area denotes the chance level.

3 Results

3.1 Influence of feature pre-processing on classification performance

Time constant optimisation. Figure 5 shows the classification performance as a function of feature low-pass filter time constant τ in all tested projects. The optimal value for project 1 was 297 ms, resulting in a classification performance of 81.8%. In project 2, the optimal time constant was 250 ms with a performance of 82.9%. In the third project, the time constant τ was 8 s, leading to a classification performance of 78.4%.

In all cases, the feature orthogonalisation improved the performance. The maximum performance was always achieved with the proposed method for feature selection. Using the PCA alone increased the performance only marginally. Both methods in combination do not give better performance results than the proposed method alone.

Feature selection. Figure 6 shows the performance of individual features in the three different projects. In project 1, the proposed feature selection method reduced the dimensionality to 12 features. The performance of individual features ranged from 19.1% to 44.4%. 42.7% of the selected features were velocity-related features. In project 2, a set of 17 features was found to be optimal; individual performance ranged from 13% to 28.4%. 35.3% of the features were velocity-related. In the last project, only 9 features were sufficient for optimal classification, with individual performance between 26.3% and 48.2%. In this case, 66.7% of the features were related to motion.

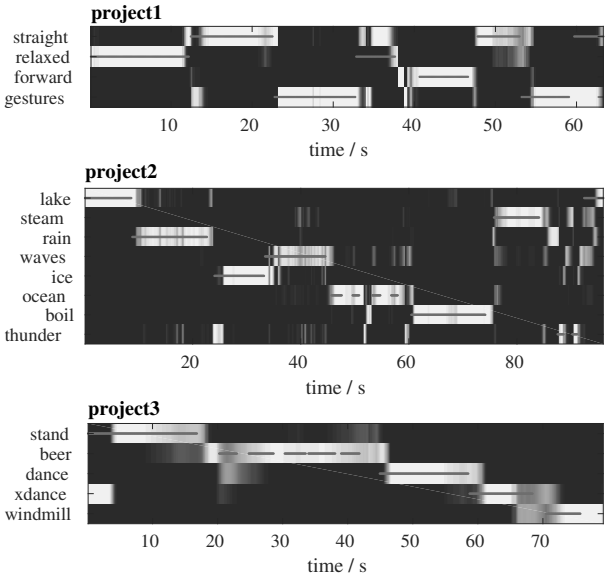


Figure 7: Posterior class probability as a function of time for the test data, with the labelled classes indicated by red lines.

3.2 Classification performance with optimised parameter sets

Figure 7 shows the posterior probabilities as a function of time. It can be noticed that classification errors mostly occurred at class transitions. With the longer time constants of project 3, a lag of classification at each transition can be seen.

The confusion matrix is shown in Figure 8. In project 1, the least confusions were achieved for the `forward` class. Typical confusions were between the classes `straight` and `relaxed`, as well as between `gestures` and `straight`. In project 2, the least confusions were found for

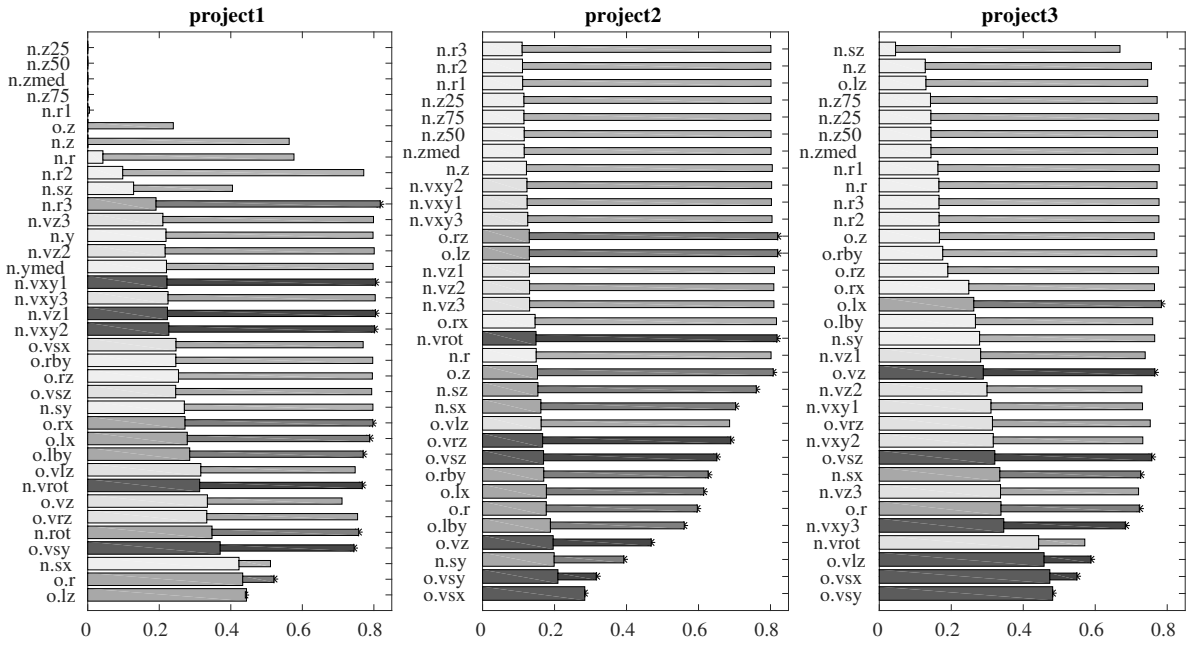


Figure 6: Classification performance of single features (thick bars) and the cumulative classification performance (thin bars). Stars denote the features which were selected by the proposed orthogonalisation method. Blue colours denote velocity-related features.

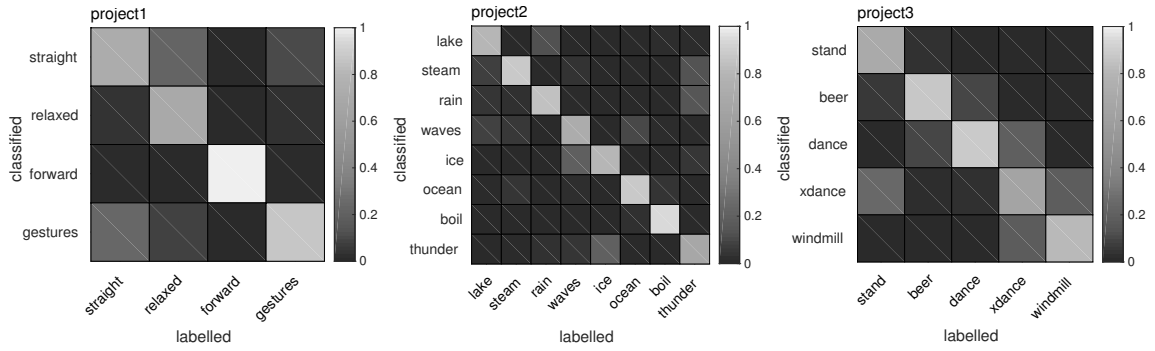


Figure 8: Confusion matrices (average posterior probability for each class in the test data set) of the three different projects.

the classes `boil`, `ocean` and `steam`. The class `thunder` was often confused with the classes `rain` or `steam`. In the third project, more confusions can be noticed. Most confusions can be found for the classes `xdance` and `windmill`, and between the classes `beer` and `dance`.

4 Discussion

The results show that a robust classification of gestures and postures based on a low-level feature set is possible, even with a naïve Bayesian classifier and a small feature space. The preprocessing of features indicated that a orthogonalisation of the feature space in a statistical sense is less important than the selection of features with an optimal class separation. However, it is still unclear whether another order of combination of orthogonalisation methods or a dimension-reduction in the PCA would further increase performance.

In this study, only number of low-level features was used. A high-level feature space, e.g., skeleton modelling, might be beneficial for robust classification of complex and high-level gestures. On the other hand, using such low-level features does not require any model assumptions. An intermediate solution could be an advanced segmentation of the point cloud.

5 Conclusions

In this study it was shown that even with a small and low-level point-cloud based feature space a robust classification of gestures and postures is possible. The tested applications covered research, art and entertainment, with four to eight classes in each application. The proposed method of feature-space optimisation by selecting a subset of the features was shown to result in better classification performance than a statistical orthogonalisation method. Low-pass filtering of features with application-specific time constants allowed for a trade-off between stable classification and fast reactions at class transitions. Classification performance of approximately 80% was achieved in all applications. Automatic classification of gestures and postures for hearing research applications with the 'subject-in-the-loop', i.e., with a behavioural feedback loop, seems feasible.

6 Acknowledgements

This study was funded by DFG research grant 1732 "Individualisierte Hörakustik" and by "klangpol Netzwerk Neue Musik Nordwest". We would like to thank all test subjects who participated in the training phase.

References

Ahmad Ashari, Iman Paryudi, and A Min Tjoa. 2013. Performance comparison between naïve bayes, decision tree and k-nearest neighbor in searching alternative design in an energy simulation tool. *Int. J. Adv. Comput. Sci. Appl*, 4(11).

Carlos Busso, Murtaza Bulut, Chi-Chun Lee, Abe Kazemzadeh, Emily Mower, Samuel Kim, Jeannette N Chang, Sungbok Lee, and Shrikanth S Narayanan. 2008. Iemocap: Interactive emotional dyadic motion capture database. *Language resources and evaluation*, 42(4):335.

Miha Ciglar. 2008. " 3rd. pole"-a composition performed via gestural cues. In *NIME*, pages 203–206.

Marco Donnarumma. 2011. Xth sense: a study of muscle sounds for an experimental paradigm of musical performance. In *ICMC*.

William T Freeman and Craig D Weissman. 1997. Hand gesture machine control system, January 14. US Patent 5,594,469.

Giso Grimm, Tobias Herzke, Daniel Berg, and Volker Hohmann. 2006. The Master Hearing Aid – a PC-based platform for algorithm development and evaluation. *Acta Acustica united with Acustica*, 92:618–628.

Giso Grimm, Tobias Herzke, and Volker Hohmann. 2009. Application of linux audio in hearing aid research. In Frank Neumann, editor, *Proceedings of the Linux Audio Conference*, pages 61–66, Parma, Italy. Istituzione Casa della Musica.

Amit Gupte, Sourabh Joshi, Pratik Gadgul, Akshay Kadam, and A Gupte. 2014. Comparative study of classification algorithms used in sentiment analysis. *International Journal of Computer Science and Information Technologies*, 5(5):6261–6264.

Tobias Herzke, Hendrik Kayser, Frasher Loshaj, Giso Grimm, and Volker Hohmann. 2017. Open signal processing platform for hearing aid research (openMHA). In *Proceedings of the Linux Audio Conference*.

Renato de Souza Melo, Andrea Lemos, Carla Fabiana da Silva Toscano Macky, Maria Cristina Falcão Raposo, and Karla Mônica Ferraz. 2015. Postural control assessment in students with normal hearing and sensorineural hearing loss. *Brazilian journal of otorhinolaryngology*, 81(4):431–438.

Richard Paluch, Matthias Latzel, and Markus Meis. 2015. A new tool for subjective assessment of hearing aid performance: Analyses of interpersonal communication. In *Proceedings of the International Symposium on Auditory and Audiological Research*, volume 5, pages 453–460.

Jan Richarz, Thomas Plotz, and Gernot A Fink. 2008. Real-time detection and interpretation of 3d deictic gestures for interaction-with an intelligent environment. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE.

Jamie Shotton, Toby Sharp, Alex Kipman, Andrew Fitzgibbon, Mark Finocchio, Andrew Blake, Mat Cook, and Richard Moore. 2013. Real-time human pose recognition in parts from single depth images. *Communications of the ACM*, 56(1):116–124.

Shane Torbert. 2012. *Applied computer science*. Springer.

VoiceOfFaust

Bart Brouns
studio magnetophon
Biesenwal 3
Maastricht, Netherlands, 6211 AD
bart@magnetophon.nl

Abstract

VoiceOfFaust turns any monophonic sound into a synthesizer, preserving the pitch and spectral dynamics of the input.

There are 7 synthesizer and two effect algorithms:

- a classic channel vocoder
- a couple of vocoders based on oscillators with controllable formants:
  - CZ resonant oscillators
  - PAF oscillators
  - FM oscillators
  - FOF oscillators
- FM with modulation by the voice
- ring-modulation
- Karplus-Strong used as an effect
- Phase modulation used as an effect

Keywords

Synthesis, Signal Processing, Audio Plugins.

1 Introduction

VoiceOfFaust turns any monophonic sound into a synthesizer, preserving the pitch and spectral dynamics of the input. It is written in Faust [1], and uses a pitch tracker in Pure Data [2].

It consists of:

- an external pitch tracker: helmholtz~ [3] by Katja Vetter.
- a compressor/expander, called qompander [4], ported to Faust.

There are 7 synthesizer and two effect algorithms:

- a classic channel vocoder
- a couple of vocoders based on oscillators with controllable formants:
  - CZ resonant oscillators
  - PAF oscillators
  - FM oscillators
  - FOF oscillators
- FM with modulation by the voice
- ring-modulation
- Karplus-Strong used as an effect
- Phase modulation used as an effect

The features include:

- all oscillators are synchronized to a single saw-wave, so they stay in phase, unless you don't want them to
- powerful parameter mapping system lets you set different parameter values for each band, without having to set them all separately
- formant compression/expansion: Make the output spectrum more flat or more resonant, at the twist of a knob.
- flexible in and output routing: change the character of the synth.
- all parameters, including routing, but except the octave, are step-less, meaning any 'preset' can morph into any other.
- multi-band deEsser and reEsser
- optionally use as a master-slave pair:
  The master is a saw-oscilator driven by the (external) pitchtacker, and the slaves contain everything else, synced to the master.
  This makes it possible to run the slaves as plugins.
- configuration file:
  Through this file, lot's of options can be set at compile time, allowing you to adapt the synth to the amount of CPU power and screen real-estate available.
  Some of the highlights:
- number of bands of the vocoders
- number of output channels
- whether we want ambisonics output
- whether a vocoder has one set of oscillators, or a separate set of oscillators per output.

2 Vocoders

2.1 Common features of all vocoders

2.1.1 Parameter mapping system

The parameters for the vocoders use a very flexible control system:

Each parameter has a bottom and a top knob, where the bottom changes the value at the lowest formant band, and the top the value at the highest formant band.

The rest of the formant bands get values that are evenly spaced in between.

For some of them that means linear spacing, for others logarithmic spacing.

For even more flexibility there is a parametric mid:

You set it's value and band number and the parameter values are now:

- 'bottom' at the lowest band, going to:
- 'mid value' at band nr 'mid band', going to:
- 'top value' at the highest band.

Kind of like a parametric mid in equalizers.

If that's all a bit too much, just set ``para`` to 0 in the configuration file, and you'll have just the top and bottom settings.

2.1.2 Formant compression/expansion

Scale the volume of each band relative to the others:

- 0 = all bands at average volume
- 1 = normal
- 2 = expansion

expansion here means:

- the loudest band stays the same
- soft bands get softer

Because low frequencies contain more energy than high ones, a lot of expansion will make your sound duller.
To counteract that, you can apply a weighting filter, settable from

- 0 = no weighting
- 1 = A-weighting
- 2 = ITU-R 468 weighting

2.1.3 DeEsser

To tame harsh esses, especially when using some formant compression/expansion, there is a deEsser:
It has all the usual controls, but since we already are working with signals that are split up in bands, with known volumes,
it was implemented rather differently:

- multiband, yet much cheaper,
- without additional filters, even for the sidechain,
- and with a dB per octave knob for the sidechain, from 0dB/oct (bypass), to 60dB/oct (fully ignore the lows).

It also has a (badly named) noise strenght parameter: it uses the fidelity parameter from the external pitchtracker to judge if a sound is an S.
When you turn it up, the deEsser gets disabled when the pitchtracker claims a sound is pitched.
See [3] for more info.

2.1.4 ReEsser

Disabled by default, but can be enabled in the configuration file.
It replaces or augments the reduced highs caused by the deEsser.

2.1.5 DoubleOscs

This is a compile option, with two settings:

- 0 = have one oscillators for each formant frequency
- 1 = creates a separate set of oscillators for each output channel, with their phase modulations reversed.

2.1.6 In and output routing

The vocoders can mix their bands together in various ways:
We can send all the low bands left and the high ones right, we can alternate the bands between left and right, we can do various mid-side variations
we can even do a full Hadamard matrix.
All of these, and more, can be cross-faded between.
In the classicVocoder, a similar routing matrix sits between the oscillators and the filters.

