THE HYBRID MOBILE INSTRUMENT: RECOUPLING THE HAPTIC, THE PHYSICAL, AND THE VIRTUAL

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF MUSIC
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Romain Pierre Denis Michon
June 2018

This dissertation is online at: http://purl.stanford.edu/rd318qn0219

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Chris Chafe, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Julius Smith, III, Co-Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Ge Wang

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Matthew Wright,

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

The decoupling of the "controller" from the "synthesizer" is one of the defining characteristic of digital musical instruments (DMIs). While this allows for much flexibility, this "demutualization" (as Perry Cook termed) sometimes results in a loss of intimacy between the performer and the instrument.

In this thesis, we introduce a framework to craft "mutualized" DMIs by leveraging the concepts of augmented mobile device, hybrid instrument design, and skill transfer from existing performer technique.

Augmented mobile instruments combine commodity mobile devices with passive and active elements that can take part in the production of sound (e.g., resonators, exciter, etc.), while adding new affordances to the device and changing its form and overall aesthetics. Screen interfaces can be designed to facilitate skill transfer, accelerating the learning and the mastery of such instruments.

Hybrid instrument design mutualizes physical and "physically-informed" virtual elements, taking advantage of recent progress in physical modeling and digital fabrication. This design ethos allows physical/acoustical elements to be substituted with virtual/digital ones and vice versa (as long as it is physically possible).

A set of tools to design hybrid mobile instruments is introduced and evaluated. Overall, we demonstrate how this approach can help digital luthiers to think about DMI design "as a whole" in order to create mutualized instruments. Through a series of case studies, we discuss aesthetic and design implications when making such hybrid instruments.

Acknowledgments

Where to start? So many people need to be thanked for their various contributions to this work.

First, I owe so much to my four advisors Professors Julius O. Smith, Chris Chafe, Ge Wang, and Matt Wright. Julius was my former advisor when I came to CCRMA as a visiting researcher in 2011. He's one of the most enthusiast and curious person I know. He provided me with constant inestimable feedback, and I wouldn't be here today without his help and support. Chris Chafe has done so much for me since I started my PhD. He got me involved in so many fun and interesting projects. He's the "benevolent father" of CCRMA, looking at all his kids. His knowledge and understanding of our field as well as his thoughtful advices really helped me move forward in my PhD journey. I collaborated with Ge on various projects related or unrelated to this thesis in recent years. His energy, craziness, curiosity, and ideas are an endless source of inspiration to all of us. CCRMA as I know it would not be the same without Ge. He's also a very good friend. Matt's feedback has been priceless both for this thesis and also for the various papers that we co-authored. His sense of detail and his rigor helped me strengthen my work.

I want to thank Yann Orlarey who's been my mentor and my "spiritual father" for the past eight years. Yann has been a constant source of inspiration for me since we met. His kindness, ideas, and awareness of ongoing research and technologies in our field (and beyond) keep inspiring me. Thanks Yann!

Thanks to my parents, Gilles and Michèle Michon, to my sister Ninon Michon (Doucette), to my grandmother Marie Michon, to my uncle, aunt, and cousin Serge, Denise, and Sophie Ferreri for their love and unswerving support, even though I'm not sure if they fully understand what I'm doing.

I would probably not being dedicating my life to music technology if I didn't meet Laurent Pottier. Laurent was my former advisor and instructor when I did my bachelors and my masters at Saint-Etienne University. He transmitted me his passion for our field and introduced me to John Chowning who helped me becoming a visiting researcher at CCRMA.

I collaborated with Sara Martin to make `mesh2faust`. This was such a fun experience and this work would not have been possible without her mathematical and physical modeling skills.

My dear friend John Granzow and I co-taught most of the workshops that served as the evaluation

of the various frameworks presented in this thesis. His work has always been a great source of inspiration for me.

Stéphane Letz provided me with invaluable technical help for the various tools presented in this thesis. He's just so good! GRAME and the Faust community are so lucky to have him...

Thanks to Sasha Leitman for making the MaxLab such a productive and cool environment. She helped me a lot in the first years of my PhD as I got interested in the NIME field and musical interaction design in general. I really miss our chats and her Bloody Marys!

Thanks to Nette Worthey for being the "CCRMA mum." We're all so grateful for her investment in our community and her kindness.

CCRMA and the Stanford Music Department are the most incredible and inspiring places I was given to work at. This is partly thanks to their staff and faculty. In particular, I'd like to thank Fernando Lopez-Lezcano, Jonathan Abel, Jonathan Berger, Takako Fujioka, Chris Jette, Jay Kadis, Dave Kerr, Carr Wilkerson, Debbie Barney, Velda Williams, Mario Champagne, Carlos Sanchez, John Chowning, and Jaroslaw Kapuscinski for their help and support throughout my stay here.

My friends at Stanford played and invaluable role in this work. People here are both so talented and down to earth. In particular, I'd like to thank Elliot Canfield-Dafilou, Spencer Salazar, Nolan Lem, Madeline Huberth, Tim O'Brien, Constantin Basica, Alex Chechile, François Germain, Emily Graber, Nick Gang, Ethan Geller, Woodrow Herman, Kurt Werner, Chet Gnegy, Myles Borins, Pablo Castellanos Macin, Priyanka Shekar, Eoin Callery, Rahul Agnihotri, Jack Atherton, Hongchan Choi, Esteban Maestre, Ed Berdahl, Luke Dahl, Rob Hamilton, Jay Dunlap, Orchisama Das, Mark Rau, Julie Herndon, Jorge Herrera, Sarah McCarthy, Kara Riopelle, Charlie Sdraulig, Kitty Shi, Jeremy Hsu, Alex Colavin, Dan Schlessinger, Victoria Grace, Blair Kaneshiro, Aury Washburn, Lauri Savioja, Lonce Wyse, Iran Roman, Séverine Ballon, and Victoria Chang.

My friends back home always supported me throughout this adventure and always made me feel like I never left. Thanks to Léo and Flavie Brossy, Clément Terrade, Clément Paulet, Julien Sthème de Jubécourt, Ludovic Chaux, Anthony Paillet, Luisa Kabala, Marine Staron, Angie Harry, and Dimitri Bouche.

Thanks to Stefania Serafin, Juraj Kojs, Dan Overholt, and Cumuhr Erkut for giving me the opportunity to teach workshops in their respective institutions. Thanks also to the all the workshop participants who served as my guinea pigs! I owe a lot to Stefania for proof-reading this thesis and giving me invaluable feedbacks.

Other people in our field helped me put this work together and have been a source of inspiration since I became interested in music technology. In particular, I'd like to thank some of the members of the faculty and staff at the National University of Ireland in Maynooth (NUIM) for being my brilliant instructors during my year there: Victor Lazzarini, Joe Timoney, and Mathieu Hodgkinson. Thanks to my "FAUST colleagues" too: Dominique Fober, Albert Graef, Emilio Gallego Arias, Pierre-Amaury Grumiaux, and Pierre Jouvelot.

Thanks to Pat Scandalis and Nick Porcaro at moForte for our collaborations, for giving my access to their network, and for helping me in various ways during the course of my PhD.

Thanks to Professor Doug James for accepting to be the chair of my oral defense committee!

Thanks to Stanford University for offering me so many opportunities and for being such an incredible place.

Thanks to GRAME in Lyon for hosting me multiple times in the frame of scientific residencies and for funding the development of FAUST. I'm so proud to be from a country where artistic creation and scientific research are supported by the state.

Finally, thanks to everyone I forgot!

Contents

List of Tables

List of Figures

Introduction

Overview

Unlike most acoustic musical instruments where control and sound generation and diffusion happen on the same entity, Digital Musical Instruments (DMIs) are often not standalone and made out of several units. While the idea of abstracting the controller from the sound generator was first introduced by early organs thousands of years ago, this paradigm became one of the defining features of DMIs and was generalized by the use of standards such as MIDI. This modularity gave birth to a new form of "high level lutherie"[1] that allowed the performer to become the designer of his/her own instruments.

While the "virtualization" and the dissociation/"de-mutualization" [42] of the different constituting elements of musical instruments offered infinite new possibilities, it also impacted their overall coherence. In particular, the ability to control "any sound" with "any interface," might result in some cases in a "loss of intimacy between human player and instrument [42]." In the commercial world, where a large part of the interfaces target pitch control, this problem has been partially solved by limiting mapping strategies to the ones induced by keyboard interfaces (i.e., note on/off, pitch, pitch bend, etc.). While this type of paradigm works well in some cases, it might be more limiting when extended continuous control is required such as with "virtual wind instruments." The modularity of DMIs, combined with the technologies that they imply, also impact their "standalone aspect," making them less portable and sometimes preventing the performer to be "completely one" with his instrument.

In this thesis, we aim at creating mutualized DMIs by bringing more physicality to digital lutherie [84]. We provide a framework where musical instrument design can be approached in a two-dimensional way, reconciling the haptic, the physical, and the virtual. Instrument parts can either be physical/acoustical or virtual/digital and substituted with one another. Recent progress in musical instrument physical modeling and digital fabrication (with 3D printing in particular) facilitates this type of approach, opening the way to a new kind of "hybrid lutherie."

[1] In this thesis, "lutherie" will refer to the design and making of musical instrument in general.

Mobile devices are at the heart of our framework. By providing power through their battery, built-in sensors (i.e., touchscreen, motion sensors, etc.), real-time DSP[2] capabilities, etc., they constitute a promising platform to implement the virtual/digital portion of our hybrid instruments.

Leveraging performers skills is a crucial factor in making successful musical instruments [41]. Thus, the idea of implementing skills transfer when designing mobile hybrid instruments will be explored, in particular in the frame of touchscreen interfaces.

We hope that centering DMIs design on physically-informed virtual/digital and physical/acoustical elements will facilitate the design of mutualized DMIs, the use of mobile devices as their "core" allowing them to be easily reproduced/deployed and completely standalone.

Outline

Chapter 1 – Background

Chapter 1 provides a review of the literature of the various topics relevant to this dissertation. We first demonstrate how physical musical interfaces have been used to control "virtual acoustic instruments"[3] and to implement skill transfer for various families of instruments (e.g., wind, string, percussion, etc.). We show that making "mutualized" DMIs preserving the overall coherence between the controller and the sound synthesis unit is often a complicated task. Various techniques used to solve this issue such as haptic feedback are briefly reviewed.