2.1.7 Phase parameters

Since all[1] formants are made by separate oscillators that are synced to a single master oscillator, you can set their phases relative to each other.
This allows them to sound like one oscillator when they have static phase relationships, and to sound like many detuned oscillators when their phases are moving.

[1] except for the classicVocoder.

Together with the output routing, it can also create interesting cancellation effects.
For example, with the default settings for the FMvocoder, the formants are one octave up from where you'd expect them to be.
When you change the phase or the output routing, they drop down.

These settings are available:

- static phases
- amount of modulation by low pass filtered noise
- the cutoff frequency of the noise filters

2.2 Features of individual vocoders

2.2.1 ClassicVocoder

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/classicVocoder-svg/process.svg

A classic channel vocoder, with:

- a "super-saw" that can be cross-faded to a “super-pulse", free after Adam Szabo [5].

* flexible Q and frequency setting for the filters
* an elaborate feedback and distortion matrix around the filters

The gui of the classicVocoder has two sections:
First oscillators, containing the parameters for the carrier oscillators.
These are regular virtual analog oscillators, with the following parameters:

- cross-fade between oscillators and noise
- cross-fade between sawtooth and pulse wave
- width of the pulse wave
- mix between a single oscilators and multiple detuned ones
- detuning amount

Second filters, containing the parameters for the synthesis filters:

- bottom, mid and top set the resonant frequencies
- Q for bandwidth
- a feedback matrix. each filter gets fed back a variable amount of:
  - itself
  - it's higher neighbor
  - it's lower neighbor
  - all other filters
  - distortion amount
  - DC offset

2.2.2 CZvocoder

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/czVocoder-svg/process.svg

This is the simplest of the vocoders made out of formant oscilators.
The oscillators where ported from a pd patch by Mike Moser-Booth [6].

You can adjust:

- the formant frequencies
- the phase parameters

2.2.3 PAFvocoder

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/PAFvocoder-svg/process.svg

The oscillators where ported from a pd patch by Miller Puckette [7].

It also has frequencies and phases, but adds index for brightness.

2.2.4 FMvocoder

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/FMvocoder-svg/process.svg

The oscillators where based on code by Chris Chafe [8].

Same parameters, different sound.

2.2.5 FOFvocoder

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/FOFvocoder-svg/process.svg

Original idea by Xavier Rodet [9].
based on code by Michael Jørgen Olsen [10].
Also has frequencies and phases, but adds:

- skirt and decay:
  Two settings that influence the brightness of each band
- Octavation index
  Normally zero. If greater than zero, lowers the effective frequency by attenuating odd-numbered sinebursts.

Whole numbers are full octaves, fractions transitional.
Inspired by an algorithm in Csound [11].

3 Other synthesizers

These are all synths that are not based on vocoders.

3.1 Features of individual synths

3.1.1 FMsinger

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/FMsinger-svg/process.svg

A sine wave that modulates its frequency with the input signal.
There are five of these, one per octave, and each one has:

- volume
- modulation index
- modulation dynamics
  This fades between 3 settings:
  - no dynamics: the amount of modulation stays constant with varying input signal
  - normal dynamics: more input volume equals more modulation
  - inverted dynamics: more input equals less modulation.

3.1.2 CZringmod

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/CZringmod-svg/process.svg

Ringmodulates the input audio with emulations of Casio CZ oscillators.
Again five octaves, with each octave containing three different oscillators:

- square and pulse, each having volume and index (brightness) controls
- reso, having a volume and a resonance multiplier:
  This is a formant oscillator, and it's resonant frequency is multiplied by the formant setting top right.
  It is intended to be used with an external formant tracker.
- There is a global width parameter that controls a delay on the oscillators for one output.

The delay time is relative to the frequency.
Because this delay is applied to just the oscillators, and before the ringmodulation, the sound of both output channels arrives simultaneously.
This creates a mono-compatible widening of the stereo image.

3.1.3 KarplusStrongSinger

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/KarplusStrongSinger-svg/process.svg

This takes the idea of a Karplus Strong algorithm [12], but instead of noise, it uses the input signal.
The feedback is ran trough an allpass filter, modulated with an LFO; adapted from the nonLinearModulator in instrument.lib.
To keep the level from going out of control, there is a limiter in the feedback path.
Parallel to the delay is a separate nonLinearModulator.
Globally you can set:

- octave
- output volume
- threshold of the limiter

For the allpass filters you can set:

- amount of phase shift
- difference in phase shift between left and right (yeah, I lied, there are two of everything)
- amount of modulation by the LFO
- frequency of the LFO, relative to the main pitch
- phase offset between the left and right LFO's.

To round things off there is a volume for the dry path and a feedback amount for the delayed one.

3.1.4 KarplusStrongSingerMaxi

Block-diagram:
https://magnetophon.github.io/VoiceOfFaust/images/KarplusStrongSingerMaxi-svg/process.svg

To have more voice control of the spectrum, this one has a kind of vocoder in the feedback path.
Since we don't want the average volume of the feedback path changing much, only the volumes relative to the other bands, the vocoder is made out of equalizers, not bandpass filters.
You can adjust it's

- strength: from bypass to 'fully equalized'

- cut/boost; steplessly vary between
  - -1 = all bands have negative gain, except the strongest, which is at 0
  - 0 = the average gain of the bands is 0.
  - +1 = the all bands have positive gain, except the weakest, which is at 0
- top and bottom frequencies
- Q factor

4 Master-slave

This is a workaround for the need for an external pitchtracker, making it possible to use the synths and effects as plugins.
It has the nice side effect that your sounds become fully deterministic:
because a pitchtracker will always output slightly different data, or at least at slightly different moments relative to the audio, the output audio can sometimes change quite a bit from run to run.
The master is a small program that receives the audio and the OSC messages from the external pitch tracker, and outputs:

- a copy of the input audio
- a saw wave defining the pitch and phase
- the value of fidelity, from the pitch tracker, as audio.

The slaves are synths and effects that input the above three signals.
The outputs of the master can be recorded into a looper or DAW, and be used as song building blocks, without needing the pitch tracker.
This makes it possible to switch synths, automate parameters, etc.

5 Strengths and weaknessesses of Faust

The Faust language has some big advantages.
The common perks of the language apply. For me, the biggest ones are:

- Quick implementation of ideas.
- If it sounds right, it is right. There won't be any crashes, memory leaks or other bugs.
- Write once, deploy everywhere.
- The block diagrams help with debugging and documentation.
- Fast running code.
- Automatic parrallelisation.

In this project it was also very helpful to be able to easily parameterize things like the number of bands. Related: the input and output routing wouldn't be nearly as easy and fun to implement in most languages, as they lean heavily on Fausts splitting and combinatory operators.

Since this idea has been implemented in PureData earlier, it makes sense to mention two big advantages over that:

1. Text-interface, enabling quicker notation of ideas, version-control and a mouseless workflow.
2. Single sample feedback loops, as used in the classicVocoder.

The downsiders of Faust to me are a steep learning curve and error messages that are often very verbose and unclear.

6 Use cases

The author has used VoiceOfFaust mostly for voice transformation in a musical context, but it has also come in handy to turn a bass-guitar into a synth [14].

7 Deployment

VoiceOfFaust heavily leans on knowing the pitch of the input signal. Since it's not yet possible to do decent pitchtracking in Faust, an external pitchtracker which sends the pitch trough OSC is used.

This limits the usable architectures to the ones supporting OSC.

Specifically, it would be nice to have VoiceOfFaust as a plugin within a DAW, but that is not directly possible.

The master slave architecture is a usable workaround.

To compile VoiceOfFaust, run one of the compilation scripts that support OSC, for example:

faust2jack -osc FMvocoder.dsp

To run it, you can use one of the scripts in the launchers directory, for example:

./FMvocoder_PT

This will start puredata with the pitchtracker patch plus a synth, and connect everything trough jack.

8 Acknowledgements

Many thanks to the developers of Faust [1] and Pure Data [2] for making dsp so accesable yet powerfull.

References

[1] http://faust.grame.fr

[2] https://puredata.info

[3] http://www.katjaas.nl/helmholtz/helmholtz.html

[4] http://www.katjaas.nl/compander/compander.html

[5] https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2010/rapporter10/szabo_adam_10131.pdf

[6] http://forum.pdpatchrepo.info/topic/5992/casio-cz-oscillators

[7] http://msp.ucsd.edu/techniques/v0.11/book-html/node96.html

[8] http://chrischafe.net/glitch-free-fm-vocal-synthesis

[9] http://anasynth.ircam.fr/home/english/media/singing-synthesis-chant-program

[10] https://ccrma.stanford.edu/~mjolsen/220a/fp/Foflet.dsp

[11] https://csound.github.io/docs/manual/fof2.html

[12] https://en.wikipedia.org/wiki/Karplus%E2%80%93Strong_string_synthesis

[13] https://github.com/magnetophon/VoiceOfFaust

[14] http://magnetophon.nl/sounds/BucketBoyz/ShiningBrightLight.mp3

On the Development of C++ Instruments

Victor LAZZARINI
Maynooth University
Maynooth
Co. Kildare
Ireland,
Victor.Lazzarini@nuim.ie

Abstract

This paper brings together some ideas regarding computer music instrument development with respect to the C++ language. It looks at these from two perspectives, that of the development of self-contained instruments with the use of a class library and that of programming of plugin modules for a music programming system. Working code examples illustrate the paper throughout.

Keywords

Computer Music Instruments, C++, Music Programming

1 Introduction

Whatever we do, if we are in the business of making music solely or primarily with computers, one way or another, at some point, we will meet computer music instruments[Lazzarini, 2017a] . Whether we are making electroacoustic music, algorithmic composition, live coding, tracking, creating pop tunes, we will find ourselves manipulating these. They can present themselves through music programming systems [Lazzarini, 2013] , such as Csound [Lazzarini et al., 2016] or Faust [Orlarey et al., 2004], or as software synthesizers, plugins, audio processing programs, etc. There is a wide variety of forms. In this paper, I would like to contemplate one of these that involves libraries, compilers, and the C++ language.

C++ was once described as having "the elegance, the power, and the simplicity of a hand grenade", which to me, as a die-hard pure C programmer sounds about right. However, I must admit that its latest standards, ISO C++11[ISO/IEC, 2011] , C++14[ISO/IEC, 2014] , and the forthcoming C++17,[ISO/IEC, 2017] arriving in quick succession as they are, are making this monstrous language more attractive. Now finally we can write a nice lambda and pass it to a map to process a list, for example. The standard library, borne out of the much appreciated, much maligned, standard template library (STL), has actually become quite usable. There is still enough complexity for one to get entangled, however, but with moderation and good design, we can make it work for us.

This paper will examine two approaches of C++ instrument making. The first one is based on employing a signal processing library to write simple, straightforward, programs that can be ported to various platforms. The second is to create components, plugins, for Csound using a framework that sits atop the system implementation in C. It is mostly directed at computer music practitioners who can converse in C/C++, and it will be fully illustrated by working code, which can also be found somewhere in an online repo (links will be given).

2 AuLib Instruments

Towards the end of 2016, I decided to collect a number of digital signal processing (DSP) algorithms that I had been writing or studying throughout the years into a simple, lightweight, flexible C++ library, called AuLib[1]. One of my aims was to document these uniformly in efficient and readable code so that they could become somewhat of a reference for me and others. I was also rewriting some of my teaching materials and this became part of them. Following a number of refactoring steps, I settled on a design that followed modern C++ standards in employing the standard library as much as possible to handle resources and keeping the code as simple and lightweight as possible.

When designing a class library, there are two distinct possibilities (amongst the various decisions we have to make) with respect to object hierarchies. One is that we can define a base class for DSP objects that has a shared processing interface, that is one (or more) DSP methods that

[1] `github.com/vlazzarini/aulib/`

are specialised in derived classes. A means of connecting objects is provided separately from this, and once connected, we can place the objects in a list of references or pointers to the base class and call the processing method of each in turn to get an output signal. This is what is at the heart of processing engines such as the one in PD, Csound, Faust. If the aim is to create a library whose main objective is to be employed as an engine for some higher-level programming or patching system, this is the way to go. The Sound Object Library[Lazzarini, 2000] was designed this way and it really paid off when it was later wrapped up in Python.

The alternative is to relax this constraint and not provide a unified processing interface, leave it to derived classes to define their own. The advantage of this is that each class can have different ways to handle input parameters to processing methods, depending on what they are supposed to do. So an oscillator might have amplitude and frequency as parameters, in scalar or vectorial forms, or no parameters at all (for say fixed values of amplitude and frequency). It can provide a bunch of overloads to handle each case. A filter will have an input signal and optional parameters, depending on the type. A frequency-domain object might take a spectral frame. This, on one hand, simplifies connections (we can define them at the processing point, rather than separately), and on the other makes it hard to use in sound engine applications where the interface needs to be shared.

Given that the objective here for this library was to provide a working context for a diverse set of algorithms, and to provide a flexible means of using them in programs, I have opted for the second approach. This would provide greater freedom to create exactly the right form to hold each DSP formulation. Now, given this context, it is still desirable to use the class structure afforded by C++ to re-use code fully. This meant to design a base class that was a container for an audio signal, providing the typical fundamental operations we would like to perform on it. For me, this meant: scaling (multiplying by a scalar), offsetting (adding a scalar), mixing (adding a vector/signal) and ring modulating (multiplying by a vector/signal). Granted, in an audience of music and audio developers, we are likely to find multiple definitions of what fundamental operations on signals are, but I am drawing the line here (ok maybe not quite, but let's keep at this for the moment). Attributes such as number of channels (interleaved), sampling rate and vector size are also needed, and of course the audio signal vector itself.

This makes up the `AudioBase` class of the library, which begins like this:

```
class AudioBase {
protected:
  uint32_t m_nchnls;
  uint32_t m_vframes;
  std::vector<double> m_vector;
  double m_sr;
  uint32_t m_error;
```

Having a fundamentally neutral base, with no hint of what a DSP object might want to implement allows me to use it for absolutely everything I can think of, or almost. So of the 50-odd classes currently sitting in the library, only four are not derived from `AudioBase` (fig. 1). It is specialised for common time-domain operations (oscillators, filters, envelopes, etc.), for spectral processing (short-time Fourier transform, phase vocoder), for function tables, for audio input/output, and even for higher-level instrument models. Code re-use is truly maximised.

3 Developing Instruments

A detailed description of the library design is offered elsewhere [Lazzarini, 2017c]. In this paper, we will to look at using it for C++ instrument development. So let's explore some cases[2].

3.1 Basic examples

We begin with a trivial case: a ping instrument, written as a command-line program. This just plays a 440Hz, -6dB sine wave to the output for a couple of seconds. The code, without its safety checks etc, can be abbreviated as this seven-liner:

```
int main() {
  Oscil sig(0.5,440);
  SoundOut output("dac");
  for (int i = 0; i < def_sr * 2;
          i += def_vframes)
    output.write(sig.process());
  return 0;
}
```

Frequency and amplitude are not changing, so I pick the `process()` overload with no parameters and stick its return value straight into the output `write()` method. The two classes

[2] all examples available in the `examples` directory of the AuLib repository.

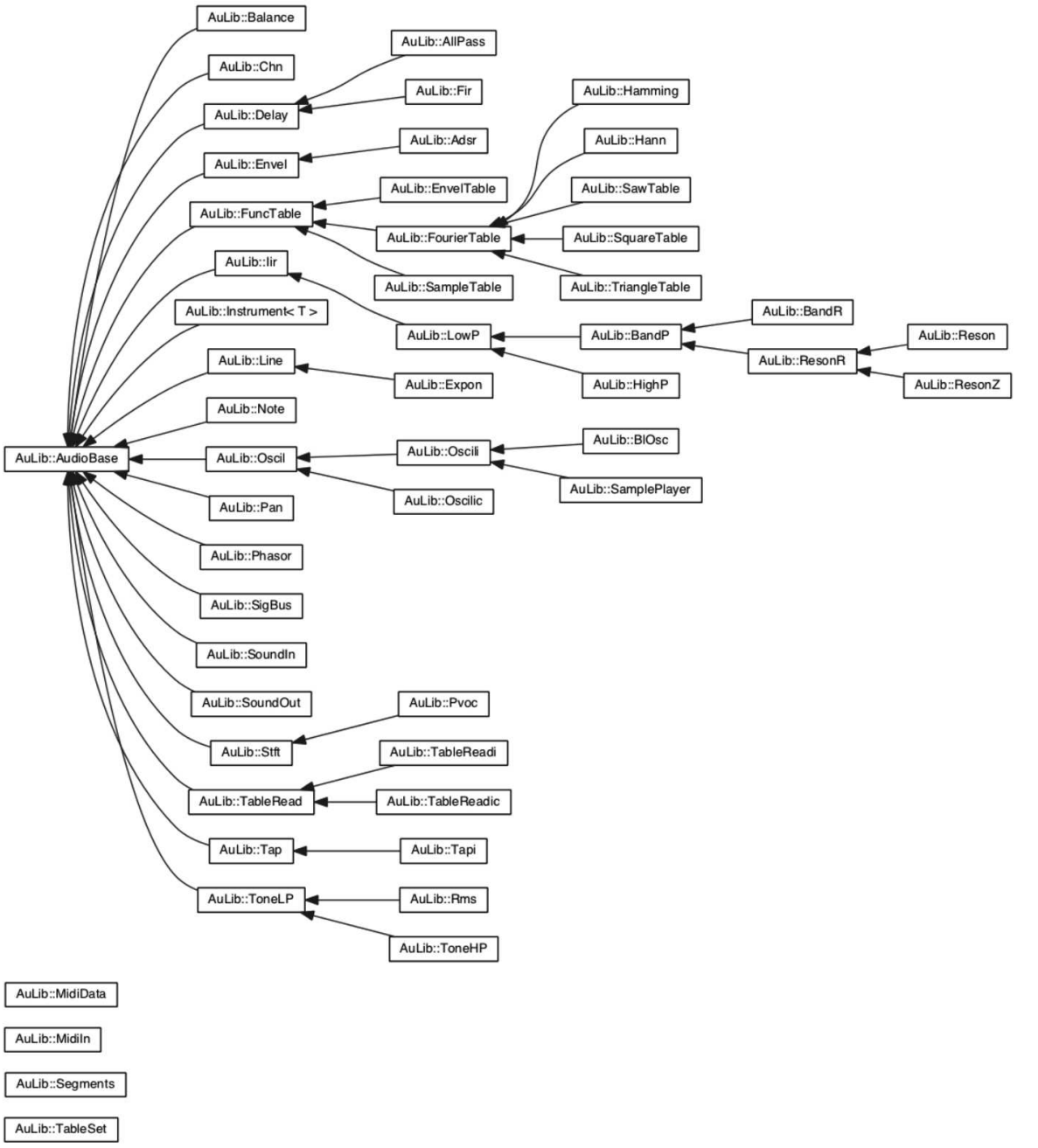


Figure 1: The AuLib class library

share the base, but they have distinctly-named and defined processing methods.

Let's try something slightly less simplistic. A similarly-placed instrument but now with a sweeping resonant filter acting on a sawtooth wave:

```
int main() {
  TableSet saw(SAW);
  BlOsc sig(0.5, 440., saw);
  ResonR fil(1000, 1.);
  Balance bal;
  SoundOut output("dac");
  for (int i = 0; i < def_sr*10;
    i += def_vframes) {
     sig.process();
     fil.process(sig,
      1000. + 400. * i / def_sr);
     bal.process(fil, sig);
      output.write(bal);
  }
  return 0;
}
```

TableSet creates a set of tables for a band-limited oscillator. The filter centre frequency is

varied over time, and we feed its output into a balancing operator that uses a comparator to keep amplitudes under control.

To demonstrate how the base class-defined operations can be useful, we have a simple FM example

```
int main() {
  double fm = 440., fc = 220., ndx = 5.;
  Oscili mod, car;
  SoundOut output("dac");
  for (int i = 0; i < def_sr*10;
         i += def_vframes) {
    mod.process(ndx * fm, fm);
    car.process(0.5, mod += fc);
    output.write(car);
  }
  return 0;
}
```

Note the use of the overloaded sum-assignment operator in `mod += fc` to add the modulator signal to the carrier (scalar) frequency.

3.2 Instrument Models

Clearly, the examples above are more demonstrations of how instruments can be set up. This would, in a more realistic scenario, be placed in plugin or GUI application wrapping code, where they can become useful. The AuLib also provides some modelling of instruments and instances of these. We can show how these work in a straightforward application case: a polyphonic MIDI synthesiser.

The AuLib class `Note` provides the base for an instance of a sound object, which can be for example, a synthesiser voice. This holds basic parameters such as amplitude, cps pitch, etc. that we can use to control a sound object. To use it, we derive our own, and specialise its `dsp` method, placing our sound processing code there.

```
class SineSyn : public Note {
  // signal processing objects
  Adsr m_env;
  Oscili m_osc;

  // DSP override
  virtual const SineSyn &dsp() {
    if (!m_env.is_finished())
      set(m_osc(m_env(), m_cps));
    else clear();
    return *this;
  }
...
```

The sound synthesis is again, trivial, to keep the example focused: an envelope and a sine wave oscillator. But note that we have a new convenient interface: using the classes `operator()`, we connect objects more easily one into another. This syntax reinforces the connection metaphor, envelope, alongside pitch, into oscillator. Given that the class is derived from `AudioBase`, we set its vector to the result of the processing.

Additionally, we want to specialise two other methods: for sound onset and sound termination:

```
// note off processing
virtual void off_note() {
    m_env.release();
}

// note on processing
virtual void on_note() {
   m_env.reset(m_amp, 0.01,
     0.5, 0.25 * m_amp, 0.01);
}
```

This plus the constructor completes our `Note`-derived class. Now we want to model the whole synthesiser, not just its voices. To do this, we can use the `Instrument` template class, instantiated with the required number of voices and our note class:

```
Instrument<SineSyn> synth(8);
```

An important aspect of this class is that it has a `dispatch()` method that takes in five parameters (message type, channel, data1, data2, time stamp) and responds to two message types (NOTE ON, NOTE OFF). While these are the same as the MIDI channel messages, we are just re-using the metaphor here. The call to `dispatch()` does not need to originate from MIDI or be limited to the usual MIDI data ranges. Specialisations of instrument can re-implement message handling to allow for other types. `Instrument` also handles polyphony using last-note priority, and this can also be overriden in derived classes.

Given that the example will use MIDI input, the library supports a simple MIDI listener class that takes an `Instrument` object (or from any type implementing `dispatch()` and `process()` and responds to messages. The complete program becomes very straightforward (trivial signal handler implementation omitted):

```
int main() {
 int dev;
 Instrument<SineSyn> synth(8);
 SoundOut out("dac");
 MidiIn midi;
 std::signal(SIGINT, signal_handler);

 std::cout <<
```

```
    "Available MIDI inputs:\n";
  for(auto &devs:
         midi.device_list())
    std::cout << devs << std::endl;
  std::cout << "choose a device: ";
  std::cin >> dev;

  if (midi.open(dev) ==
      AULIB_NOERROR) {
    std::cout <<
     "running...
      (use ctrl-c to close)\n";
    while (running)
       // listen to midi on
       // behalf of synth
       out(midi.listen(synth));
  } else
    std::cout <<
     "error opening device...\n";
   std::cout << "...finished \n";
   return 0;
}
```

Again, with a few lines of code, we can get a basic MIDI synthesiser instrument. Although the synthesis is simple, it can be shown that the effort involved in more complex examples scales well. It is just a case of using other signal processing objects in different arrangements.

4 Csound Plugins

The second case of C++ instrument development we will look at focuses on creating components (plugins) that can be employed in a music programming language. Unit generators in Csound are known as *opcodes* and the system has a well-document C interface for the purpose of adding new ones of these to it. It also has a C++ base class that has been used for a small number of opcode plugin libraries that come with the system.

With the intention of enabling a more complete and well-integrated C++ support for plugin opcode development, I have introduced the Csound Plugin Opcode Framework[3] CPOF (pronounced *see-pough* or *cipó* = *vine* in Portuguese[4]). The actual framework part of it is fairly light, consisting of two template base classes, but it also contains an extensive set of utility classes that wrap Csound C code for C++ use in a very idiomatic way (table 1). CPOF is discussed extensively in [Lazzarini, 2017b].

[3]available as part of Csound, `github.com/csound/csound`, with code examples in the `examples/plugin` directory.

[4]as in: C++ gives you enough vine, or rope, for you to either hoist yourself up a tree, or hang yourself fairly decently.

Class	Description
`Csound`	The Csound engine
`Params`	Opcode parameters
`AudioSig`	Audio signals
`Fsig`	Spectral signals
`Pvframe<T>`	Spectral data frames
`Pvbin<T>`	Spectral data bins
`Vector<T>`	Array variables
`Table`	Function tables
`AuxMem<T>`	Dynamic memory
`Thread`	Multithreading
`Plugin<N,M>`	Plugin base class
`FPlugin<N,M>`	Spectral plugin base class

Table 1: Classes provided by CPOF.

5 Plugin Examples

The Csound language has a variety of internal data types that its opcodes can process. We will look at each one of these with a programming example.

5.1 Init-time opcodes

In Csound, code that is run only once per instantiation (or again on explicit re-initisation) employs init-time variables. These are scalar types holding a floating-point number (the `MYFLT` type defined by the system). Plugin opcodes for these types are derived from `Plugin` and are instantiated templates taking the number of output and input arguments (respectively) as parameters. The following examples uses the standard library Gaussian generator to produce a random number using the normal distribution. The first input argument is the mean, followed by the deviation, and the seed:

```
#include <plugin.h>
struct Gauss :
  csnd::Plugin<1, 3>{
  std::normal_distribution<MYFLT> norm;
  std::mt19937 gen;

  init init(){
   csnd::constr(&norm, inargs[0],
      inargs[1]);
   csnd::constr(&gen, inargs[2]);
   outargs[0] = norm(gen);
   csnd::destr(&norm);
   csnd::destr(&gen);
  }
};
```

Note that because Csound instantiates the plugin object and it does not know anything about C++ constructors, we need to explicitly construct the objects `norm` and `gen`. When we

are done, we need to destruct them as they are likely to have allocated resources, which we do not want to be left dangling. The `Plugin` base class gives us the `inargs` and `outargs` objects, which contain the input and output arguments respectively.

In order for the plugin to be added to Csound's collection of opcodes, we need to register it. To do this, we implement the `csnd::on_load()` function, where we place a call to the `csnd::plugin<T>()` template method, passing the argument types (`"i"`) and the action time of the opcode (`thread::i`), as well as the opcode name we will use ("guassian"):

```
#include <modload.h>
void csnd::on_load(Csound
     *csound) {
csnd::plugin<Gauss>(csound, "gaussian",
  "i", "iii", csnd::thread::i);
}
```

5.2 Control-rate opcodes

The next data type we can tackle is the one used control-rate variables (k). This is also a scalar, but now the opcode is active at performance time (as well as init). A control-rate version of the `gaussian` opcode would look like this:

```
struct GaussP :
 csnd::Plugin<1, 3>{
  std::normal_distribution<MYFLT>
    norm;
  std::mt19937 gen;

  int init(){
   csnd::constr(&norm, inargs[0],
      inargs[1]);
   csnd::constr(&gen, inargs[2]);
   csound->plugin_deinit(this);
   return OK;
  }

  int deinit(){
   csnd::destr(&norm);
   csnd::destr(&gen);
   return OK;
  }

  int kperf() {
    outargs[0] = norm(gen);
    return OK;
  }
};
```

We can see that we now supplied the `kperf()` that will be called repeatedly during performance. Another difference is that we have to provide a `deinit()` to call the destructors, which will be called when performance ends. This method needs to be registered separately with Csound through the `plugin_deinit()` template function. We register this version of the opcode with:

```
csnd::plugin<GaussP>(csound,"gaussian",
 "k", "iii", csnd::thread::ik);
```

5.3 Audio-rate opcodes

For audio signals, we need to implement the `aperf()` method. The variable now is a vector, so we have to use an `AudioSig` object to hold it. The following example shows an `aperf()` method that can be added to `GaussianPerf` to implement an audio rate opcode:

```
int aperf(){
 csnd::AudioSig out(this, outargs(0));
 for(auto &sample : out)
      sample = norm(gen);
 return OK;
}
```

The same class can then be registered for an audio-rate output:

```
csnd::plugin<GaussP>(csound, "gaussian",
 "a", "iii", csnd::thread::ia);
```

5.4 Spectral signals

Spectral signals in Csound are carried from opcode to opcode using fsig variables. These are self-describing variables holding one frame of frequency-domain data, plus associated information about the stream. In CPOF, we manipulate these using the `pv_stream` class. Similarly to audio signals we can get the fsig data off arguments into objects of these types for processing. An opcode is responsible for initialising its own output stream, which we can do at init time. Stream frames can be decomposed in separate bins held by `pv_bin` objects.

The example below shows a plugin that implements spectral tracing [Wishart, 1996] defined as retaining only the loudest N bins in each frame. Some important aspects to note about this code: (a) spectral processing occurs at a rate determined by the frame analysis rate, so we run it a k-rate and process frames as they become available; (b) a framecount, a member variable of the FPlugin base class, is kept for this. (c) The `AuxMem` is used to manage a heap-allocated block of memory to keep bin amplitudes; and (d) we add the types as a static constant member of the class, which simplifies the plugin registration call.

The basic algorithm is as follows:

1. get the amplitudes from each bin;

2. find the nth loudest;
3. use this as a threshold to filter the frame date, keeping only the bin holding amplitudes above it.

```
#include <plugin.h>
#include <algorithm>

struct PVTrace : csnd::FPlugin<1, 2> {
 csnd::AuxMem<float> amps;
 static constexpr
  char const *otypes = "f";
 static constexpr
  char const *itypes = "fk";

 int init() {
  if(inargs.fsig_data(0).isSliding())
   return csound->init_error(
     Str("sliding not supported"));

  if(inargs.fsig_data(0).fsig_format()
     !=csnd::fsig_format::pvs &&
     inargs.fsig_data(0).fsig_format()
     !=csnd::fsig_format::polar)
      return csound->init_error(
      Str("fsig format not supported"));

  amps.allocate(csound,
    inargs.fsig_data(0).nbins());

  csnd::Fsig &fout =
   outargs.fsig_data(0);
  fout.init(csound,
   inargs.fsig_data(0));

  framecount = 0;
  return OK;
 }

 int kperf() {
  csnd::pv_frame &fin =
   inargs.fsig_data(0);
  csnd::pv_frame &fout =
   outargs.fsig_data(0);

  if(framecount < fin.count()) {
   int n =  fin.len() - (int)inargs[1];
   float thrsh;

   std::transform(fin.begin(),fin.end(),
     amps.begin(), [](csnd::pv_bin f){
      return f.amp(); });

   std::nth_element(amps.begin(),
     amps.begin()+n, amps.end());
   thrsh = amps[n];

   std::transform(fin.begin(), fin.end(),
    fout.begin(),
     [thrsh](csnd::pv_bin f){
      return f.amp() >= thrsh ?
        f : csnd::pv_bin(); });

   framecount = fout.count(fin.count());
  }
  return OK;
 }
};

#include <modload.h>
void csnd::on_load(Csound *csound) {
 csnd::plugin<PVTrace>(csound,
   "pvstrace", csnd::thread::ik);
}
```

The standard library algorithms are very well suited to implementing these steps. The code becomes very compact and fairly readable.

5.5 Array variables

Csound has a container type, array, which can be used to create vectors of built in types. CPOF provides a template class `Vector<T>` to wrap array arguments conveniently for manipulation. The typedef `myflt_vector` is an instantiation of this template for real values (`MYFLT`). The following example combines the use of lambdas and templates to create a whole family of binary (two-operand) operators for numeric (scalar) arrays. It can be used for init and k-rate opcodes. The processing is placed on a separate function to avoid code duplication. It is just a matter of mapping the inputs into the outputs through the application of a given function.

```
template <MYFLT (*bop)(MYFLT, MYFLT)>
struct ArrayOp2 : csnd::Plugin<1, 2> {

  int process(csnd::myfltvec &out,
              csnd::myfltvec &in1,
              csnd::myfltvec &in2) {
   std::transform(in1.begin(), in1.end(),
    in2.begin(), out.begin(),
    [](MYFLT f1, MYFLT f2) {
     return bop(f1, f2); });
   return OK;
  }

  int init() {
    csnd::myfltvec &out =
     outargs.myfltvec_data(0);
    csnd::myfltvec &in1 =
     inargs.myfltvec_data(0);
    csnd::myfltvec &in2 =
     inargs.myfltvec_data(1);

    if (in2.len() < in1.len())
     return csound->init_error(
      Str("second input array"
            " is too short\n"));

    out.init(csound, in1.len());
    return process(out, in1, in2);
  }

  int kperf() {
   return
```

```
    process(outargs.myfltvec_data(0),
        inargs.myfltvec_data(0),
        inargs.myfltvec_data(1));
  }
};
```

This class template then is instantiated to create various opcodes based on different two-operand functions:

```
csnd::plugin<ArrayOp2<std::atan2>>
 (csound, "taninv",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::atan2>>
 (csound, "taninv",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::pow>>
 (csound, "pow",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::pow>>
 (csound, "pow",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::hypot>>
 (csound, "hypot",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::hypot>>
 (csound, "hypot",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmod>>
 (csound, "fmod",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmod>>
 (csound, "fmod",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmax>>
 (csound, "fmax",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmax>>
 (csound, "fmax",
  "k[]", "k[]k[]", csnd::thread::ik);
csnd::plugin<ArrayOp2<std::fmin>>
 (csound, "fmin",
  "i[]", "i[]i[]", csnd::thread::i);
csnd::plugin<ArrayOp2<std::fmin>>
 (csound, "fmin",
  "k[]", "k[]k[]", csnd::thread::ik);
```

This is a good example of how we can apply modern a C++ idiom to create compact code for the generation of a family of related opcodes.