Next we give an overview of two special kinds of DMIs: augmented and acoustically driven hybrid instruments. Augmented instruments are based on existing acoustic instruments that are enhanced with digital elements to add new affordances or to modify their sound. Acoustically driven hybrid instruments use physical/acoustical elements to drive virtual ones.

A review of the field of mobile music is then provided. A strong emphasis is given to the question of skill transfer on mobile devices and how it can be used as a way to accelerate the learning process of new DMIs.

Next, various types of mobile device augmentations towards musical instrument design are presented. We demonstrate that built-in elements on smart-phones and tablets offer already a wide range of features and that small, non-invasive augmentations are often enough to turn mobile devices into high quality musical instruments.

We then give a brief overview of the field of physical modeling of musical instruments. We focus on computationally cheap techniques that can run on mobile devices.

Finally, we show how digital fabrication, with 3D printing in particular, has been used in lutherie to make existing and novel acoustic or electronic musical instruments.

[2] Digital Signal Processing

[3] In this thesis, we call "virtual acoustic instruments" any synthesizer designed to reproduce the sound of existing acoustic instruments. This is not limited to specific techniques such as physical modeling.

Chapter 2 – Genesis

Chapter 2 presents a series of early projects/instruments of our own that led to the work presented in this thesis. We start first with the FÉRAILLOPHONE, the HYBRIDSCREEN, the BLACKBOX, and the CHANFORGNOPHONE. All these instruments are based on the idea of using acoustic excitations to drive physical models. Next, a series of instruments based on augmented iPads is presented. Finally, various versions of the BLADEAXE – a guitar physical model controller – are reviewed. We emphasize the iterative design process that was involved when creating these instruments. The final version of the BLADEAXE presented in the last section of this chapter served as the basis for the framework introduced in this dissertation.

Chapter 3 – MOBILEFAUST: Facilitating Musical Apps Design and Skill Transfer

Chapter 3 introduces a series of tools based on the FAUST programming language[4] [141] to facilitate the design of musical mobile apps serving as the "glue" for the rest of our framework.

First, we present two early systems that inspired the other tools presented in this chapter: `faust2android` and `faust2ios`. They allow for the conversion of FAUST code to Android and iOS apps respectively. The user interface (UI) of the generated apps is based on the standard UI description provided in the FAUST code.

Next, we introduce `faust2api`, a tool to generate DSP engines from FAUST code for a wide range of platforms including Android and iOS. The high level APIs[5] produced by this system allow for the interaction with the DSP engine in a very simple way. It features OSC and MIDI support, polyphony handling, built-in sensors mapping, etc.

`faust2api` serves as the basis for `faust2smartkeyb` which is introduced in the following section. Like `faust2android` and `faust2ios`, `faust2smartkeyb` can be used to generate apps for Android and iOS from FAUST code. The standard FAUST UI is replaced by a SMARTKEYBOARD interface that can be configured from the FAUST code. SMARTKEYBOARD offers a new approach to touchscreen musical interface design based on a keyboard matrix.

Finally, a study around the use of `faust2smartkeyb` to make mobile-device-based musical instruments focusing on skill transfer is presented. Various instrument types and paradigms are covered (e.g., plucked strings, bowed strings, percussion, wind instruments, etc.).

Chapter 4 – Passively Augmenting Mobile Devices

Chapter 4 focuses on the idea of passively augmenting smart-phones and tablets. This type of device hosts a wide range elements (e.g., sensors, microphone, speaker, etc.) that can be easily enhanced or

[4] `http://faust.grame.fr/` – All URLs presented in this thesis were verified on Feb. 6, 2018.
[5] Application Programming Interface

modified using digitally fabricated passive elements. First, we introduce MOBILE3D, an OpenScad library to facilitate the design of this type of augmentations. We then give an exhaustive overview of the taxonomy of the various types of passive augmentations that can be implemented on mobile devices through a series of case studies and we demonstrate how they leverage existing components of the device. Finally, we evaluate our framework and propose future directions for this type of research.

Chapter 5 – Actively Augmenting Mobile Devices With Sensors

Chapter 5 provides a framework and a method to make active mobile device augmentations. Active sensor augmentations involve the use of electronic elements and provide more flexibility to musical instrument designers than passive ones. On the other hand, they are usually more invasive and their design is more complex.

First, we present NUANCE, an iPad-based instrument where a set of sensors is used to add force sensitivity to the touchscreen of the device. It can be seen as a first step towards the framework presented in the following sections. Next, we introduce different strategies to transmit sensor data to mobile devices. We then use the conclusions from this study to build our active sensors augmentation framework. Finally, we evaluate this framework through a series of example instruments created by students in the frame of a workshop.

Chapter 6 – Developing the Hybrid Mobile Instrument

Chapter 6 introduces and develops the concept of "hybrid mobile instrument." Hybrid instrument design mutualizes physical and "physically-informed" virtual elements, taking advantage of recent progress in physical modeling and digital fabrication (see §1). This design ethos treats each component of the instrument in a two-dimensional way, allowing physical elements to be substituted with virtual ones and vice versa (as long as it is physically possible).

First, we give an overview of our framework to design musical instruments combining digitally fabricated physical elements and virtual elements. Challenges and technical difficulties presented by specific cases are discussed and we try to link these new methods to the tools presented in the previous sections.

We then introduce the FAUST Physical Modeling Library (FPML): a tool to easily design physical models of musical instruments from scratch using FAUST. Various case studies of models implemented using this system are provided.

Finally, we present `mesh2faust`, a tool to generate modal physical models from their graphical representation. We demonstrate how it can be used in combination the FAUST Physical Modeling Library to create custom models that can be easily converted to physical objects using digital fabrication.

The concept of hybrid mobile instrument introduced in this thesis is summarized in Figure 1 where the various tools implemented as part of this work are underlined.

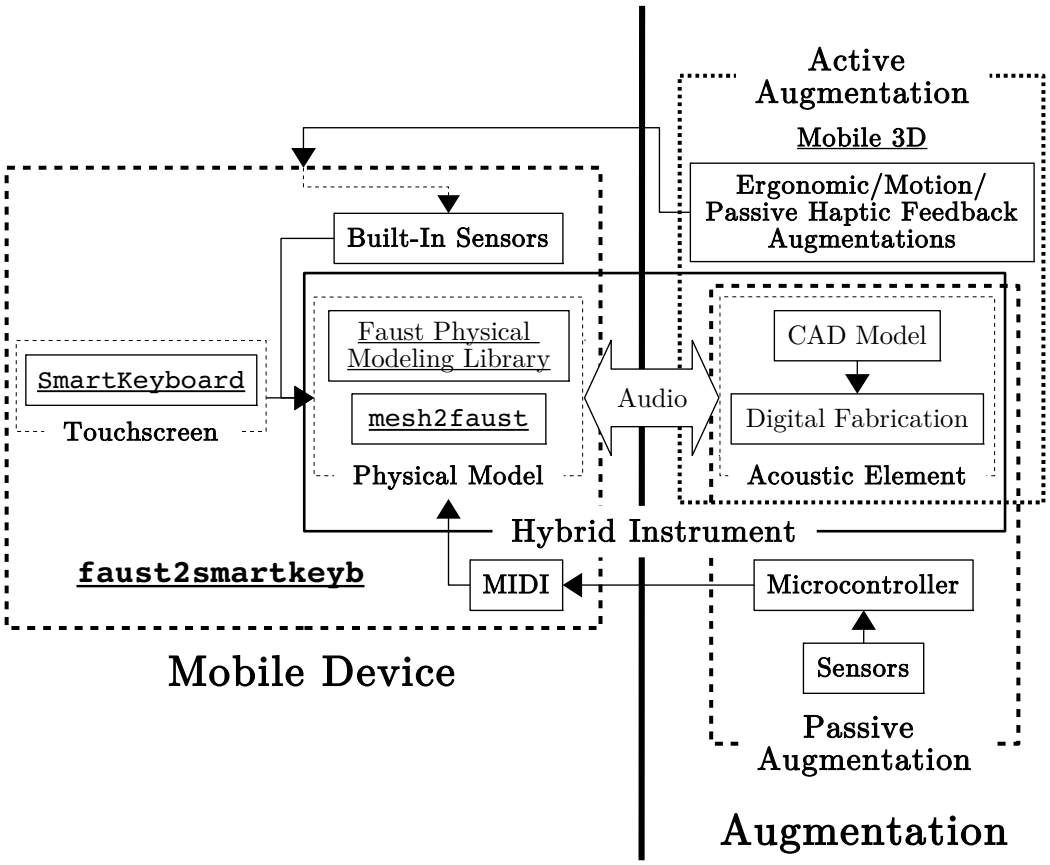


Figure 1: Overview of the Hybrid Mobile Instrument.

Chapter 1

Background

> "Digital lutherie is in many respects very similar to music creation. It involves a great deal of different know-how and many technical and technological issues. At the same time, like in music, there are no inviolable laws. That is to say that digital lutherie should not be considered as a science nor an engineering technology, but as a sort of craftsmanship that sometimes may produce a work of art, no less than music." (Sergi Jordà [84])

1.1 Physical Interfaces and Virtual Instruments: Remutualizing the Instrument

1.1.1 The Rise of Musical Interfaces

The concept of musical controller is not new and was perhaps invented when the first organs were made centuries ago. However, the rise of analog synthesizers in the middle of the twentieth century, followed a few decades later by digital synthesizers almost systematized the dissociation of the control-interface and sound-generation in musical instrument design.

> "The computer, as a design medium, is profoundly under-constrained – a virtual space without no physical rules [198]."

This gave birth to a new family of musical instruments known as "Digital Musical Instruments" (DMIs).

Marc Battier defines DMIs from a "human computer interaction (HCI) standpoint" as

> "Instruments that include a separate gestural interface (or gestural controller unit) from a sound generation unit [15]."

Thus, this feature that originally resulted from logical engineering decisions encouraged by the use of flexible new technologies, became one of the defining component of DMIs. This characteristic has been extensively commented upon and studied in the NIME[1] literature. Thor Magnusson extends Battier's definition by highlighting the arbitrary aspect of the interface in respect to the sound generation unit in DMIs:

> "The tangible user interfaces that apparently constitute many digital musical instruments are but arbitrary peripherals of the instruments' core – that is, a core that is essentially a symbolic system of computational design [109]."