6 Conclusions

Perhaps one of the conclusions of this paper is that C++ is not such a terrible choice for the implementation of computer music instruments. While C is still the preeminent language for audio signal processing, the latest C++ standards have made that language somewhat more interesting, providing almost a blend of high-level scripting with a (hopefully) efficient implementation.

References

ISO/IEC. 2011. ISO international standard ISO/IEC 4882:2011, programming language C++.

ISO/IEC. 2014. ISO international standard ISO/IEC 14882:2014, programming language C++.

ISO/IEC. 2017. Working draft, standard for programming language C++.

V. Lazzarini, J. ffitch, S. Yi, J. Heintz, Ø. Brandtsegg, and I. McCurdy. 2016. *Csound: A Sound and Music Computing System*. Springer Verlag.

V. Lazzarini. 2000. The SndObj sound object library. *Organised Sound*, (5):35–49.

V. Lazzarini. 2013. The development of computer music programming systems. *Journal of New Music Research*, (42):97–110.

V. Lazzarini. 2017a. *Computer Music Instruments*. Springer Verlag.

V. Lazzarini. 2017b. The csound plugin opcode framework. In *SMC 2017 (under review)*, Helsinki.

V. Lazzarini. 2017c. The design of a lightweight dsp programming language. In *SMC 2017 (under review)*, Helsinki.

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of faust. *Soft Computing*, 8(9):623?632.

T. Wishart. 1996. *Audible Design*. Orpheus The Pantomine.

Meet the Cat: Pd-L2Ork and its New Cross-Platform Version "Purr Data"

Ivica Ico Bukvic
Virginia Tech SOPA ICAT
DISIS L2Ork
Blacksburg, VA, USA 24061
ico@vt.edu

Albert Gräf
Johannes Gutenberg
University (JGU)
IKM, Music-Informatics
Mainz, Germany
aggraef@gmail.com

Jonathan Wilkes
jon.w.wilkes@gmail.com

Abstract

The paper reports on the latest developments of Pd-L2Ork, a fork of Pd-extended created by Ico Bukvic in 2010 for the Linux Laptop Orchestra (L2Ork). Pd-L2Ork offers many usability improvements and a growing set of objects designed to lower the learning curve and facilitate rapid prototyping. Started in 2015 by Jonathan Wilkes, Purr Data is a cross-platform port of Pd-L2Ork which has recently been released as Pd-L2Ork version 2. It features a complete GUI rewrite and Mac/Windows support, leveraging JavaScript and Node-Webkit as a replacement for Pd's aging Tcl/Tk-based GUI component.

Keywords

Pd-L2Ork, Purr Data, fork, usability, L2Ork

1 Introduction

Pure Data, also known as Pd, [15] is arguably one of the most widespread audio and multimedia dataflow programming languages. Pd's history is deeply intertwined with that of its commercial counterpart, Cycling 74's Max [16]. A particular strength shared by the two platforms is in their modularized approach that empowers third party developers to extend the functionality without having to deal with the underlying engine. Perhaps the most profound impact of Pd is in its completely free and open source model that has enabled it to thrive in a number of environments inaccessible to its commercial counterpart. Examples include custom in-house solutions for entertainment software (e.g. EaPd [10]), Unity3D [18] and smartphone integration via libPD [1], an embeddable library (e.g. RjDj [11], PdDroidParty [12], and Mobmuplat [9]), and other embedded platforms, such as Raspberry Pi [4].

Pd's author Miller Puckette has spearheaded a steady development pace with the primary motivation being iterative improvement while preserving backwards compatibility. Puckette's work on Pd continues to be instrumental in fostering creativity and curiosity across generations, and as the library of works relying on Pd grows, so does the importance of conservation and ensuring that Pd continues to support even the oldest of patches. However, the inevitable side-effect of the increasingly conservationist focus of the core Pd is that any new addition has to be carefully thought out in order to account for all the idiosyncrasies of past versions and ensure there is a minimal chance of a regression. This vastly limits the development pace.

As a result, the Pd community sought to complement Pure Data's compelling core functionality with a level of polish that would lower the initial learning curve and improve user experience. In 2002 the community introduced the earliest builds of Pd-extended [13], the longest running Pd variant. There were other ambitious attempts, like pd-devel, Nova, and DesireData [14], and in recent years Pd has seen a resurgence in forks that aim to sidestep usability issues through alternative approaches, including embeddable solutions (e.g. libPd) and custom front ends. Pd-extended was probably the most popular alternative Pd version which continues to be used by many, even though it was abandoned in 2013 by its maintainer Hans-Christoph Steiner due to lack of contributors to the project.

Pd-L2Ork presents itself as a viable alternative which started out as a fork of Pd-extended and continues to be actively maintained. We begin with a discussion of Pd-L2Ork's history, motivation and implementation. We then look at Pd-L2Ork's most recent off-spring nick-named "Purr Data", which has recently been released as Pd-L2Ork version 2, runs on Linux, Mac and Windows, and offers some unique new features, most notably a completely new and improved GUI component. The paper concludes with some remarks on availability and avenues for future developments.

2 History and Motivation

Introduced in 2009 by Bukvic, Pd-L2Ork [2] started as a Pd-extended 0.42.5 variant. The focus was on nimble development designed to cater to the specific needs of the Linux Laptop Orchestra (L2Ork), even if that meant suboptimal initial implementations that would be ironed out over time as the understanding of the overall code base improved and the target purpose was better understood through practice.

An important part of L2Ork's mission was educational outreach. Consequently, a majority of early additions to Pd-extended focused on usability improvements, including graphical user interface and editor functions. While some of these were incorporated upstream, a growing number of rejected patches began to build an increasing divide between the two code bases. As a result in 2010 Bukvic introduced a separately maintained Pd-extended variant, named Pd-L2Ork after L2Ork for which it was originally designed.

Over time, as the project grew in its scope and visibility, it attracted new users, and eventually a team of co-developers, maintainers and contributors formed around it. This is obviously important for the long-term viability of the project, so that it doesn't fall victim to Pd-extended's fate, and thus the development team continues to invite all kinds of contributions.

Pd-L2Ork's philosophy grew out of its initial goals and the early development efforts. It is defined by a nimble development process allowing both major and iterative code changes for the sake of improving usability and stability as quickly as possible. Another important aspect of this philosophy is releasing improvements early and often in order to have working iterations in the hands of dozens of students of varying educational backgrounds and experience, which ensured quick vetting of the ensuing solutions.

Despite an ostensibly lax outlook on backwards compatibility, to date Pd-L2Ork and Purr Data remain compatible with Pd (the `-legacy` flag can be used to disable some of the more disruptive changes). In particular, there haven't been any changes in the patch file format, so patches created in Pd still work without any ado in Pd-L2Ork and vice versa (assuming that they don't use any externals which aren't available in the target environment). Also, communication between GUI and engine still happens through sockets, so that the two can run in separate processes (running the engine with real-time priorities).

Like Pd-extended, Pd-L2Ork provides a single turnkey monolithic solution with all the libraries included in one package. This minimizes overhead in configuring the programming environment and installing supplemental libraries, and addresses the potential for binary incompatibility with Pd.

3 Implementation

Pd-L2Ork's code base increasingly diverges from Pd. It consists of many bug-fixes, additions and improvements, which can be split into engine, usability, documentation, new and improved objects and libraries, scaffolded learning and rapid prototyping. In this section we highlight some of the most important user-visible changes and additions, more details can be found in the authors' PdCon paper [3].

3.1 Engine

Internal engine contributions have largely focused on implementing features and bug-fixes requested by past and existing Pd users. Some of these include patches that have never made it to the core Pd, such as the cord inspector (a.k.a. magic glass), improved data type handling logic, and support for outlier cases that may otherwise result in crashes and unexpected behavior. Additional checks were implemented for the Jack [6] audio backend to avoid hangs in case Jack freezes. Default sample rate settings are provided for situations where Pd-L2Ork may run headless (without GUI), thus removing the need for potentially unwieldy headless startup procedures. The `$0` placeholder in messages now automatically resolves to the patch instance, while the `$@` argument can be used to pass the entire argument set inside a sub-patch or an abstraction.[1] `[trigger]`[2] logic has been expanded to allow for static allocation of values, which alleviates the need for creating bang triggers that are fed into a message with a static value.

Visual improvements: The Tk-based [19] graphical engine has been replaced with TkPath [17] which offers an SVG-enabled antialiased

[1] In Pd parlance, an *abstraction* is a Pd patch encapsulating some functionality to be used as a subpatch in other patches.

[2] Here and in the following we employ the usual convention to indicate Pd objects by enclosing them in brackets.

canvas.[3] A lot of effort went into streamlining "graph-on-parent" (Pd's facility to draw GUI elements in a subpatch on its parent), including proper bounding box calculation and detection, optimizing redraw, and resolving drawing issues with embedded graph-on-parent patches. Improvements also focused on sidestepping the limitations of the socket-based communication between the GUI and the engine, such as keyboard autorepeat detection. As a result, the `[key]` object can be instantiated with an optional argument that enables autorepeat filtering, while retaining backward compatibility.

Stacking order: Another substantial core engine overhaul pertains to consistent ordering of objects in the glist (a.k.a. canvas) stack. This has helped ensure that objects always honor the visual stacking order, even after undo and redo actions, and has paved the way towards more advanced functionality including advanced editing techniques and a system-wide preset engine.

Presets: The preset engine consists of two new objects `[preset_hub]` and `[preset_node]`. Nodes can be connected to various objects, including arrays, and can broadcast the current state to their designated hub for storing and retrieval. Multiple hubs can be used with varying contexts. The ensuing system is universal, efficient, unaffected by editing actions, and abstraction- and instance-agnostic (e.g., using multiple instances of the same abstraction is automatically supported). It supports anything from recording individual states to real-time automation of multiple parameters through periodic snapshots.

Data structures: Data structures are an advanced feature of Pd to produce visualizations of data collections such as interactive graphical scores. Pd-L2Ork enhances these with the addition of sprites and new ways to manipulate the data.

3.2 Usability

On the surface Pd-L2Ork builds on Pd-extended's appearance improvements. Under the hood, with the canvas being drawn as a collection of SVG shapes, the entire ecosystem lends itself to a number of new opportunities. The most obvious involve antialiased display, advanced shapes (e.g. Bézier curves that are also used for drawing patch cords), support for image formats with alpha channel, and advanced data structure drawing and manipulation using SVG-centric enhancements (Fig. 1).

[3]SVG = Scalable Vector Graphics, a widely used vector image format standardized by the W3C.

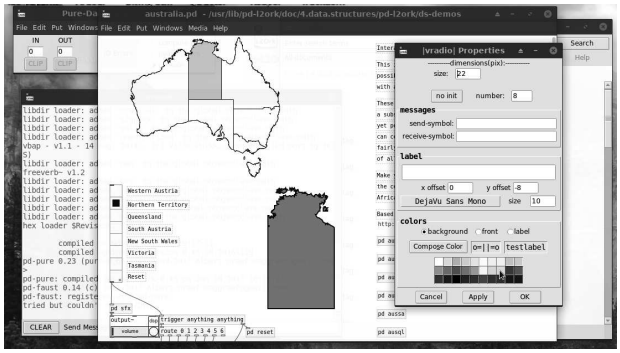


Figure 1: Pd-L2Ork running on Linux.

A majority of usability improvements focus on the editor. The *consistent stacking order* implemented in the engine has served as a foundation for the infinite undo, as well as to-front and -back stacking options that are accessible via the right-click context menu. Lots of improvements and polishing went into the *iemgui objects*, such as improved positioning, enhanced properties dialogs and graph-on-parent behavior.

The old *autotips patch* was integrated (and improved upon). The *tidy up* feature has been redesigned to offer a two-step realignment of objects. (The first key press aligns the objects on a single axis, while the second respaces them, so that they are equidistant from each other.) *Intelligent patching* was implemented to provide four variants of automatically generating multiple patch cords based on user's selection, and to provide additional ways of creating multiple connections (e.g. SHIFT + mouse click). The *canvas scrolling* logic has been overhauled to minimize the use of scrollbars, provide minimal visual footprint, and ensure most of the patch is always visible.

Pd-L2Ork supports *drag and drop* and has support for pasting *Pd code snippets* (using Pd's "FUDI" format) directly onto the canvas. The *copy and paste* engine has been overhauled to improve buffer sharing across multiple application instances. The entire graphics engine is *themeable* and its settings are by default saved with the rest of the configuration files.

3.3 Object Libraries

Apart from the core Pd objects and improvements described in the Engine section above, Pd-L2Ork offers a growing number of revamped

objects while also pruning redundant and unnecessary objects.

Special attention was given to supporting the Raspberry Pi (RPi) platform with a custom set of objects designed specifically to harness the full potential of the RPi GPIO and I2C interfaces, including `[disis_gpio]` and `[disis_spi]` [4]. The cyclone library has received new documentation and a growing number of bugfixes and improvements. Ggee library's `[image]` has received a significant overhaul and became the catchall solution for image manipulation. In addition to the standard Pd-extended libraries, Pd-L2Ork has reintroduced `[disis_munger~]` and an upgraded version of the `[fluid~]` soundfont synth external which depend on the flext library. Other libraries include fftease, lyonpotpourri, and RTcmix~. Pd-L2Ork bundles advanced networking externals `[disis_netsend]` and `[disis_receive]`, convenience externals like `[patch_name]`, and abstractions (e.g., those of the K12 learning module [5], and a growing number of L2Ork-specific abstractions designed to foster rapid prototyping). A few libraries have been removed due to lack of support and/or GUI object implementations that utilize hardwired Tcl-specific workarounds.

3.4 Introspection

Most interpreted languages have mechanisms to do introspection. Pd-L2Ork features a collection of "info" classes for retrieving the state of the program on a number of levels, from the running Pd instance to individual objects within patches. Four classes provide the basic functionality:

- `[pdinfo]` reflects the state of the running Pd instance, including dsp state, available/connected audio and midi devices, platform, executable directory, etc.
- `[canvasinfo]` is a symbolic receiver for the canvas, abstraction arguments, patch filename, list of current objects, etc. The object takes a numeric argument to query the state of parent or ancestor canvases.
- `[classinfo]` offers information about the currently loaded classes in the running instance. This includes creator argument types, as well as the various methods.
- `[objectinfo]` returns bounding box, class type, and size for a particular object on the canvas.

While the introspection provided by these classes is relatively rudimentary, it alleviates the need for a large number of external libraries that add missing core functionality. For example, Pd-L2Ork ships with several compiled externals whose purpose is to fetch the list of abstraction arguments. These externals all have different interfaces and are spread across various libraries. Having one *standard* built-in interface for fetching arguments that behaves similarly to other introspection interfaces improves the usability of the system. Furthermore, opening up rudimentary introspection to the user increases the composability of Pd. Functionality that previously only existed inside the C code can now be implemented as an abstraction (i.e., in Pd itself). These don't require compilation and are more accessible to a wider number of users to test and improve them.

4 Purr Data a.k.a. "The Cat"

Despite all of the improvements it brings to the table, Pd-L2Ork still employs the same old Tcl/Tk environment to implement its graphical user interface. This is both good and bad. The major advantage is compatibility with the original Pd. On the other hand, Tcl/Tk looks and feels quite dated as a GUI toolkit in this day and age. Tcl is a rather basic programming language and its libraries have been falling behind, making it hard to integrate the latest GUI, multimedia and web technologies. Last but not least, Pd-L2Ork's adoption was severely hampered by the fact that it relies on some lesser-used Tcl/Tk extensions (specifically, Tk-Path and the Tcl Xapian bindings) which are not well-supported on current Mac and Windows systems, and thus would have required substantial porting effort to make Pd-L2Ork work there.

Purr Data was created in 2015 by Wilkes to address these problems.[4] The basic idea was to replace the aging Tcl/Tk GUI engine with a modern, open-source, well-supported cross-platform framework supporting programmability and the required advanced 2D graphical capabilities, without being tied into a particular GUI toolkit again.

[4] Readers may wonder about the nick-name of this Pd-L2Ork offspring, to which the author in his original announcement at `http://forum.pdpatchrepo.info` only offered the explanation, "because cats." Quite obviously the name is a play on "Pure Data" on which "Purr Data" is ultimately based, but it also raises positive connotations of soothing purring sounds.

Employing modern web technologies seemed an obvious choice to achieve those goals, as they are well-supported, cross-platform and toolkit-agnostic, programmable (via JavaScript), and offer an extensive programming library and built-in SVG support (as a substitute for Pd-L2Ork's use of TkPath which incidentally follows the SVG graphics model).

There are basically two main alternatives in this realm, nw.js[5] a.k.a. "node-webkit" and Electron[6]. These both essentially offer a standalone web browser engine combined with a JavaScript runtime. nw.js was chosen because it offers some technical advantages deemed important for Purr Data (in particular, an easier interface to create multi-window applications and better support for legacy Windows systems).

So, in a nutshell, Purr Data is Pd-L2Ork with the Tcl/Tk GUI part ripped out and replaced with nw.js. Purr Data's GUI is written entirely in JavaScript, which is a much more advanced programming language than Tcl with an abundance of libraries and support materials. Patches are implemented as SVG documents which are generally much more responsive and offer better graphical capabilities than Tk windows. They can also be zoomed to 16 different levels and themed using CSS, improving usability. The contents of a patch window is drawn and manipulated using the HTML5 API. Thus the code to display Pd patches is very portable and will work in any modern GUI toolkit that has a webview widget.

There are also some disadvantages with this approach. First, Tcl code in Pd's core and in the externals needs to be ported to JavaScript to make it work with the new GUI; we'll touch on this in the following subsection.

Second, the size of the binary packages is much larger than with Pd-L2Ork or Pd-extended since, in order to make the packages self-contained, they also include the full nw.js binary distribution. This is a valid complaint about many of the so-called "portable desktop applications" being offered these days, but in the case of Purr Data it is mitigated by the fact that plain Pd-L2Ork is not exactly a slim package either.

Third, the browser engine has a much higher memory footprint than Tcl/Tk which might be an issue on embedded platforms with very tight memory constraints.

[5] `https://nwjs.io/`

[6] `https://electron.atom.io/`

So far, none of these issues has turned out to be a major road-block in practice. The most serious issue we're facing right now probably is that externals using Pd's Tcl/Tk facilities need to have their GUI code rewritten to make it work with Purr Data; this is a substantial undertaking and thus hasn't been done for all bundled externals yet.

4.1 Implementation

Using JavaScript in lieu of Tcl as the GUI programming language poses some challenges. Tcl commands with Tk window strings are hard-coded into the C source files of Pd. This means that any port to a different toolkit must either replace those commands with an abstract interface, or write middleware that turns the hard-coded Tcl strings into abstract commands. Given the complexity of Tcl commands in both the core and external libraries, that middleware would essentially have to re-implement a large part of the Tcl interpreter.

Consequently, Purr Data opted for the former approach of directly implementing an abstract interface. This takes the form of a JavaScript API providing the necessary GUI tie-ins to the engine and externals, which is called from the C side using a new set of functions (`gui_vmess` et al) which replace the corresponding functions of Pd's C API (`sys_vgui` etc.). As already mentioned, this means that externals which use these facilities need to have their GUI code rewritten to make it work with the new GUI. (Affected externals will work, albeit without their GUI features.)

Adding to the porting difficulty is the fact that Pd has no formal specification, and its GUI interface follows no common design pattern for 2D graphics. For example, the graph-on-parent window appears at a glance as a viewport that clips to a specified bounding box. However, the bounding box itself behaves inconsistently–for built-in widgets like `[hslider]` or `[bng]` it clips (per widget, not per pixel), but for graphed arrays, data structure visualizations, and widget labels it does no clipping at all.

To get to grips with these problems, Purr Data's JavaScript GUI implementation draws and manipulates Pd patch windows using the HTML5 API, which is widely documented and used. The Pd canvas itself is implemented as an SVG document. SVG was chosen because it is a mature, widely-used 2D API. Also, larger canvas sizes have little to no performance impact

on the responsiveness of the graphics. Since Pd patches can be large, this makes SVG a better choice for drawing a Pd canvas than the standard HTML5 canvas.

4.2 Leveraging HTML5 and SVG to Improve Pd Data Structures

Purr Data employs a small subset of the SVG specification to implement quite substantial improvements to data structure visualization. Inheriting from a pre-existing standards-based 2D API has several advantages over an ad-hoc approach. First, if implemented consistently, the existing SVG documentation can be used to test and teach the system. Second, it is not necessary to immediately understand all the design choices of the entire specification in order to implement parts of it. Since those parts have been used and tested in a variety of mature applications, it makes it easier to avoid mistakes that often riddle designs made by developers who aren't graphics experts. Finally, there is less risk of a standards-based API becoming abandoned than a more esoteric API.

To improve data structure visualizations, several `[draw]` commands were added to support the basic shape/object types in SVG. The currently supported types are `circle`, `ellipse`, `rect`, `line`, `polyline`, `polygon`, `path`, `image`, and `g`.[7] Each has a number of methods which map directly to SVG graphical attributes. Methods were also added for Document Object Model (DOM) events to trigger notifications to the outlet of each object.

The screenshot in Fig. 2 shows the "SVG tiger" drawn from a few hundred paths found inside the `[draw g]` object. Even though the drawing is complex, Purr Data caches the bounding box for the tiger object to prevent the hit-testing from causing dropouts. One can mouse over the tiger and trigger real-time audio synthesis.

It is also possible to set parameters for most of the `[draw]` attributes. For instance, the message `opacity z` can be sent to set a shape's opacity to be whatever the value of the field `z` happens to be for a particular instance of the data structure. As soon as the value of `z` changes, Pd then automatically updates the opacity of the corresponding shape accordingly.

[7] The latter `g` element denotes a "group", which is implemented as a special kind of subpatch that allows the attributes of several `[draw]` commands to be changed simultaneously.

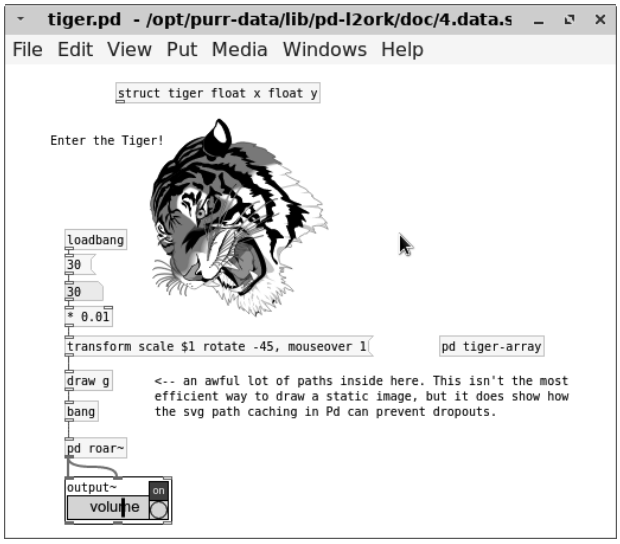


Figure 2: Interactive SVG data structure.

4.3 Custom GUI Elements

As the SVG tiger example shows, Purr Data makes it possible to bind HTML5 DOM events to SVG shapes. Reporting the events is not enabled by default, but can be switched on by simply sending the appropriate Pd message to the `[draw]` object, such as the `mouseover 1` message in Fig. 2. Each `[draw]` object has an outlet which then emits messages when events like mouse-over, movements and clicks are detected.

It goes without saying that this considerably expands Pd's capabilities to deal with user interactions, e.g., if the user wants to modify elements of a graphical score in real-time. But it also paves the way for enabling users to design any kind of GUI element in plain Pd, without having to learn a "real" programming language and its frameworks.

For instance, Fig. 3 shows a collection of three knobs drawn using the new SVG `[draw]` commands, whose values (represented by the `r` field in the `nub` data structure, which is linked to the rotation angles of the knobs) can be manipulated by dragging the mouse up or down. The values can then be read from the data structure using Pd's built-in `[get]` object and used for whatever purpose, just like with any of the built-in GUI elements.

Pd offers a rather limited collection of built-in GUI elements to be used in patches, and extending that collection needs a developer proficient in both C and Tcl/Tk. Purr Data's new SVG visualizations totally change the game, because any Pd user can do them without specialized programming knowledge. We thus expect the

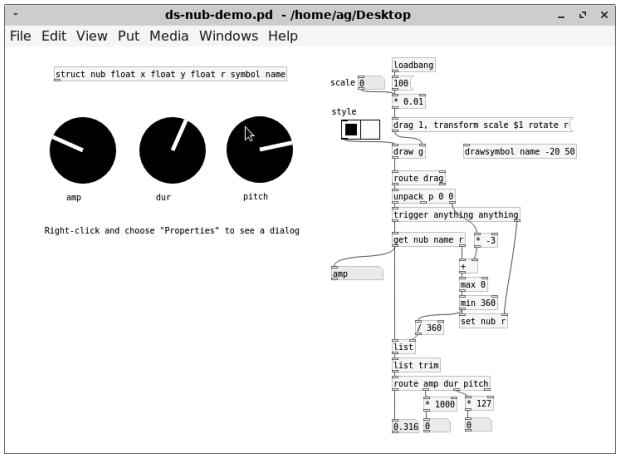


Figure 3: Custom GUI elements.

facilities sketched out above to be used a lot by Pd users who want to enrich their patches with new kinds of GUI elements. As soon as it becomes possible to conveniently package such custom GUI elements as graph-on-parent abstractions, we hope to see the proliferation of GUI element libraries which can then be used by Pd users and modified for their own purposes.

5 Getting Pd-L2Ork

The sources of Pd-L2Ork and Purr Data are currently being maintained in two separate git repositories.[8] There are plans to merge the two repositories again at some point, so that both versions will become two branches in the same repository, but this has not happened yet.

For Purr Data there is also Github mirror available at `https://agraef.github.io/purr-data/`. This is mainly used as a one-stop shop to make it easy for users to get their hands on the latest source and the available releases, including pre-built packages for Linux, macOS and Windows.

Because of Pd-L2Ork's addons and its comprehensive set of bundled externals, the software has a lot of dependencies and a fairly complicated (and time-consuming) build process. So, while the software can be built straight from the source, it is usually much easier to use one of the available binary packages:

- Virginia Tech's official Pd-L2Ork packages are available at `http://l2ork.music.vt.edu/main/make-your-own-l2ork/software/`.
- Jonathan Wilkes' Purr Data packages can be found at `https://github.com/agraef/purr-data/releases`.
- JGU also offers Pd-L2Ork and Purr Data packages for Ubuntu and Arch Linux. Web links and installation instructions can be found at `http://l2orkubuntu.bitbucket.org/` and `http://l2orkaur.bitbucket.org/`, respectively.

The JGU packages can be installed alongside each other, so that you can run both "classic" Pd-L2Ork and Purr Data on the same system. (This may be useful, e.g., if you plan to use Pd-L2Ork's K12 mode which has not been ported to Purr Data yet.) We mention in passing that JGU's binary package repositories also contain Pd-L2Ork and Purr Data versions of the Faust and Pure extensions which further enhance Pd's programming capabilities.[9]

6 Future Work

After Purr Data's initial release as Pd-L2Ork 2.0 in February 2017, "classic" Pd-L2Ork has become version 1.0 and went into maintenance mode. While development will continue on the Purr Data branch, we will keep the original Pd-L2Ork available until all of Pd-L2Ork's features have been ported or have suitable replacements in Purr Data.

Purr Data has matured a lot in the past few months, but like any project of substantial size and complexity it still has a few bugs and rough edges we want to address after the initial release, in particular:

- Port the remaining missing features from Pd-L2Ork (autotips and K12 mode).
- Port legacy Tcl code that is still present in the GUI features of some of the 3rd party externals.
- Some code reorganization is in order, along with a complete overhaul of the current build system.

One interesting direction for future research is leveraging the new SVG visualizations as a means to create custom GUI elements in plain Pd, i.e., as ordinary Pd abstractions. This will make it much easier for users to create their

[8] cf. `https://github.com/pd-l2ork/pd` and `https://git.purrdata.net/jwilkes/purr-data`

[9] Grame's Faust and JGU's Pure are two functional programming languages geared towards signal processing and multimedia applications [7, 8].

own GUI elements, and will hopefully encourage community contributions resulting in libraries of custom GUI objects ready to be used and modified by Purr Data users.

With the expansion onto other platforms, Pd-L2Ork's key challenge is ensuring sustainable growth. As with any other open-source project of its size and scope, this can only be achieved through fostering greater community participation in its development and maintenance, so please do not hesitate to contact us if you would like to help!

7 Acknowledgements

The authors would like to thank the original Pd author Miller Puckette, numerous community members who have complemented the Pd ecosystem with their own creativity and contributions, including Hans Christoph Steiner and Mathieu Bouchard. We would also like to thank the L2Ork sponsors and stakeholders without whose support Pd-L2Ork would have never been possible nor sustainable.

References

[1] P. Brinkmann, P. Kirn, R. Lawler, C. McCormick, M. Roth, and H.-C. Steiner. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention*, volume 291. Citeseer, 2011.

[2] I. Bukvic, T. Martin, E. Standley, and M. Matthews. Introducing L2ork: Linux Laptop Orchestra. In *Interfaces*, pages 170–173, 2010.

[3] I. Bukvic, J. Wilkes, and A. Gräf. Latest developments with Pd-L2Ork and its development branch Purr-Data. PdCon 2016, New York, NY, USA. `http://ico.bukvic.net/PDF/PdCon16_paper_84.pdf`, 2016.

[4] I. I. Bukvic. Pd-L2ork Raspberry Pi Toolkit as a Comprehensive Arduino Alternative in K-12 and Production Scenarios. In *NIME*, pages 163–166, 2014.

[5] I. I. Bukvic, L. Baum, B. Layman, and K. Woodard. Granular Learning Objects for Instrument Design and Collaborative Performance in K-12 Education. In *New Interfaces for Music Expression*, pages 344–346, Ann Arbor, Michigan, 2012.

[6] P. Davis and T. Hohn. Jack audio connection kit. In *Proc. Linux Audio Conference, LAC*, volume 3, pages 245–256, 2003.

[7] A. Gräf. Signal Processing in the Pure Programming Language. In *Proceedings of the 7th International Linux Audio Conference*, pages 137–144, Parma, 2009. Casa della Musica.

[8] A. Gräf. Pd-Faust: An integrated environment for running Faust objects in Pd. In *Proceedings of the 10th International Linux Audio Conference*, pages 101–109, Stanford University, California, US, 2012. CCRMA.

[9] D. Iglesia. MobMuPlat (iOS application). *Iglesia Intermedia*, 2013.

[10] K. Jolly. Usage of pd in spore and darkspore. In *PureData Convention*, 2011.

[11] J. Kincaid. RjDj Generates An Awesome, Trippy Soundtrack For Your Life. `http://social.techcrunch.com/2008/10/13/rjdj-generates`.

[12] C. McCormick, K. Muddu, and A. Rousseau. PdDroidParty-Pure Data patches on Android devices. *Retrieved January*, 21, 2014.

[13] [PD-announce] MacOSX installers for pd 0.36 and pd 0.36 extended (CVS).

[14] pd forks WAS : Keyboard shortcuts for "nudge", "done editing". `http://permalink.gmane.org/gmane.comp.multimedia.puredata.general/79646`.

[15] M. Puckette. Pure Data: another integrated computer music environment. *In Proceedings, International Computer Music Conference*, pages 37–41, 1996.

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

[17] TkPath. `http://tclbitprint.sourceforge.net/`.

[18] Unity - Game Engine. `https://unity3d.com`.

[19] B. B. Welch. *Practical programming in Tcl and Tk*, volume 3. Prentice Hall Upper Saddle River, 1995.

Posters / SpeedGeeking

Impulse-Response and CAD-Model-Based Physical Modeling in Faust

Pierre-Amaury Grumiaux[1], Romain Michon[2], Emilio Gallego Arias[1], and Pierre Jouvelot[1]
[1]MINES ParisTech, PSL Research University, France
[2]CCRMA, Stanford University, USA

Abstract

We present a set of tools to quickly implement modal physical models in the FAUST programming language. Models can easily be generated from any impulse response or 3D graphical representation of a physical object.

This system targets users with little knowledge in physical modeling and willing to use this type of synthesis technique in a musical context.