This feature created new problematics in musical instrument design such as that of "adequate control:"

> "A challenge is how to make controllability and interactivity central design principles in sound modeling. It is widely believed that the main missing element in existing synthesis techniques is adequate control." (Gerhard Widmer et al. [203])

Other consequences of the "separation of the DMI into two independent units" are pointed out by Marcello Wanderley:

> "This separation is most of the time impossible in the case of acoustic instruments, where the gestural interface is also part of the sound generation unit. [...] Clearly, this separation of the DMI into two independent units is potentially capable of extrapolating the functionalities of a conventional musical instrument, the latter tied to physical constraints. On the other hand, basic interaction characteristics of existing instruments may be lost and/or difficult to reproduce, such as tactile/force feedback [195]."

Perry Cook provides an exhaustive overview of the risks associated with "abstracting the controller from the synthesizer" [42], which might sometimes result in a "loss of intimacy" between performer and instrument. More specifically, he associates the flaws of "demutualized instruments" to:

- the lack of haptic feedback, which has been extensively studied [144, 195, 99], especially in the framework of the control of physical models of musical instruments [17, 99] (see §1.1.7),
- the lack of "fidelity in the connections from the controller to the generator,"
- the fact that "no meaningful physics goes on in the controller."

Ge Wang reinforces Perry Cook's view by drawing parallels with the principles of form and function used in architecture:

[1] New Interfaces for Musical Expression

> "With acoustic instruments, function suggests from. With computer-based instruments, form is decoupled from function [198]."

Prior to the "digital sound synthesis era," the variety of interfaces to control sound synthesis was rather limited. Piano-keyboard-like controllers were the preferred interface to control pitch. This characteristic is embodied by the differences between Buchla (no piano keyboard) and Moog (piano keyboard is used to control pitch) synthesizers from that period [145, 148]. This paradigm remains a standard nowadays in commercial DMIs and is tightly related to the question of skill transfer. However, the lack of continuous control on standard piano keyboards limited its use as a generic interface for the control of pitch. Thus, as early as in the 1970s, synthesizers such as the Minimoog allowed to bend the pitch of a designated key on the keyboard using a pitch wheel. Similarly, early commercial digital synthesizers such as the Yamaha DX7 [206] allowed for the continuous control of the parameters using a breath controller.

In parallel to these changes and as sound synthesis techniques grew more sophisticated and computers more powerful, the ability to synthesize the sound of existing acoustic instruments became possible. To control such sound generators that we'll call "virtual acoustic instruments,"[2] a wide range of interfaces were created since the end of the 1970s. An interesting characteristic of this type of controller is that their overall design is often influenced by the synthesizer they're controlling [196], potentially loosing their generic nature [109].

This paradigm is pushed even further in the case of physical models of musical instruments (see §1.5) as they implement virtual objects that are tight by physical constraints. In this respect, DMIs using physical modeling are closer to acoustic musical instruments than DMIs based on other sound synthesis techniques.

> "Computer interfaces can dissociate gesture from result to varying degrees by the way that software intermediates the relationship between gesture and resulting sound. (A one-to-one correspondence such as a mallet striking a marimba is an example of a simple gesture-result relationship, while a finger pushing the play button on a CD player exemplifies the opposite extreme in which a simple neutral gesture produces a complex musical result.) Jordà [84] evaluates this relationship as the 'efficiency' of the interface, defined as the ratio of 'musical output complexity' to 'control input complexity,' but acknowledges that these are 'quite fuzzy terms,' and that while computer-mediated controllers can provide more 'efficiency' than most acoustic instruments, they often lack the 'expressiveness' (flexibility, diversity of micro-control, etc.) of traditional instruments." (Christopher Dobrian et al. [52])

In the rest of this section, we demonstrate how various commercial and non-commercial DMIs implementing virtual acoustic instruments attempt to re-mutualize their interface and their sound

[2] We call virtual acoustic instrument any sound generator reproducing the sound of existing acoustic instruments (synthesized or sampled) or implementing physical models of completely novel instruments.

generation unit. We'll show that this type of sound generator tends to break the generic aspect of musical controllers by forcing them to implement specific gestures to control specific sounds, reducing their efficiency [52]. The importance of skill transfer in the design of such controllers is highlighted, especially in the case of commercial interfaces that tend to be centered on this feature. We also study the special case of interfaces implementing haptic feedback. This section focuses on DMIs based on the interface/synthesizer paradigm. Instruments based on a more hybrid/unified approach will be presented in §1.2. Only representative instruments are presented here: this section is not meant to be exhaustive.

1.1.2 Keyboard-Based Interfaces

Keyboard controllers can probably be considered as the most widely used type of musical interface. They come in various formats and are often used to implement specific types of pitch mappings (e.g., isomorphic [112], chromatic, etc.), enabling skill transfer.

A special category comprises chromatic piano keyboards, that are mastered by countless musicians, and can be found on a large range of commercial synthesizers from any period. They continue to be the most standard interface for the control of pitch.

In this section, we focus on all the types of keyboards that are not related to a specific acoustic musical instrument. These other special cases (e.g., guitar necks, violin necks, etc.) will be treated in the following sections.

Commercial Interfaces

There exists dozens of commercial controllers implementing standard chromatic piano keyboards that will not be presented in this section. Instead, we'll focus on interfaces providing special features transcending the standard use of this kind of controller.

While piano keyboards offer many advantages (e.g., speed of playing, polyphony, universality, etc.), they are limited by their lack of continuous control. This problem is partially solved on most commercial keyboards by rotary potentiometers also called "wheels," placed on the left side of the keyboard. The most common parameter controlled by wheels is pitch through the "pitch bend wheel," but other parameters such as "brightness" or "portamento" are common as well.

More innovative (and controllable) solutions were developed early on by various musical instrument manufacturers. Yamaha added the possibility to plug a breath controller to some of its synthesizers as early as in the first version of the DX7 [206]. This feature provided a more natural way to control the sound of wind instrument synthesizers. It remained available on later products of this brand such as the VL1 [205] that was the first commercial product to make use of the waveguide physical modeling technique (see §1.5). The VL1 implemented a wide range of existing and novel acoustic instruments, and offered extended expressive capabilities [159].

The ROLI SeaBoard [94, 157] is a more recent musical controller taking an interesting approach to improve piano keyboard interfaces by making it possible to slide between keys and continuously control the gain of generated sounds. The SeaBoard proved to be well suited to control physical models of musical instruments such as the one of the SWAM engine,[3] one of their commercial partners. There are many videos demonstrating this on the ROLI website.[4]

The Haken Audio Continuum Fingerboard[5] is similar to the Seaboard in many respects as it allows for the continuous control of the pitch and the gain of the sounds it generates. Another musical controller taking a similar approach is the LinnStrument,[6] which also permits to continuously control the pitch and the velocity parameters of sound synthesizers through a silicone-three-dimensional interface implementing a configurable pitch matrix.

Finally, various "less common" controllers implement other types of chromatic keyboards such as the WholeTone Revolution[7] or the C Thru Music Keyboard Series.[8] Both of them are based on isomorphic keyboards [112], targeting skill transfer for accordion players or other types of existing traditional instruments.

Non-Commercial/Academic Interfaces

There are dozens of examples of keyboards implementing custom pitch mappings in the NIME community. Steven Maupin gives a good overview [112] of this type of instruments.

Despite the various examples presented in the previous section, piano keyboard-based interfaces remain relatively "standard," and there are only a few examples of custom non-commercial piano keyboard-based interfaces. Christian Heinrichs "augmented" a MIDI piano keyboard with capacitive touch sensors to detect the X/Y position of fingers on each key [72]. This type of interface is relatively close to the SeaBoard and the Continuum Fingerboard presented in the previous section in that respect. However, its mechanical keys provide more haptic feedback to the performer than its commercial counterparts, which also makes slides between keys harder to execute.

1.1.3 Wind Instrument Controllers

Wind instrument controllers target a totally different set of skills and gestures, and offer different affordances than keyboard-based interfaces such as the ones presented in the previous section. While on most keyboards, a key is associated to a pitch (or a combination of pitches) and sometimes a voice, pitch selection is made through a combination of keys on wind instruments, making them consequently monophonic in many cases. On the other hand, the complex type of continuous interaction happening between the performer and the mouthpiece makes them very expressive. This

[3] https://www.samplemodeling.com/
[4] https://roli.com/
[5] http://www.hakenaudio.com/Continuum/hakenaudioovervg.html
[6] http://www.rogerlinndesign.com/linnstrument.html
[7] http://wholetone.jp/
[8] http://c-thru-music.com/cgi/index.cgi

section gives an overview of the different kinds of wind instrument controllers and how they leverage specific sets of skills.

Commercial Interfaces

Commercial wind instrument controllers might be the class of interface embodying the best the concept of skill transfer for the control of sound synthesizers. They allow wind instrument performers to use their skills to control virtual musical instruments. Most of them host various continuous sensors such as a built-in breath controller, valves, etc.

The first "electronic wind instrument" was the Lyricon [8, 75]. It combined a soprano saxophone-like interface with an analog synthesizer. It greatly influenced the design of the MIDI controllers of the Yamaha WX series [202] that implemented saxophone, clarinet, and flute fingerings. Akai created several series of equivalent controllers: the EWI,[9] that was similar to controllers of the WX series, and the *EVI*,[10] that was designed to facilitate skill transfer for "valve instrument" performers (e.g., trumpet, trombone, etc.).

Roland takes a slightly different approach with the Aerophone AE-10 [156]. While this instrument implements saxophone fingerings in a similar way than Yamaha WX controllers, it also host an embedded synthesizer, a battery, and a built-in speaker, making it completely standalone and much closer to acoustic musical instruments in that respect.

Several design features common to all the controllers of this family are worth being highlighted. First, while these controllers implement fingerings close to that of existing "traditional" acoustic instruments, they often aren't an exact translation of their acoustic counterpart, mostly to provide more flexibility and make them more generic. That's how flute, saxophone, and clarinet fingerings can all be used with the controllers from Yamaha WX series. By doing so, they allow performers to reuse a big part of their skills, but also force them to go through a (re)learning phase, necessary to fully master these controllers. In that case, a big part of the work of the controller designer is to make sure that this step will be as short and as seamless as possible.

> "Copying an instrument is dumb, leveraging expert technique is smart." (Perry Cook [41])

Beyond skill transfer, this type of controller also completely influences the way the performer engages with the sound generator and the type of gestures implied in its control. Additionally, it is interesting to note how the general form factor of such interfaces is influenced by their "traditional" acoustical counterparts.

Wind instrument musical controllers are designed to leverage a different set of expressive features than the piano keyboard controllers presented in §1.1.2. While they often provide more "natural"

[9] Electronic Wind Instrument
[10] Electronic Valve Instrument

continuous controllers to the performer, they can't control polyphonic sound synthesizers. Thus, despite their versatility, their design is strongly influenced by the type of "virtual acoustic instrument" they are controlling.

Non-Commercial/Academic Interfaces

From an organological and human-interaction standpoint, most non-commercial wind instrument controllers are quite similar to their commercial counterparts and reuse the same principles [40, 41, 131]. A good example of such interface is Garry Scavone's PIPE [163]. It is similar in many respects to controllers from the Yamaha WX series presented in the previous section. Various examples of "augmented" wind instruments take an innovative approach to wind instrument controller design and are presented in §1.2.

1.1.4 String Instrument Controllers

Designing controllers for string instruments presents different challenges than for other types of musical interfaces such as the one presented in the previous sections. For example, string instruments offer a more direct way to interact with parts responsible for the production of sounds than keyboard based instruments. While strings can be driven by external tools such as a bow, they can also be plucked by fingers, allowing the performer to directly interact with them. The feel of a string on a finger as well as the types of affordances that it offers is hard to recreate with musical controllers. This section gives an overview of the different types of string instrument controllers available on the market and made in the non-commercial/academic community.

Commercial Interfaces

Commercial string instrument controllers take a similar approach than wind instrument interfaces (see §1.1.3) by being centered on skill transfer. The guitar is the only instrument represented in this category (probably due to its extensive use in pop music). The product that embodies this paradigm the best is perhaps the SynthAxe [74]. This MIDI interface offers the same control capabilities than a guitar and preserves its form factor by providing a neck and physical strings for strumming, pitch selection, vibrato, etc. It also has various continuous controllers such as a whammy bar and pressure sensitive buttons that can be assigned to different parameters. Only a few SynthAxes were made in the 1980s and they are not produced anymore.

Starr Labs' Ztar series[11] [180] takes a very similar approach by providing an interface based on physical strings for strumming and a neck with buttons for controlling the pitch. The same company also makes various MIDI "ZBoards" implementing guitar necks as keyboards – another

[11] https://www.starrlabs.com/

good example of interface oriented towards skill transfer. Similarly, moForte and Wizdom Music's GeoShred[12] implements the pitch mapping of a guitar neck on an iPad touchscreen.

The JamStick[13] is a cheaper MIDI guitar interface whose main focus is on teaching by connecting it to a dedicated smart-phone app. It is also based on physical strings. It is interesting to highlight the fact that this feature can be found on all the controllers of this category. Sometimes, the design of interfaces can be so influenced by "traditional" instruments that they might become based on elements primarily designed to generate sound, even though they are not used for that purpose in this case. An interesting alternative to this is presented at the end of this section where acoustic musical instruments are used as controllers.

Non-Commercial/Academic Interfaces

Unlike their commercial counterparts, non-commercial string instrument interfaces implement various types of string instruments. The category that might be best represented in this family are bowed string instruments. Charles Nichols' vBow [138, 139] was specifically designed to control violin physical models through an interface providing the same affordances and having the same form factor than an acoustic violin. An interesting feature of the vBow is that it provides haptic feedback to the performer. This concept is further developed in §1.1.7. Nichols also presents an extended study on the control of bowed string instruments [139].

Dan Trueman's BoSSA [187] takes an interesting approach by being both a controller and a sound diffuser, making it much closer to an acoustic instrument that most DMIs. It pushes further this paradigm by using a speaker array to radiate sound in a similar way as acoustic instruments. While the position of the various sensors on the instrument allows it to be controlled like a violin or a cello, it is slightly different than the one on its traditional counterpart. According to Trueman, this doesn't seem to affect the performer's ability to transfer his playing skills.

On the same topic, Stefania Serafin extensively studied the control of bowed string physical models [167, 169], in particular through the use of a graphical tablet [168]. In that case, the performer is able to conceptually use the same kind of gestures as with a bow to control the instrument. This can also be considered as a form of skill transfer, even though the interface is completely different from the original instrument. The use of graphical tablets to control sound synthesizers has been extensively studied (see §1.3.1) [48, 210].

While there exists various guitar controllers in the commercial world, this category is not well represented in the non-commercial one. Christian Heinrichs' Hybrid Keyboard-Guitar Interface has already been presented in §1.1.2 and was specifically designed for the control of guitar virtual instruments, even though its use can be easily extrapolated. Matti Karjalainen et al.'s Virtual Air Guitar [86] explores the use of a custom hand-held devices to control guitar physical models by

[12] http://www.moforte.com/
[13] https://jamstik.com/

capturing the performer's movements.

1.1.5 Percussion Instrument Controllers

By definition, percussion instrument controllers are intrinsically linked to the type of virtual musical instrument they're controlling, and thus a good example of "mutualized" DMIs. They're designed to acquire strike, tap, etc. gestures – which in the "real/physical world" always result in percussive sounds. Even more than that, the shape and format of this type of controllers – at least in the commercial world – are so close to their acoustic counterparts that they naturally suggest a specific set of gestures to the performer to interact with them [136].

Commercial Interfaces

Electronic drum sets (e.g., Roland's VDrums Series,[14] Yamaha's DTX series,[15] etc.) are probably the type of controllers representing the best this family of interfaces, both because of their popularity and their plurality (there exists dozens of models on the market). Most of them conserve the form factor of acoustic drum sets, allowing for a comprehensive transfer of skills. They represent a generic interface to control any virtual drum set, and are somewhat limited to that. By detecting the position of the strikes, they help re-mutualize the interface and the sound synthesis process. In other words, their limited range of applications makes them much more coherent from a physical and gestural standpoint.

Similarly to keyboard synthesizers, electronic drum sets are "traditionally" sold with their own sound synthesizers, making them more standalone than most of the controllers presented in §1.1.3 and §1.1.4. Finally, the strong decoupling between the excitation (i.e., strike, tap, etc.) and the resulting sound allows for the implementation of very effective passive haptic feedback (see §1.1.7).

A more generic type of percussion instrument controller are percussion pads. Less invasive than electronic drum sets, they can capture a wide range of gestures (e.g., strike, tap, etc.) and are very versatile. Unlike electronic drum sets, their form factor greatly differs from any traditional acoustic instrument. Most major electronic musical instrument manufacturers make various kinds of drum pads (e.g., Roland's SPD Series,[16] Yamaha's DTX-MULTI series[17]). Some companies take a more innovative approach to this by making more "high end" products such as the Jambé,[18] a percussion pad utilizing a proprietary format to transmit control data to iOS devices at a very high rate, necessary to capture fast drumming gestures (e.g., roll, etc.).

"Acoustically driven hybrid" and "augmented" percussion interfaces are relatively well represented in the commercial world (e.g., ATV aFrame, Korg WaveDrum, etc.) and will be presented in §1.2.

[14] https://www.roland.com/global/categories/drums_percussion/v-drums_kits/
[15] https://usa.yamaha.com/products/musical_instruments/drums/el_drums/drum_kits
[16] https://www.roland.com/global/categories/drums_percussion/percussion/
[17] https://usa.yamaha.com/products/musical_instruments/drums/el_drums/drum_kits
[18] http://getjambe.com/

Non-Commercial/Academic Interfaces

Non-commercial percussion interfaces are often either more specific (linked to an acoustic instrument in particular and its related set of gestures) or provide more innovative approaches to motion capture than their commercial counterparts. Max Mathews' Radio Baton [27] might be the most representative controller of this second category by allowing the detection of the position and velocity of strikes, as well as the distance of the sticks ("batons") from the interface. Mike Collicutt et al. [39] provide an overview of this type of interfaces.

On the other hand, a wide range of less generic percussion instrument controllers directly associated to specific acoustic musical instruments were developed in the NIME community. Similarly to their commercial counterparts, this type of interface is more focused on skill transfer than the ones presented at the beginning of this section. Good examples of such controllers are Ajay Kapur's Electronic Tabla [85] or Diana Young's AoBachi [208], a controller for Japanese percussion instruments.

The special case of "acoustically driven hybrid" and "augmented" non-commercial percussion controllers will be treated in §1.2.

1.1.6 Other Controllers

There exists dozens of other musical controllers – both commercial and non-commercial – that are either more generic and not associated to a specific family of musical instruments, or that present a unique interface involving its own set of skills. While such controllers are not presented here because they are beyond the scope of this section, some specific cases are worth being investigated. In particular, controllers that are not directly related to a specific acoustic instrument but that rely on the combination of multiple sets of skills are presented.

The Artiphon INSTRUMENT 1[19] combines multiple interfaces and sensors, allowing for the use of various sets of musical skills at once. It hosts a strumming interface and a continuous-pressure-sensitive pitch matrix – not unlike the one on the LinnStrument (see §1.1.2) – and can be used as a guitar, violin, cello, keyboard, etc. controller. It is wireless, battery powered, and also has built-in speakers, making it an almost completely standalone instrument (sound synthesis must be done on a third-party device such as a smart-phone). The generic and standalone aspect of the INSTRUMENT 1, as well as its design oriented on skill transfer, make it especially relevant in the framework of this dissertation.

Another interface combining various sets of instrumental skills in a unique and novel way is the Karlax.[20] While it borrows the form factor and the fingerings of some wind instruments, it allows the performer to carry out more gestures (e.g., it can be twisted, etc.) and doesn't have a breath

[19] https://artiphon.com/
[20] http://www.karlax.com/

controller. As a generic interface, it was not designed to control a specific set of virtual acoustic instrument.

1.1.7 Haptic Feedback

The lack of haptic/force feedback in interfaces used to control virtual musical instrument is a known issue whose impact on the ability to perform has been extensively studied [144]. By increasing the tangibility of virtual instruments, the goal of this type of system is to allow the performer to experience similar sensations as with acoustic musical instruments.

Perry Cook demonstrates that interfaces offering haptic feedback tend to reinforce the intimacy between human player and instrument (see §1.1.1), implementing "mutualized" DMIs [42]. Indeed, by linking virtual elements to tangible features, they induce an advanced level of cohesion between the different elements constituting the DMI.