Keywords

Physical Modeling Synthesis, Faust Language, Digital Signal Processing

1 Introduction

The FAUST programming language has proven to be well suited to implement physical models of musical instruments [Michon and Smith, 2011] using waveguides [Smith, 2010] and modal synthesis [Adrien, 1991].

In this short paper, we present two Python scripts[1] allowing to easily generate FAUST modal physical models: `ir2dsp.py` and `mesh2dsp.py`.

- `ir2dsp.py` takes an audio file containing an impulse response as its main argument and converts it into a FAUST file implementing the corresponding modal physical model.
- `mesh2dsp.py` outputs the same type of model but takes an `.stl`[2] file containing the specification of any 3D object designed with a CAD[3] program as its main argument.

FAUST programs generated by `ir2dsp.py` and `mesh2dsp.py` are ready to use and can be compiled to any of the FAUST targets (standalone applications, plug-ins, etc.).

After briefly describing these two tools, we'll evaluate them and provide directions for future works.

2 Faust Modal Physical Model

Any linear percussion instrument can be implemented using a bank of resonant bandpass filters [Smith, 2010]. Each filter implements one mode (a sine or cosine function) of the system and can be configured by providing three parameters: the frequency of the mode, its gain, and its resonance duration (*T60*).

Such a filter can be easily implemented in FAUST using a biquad filter (`tf2`) and by computing its poles and zeros for a given frequency (`f`) and *T60* (`t60`):

```
modeFilter(f,t60) = tf2(b0,b1,b2,a1,a2)
with{
      b0 = 1;
      b1 = 0;
      b2 = -1;
      w = 2*PI*f/SR;
      r = pow(0.001,1/float(t60*SR));
      a1 = -2*r*cos(w);
      a2 = r^2;
};
mode(f,t60,gain) =
      modeFilter(f,t60)*gain;
```

The `modeFilter` function can be easily applied in parallel in FAUST using the `par` operator to implement any modal physical model:

```
model =
      _ <:
      par(i,nModes,
          mode(freq(i),t60(i),gain(i)))
      :> _;
```

The FAUST-generated block diagram corresponding to this code, with `nModes = 4`, `freq(i)= 100*(i+1)`, and `(t60(i),gain(i))` as successively (0,9,0.9), (0.8,0.9), (0.6,0.5) and (0.5,0.6), can be visualized in Figure 1.

This type of model can be easily excited by a filtered noise impulse (see Figure 2). The cut-off frequency of the lowpass and highpass filters

[1]`https://github.com/rmichon/pmFaust/` – All URLs in this paper were verified on 07/04/2017.
[2]STereoLithography.
[3]Computer-Aided Design.

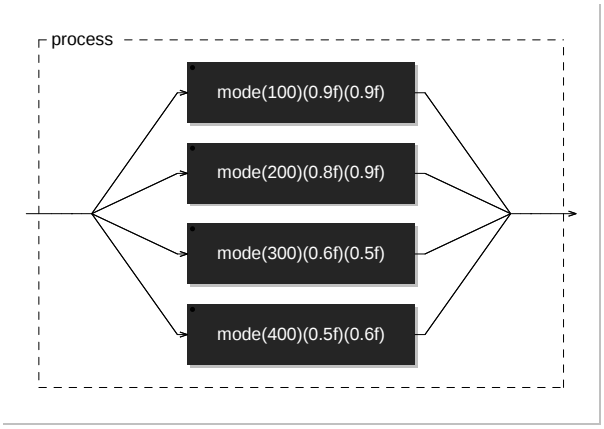


Figure 1: Block diagram of a FAUST modal physical model.

can be used to excite specific zones of the spectrum of the model and to choose the "excitation position." Since this system is linear, the same behavior could be achieved by scaling the gain of the different modes, but the filter approach that we use here will better integrate to our modular physical modeling synthesis toolkit, briefly presented in §6.

Figure 2: Excitation generator algorithm used to drive our modal physical models.

3 ir2dsp.py

`ir2dsp.py` takes an audio file containing an impulse response as its main argument. After performing the Fast Fourier Transform (FFT) on it, modes information is extracted by carrying out peaks detection. The *T60* of each mode is computed by measuring its bandwidth at -3 dB.

Modes information is formatted by `ir2dsp.py` to be plugged to a generic modal FAUST physical model similar to the one described in §2. The output of the Python program is a ready-to-use FAUST file implementing the model.

The goal of this tool is not to create very accurate models but rather to be able to strike any object (e.g., a glass, a metal bar, etc.), record the resulting sound, and turn it into a playable digital musical instrument.

4 mesh2dsp.py

The output of `mesh2dsp.py` is the same as `ir2dsp.py` (see §3), but it takes a `.stl` file as its input instead of an impulse response. `stl` is a common format supported by most CAD programs to export the description of 3D objects.

After converting the provided `.stl` file into a mesh, `mesh2dsp.py` performs a Finite Element Analysis (FEA) using Elmer [4] Various parameters such as the Young Modulus, the Poisson Coefficient, and the density of the material of the object must be provided to carry out this task.

The result of the analysis is a set of eigenvalues and mass participations for each mode. Eigenvalues are then converted to mode frequencies and mass participations to mode gains. Unfortunately, this technique doesn't allow to calculate the *T60* of the modes which can be configured by the user directly from the FAUST program.

5 Evaluation

To evaluate the accuracy of `ir2dsp.py`, we recorded the impulse response of a can and generated its corresponding modal physical model. Figure 3 shows the spectrogram of the impulse response of the can and Figure 4 the spectrogram of the impulse response of the physical model generated by `ir2dsp.py`. `ir2dsp.py` was configured to detect peaks at a minimum value of -20 dB and at least 100 Hz spaced from each other. We see that the synthesized sound is pretty close to the recorded version. *T60*s are not perfectly accurate since they were calculated by measuring the bandwidth of the mode. Tracking their evolution in the time domain would provide better results; thus we plan to use this technique in the future instead.

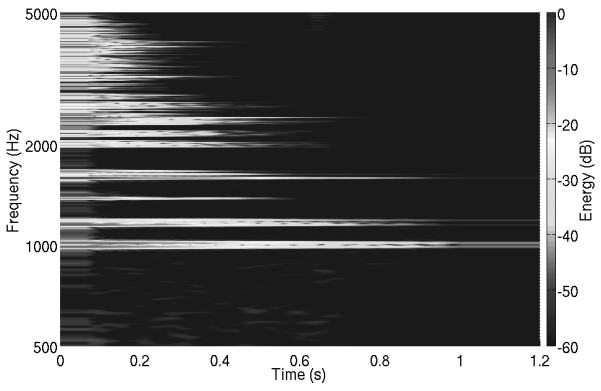


Figure 3: Spectrogram of an impulse response of a can

[4] `https://www.csc.fi/web/elmer/`

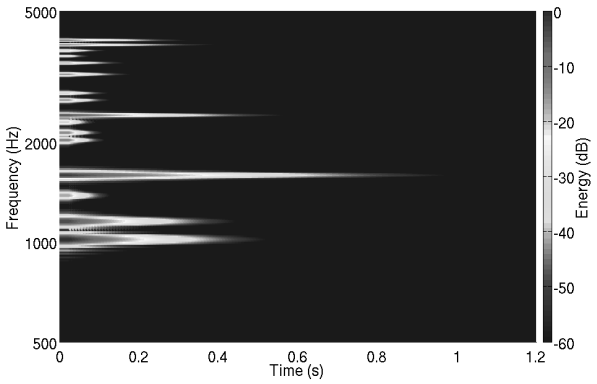


Figure 4: Spectrogram of the output of the modal model generated with `ir2dsp.py` from a can IR

`mesh2dsp.py` was tested with the geometric 3D model of a solid bar and provided good subjective auditory results. More objective analysis is clearly needed here.

6 Future Work

This work has been carried out as part of a larger project on designing a physical modeling toolkit for the FAUST programming language. `ir2dsp.py` and `mesh2dsp.py` will be integrated to it.

We plan to improve `ir2dsp.py` by using a better *T60* measurement algorithm. Indeed, the *T60* of each mode is currently computed by measuring its bandwidth after taking the FFT of the entire impulse response. A better approach would be to extract this information from a time-frequency representation of the signal (i.e., spectrogram), which would be more accurate.

Finally, we would like to try other open-source packages than Elmer to carry out the FEA in `mesh2dsp.py` to get better results and to smooth its integration in our FAUST physical modeling toolkit.

7 Conclusion

We presented a series of prototype tools allowing to design at a very high level ready-to-use physical models of musical instruments. Models can be generated from impulse responses or 3D graphical representations of physical objects.

While the models generated by this system are far from being accurate, we believe that it provides a convenient way for composers and musicians to design expressive custom instruments usable in a musical context.

8 Acknowledgements

Our thanks go to Yann Orlarey for his help with the use of Faust.

References

Jean-Marie Adrien. 1991. The missing link: Modal synthesis. In *Representations of Musical Signals*, chapter The Missing Link: Modal Synthesis, pages 269–298. MIT Press, Cambridge, USA.

Romain Michon and Julius O. Smith. 2011. Faust-STK: a set of linear and nonlinear physical models for the Faust programming language. In *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, pages 199–204, Paris, France, September.

Julius Orion Smith. 2010. *Physical Audio Signal Processing for Virtual Musical Instruments and Digital Audio Effects*. W3K Publishing.

Fundamental Frequency Estimation for Non-Interactive Audio-Visual Simulations

Rahul AGNIHOTRI, Romain MICHON and Timothy S. O'BRIEN
CCRMA (Center for Computer Research in Music and Acoustics),
Stanford University,
Stanford, CA, 94305
{ragni, rmichon, tsob}@ccrma.stanford.edu

Abstract

This paper aims to demonstrate the use of fundamental frequency estimation as a gateway to create a non-interactive system where computers communicate with each other using musical notes. Frequency estimation is carried out using the YIN algorithm, implemented using the ofxAubio add-on in openFrameworks. Since we have used only open-source technologies for the implementation of this project, it can be executed on any platform: Linux, Macintosh OS or Windows OS. As a simulation, this system is used to depict a conversation between people at a round-table event and presented as an audiovisual art installation.

Keywords

Pitch Estimation, MIR, YIN algorithm, Aubio, Audio Simulation.

1 Introduction

Pitch detection algorithms have been used in various contexts in the past:

- audio editing programs (pitch correction and time scaling) such as Melodyne[1],
- analysis of complicated melodies of world music cultures (Indian Classical music),
- music notation programs like Sibelius[2],
- MIDI interfaces such as the Roland GI-20 to get data from guitar MIDI pickups.

Since it lies firmly within the domain of music information retrieval (MIR), pitch estimation has many applications in recommender systems, sound source separation, genre categorization and even music generation. With the popularity of machine learning, neural-networks, and data mining, audio signal processing tools are utilized more and more to create user-specific systems. When considering applications for pitch detection and source separation, we often consider the example of identifying individual speakers at a round-table event. Computers, unlike humans, have a difficult time identifying the words of a particular person if multiple people are communicating with each other simultaneously. Building on that concept, we simulate a conversation between multiple computers using musical notes. The YIN pitch detection algorithm is employed to detect trigger notes, to which other computers in the network respond.

Since we desire a continuous conversation, we must use an efficient detection algorithm with the following features:

- real time response,
- minimal latency,
- accurate identification in the presence of noise.

We need to be careful about both the latency of the attack and of the pitch detection algorithm since if a note is played, the human ear needs at least seven periods of a waveform to identify its pitch. Hence, note onsets and note pitches are not directly related [3]. Additionally, we must ensure that the pitch recognition algorithm is reasonably robust to the sort of noise which is inevitable in a performance scenario.

After comparing different methods for pitch estimation, as in [3], we chose the YIN algorithm for its real-time tracking ability. YIN is a time-domain algorithm based on the autocorrelation method for estimation. [2]. Using the common autocorrelation method, its error rates are analyzed and corrected for every new iteration to ensure the best possible accuracy. Using YIN is beneficial since it can accurately analyze higher frequencies which we might use as trigger notes in our system. To make sure that we have the lowest possible pitch identification latency and have a very small frame size for incoming

[1] `http://www.celemony.com/en/melodyne/what-is-melodyne`. URLs in this paper were verified on Feb. 16, 2017.

[2] `http://www.avid.com/sibelius`

audio, we use the YIN algorithm implemented in the aubio framework [1], extended further as an addon in openFrameworks[3] for our computer network.

2 Methods and Implementation

Since fundamental frequency identification works well on monophonic sounds (e.g., a guitar solo or any wind instrument), it was decided to use this approach to reduce the problem of incorrect triggers for the other computers in the network. Individual notes are generated one after the other at randomized tempos (to mimic the prosody of human speech) in a particular musical scale. Every other computer has its own "voice" which does not overlap with that of any other computer in the network. At a given point in time, multiple "people" can "speak" simultaneously.

The YIN algorithm is accurate enough to sufficiently identify the trigger notes at any given time within the chaotic yet pleasing tone of the conversation, and the computers react accordingly. Visual feedback is provided to portray whether a computer is voicing itself or is remaining silent in response to the trigger note. For the test system, three computers, each with their own voice, constituted the network.

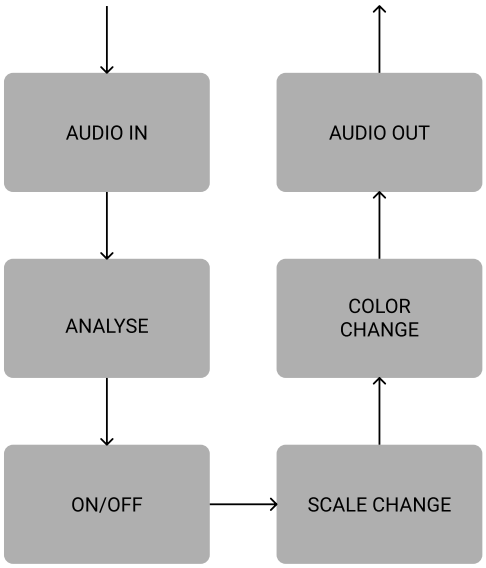


Figure 1: Overview of the system

2.1 Audio

Each voice for the "speaker" in the system is a musical scale. For our demonstration, each computer was set to the G, C and D Major scale respectively. The choice for these particular scales was arbitrary and baseless. The `ofxStk`[4] add-on was used to generate the sounds using the `Moog` synthesis class. The generated sound has a reverb effect applied to it in order to create a wider stereo image when all computers are in place.

In order to mimic the characteristics of human speech, there is no fixed tempo for the system. When a particular trigger note is heard by a computer, it goes silent and shifts its scale by an octave higher or lower and also changes its corresponding trigger note in order to keep the whole system ambiguous. The ambiguity lies in the fact that when all computers are communicating simultaneously, it is difficult to identify what the actual trigger note is and maintains the illusion of an improvised conversation. The exact flow of the network is represented in Figure 1. This flow remains constant for any new computer added to the system.

2.2 Graphics

Graphical feedback is used to convey whether a particular computer is active or silenced. We used the `ofxParticles` add-on to implement particle physics in order to visually represent the current state of our system.

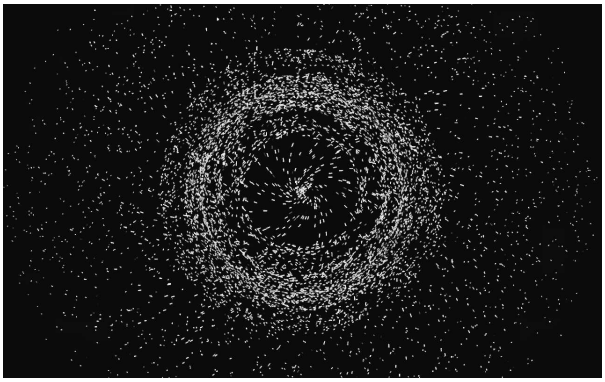


Figure 2: Screenshot of the particle physics graphics

White particles at any given instant represent an "in-active state" of the computer, and colored particles are used when the computer is active (as shown in Figure 2). The color of the particles for each computer changes when it becomes active after being inactive for a random amount of time, as if joining the conversation

[3] See `https://github.com/aubio/ofxAubio` and `http://openframeworks.cc/`

[4] `https://github.com/Ahbee/ofxStk`, which encapsulates the original Synthesis ToolKit, `https://ccrma.stanford.edu/software/stk/`

again to state its opinion. Particles are emitted from the exact center, spiral outward, eventually fade out. There is a pseudo-gravitation effect that makes the particles orbit around the center of the screen after their spiral trajectory. We designed this visual tool to create a psychedelic effect for the audience.