While haptic feedback can be as simple as transducing the sound of a virtual instrument to an interface – implementing "vibrotactile feedback" [36, 110, 26] – more advanced techniques can also be used. In particular, the paradigm presented above is pushed even further when haptic interfaces are designed to work with physical models of musical instruments [17, 21]. In that case, the controller can become the "real/physical world" physical extension of the virtual instrument, facilitating the design of mutualized DMIs. Pioneering works in this field were done by Claude Cadoz et al. [99] through their Transducteur Gestuel Rétroactif (TGR) [33] and the Cordis-Anima environment [34]. This tool allows for the implementation virtual physical structures using masses and springs. Connections to the physical world can be made at different locations in such structures through a TGR interface. The energy present at each location is transmitted to the controller and converted to a mechanical force using electric motors, providing seamless bidirectional connections between the virtual and the physical elements of the instrument. The high level of modularity of the TGR allows it to be used to implement a wide range of gestures. For example, Stephen Sinclair et al. used it to make a bow controller [172].

Various other projects investigated the use of haptic feedback in interfaces for the control of physical models. David Howard et al. [77, 78] take a similar approach than Cadoz et al. with their Cymatic environment [154]. Edgar Berdahl tried to generalize the principles used in the TGRs to make them more accessible and to control other types of physical models. Thus, his environment Synth-A-Modeler [18] enables to implement physical models of musical instruments by combining several modeling techniques through a graphical interface. The performer can interact with the virtual portion of the instrument using FireFaders [19].

Other haptic controllers closer to the one presented earlier in this chapter are specifically designed to interact with existing virtual acoustic instruments. Charles Nichols' vBow [138, 139] is a good example of such controller and was made to control a waveguide physical model of a violin. Haptic feedback is provided to the performer through a motorized bow directly mounted on the body of

the interface.

An interesting feature of the approach taken by Cadoz and colleagues is that it doesn't target the implementation of existing acoustic musical instrument. Instead, it provides a modular physical and virtual environment to design completely novel mutualized instruments.

1.2 Augmented and Acoustically Driven Hybrid Instruments: Thinking DMIs As a Whole

Most of the instruments presented in the previous section are based on the combination of a controller and a synthesizer. Even though in some cases they might be part of the same entity, they remain physically and conceptually separated. Instruments presented in this section are special kinds of DMIs combining acoustical and virtual elements to make sound. By doing so, they blur the interface/synthesizer boundary, making them more mutualized and unified as a whole, partly solving the issue presented in §1.1.

A simple case of such instruments are "augmented instruments." They are based on existing acoustic instruments that are augmented using sensors and some digital effect/synthesizer. This type of instrument is presented in §1.2.1. "Acoustically driven hybrid instruments" use both physical and virtual elements and are not based on existing instruments. They are presented in §1.2.2.

1.2.1 Augmented Instruments

> "The current research field on augmented instruments is motivated by the assumption that the combination of traditional acoustic instruments with today's sound technology yields a high potential for the development of tomorrow's musical instruments." (Otso Lähdeoja [105])

Augmentations can take several forms. They participate in the sound production of the instrument using acoustic, electronic, or digital/virtual elements that can be mounted on the instrument or be independent from it (e.g., a computer). In the case of electronic and digital augmentations, sensors are usually placed on the body of the instrument to control the parameters of the augmentations. The choice of the type of sensors and their placement is usually a key factor in a successful augmented instrument. One of the main challenges pointed out by Lähdeoja when designing such instruments is that gestures are usually already completely saturated in acoustic instruments. Thus, the control of augmentations must seamlessly integrate to existing sets of gestures associated with a specific instrument.

The Rickenbacker Electro A-22 "frying pan" electric guitar [179] that was released in 1931 might be considered as one the first examples of electronic augmentation of an acoustic instrument. The body of the instrument is replaced by an amplifier and the acoustic elements of the instrument

(i.e., strings) are interfaced with the electronic ones through electromagnetic pickups. For the same reason, electric guitars can also be considered as "acoustically driven hybrid instruments" (see §1.2.2).

Disklaviers can be considered as a form of augmented/hybrid instrument by having the same characteristics than a traditional acoustic piano and offering the possibility to be digitally controlled. They offer a wide range of options to design completely novel musical instruments such as David Jaffe's Radio-Drum-Drive Disklavier [81] where a Radio-Baton [27] is used to control the piano keyboard. This concept was expanded to entire orchestras by Troy Rogers et al. [107].

Dozens of augmented instruments have been developed in the NIME community during the past thirty years. Many of them are presented in Eduardo Miranda's book on "New Digital Musical Instruments" [131]. The violin is particularly well represented in this category with the Augmented Violin [22], the Hyperbow [207], and the Overtone Violin [143]. Max Mathews' Electronic Violin [111] can probably be put in this category as well.

Similar projects involve making "meta-trumpets" [80, 90] or a "meta-saxophone" [31], comparable in many respects to Softwind's Synthophone.[21]

1.2.2 Acoustically Driven Hybrid Instruments: Mixing Physical and Virtual Elements

Instead of being based on existing acoustic instruments, acoustically driven hybrid instruments use acoustic elements (e.g., membrane, solid surface, strings, etc.) to drive virtual (i.e., electronic, digital, etc.) ones. Virtual elements are implemented using physical modeling techniques. Their goal is to play to the strengths of physical elements (e.g., imperfection, tangibility, randomness, etc.) and combine them with the infinite possibilities of the virtual/digital world. Some of these instruments involve an analysis step to interface a physical element to a virtual one, while some of them directly use the acoustic signal generated by a physical element to drive virtual parts.

In this section, we give an overview of the different kinds of existing acoustically driven hybrid instruments. We demonstrate how they are usually more physically coherent than other types of DMIs and how this potentially makes them more expressive.

Using Acoustic Signals to Control Sound Synthesis

Audio signals contain a wide range of information that can be used to extract various types of parameters [46] such as the position of a strike/pluck and its velocity, etc. This idea has been extensively exploited by Tod Machover et al. in their series of "hyperinstruments" [106]. About ten years later, Caroline Traube worked on a new technique to estimate the position of a pluck on a string by analyzing the spectrum of its sound [186]. Pluck position information was used to control sound synthesis parameters.

[21] http://www.softwind.com/synthophone.html

Around the same period, a series of projects where acoustic instruments are used as musical controllers were conducted. Cornelius Poepel used an electric viola to control the parameters of a synthesizer [149]. Camille Goudeseune conducted a similar project, but applied to the violin [66]. Adam Tindale did the same thing for a drum that was used as a drum synthesizer controller [185].

Such systems solve many problems of more "traditional" controllers by providing an interface identical to the one that performers are used to, greatly facilitating skill transfer.

A wide range of commercial products based on this principle are available and can be used as a way to control sound synthesizers. The electric guitar is particularly well suited for such use as its "natural" sound is very quiet and doesn't interfere with the synthesized one. The Fishman TriplePlay Series[22] is a good example of this type of product and can be mounted directly near the bridge of the electric guitar to use this instrument as MIDI controller. Similarly, Roland's RT-30 Series[23] are "acoustic drum triggers" that can be mounted directly onto acoustic drums to trigger virtual ones. The idea in that case is to combine acoustic and virtual drum sounds. Roland's VDrums (see §1.1.5) use a similar approach but are based on drum membranes design to be as quiet as possible when they are struck.

Mogees[24] is an interesting special case allowing for the use of any surface as a controller by extracting features from the sound generated on those surfaces. It leverages years of research done at IRCAM[25] in this field [23].

Driving Virtual Elements With Acoustical Ones

The idea of driving virtual physical models with acoustical elements has been exploited by a few projects and commercial products. While physical modeling techniques are getting more advanced (see §1.5) and can be used to implement virtual versions of most musical instrument parts, controlling them in a natural expressive way is often challenging. They might also miss some key physical features that just can't be implemented in the virtual/digital world (e.g., sound radiation, etc.).

An early example of a acoustically driven hybrid instrument is the Korg Wavedrum.[26] Sounds happening on a "real" drum membrane are captured using contact microphones and used to drive a wide range of percussion instrument physical models. Since each sound excitation created on the membrane is different, generated sounds are very natural and realistic. A similar product is ATV's aFrame[27] that uses a polycarbonate surface to capture sound excitations to send them to a built-in sound synthesizer.

This technique has been extensively investigated by Roberto Aimi in his PhD thesis on hybrid percussions [6] and used in many other projects and products. For example, Miller Puckette mounted

[22] https://www.fishman.com/products/series/tripleplay/
[23] https://www.roland.com/global/products/rt-30h/
[24] http://www.mogees.co.uk/
[25] Institut de Recherche et Coordination Acoustique/Musique, Paris (France)
[26] http://www.korg.com/us/products/drums/wavedrum_global_edition/
[27] http://www.aframe.jp/

a contact microphone on the membrane of a snare drum [152] and used its signal to drive a physical model [151]. Beep Street's Impaktor[28] takes a similar approach by allowing any surface to be turned into a percussion instrument. It comes with a contact microphone that can be connected to an iOS device running the Impaktor app. Captured sounds are directly fed into a wide range of percussion instrument physical models to drive them. It is comparable to the Mogees presented earlier in this section in many respects.

Ali Momeni made Caress [133], a musical interface leveraging this principle as well. Finger pads with different textures can be touched to create various kind of sound excitations. Similarly, Maarten van Walstijn et al. [190] used the acoustic signal generated on a conga membrane to drive a percussion instrument physical model.

The same approach has been used with plucked string instruments where sound excitations are captured and used to drive virtual strings. Daniel Schlessinger's Kalichord [165] uses piezo films to "materialize" virtual strings. The sound of plucks is captured and fed into a simple string physical model (i.e., Karplus-Strong [89]). As for percussion instruments, the expressivity embedded in the acoustic excitation is transmitted to the digital portion of the system. Edgar Berdahl designed a similar instrument using a real electric guitar with damped strings that are used to capture pluck sounds through the built-in pickup. Pluck sounds are then fed to a string physical model to drive it [20]. The BladeAxe presented in §2.2 and §2.3 also leverages this principle. Sandor Mehes et al. nonlinearly coupled a physical string to a plate physical model, allowing performers to create very expressive sounds with unpredictable behaviors [113].