2.3 Evaluation

When presenting this system to a group of observers it was noticed that despite having an underlying pattern to the changes in scale and trigger conditions, they could not detect this and the audience expressed that they believed the computers were having a conversation, albeit through musical notes. The audience also responded that having a visual feedback gave each computer a unique personality.

3 Future Work

The future scope of this project involves the integration of machine learning in order to implement musical improvisations as a response to the trigger conditions. This would make the whole system more expressive (i.e., trivial actions such as having the computer go silent or be active, etc.). Polyphonic sound identification is also a viable addition to have a more immersed experience of musicians improvising with one another.

4 Conclusion

In this paper we presented pitch estimation as a tool for a musical performance system. Since the future scope does involve the use of machine learning and data mining techniques, as is the custom with music information retrieval, this was a relevant stepping stone. Presently, this system is being modified to include multiple triggers for the whole system. We are trying to move away from the initial condition of having only one computer respond to a single trigger note but have multiple computers react to more than one trigger added into the network.

5 Acknowledgements

Many thanks to the CCRMA community for its support.

References

[1] P Brossier. Automatic Annotation of Musical Audio for Interactive Applications. *Centre for Digital Music Queen Mary University of London*, Diploma of(August):215, 2006.

[2] Alain de Cheveigné and Hideki Kawahara. YIN, a fundamental frequency estimator for speech and music. *The Journal of the Acoustical Society of America*, 111(4):1917–1930, 2002.

[3] P De La Cuadra. Efficient pitch detection techniques for interactive music. *International Computer Music Conference*, pages 403–406, 2001.

Porting WDL-OL to LADSPA/LV2

Jean-Jacques Girardot
Independent programmer
25 Rue Pierre Bérard
42000 Saint-Etienne, France
jj@girardot.name

Abstract

WDL-OL is an open source framework that is used to develop audio plug-ins in various formats (VST, VST3, AU, RTAS, AAX) for Mac and Windows operating systems. The proposition is to add the possibility to develop plug-ins in LADSPA and LV2 formats under Linux.

Keywords

Audio Plug-ins, WDL-OL, LADSPA, LV2

1 Introduction

WDL / IPlug is a simple-to-use C++ framework for developing cross platform audio plugins and targeting multiple plugin APIs with the same code. Originally developed by Schwa/Cockos, IPlug has been enhanced by various contributors, in particular Oliver Larkin, whose version seems to be the most used.

Plug depends on WDL, and that is why this project is called WDL-OL, although most of the differences from Cockos' WDL are to do with IPlug. The source code for the framework can be downloaded from the WDL repository [1].

There exists also a very active discussion list about WDL [2]

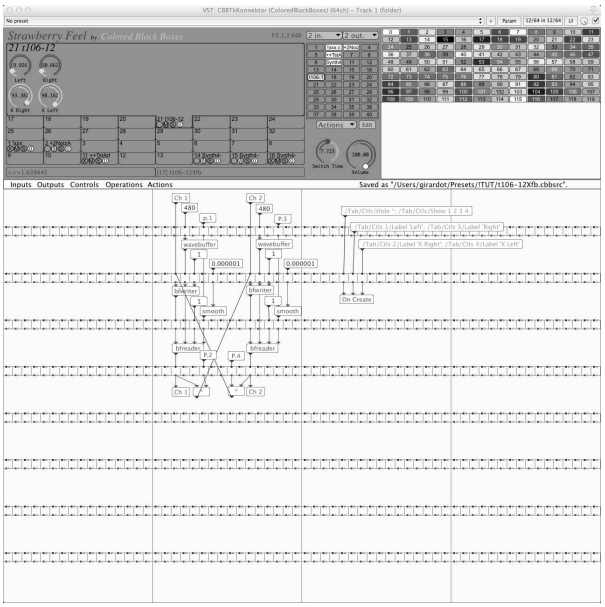
2 The Geek Meeting

The author has developed a few freeware plug-ins (pictured here: *ClockWise Orange* which manages multi tap multi delays, and *Strawberry Feel* which provides a graphical audio language) using WDL-OL, and he wishes to make them available to the Linux community. Some other plug-ins authors may whish to do the same thing, and therefore add to the plug-ins offer under Linux. For this, we need to add to WDL-OL the LADSPA/LV2 support (for DAWs, but also for tools like ChucK), and to find people knowlegeable on the subject and willing to help us in this development.

References

[1] WDL Git Repository: https://github.com/olilarkin/wdl-ol

[2] WDL discussion Forum: http://forums.cockos.com/forumdisplay.php?f=32



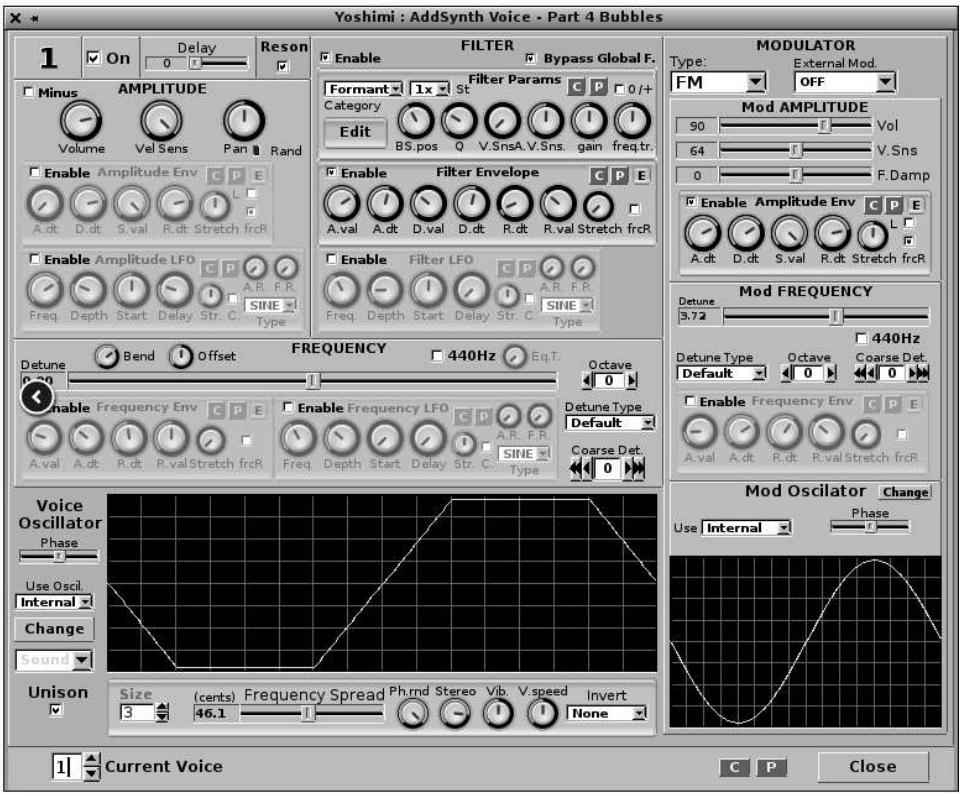
Workshops

Workshops

1-*Yoshimi and the reluctant developer by* Will Godfrey (United Kingdom)

A workshop overview of my involvement with the Yoshimi soft-synth, discussion and current status, including demonstrations

See https://sourceforge.net/projects/yoshimi/



2. *Free Software and DIY at Radio Panik* by Frederic Peters, Arthur Lacomme et Suzie Suptille (Belgium)

Radio Panik is a community FM-radio created in Brussels in 1983, it has been using, adapting and creating free software for much of its activity for years. This workshop aims to explore and discuss our current audio practices, from broadcast systems to creative tools.

See http://www.radiopanik.org/

3. *Building a local linux audio community* by Daniel Appelt (Germany)

The Open Source Audio Meeting Cologne is a monthly gathering of audio and free software enthusiasts. This workshop provides insights how such a regular event may be organized.

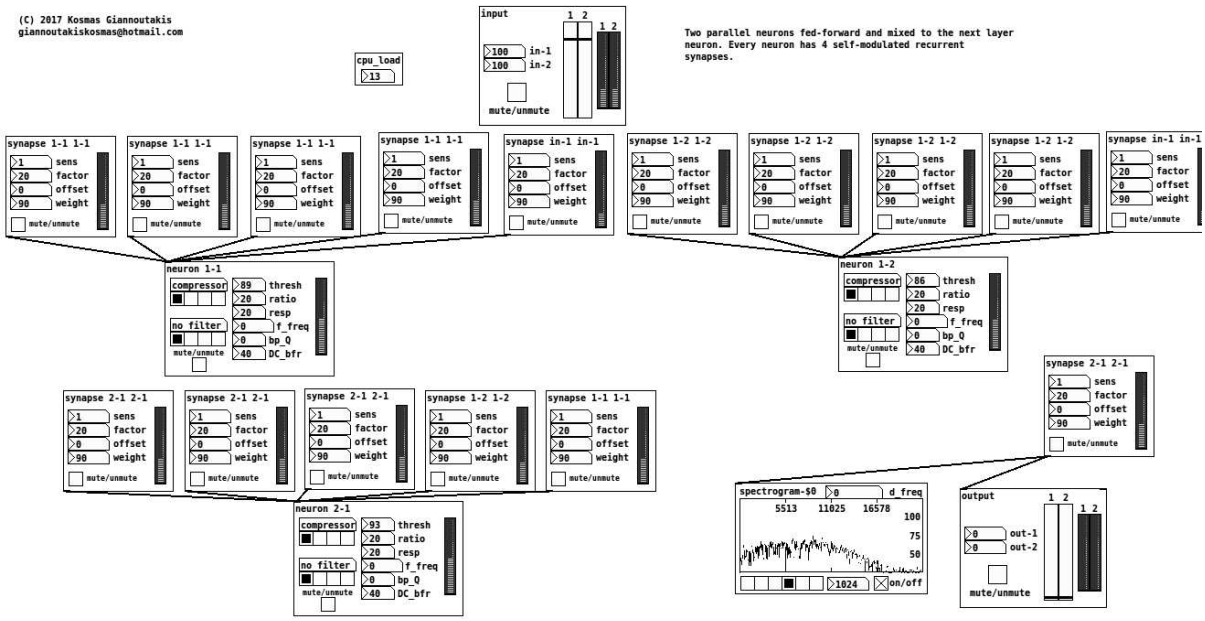
See http://cologne.linuxaudio. org

4. *Generative Music with Recurrent Neural Networks* par Kosmas Giannoutakis

(Institut für Elektronische Musik und Akustik Inffeldgasse, Graz, Austria)

In this workshop the generative music capabilities of artificial recurrent neural networks will be explored, using an abstractions library for the programming environment Pure Data, called RNMN (Recurrent Neural Music Networks). The library provides the basic building blocks, neurons and synapses, which can be arbitrarily connected, easily and conveniently, creating compound topologies. The framework

allows real time signal processing, which permit direct interactions with the topologies and quick development of musical intuition. For the workshop, a laptop with a build-in microphone, Pure Data (PD-vanilla 0.47.1 version is recommended) installed and headphones is required for the participants. Experience with visual programming is a plus but not a necessary prerequisite. It will be explained the basic principles of the framework and it will be demonstrated the construction of some basic topologies. In the end the participants can create their own topologies which can demonstrate to the other participants.



5. *Origin, features and roadmap of the MOD* Duo* by Mauricio Dwek, Gianfranco Ceccolini & Filipe Coelho (Germany)

(* Musical Operating Devices For Experienced Musicians)

In this workshop, MOD Devices tells its story and shows its heavy use of Linux Audio technologies for the MOD Duo.

The workshop will consist of:

- History on how the MOD Duo came to be
- What (Linux Audio) technologies are used inside
- Challenges and difficulties found while making the Duo
- Showing the Duo in action
- Things to come soon

The audience will be encouraged to get their hands on the device and try out its features.

See https://moddevices.com/

6. *Too Much Qstuff* To Handle* by Rui Nuno Capela (Portugal)

Following in the tradition of LAC2013@IEM-Graz, LAC2014@ZKM-Karlsruhe, LAC2015@JGU-Mainz and miniLAC2016@c_base-Berlin, this talk/workshop is once again being proposed as an informal opportunity for open debate and discussion, over the so called Qstuff* software constellation. Although starring Qtractor [4] as the main subject, all users and developers are welcome to attend, whether or not they're using any of the Qstuff*. An all-inclusive talk/workshop.

The Qstuff* are, in order of appearance:

[1] QjackCtl - A JACK Audio Connection Kit Qt GUI Interface
http://qjackctl.sourceforge.net
https://github.com/rncbc/qjackctl

[2] Qsynth - A fluidsynth Qt GUI Interface
http://qsynth.sourceforge.net
https://github.com/rncbc/qsynth

[3] Qsampler - A LinuxSampler Qt GUI Interface
http://qsampler.sourceforge.net
https://github.com/rncbc/qsampler
https://github.com/rncbc/liblscp

[4] Qtractor - An audio/MIDI multi-track sequencer
http://qtractor.org
http://qtractor.sourceforge.net
https://github.com/rncbc/qtractor

[5] QXGEdit - A Qt XG Editor
http://qxgedit.sourceforge.net
https://github.com/rncbc/qxgedit

[6] QmidiNet - A MIDI Network Gateway via UDP/IP Multicast
http://qmidinet.sourceforge.net
https://github.com/rncbc/qmidinet

[7] QmidiCtl - A MIDI Remote Controller via UDP/IP Multicast
http://qmidictl.sourceforge.net
https://github.com/rncbc/qmidictl

[8] synthv1 - an old-school polyphonic synthesizer
http://synthv1.sourceforge.net
https://github.com/rncbc/synthv1

[9] samplv1 - an old-school polyphonic sampler
http://samplv1.sourceforge.net
https://github.com/rncbc/samplv1

[10] drumkv1 - an old-school drum-kit sampler
http://drumkv1.sourceforge.net
https://github.com/rncbc/drumkv1

7. *Moony – rapid prototyping of LV2 (MIDI) event filters in Lua* by Hanspeter Portner (Switzerland)

In need of a specific event filter no yet existent for your DAW or live setup? No time or skill to write your own MIDI plugin in C/C++? Moony comes to the rescue and lets you script your filters in Lua on-the-fly for any LV2 host. Come and learn about the LV2 atom event system and rapid prototyping in Lua.

See https://open-music-kontrollers.ch/

8. *Interactive music with i-score* by Jean-Michaël Celerier

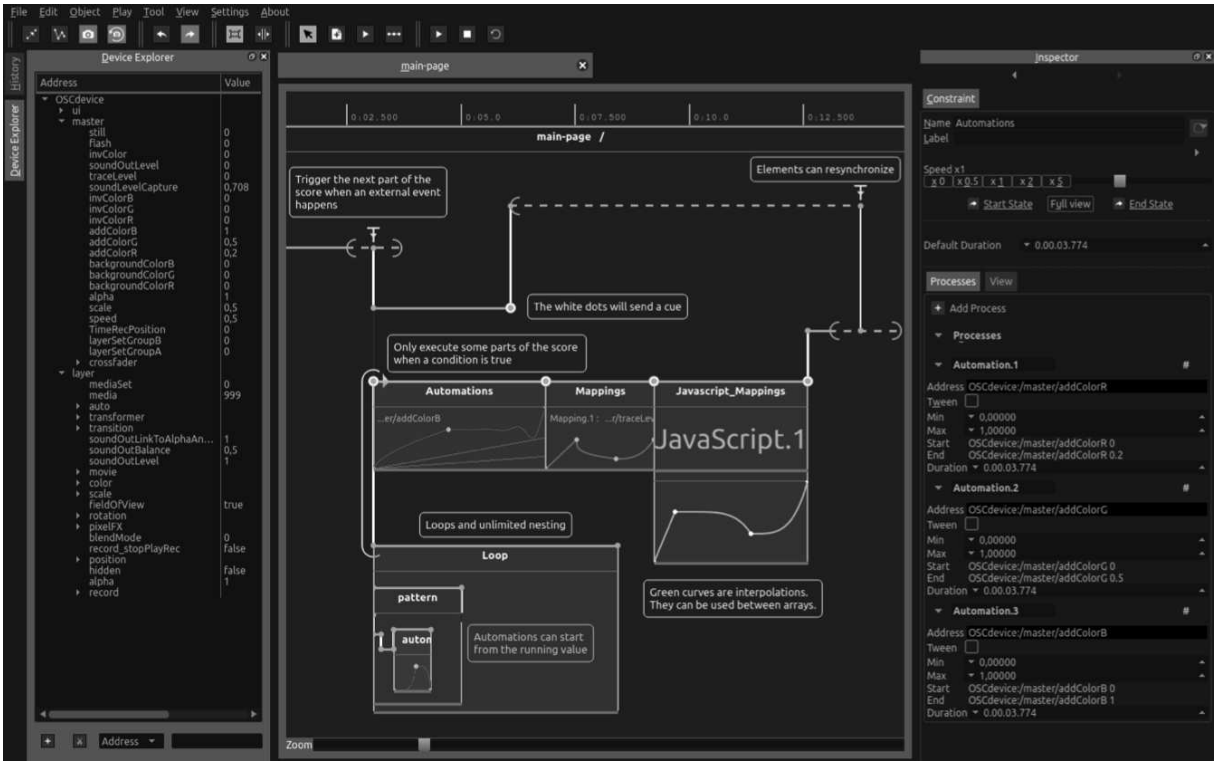
(Laboratoire Bordelais de Recherche en Informatique, France)

This workshop presents the i-score (www.i-score.org) sequencer.

It will present the challenges and the rationale that led to the creation of the software, that is, providing a dedicated tool for temporal design in an interactive context.

The construction of a score will be detailed on practical examples involving audio-visual features and interaction with a familiar creative coding environment such as PureData, openFrameworks or Processing.

See http://www.i-score.org



9. *Smartphone Passive Augmentation* by John Granzow* & Romain Michon**

(University of Michigan - United States, ** Stanford University - United States)*

In this 4 hours workshop (2 sessions) we will present Mobile3D, a library for introducing passive musical augmentations to mobile phones. The library allows participants to quickly leverage the parametric features of OpenScad a functional programming language for text based computer assisted drawing (CAD). Several 3D printers will be on hand to materialize designs. The workshop gives participants the tools to customize their smartphones for musical interaction.

See https://ccrma.stanford.edu/~rmichon/mobile3D/

Concerts

Concerts

Thursday, May 18 – 06:30pm – 08:00pm - Auditorium de la Maison de l'Université – 10 rue Tréfilerie – Saint-Etienne

Concert N°1: Mixed Music

It is a first concert featuring works of mixed music (music with acoustic instruments and electronic devices that interacts). These works are pieces whose electronic devices have been developed with real-time free software (FAUST and SuperCollider). They are played by regional musicians and students from Saint-Etienne.

1. *SmartMachine musicale* (20' – 2017 - Création)

With the students of La Salle secondary school (Saint-Etienne), Robert Chauchat, music teacher, Roméo Monteiro, percussionist, solist from Ensemble Orchestral Contemporain and Gérard Authelain, teacher and musician (GRAME).

Guided by their teacher and musicians, the students have explored the relationship between the technical object and the musical creation, but also the musical gesture (using the Smartfaust apps for smartphone designed by GRAME, National Center of Music Creation in Lyon). They have gradually created a new kind of sound scene, halfway between a performance and a visual installation. They invite you tonight to discover a space of research, questioning the aesthetic of the concert, the musician's attitude and his instrument!

Project conducted by a partnership between GRAME and Ensemble Orchestral Contemporain, with the support from DRAC Auvergne – Rhônes-Alpes and the Department of Loire.

2. *Smartbones* (20' - 2016)

With brass section students of the Conservatory of Valence (Léane Berthaud, Alice Chakroun, Noam Leenhardt, Rami Leenhardt, Meryem Ouannas, Antonin Vinay) and Pierry Bassery, musician and teacher.

In order to arouse the curiosity and the listening, the students of the conservatory of Valence, guided by the trombonist Pierre Bassery, combine repertoire, improvisation, creation and new technologies with pieces for trombone/tuba. The incorporation of smartphones on the instruments allows the musicians to play with Smartfaust apps as the continuation of the instrumental gesture. Thus these new instruments ("Smartbones"- smartphones & trombones) will generate a sound material that will lead the students to imagine choreographies in which dance, trombone and electro will be mixed and confronted.

3. *Peyote* (16' - 2016) by Sébastien Clara, mixed music for trio & electronics

With Alice Szymanski, flute, Justine Eckhaut, piano and Florent Coutanson, saxophones.

In 1936 Antonin Artaud left Europe to travel to Mexico. This departure symbolizes his break with surrealist aesthetics. "The rationalist culture of Europe has gone bankrupt and I have come to the land of Mexico to seek the bases of a magical culture that can still spring from the forces of Indian soil." However, by a romantic desire to touch the bottom or a personal inner experience to better understand his fellows, Artaud undertakes "a descent to come out of the day", a journey within his journey, a transcendental abysm.

Artaud sets out to research "the ancient solar culture". To do this, he wants to be introduced to the culture of the men and women of the Sierra Tarahumara.
Peyote retraces the dances of the Tarahumara to the glory of the sun, which undeniably influenced the work of Antonin Artaud.

Friday, May 19 – 08:00pm – 12:00pm - Auditorium de la Maison de l'Université – 10 rue Tréfilerie – Saint-Etienne

Concert N°2: Electronic Music

It is a concert of electronic music with musicians coming from the USA or from various European countries. They will play works, mostly experimental, realized with diverse and innovative digital devices, all involving Open Source software.

1. *Inaudible Harp* (10') by Bruno Ruviaro & Juan-Pablo Caceres

(Santa Clara University - USA)

Music for harp, distant, played via Internet, and a computer system for real time sound synthesis and processing.

One of "origin tales" of Ambient Music has Brian Eno stuck in a hospital bed after an accident: lying immobile in bed, he would listen to records played by visiting friends. One day it was harp music, with the volume turned so low that the plucked strings were almost inaudible. "At first I thought, 'Oh God, I wish I could turn it up," Eno remembers. "But then I started to think how beautiful it was. It was raining heavily outside and I could just hear the loudest notes of the harp coming above the level of the rain." Our telematic duo improvisation is inspired by this image.

The duo utilizes SuperCollider to generate and process sounds, and JackTrip to stream audio over the internet. JackTrip is a Linux and Mac OS X-based system used for multi-machine network performance over the Internet. It supports any number of channels (as many as the computer/network can handle) of bidirectional, high quality, uncompressed audio signal streaming. The duo usually performs with one player on location, and the other remotely from Chile or the USA.

2. *Vox Voxel* (12' – 2015) by Fernando Lopez-Lezcano* & John Granzow**

(Center for Computer Research in Music and Acoustics - Stanford University - USA, ** University of Michigan - USA)*

Music for two interpreters, a 3D printer, a Daxophone, a Korg Nanokontrol 2 and a computer.

From an IBM 720 line printer playing *Three Blind Mice* in 1954 to dot matrix printers playing love songs and Queen, mechanical noises coming from printers were slowly tamed, domesticated and controlled, and countless unproductive hours of programming time were spent in figuring out how to make those noises into musical notes, phrases and whole pieces for the enjoyment of the IT team. From deafening antique mainframe line printers to whisper quiet inkjets, all have been at the spotlight of a concert performance (or at least a basement computer room).

VoxVoxel is "composed" by designing a suitably useless 3D shape and capturing the sound of the working 3D printer using piezoelectric sensors. Those sounds are amplified, modified and multiplied through live processing in a computer using Ardour and LV2/LADSPA plugins, and output in full matching 3D sound. 3D pixels in space.

The piece is dedicated to our endangered wooden 3d printer, slowly declining with the rise of folded metal frames in entry-level machines. The wood, (if fragile) is good for

contact one view of an object and score for the piece vibrations, to amplify rhythms of the tool-path and the frequencies of stepper motors. This rare 3d printer takes six minutes to warm up its extruder. For this, it has also fabricated an array of extensions for its equally endangered human performer.

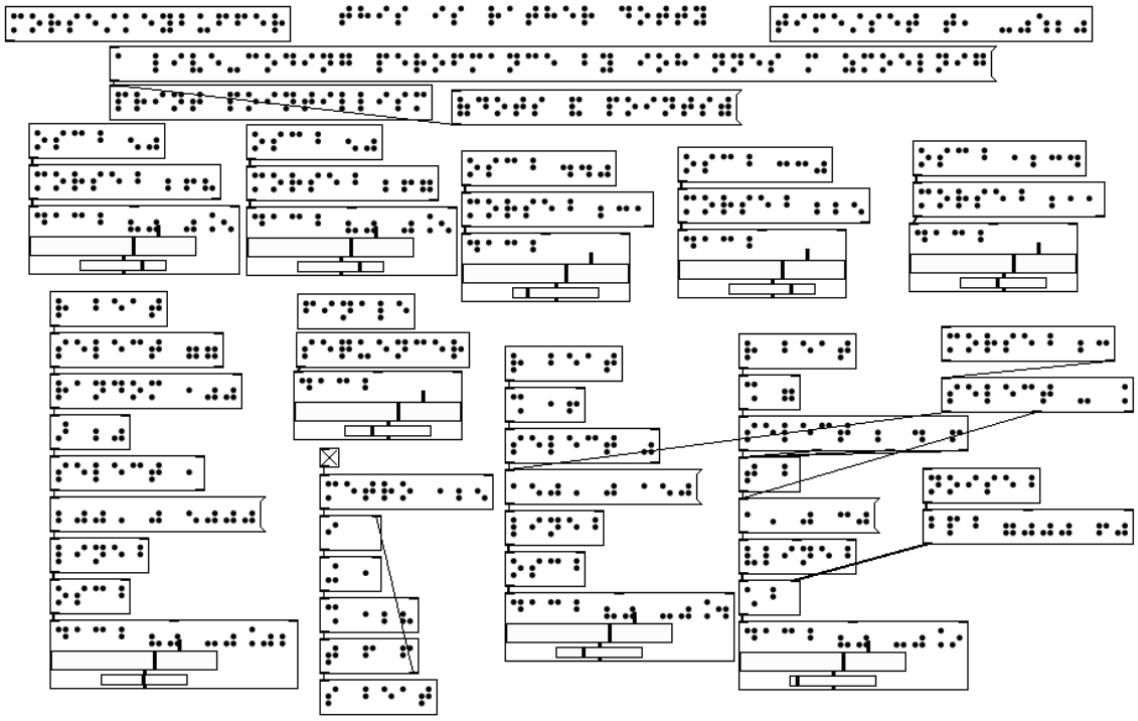
Sound Symposium 2016 performance

3. *Pointillism* (20' – 2016) by Iohannes Zmolnig

(Institute of Electronic Music and Acoustics - University of Music and Performing Arts - Graz – Austria)

Pointillism is a solo live-coding performance in Pd. Both the code representation and the generated audio use "points" and "dots" as building blocks: the music is generated using morse code patterns, the code is written in "Braille" (the dot--based writing system especially designed for blind/visually-impaired people).

The performance is an ironic statement on the popular "show--us--your--screens" paradigm, by presenting the code in a form that does not even pretend to be readable.



4. *Seven Sphere Journey* (12' - 2016) by Broch Vilbjorg

(Anti-Delusion Mechanism - Netherlands)

Music for computer generated sound, voice, dance and video projection

This project explores generative sound, graphics and algorithmic composition based on the octonion algebra. The octonions form an 8 dimensional normed division algebra. The project started 2016 and and is in ongoing development. The octonions are an extremely rich subject within mathematics, with symmetry relations to Lie algebras and not the least to the so called special Lie groups.

The unit-length octonions trace the seven-sphere: S7 - a 7 dimensional surface in 8 dimensional space. The project explores orbits in 8 dimensional space for audio synthesis and graphics.

5. *1)3V1532* (25' - 2012) by David Runge

(Elektronic Studio, TU Berlin & c-base - Germany)

1)3\/1532 (deviser) is drone/noise/experimental soundscaping.

The aural journey leads to places like simple repetitve guitar tunes, loops, feedback manipulation, modified samples of field recordings, DIY analog synthesizers, toys and the like.

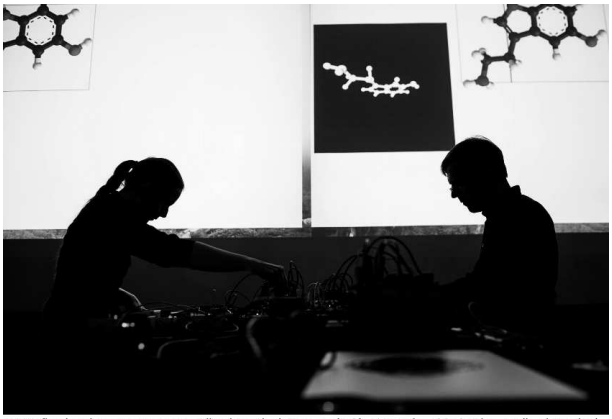
Experimentation for the greater good!

6. *5-HT_five levels to zero* (28' – 2016) by Tina Mariane Krogh Madsen & Malte Steiner

(Block4 - Germany)

For Linux computer with Pure Data, synthesizer, div pedals and instruments for noise

The concert *5-HT_five levels to zero* is based on the structural qualities of the neurotransmitter serotonin. The dynamics of the molecule will be improvised and performed live in a dynamic and counterbalanced noise act that deals with both the balances as well as the imbalances inherent, resulting in chaotic states caused by disruption of this unit in the brain. For the piece, TMS has created a score that captures the dynamics of the musical composition, where the audio will be accompanied by a projection of a visual score created in Blender and Pure Data.



5-HT_five levels to zero concert, Piksel Festival, Bergen (NO), November 2016. Photo: Piksel Festival

5-HT_five levels to zero concert, Piksel Festival, Bergen (NO), November 2016. Photo: Piksel Festival

7. *Level 5 Alert* (30' - 2016) by Frederic Peters, Arthur Lacomme and Suzie Suptille

(Radio Panik - Belgium)

Live performance; the show was created after 2016 Brussels bombings as an independent and uncontrolled mean of expression, as a recurrent collective work mixing experimental and creative writing, music and sound effects.

8. *The Infinite Repeat* (22') by Jeremy Jongepier

(linuxaudio.org – USA & MOD Devices - Germany)

For guitar, computer and voice.

A musician with over 25 years of experience and a computer with Linux. That's what it boils down to. The result: conventional, solid song-writing, with an eclectic tinge because of the choice to not walk the threaded paths coupled with an auto-didactic background, an outspoken personal taste and an open-minded world-view.

This year *The Infinite Repeat* will be all about going back to the roots and mixing that up with the latest Linux based technology. So expect some solid acoustic singer-songwriter material with a modern touch and a Linux device on the floor.

See http://theinfiniterepeat.com

Saturday, May 21 – 08:30pm – 10:00pm – L'Estancot – 10 Rue Henri Dunant - Saint-Étienne

Concert N°3: Acousmatic Music

1. *Voce 3316* (3' - stereo), Massimo Fragalà, Italy.

2. *Kecapi III* (10'56 – octophony), Patrick Hartono, Indonesia.

3. Inuti (9' - 16 channels), Helene Hedsund, United Kingdom.

4. *Profon* (7' – stereo), Magnus Johansson, Sweden.

5. *Dark Path #6* (5' – stereo), Anna Terzaroli, Italy.

6. *Kruchtkammer* (6' – quadriphony), Lukas Tobiassen, Germany.

7. *Definierte Lastbedingung* (11'40 – octophony ambisonic), Clemens Von Reusner, Germany.

8. *Suite of miniatures* (10' - stereo), Bernard Bretonneau, France

9. *Concret X* (5' – ambisoniX), Jean-Marc Duchenne, France

Multimedia Installations

Multimedia installations

1. *OUPPO* (Harris Louise - United Kingdom)

Ouppo is a generative audiovisual installation composed of four modular units for video mapping.

2. *PROFON* (Julien Ottavi - France)

Profon is a performative device composed of a collection of flower pots moved in time and space.

3. *SONIC CURRENT* (Giannoutakis Kosmas - Austria)

Sonic current is a site-specific sound installation which transform architectural locations into “sonic conscious” organisms. The transformation of the site into a body, with its sense organs (microphones) and actuators (loudspeakers), enable the site to articulate and manifest itself in an open dialogue with its visitors.

4. *THETA FANTOMES* (Apo33 Collective - France)

Thêta Fantomes is a cross-disciplinary digital game/art project. Its an art piece developed by APO33 to realise some of our ideas about using real-time neuronal data processing with game play in a hybrid transcendental experience.

5. *ZIC STREET BOX* ((Lionel Rascle)

Conservatoire de Musique de Saint-Chamond - France

SicStreetBox is an interactive equipment targeted for public demonstration and Raise awareness to the use of technologies for sound art creation. The songs used in the installation are designed by the pupils of the computer music course in Saint-Chamond music school.

6. SOUND SCULPTURES (Thomas Barbe - France)

The sound sculptures of Thomas Barbe question the link between sculptural form and sonorous generation. His creations situate the sound in the spatial environment in a tangible way. They question the link between visual form and sound activity.

lac2017.univ-st-etienne.fr

FACULTÉ
ARTS
LETTRES
LANGUES

Ecole
supérieure
d'art
et design
Saint-Etienne

Le son des choses