Other instruments such as the ones of Line 6's Variax Series[29] place the "real/virtual boundary" in a different location by having acoustical strings and a virtual body/resonator. In the case of the Variax, this allows the performer to play a wide range of guitars from a single one. This works especially well since guitar bodies are linear time invariant (LTI) systems that can be easily modeled using a modal physical model (see §1.5 and §6.2). Roland offers a similar product called the V-Guitar[30] where the sound of the strings is processed to apply the acoustical properties of various guitar bodies to it. Amit Zoran's Chameleon Guitar [213] is a pioneer in this field and also allows the performer to change the "physical" material of the front plate of the soundboard. Other types of instruments have been made using the same technique such as Friedrich Türckheim's Semi-Virtual Violin [188].

All the instruments presented so far in this section are based on one set of virtual and one set of physical elements connected together in a unidirectional way. However, many other types of instruments such as woodwinds need their elements to be connected a bidirectional way (e.g., a clarinet mouthpiece will transmit energy to the tube, but the opposite is true as well, etc.). Some instruments such as Alice Eldridge et al.'s Self-Resonating Feedback Cello [53] explored this idea.

[28] http://www.beepstreet.com/ios/impaktor/
[29] http://line6.com/variax-modeling-guitars/
[30] http://www.rolandus.com/go/v-guitar/

Similarly various experiments on the active control of some types of musical instruments have been conducted and are presented in more details in §E.

By using acoustical elements as an interface, instruments presented in this section implement a form of "passive haptic feedback," (i.e., the performer physically feels these elements while actuating them, even though they are not transmitting information from the virtual simulation via active force feedback). Moreover, they force the instrument maker to "co-design synthesis algorithms and controllers" reinforcing "the sense of intimacy, connectedness, and embodiment for the player and audience" [42].

1.3 Mobile Devices as Musical Instruments

Mobile devices (smart-phones, tablets, etc.) offer a generic platform for the design of digital musical instruments. As stand alone devices they present a promising platform for the creation of versatile instrumented for live music performance [16]. Within a single entity, sounds can be generated and controlled, facilitating the implementation of mutualized DMIs (see §1.1 and §1.2), and allowing for the creation of instruments much closer to "traditional" acoustic instruments this respect.

With mobile devices, hardware is generic and more or less the same from one device to another, regardless of the brand and operating system (e.g., iOS, Android, etc.), and it is the programmer's work to design apps to "make it more specific." In the case of mobile-device-based DMIs, the programmer truly becomes a "digital luthier" [84] able to create completely standalone instruments.

This section gives a brief overview of the field of "mobile music." Early works on using tablets as musical controllers are presented as well as more recent instruments. The research on human computer interaction done on this topic and the limitations of this type of system are also studied.

1.3.1 Towards Smart-Phones: Tablets and Tactile Interfaces

One of the main constituting element shared by all smart-phones and tablets is the touchscreen. Most interactions between the user and the software on such devices happen through this interface. While they only became "a standard" when Apple introduced the iPhone in 2007 (see §1.3.2), they were used on a wide range of devices before that [32]. The fact that they offer a generic hardware platform that can be configured for various applications exclusively through software was probably one of the keys of their success.

By offering multi-axis continuous controls, touch interfaces have always been an attractive platform for implementing musical controllers. Even though they rely on very different types of technologies, tablets controlled through an electronic pencil offer a similar interaction paradigm as touchscreens. The earliest example of this is Iannis Xenakis' UPIC (1977) [103], a custom device where a tablet equipped with an electronic pencil is used to control music synthesis. A similar approach is

used by more generic systems such as Wacom Tablets,[31] that have been extensively used as musical controllers [9, 210, 48]. The technology behind this type of system offers a relatively low latency compared to touchscreens using capacitive touch sensing such as the one presented later in this section (see §1.3.6). Electronic pencils offer a higher level of precision that fingers on a touchscreen and they're a simple solution to compensate for the lack of haptic feedback (see §1.3.5).

The tablet/pencil paradigm was exported to Personal Digital Assistants (PDA) that became quite popular in the 1990s and early 2000s. As touch sensing technologies improved, many PDAs abandoned electronic pencils, allowing users to interact with the screen directly with their fingers. The computer music community explored the potential of such devices [64], and a great part of the theory behind the design of touchscreen-based musical interfaces dates back from this period [65].

In the second half of the 2000s, a wide range of large multi-touch tablet music controllers sprung up and offered new paradigms to program and control digital musical synthesizers. The most "famous" of them is probably the Reactable[32] [83] where tangible objects can be placed on a screen to build sound synthesizers and control them. This idea was exploited by other controllers presented in §1.3.5. An iPad version of this system also exists. A very similar approach is taken by Auraglyph,[33] [161] an iPad app allowing the user to "hand sketch" digital synthesizers. Finally, the Lemur [96] was a versatile programmable multi-touch musical controller that now exists as an iPad app.[34] It is interesting to note that this trend is common to almost all the devices of this category [49].

With all these systems came the idea of abstracting existing instrumental gestures to transfer them to this type of interface. In other words, how can a touchscreen be used to facilitate skill transfer for specific types of musical instruments? This concept is well summarized by Günter Geiger:

> "One of the challenges when designing musical interfaces for the touch screen is to come up with a set of interaction principles, that can be regarded as fundamental, studied and practiced as such in order to lead to a higher level of control of the instrument (or, more general, the interface) [65]."

This idea is further studied and extended in §1.3.2 and §3.4.

1.3.2 Smart-Phone-Based Musical Instruments

The idea of using smart-phones as musical instruments emerged at the end of the first half of the 2000s [63]. Beside Günter Geiger's version of PureData[35] for Pocket PC briefly presented in §1.3.1, Atau Tanaka also studied the use of PDA as musical instruments [183]. He also explored the idea of

[31] http://www.wacom.com/
[32] http://reactable.com/
[33] https://ccrma.stanford.edu/~spencer/auraglyph/
[34] https://liine.net/en/products/lemur/
[35] https://puredata.info/

augmenting these devices with external sensors (see §5) which makes his work particularly relevant here.

Greg Schiemer et al.'s Pocket Gamelan [164] can probably be considered as the first smart-phone-based musical instrument. It featured real-time sound synthesis and playback as well as network capabilities through blutooth on a Nokia phone. The idea of passively augmenting the phone (see §4) was also explored by placing the device in a bag attached to a string used to spin the instrument during the performance.

In 2007, Apple released its first iPhone which completely revolutionized and redefined the emerging field of mobile music by providing an easily programmable platform with a multi-point touchscreen and an extended amount of memory. Pioneering work in using the iPhone as a musical instrument was carried out at CCRMA (Stanford University) and Smule[36] by Ge Wang. One of the most representative instrument from this period is probably Ocarina [197].[37] It embodies most of the concepts that we try to convey here by providing a standalone, physically coherent DMI, leveraging in a powerful way the various elements available on the mobile device while enabling skill transfer. For instance, the "four holes" interface to select pitch is perfectly adapted to the iPhone's touchscreen which is limited to five simultaneous touch points. A breath sensor is implemented using the built-in microphone and sound is synthesized and played back through the built-in speaker. From a design standpoint, this instrument is as easy and convenient to play as its acoustic counterpart, a rare quality for this type of DMIs. The performer just has to take his smart-phone and play without having to power or plug anything. Additionally, performances can be shared to the rest of the world providing unique social capabilities.

The paradigm initiated by the iPhone where smart-phones use a multi-point touchscreen as their main interface, host sensors such as an accelerometer, a gyroscope, a GPS, etc. quickly became a standard. The use of this new class of devices as musical instruments was extensively studied in the following years, both in terms of leveraging built-in sensors for musical expression [56, 58, 57, 184] and turning the touchscreen into a versatile music controller (see §1.3.4).

As part of this enthusiasm for mobile music, mobile phone orchestras sprung up in the second half of the 2000s. The Stanford Mobile Phone Orchestra (MoPhO)[38] [199] was one of the pioneers. As a one quarter class, students had to design their own instruments (apps) and write a musical piece for them, tightly following Norbert Schnell's idea of composed instrument [166]. The format of MoPhO instruments, performances, and pieces was not limited, and mobile devices could be treated as completely standalone musical instruments or as "a whole," using their network capabilities [30]. This last paradigm [76] is pushed further by specific ensembles where the mobile phone orchestra is treated as a singular entity. For example, Sébastien Piquemal et al. [44] use mobile devices to spatialize the sound of their pieces and provide the members of the audience (which in this case

[36] https://www.smule.com/
[37] https://www.smule.com/ocarina/original/
[38] http://mopho.stanford.edu/

are also the performers) with a simple interface. Their instruments are implemented as a web app, facilitating their distribution. Other performances/ensembles try to integrate smart-phone-based instruments to "traditional" orchestras. In his *Geek Bagatelles*,[39] Bernard Cavanna treats smart-phones as standlone musical instruments, just as any other instrument in the orchestra. Similarly, Xavier Garcia wrote a series of pieces[40] where smart-phone-based instruments are used concurrently with traditional ones. All the apps used for these two last projects were developed using the tools presented in §3.1.

Since the release of the iPhone in 2007 and the consequent "standardization" of smart-phones, this type of device didn't know any significant evolution. Their computational power significantly increased, allowing us to run complex synthesis algorithms in real-time. Similarly, the overall quality of the various sensors and technologies used on mobile devices improved, enabling a more precise control as well as reduced latency. The latest generations of high-end devices also allow for the measurement of the pressure applied by a specific finger on the touchscreen. This feature is very useful to create natural behaviors as it allows for the implementation of interactions happening on acoustic instruments. It has been exploited by various musical apps such Roli's Noise.[41] A solution for larger screen devices lacking pressure sensitivity is presented in §5.1.

While other manufacturers (e.g., Samsung, Google, etc.) and operating systems (e.g., Android, Windows, etc.) shortly started competing with Apple and its iOS[42] after 2007, it remained for a very long time the only viable option for mobile music (see §3.1.2). As such, most of the examples presented here were designed to work on iOS only.

Nowadays, a wide range of commercial apps turning smart-phones into musical instruments are available. For example, GarageBand for iOS[43] offers a wide range of touchscreen interfaces targeting performer skill transfer that are briefly presented in more details in §1.3.4.

1.3.3 Larger Screen Mobile Devices

While graphical tablets and large tactile interfaces and PDAs existed for decades (see §1.3.1), Apple released the iPad in 2010. It can be considered as the first commercially successful device of this type and it completely redefined (or even defined) their "standards," the same way as the iPhone did three years earlier for smart-phones. While the iPad (and what we now call "tablets" nowadays in general) has almost the same features as a smart-phone (e.g., touchscreen, built-in sensors, etc.), its large screen allowed for different affordances, particularly useful to implement musical interactions leveraging performer skill transfer (see §1.3.4).

As for the iPhone, pioneering work in exploring the potential of the iPad as a musical instrument

[39] `http://www.grame.fr/prod/geek-bagatelles/`
[40] `http://www.grame.fr/prod/smartfaust/`
[41] `https://roli.com/products/software/noise/`
[42] Apple's operating system for the iPhone and the iPad.
[43] `https://www.apple.com/ios/garageband/`

was carried out by Ge Wang et al. at CCRMA and Smule. Magic Fiddle [200] is one of the first apps for the iPad that targeted live music performance for this type of device. The screen interface implemented a violin neck, allowing for performer skill transfer (see §1.3.4), as well as various social and learning features.

iPads are, at the time of writing, widely used on stage as musical instruments. However, they are mostly used to replace laptops as sound synthesizers and to carry out basic interactions on the touch screen, the main musical interface being external (e.g., MIDI drum pads, keyboard, etc.). There exists dozens of apps based on this paradigm, and most electronic musical instrument manufacturers (e.g., Korg, Yamaha, Roland, etc.) now make this type of products. As such, the iPad provides a framework allowing us to build systems at a reduced cost that can compete with "traditional" digital synthesizers and workstations that have been dominating the market since the mid 1980s.

While the iPad/MIDI keyboard couple can be considered as a standard nowadays, only a few apps are based on the same paradigm as Magic Fiddle and turn the iPad into a completely standalone instrument, free from any external device. The design of the interface of this type of app is crucial to its success as an "accepted" musical instrument and must leverage existing skills for performers to be able to quickly master them. In the following section, the interfaces of such apps and the role they play facilitating skill transfer are studied.

1.3.4 Touchscreen and Skill Transfer

With the rise of mobile devices with larger multi-touch screens (see §1.3.3) in recent years, the appeal of making apps turning the iPad into a standalone musical instrument became much greater. Beside Smule's Ocarina presented in §1.3.2, there are very few other examples of such apps for smaller screen devices such as the iPhone. Thus, most of the apps presented in this section were designed to only work on the iPad.

In order for standalone iPad-based instruments to be successful, both from a musical and a commercial standpoint, they must be easily approachable by musicians. This implies the implementation of interfaces leveraging both the capabilities of multi-point touchscreens and existing musical skills sets. As described in §1.3 and by Günter Geiger in [65], it then all becomes a matter of abstracting the gestures linked to a specific instrument and conceptually reproduce them on a touchscreen.

Through his company Wizdom Music,[44] Jordan Rudess has been one of the pioneer in developing iPad apps usable on stage for live performance. He created a wide range of very innovative interfaces specifically designed for multi-point touchscreens and leveraging the various capabilities of the iPad. While many of them are meant to be controlled with a MIDI keyboard (e.g., Jordanotron, SampleWiz, etc.), some are designed to be more standalone such as Geo Synthesizer.

The interface of Geo Synthesizer served as the model for the app that probably embodies best the

[44] http://www.wizdommusic.com/

idea of turning the iPad into a self-contained instrument on the market: GeoShred.[45] It is the fruit of a collaboration between Wizdom Music and moForte.[46] Its primary goal is to turn the iPad into an electric guitar physical model controller/instrument. GeoShred offers an interface implementing a customizable isomorphic keyboard configured by default as a guitar neck. In other words, an arbitrary number of chromatic keyboards with an arbitrary number of keys are put in parallel of each other. Each keyboard row materializes a string, is mono, and implements "voice stealing," which means that priority is always given to the last finger to touch the "string." By default, keyboard strings are placed one fourth apart of each other, the same way as strings on a guitar. Notes are triggered when a new key is pressed. Slides and vibrato can be carried out. Features only implementable on a virtual system are also available such as the ability to automatically switch octaves when triads are performed successively. The GeoShred interface also emphasizes visual feedback, which is a common technique to compensate for the lack of haptic feedback on touchscreens (see §1.3.5).

GeoShred's interface is half way between a keyboard (since notes can be triggered when they are touched) and a guitar neck for the mapping of pitches and the general behavior of the system (more details about this type of mapping are given in §3.4.1). A large number of professional and renowned performers (i.e., Jordan Rudess himself) learned to master this instrument and reached a level of virtuosity comparable to that achievable on a traditional acoustic instrument.

While GeoShred is currently one of the only apps on the market treating the iPad as a completely standalone instrument, other apps use the multi-touch screen in innovative ways to implement various kinds of skill transfer. Apple's GarageBand for iOS[47] is one of them and provides a wide range of "smart" interfaces for various families of musical instruments (e.g., guitars, bass guitars, drums, pianos, string ensembles, etc.). Most of them try to copy/implement the pitch mapping and affordances (e.g., physical behavior of a string, ability to slide and carry out vibrato, etc.) of their real/physical world counterparts, in a similar way as Magic Fiddle does (see §1.3.3). Various modes can be used to quantize the pitch or limit the system to specific musical scales. More standard interfaces such as a configurable piano keyboard are also available. ThumbJam[48] is another example of such app and has similar goals as GarageBand.

Since the work done on graphic tablets, the study and the implementation of multi-touch screen interfaces leveraging instrumental skill transfer has been somewhat left out (see §1.3.1). A few systems such as TouchOsc[49] allow us to easily design iOS-based custom musical controllers, potentially allowing for the implementation of some forms of skill transfer. Olivier Perrotin et al. worked on a common problem when controlling pitch with touch interfaces: the ability to accurately select a pitch while being able to continuously control it (e.g., glissandos, vibrato, etc.) [147]. Similar techniques

[45] http://www.wizdommusic.com/products/geoshred.html
[46] http://www.moforte.com/
[47] https://www.apple.com/ios/garageband/
[48] http://thumbjam.com/
[49] https://hexler.net/software/touchosc/

are used in commercial apps such as GeoShred, GarageBand, etc. We introduce a slightly different method addressing this problem in §3.3.4.

1.3.5 Touchscreen and Tangibility

One of the main limitations of touchscreens are their lack of haptic feedback. While this might be compensated by the use of realistic and sometimes "exaggerated" graphical feedbacks, this remains a significant problem in the framework of digital musical instruments design where this type of features play a crucial role [140]. A wide range of solutions to this have been suggested and are briefly presented in this section.

Various types of active solutions have been proposed such as the use of mechanical and electrotactile feedback. Bruce Banter created a touchscreen equipped with actuators capable of generating vibrations, pulses, textures, etc. in response to user interactions [12]. Similarly, Ecran Altinsoy et al. added electrotactile feedback to a touchscreen, allowing for the creation of virtual tangible zones with different textures [7]. None of these two technologies have been used for musical applications though.

As demonstrated in §1.3.1, electronic pencils are a simple way to make the interaction with a touchscreen more tangible. This idea is pushed further by Müller et al. who created a system providing mechanical feedback through a motorized stylus [137]. This interface was specifically designed for musical applications and it allowed for the simulation a wide range of textures as well.

Other solutions to the lack of haptic feedback on touchscreens involve the use of tangible objects that can be placed directly onto the screen, which can also be seen as a form of augmentation (see §1.4.1). Completely passive solutions have been proposed by Sven Kratz et al. [93] as well as others [38, 101] where virtual objects can be materialized using physical ones. Basic interactions are possible and elements can be rotated, stretched, moved, etc. Information is transmitted to the screen using passive mechanical elements interacting with the capacitive touch sensing. This principle has been adapted to musical interface design by Edward Rutter [160] who created a collection of physical sliders, knobs, etc. directly usable on the iPad screen. Other projects involve the use of active systems to transmit data to the screen by electronically interfering with capacitive touch sensing using frequency modulation [209].

A different approach to solve this problem partly inspired by some ideas presented in [65] is introduced in §2.1.5. A thin "texturized" polycarbonate sheet is placed on the screen of the device, both to make it more tangible and to capture sounds happening on it through a set of contact microphones.

1.3.6 Limitations

Despite their countless qualities as a platform to make digital musical instruments, mobile devices were never designed to be used as such. While they can be easily turned into tools to carry out

specific tasks through their apps, their hardware remains very general and generic, and most mobile device-based DMIs can hardly compete with their real/physical world counterparts. §1.4, §4, and §5 all address this issue.

Beside the lack of haptic feedback of the touchscreen studied in §1.3.5, mobile devices have many other limitations. They are briefly reviewed here.

Other limitations of touchscreens are related to their use of capacitive touch sensing. This technology made tremendous progress during the past ten years, mostly because it is now used on almost all smart-phones. However, it remains relatively slow with a "typical" latency of about 35ms (see §3.2.3), often lacks the ability to sense pressure (see §1.3.2), and limits the number of simultaneous touch-points on some devices (e.g., five for the iPhones vs. ten for the iPad).

While mobile devices can be used to implement skill transfer by abstracting gestures associated with existing instruments (see §1.3.4), this often implies to hold them in a specific way. Smart-phones were primarily designed to be held in one hand – as a telephone – but Smule's Ocarina suggests the user to hold it with two hands, like a wind instrument. Since mobile devices don't have handles, the performer must always have at least one finger on the touchscreen to prevent it form falling, which is not very comfortable. Similarly, Magic Fiddle suggests to hold the iPad like a violin by placing it on the shoulder of the performer, which is not very practical. Various solutions to those purely ergonomic issues are proposed in §4.3.

Despite the fact that they are a key component of smart-phones "standaloneness," their built-in speakers are often weak and not good. While this limitation mostly comes from the size of the devices, there exists passive and active solutions that will be presented in §4.2.2 and §5, respectively.

The quality of real-time audio support greatly varies from one platform to another (i.e., iOS vs. Android), and has a significant impact on the performances of these systems, their deployability, and their audio latency. More details about these issues are provided in §3.2.

Similarly, mobile devices are usually less powerful than desktop computers (although the latest generations of high-end iPads tend to compete with most laptops). This might be a problem when designing DMIs using complex physical models such as the ones presented in §6 or when dealing with polyphonic instruments.

Finally, the lack of openness of some platforms can often be a problem for the deployment of some DMIs and the implementation of complex augmentations. For example, iOS apps can only be installed through Apple's App Store, and the requirements for apps to be available on this platform are high and often limiting. Similarly, connecting external devices to iPhones and iPads can be complicated since only MIDI devices are recognized and accepted on this platform. Various solutions to this problem are presented in §5.2. Android doesn't suffer from this problem and remains relatively open.

1.4 Augmenting Mobile Devices

The versatility of mobile devices is probably their main strength. They are generic hardware platforms that can be made specific through their software (apps) to carry out a wide range of tasks. On the other hand, mobile devices were not designed to do "everything" and their genericity is not always adapted to specific applications. §1.3 gives a good overview of this type of limitations.

One way to solve this problem is to use various kinds of elements that can be directly connected to mobile devices (e.g., MIDI controllers in the musical instrument world, etc.) to expand their affordances and functionalities. The main limitation of this type of solution is that it tends to make mobile devices less standalone. While this might not be an issue in some cases (e.g., keyboard-based instruments), it can be quite limiting for other types of instruments involving extended movements and gestures.

The main way to solve this problem is to make the external element part of the mobile device by turning it into a prosthetic that can be mounted on the smart-phone or the tablet itself. Various kinds of such augmentations, that can be both passive or active, have been designed and target a wide range of applications ranging from credit card readers to musical instruments. This paradigm was pushed further by the Ara,[50] a smart-phone series commercialized by Google and specifically designed to be modular and augmented.

This section gives an overview of the different types of augmentations of mobile devices towards musical instrument design through a series of examples.

1.4.1 Passive Augmentations

Many mobile device augmentations are completely passive and don't require to be connected to the device (as we'll see in §1.4.2, this can be sometimes technically challenging). Their goal is then to leverage existing components (e.g., built-in speakers, sensors, etc.) to improve their capabilities for specific applications. A popular example of this type of augmentation is the Google Cardboard,[51] where a cardboard box equipped with simple lenses is used to turn a smart-phone into a Virtual Reality (VR) headset.

In the world of audio, the most "famous" type of passive augmentations are probably passive amplifiers (e.g., Etsy amplifiers,[52] etc.). They plug directly to the built-in speaker of the mobile device to further amplify its sound using purely acoustic components. Thus, from an aesthetics and conceptual standpoint, they are very close to the instruments presented in §6.

Other types of passive augmentations leverage built-in sensors (see §4) by facilitating specific gestures. The AAUG Motion Synth Controller,[53] which allows the performer to hold the iPhone

[50] https://atap.google.com/ara/
[51] https://vr.google.com/cardboard/
[52] https://www.etsy.com/market/iphone_amplifier/
[53] http://www.auug.com/

with one hand is a good example of this type of augmentation. Performers can comfortably interact with the touchscreen while being able to use the built-in motions sensors.

Some projects focus on using/hacking built-in elements of mobile devices to implement passive or "semi-passive" external sensors. For example, in their Acoustruments project, Gierad Laput et al. [95] use the built-in microphone and speaker of smart-phones to implement passive, acoustically-driven controls for handheld devices. A broadband signal produced by the speaker is driven to the microphone using a tube whose topology can be modified by the user. Variations in the spectrum of the generated sound are tracked and high level parameters are extracted from them. Similarly, the passive tangible touchscreen objects [65, 93, 101] presented in §1.3.5 are also part of this category.

An important limitation of mobile device augmentations is their device specificity. For example, an iPhone 5 augmentation will not work with and iPhone 6 augmentation. A solution to this problem involving the use of digital fabrication and parametric CAD models is proposed in §4.1.

1.4.2 Active Augmentations

While passive augmentations are lightweight and usually perfectly integrate to mobile devices, they are limited in their scope and applications as they entirely rely on active elements already present on the device. On the other hand, active augmentations are more versatile, but are more technically challenging and hard to design. Indeed, since they rely on electronic components, they need to be powered. While mobile devices are capable of powering external elements through their USB, lightning, etc. port, it is often limited to a low amperage, requiring sometimes the use of an external power supply such as a battery or a power adapter. This type of elements might have an impact on the ergonomics and overall design of the augmentation. Transmitting data to the mobile device can also be problematic as plug types greatly vary from one device to another (e.g., USB, lightning, etc.). Similarly, even though they are getting better, wireless solutions often remain unadapted to musical applications because of latency (see §5.2). Additionally, some platforms such as iOS limit the types of elements that can be connected to mobile devices. A survey of the different techniques to transmit data to smart-phones and tablets is provided in §5.2.

Active augmentations can be as simple as plugging speakers to the jack output of mobile devices. This idea has been extended and used in an innovative way by some mobile phone orchestras such as MoPhO (see §1.3.2) where external battery-powered speakers are mounted on the hands of the performer using gloves. This solution is obviously more efficient than the use of passive amplifiers (see 1.4.1) and doesn't affect the standalone aspect of the device.

A popular type of augmentation in the audio world are external audio interfaces that can be mounted directly onto the device. For example, Alesis' iO Dock[54] turns the iPad into a simple audio work station. Similarly, the Sonoma Guitar Jack[55] simplifies the connection of an electric guitar to

[54] http://www.alesis.com/products/view/io-dock/
[55] https://www.sonomawireworks.com/guitarjackmodel2/

an iPhone, etc.

Other products use the iPad and the iPhone at the heart of electronic musical instruments as a sound synthesizer or audio effect processor. Akai's SynthStation49[56] augments the iPad with a MIDI keyboard and a wide range of controllers (e.g., knobs, sliders, etc.). DigiTech's iPB-10 Programmable Pedalboard[57] is an iPad-based guitar pedal effect board. Akai's MPC Fly 30[58] is an iPad sleeve turning it into a drum pad and a sampler. There are many more examples of this type of augmentations on the market.

The gTar[59] represents a specific case in this world of augmented smart-phone-based musical instruments as it targets education. It is a full-size guitar controller where frets on the neck are implemented with buttons. Its general design borrows a lot from the Starr Labs Ztar presented in §1.1.4. An iPhone can be docked to the instrument to synthesize sound and to run a wide range of programs to learn how to play guitar.

Another good example of augmented iPhone-based musical instrument is the early prototype of the Artiphon INSTRUMENT 1 briefly presented is §1.1.5 where the mobile device was literally placed at the heart of the instrument.

Overall, most of the products presented in this section date back to the early 2010s. They are all discontinued and haven't been replaced by new ones. Despite the appeal of using mobile devices as the basis for the implementation of a wide range of DMIs, their quick evolution, both in terms of technology and shape, became a significant limitation for the manufacturers of such instruments. Relying on a specific device appeared to be very risky as their manufacturers (e.g., Apple) constantly change their design and shape, making most augmentations obsolete only after a few months.

All commercial products presented in this section use mobile devices to carry out sound synthesis or processing tasks, however, they completely discard the various sensors already present on the device (e.g., touchscreen, motion sensors, etc.). Inversely, the instruments presented in this thesis in general, are committed to start from the mobile device, leverage its existing elements, and finally design augmentations for missing affordances.

1.5 Physical Modeling

In this section, we give a brief overview of waveguide [177] and modal synthesis, and we show how they can be used to efficiently and easily implement a wide range of musical instrument parts. We also highlight the link between the virtual/digital and the physical/acoustical world from the musical instrument maker/designer perspective, enabled by the combination of physical modeling and digital fabrication. Finally, we show how the FAUST programming language has been used to implement

[56] http://www.akaipro.com/product/synthstation49/
[57] http://digitech.com/en/products/ipb-10-programmable-pedalboard/
[58] http://www.akaipro.com/mpc-fly-30/
[59] https://www.kickstarter.com/projects/incident/gtar/

physical models of musical instruments in the past.

1.5.1 Digital Waveguides and Modal Synthesis

Smith et al. extended the Karplus-Strong algorithm [82] and generalized it to other types of instruments through digital waveguide models [173, 88, 194]. Such models are "built out of digital delay-lines and filters (and nonlinear elements), and they can be understood as propagating and filtering sampled traveling-wave solutions to the wave equation (partial differential equation), such as for air, strings, rods, and the like" [177].

The main advantages of digital waveguide models are their simplicity and efficiency while still sounding adequately real. They allow for the accurate modeling of a wide range of instruments (string and wind instruments, tonal percussions, etc.) just with a single filtered delay loop. Efficiency was a key factor in the success of waveguide physical modeling at a period when CPUs were not as powerful as nowadays. This technique was used in many commercial synthesizers in the 1990s such as the Yamaha VL1 (see §1.1.2) as well as by various composers [37, 92].

While any instrument part implementing a quasi harmonic series (e.g., a linear string, tube, etc.) can be modeled with a single digital waveguide, this technique is not suitable for strongly non-harmonic systems.

Modal synthesis [5] consists of implementing each mode of a linear system as an exponentially decaying sine wave. Each mode can then be configured with its frequency, gain, and resonance duration (T60). Since each partial is implemented with an independent sine wave generator (see §6.3.2), this technique is more computationally expensive than waveguide modeling for systems with many modes. The parameters of a modal synthesizer (essentially a list of frequencies, gains, and T60s) can be calculated from the impulse response of a physical object [87] or by using the Finite Element Method (FEM) on a volumetric mesh (see §6.3.1) [29]. This technique strengthens the link between physical elements and their virtual counterparts as it allows for the design of an instrument part on a computer using a CAD software, and turn it into a physical model that could also be materialized using digital fabrication (see §1.6). This concept is further developed in §6.

Other methods such as finite-difference schemes [24] can be used to implement physical models of musical instruments and provide more flexibility and accuracy in some cases. However, most of them are computationally more expensive than waveguide models and modal synthesis. An overview of these techniques is provided in [177]. Since this thesis is targeting the use of physical models on mobile platforms with a limited computational power, we're focusing on CPU-efficient techniques.

1.5.2 Physical Modeling Environments

Designing physical models of musical instruments from scratch can be hard and requires a deep knowledge of the different techniques used for this purpose. On the other hand, working with a high-level digital representation of physical objects (e.g., instrument body, strings, membranes, etc.)

to assemble them into existing or new instruments is very appealing. There exists a few environments to carry out this type of tasks. They are briefly presented in this section.

Modalys [54, 35] has been developed for years by IRCAM[60] and probably represents best this type of environments. It uses different physical modeling techniques such as modal synthesis and digital waveguides to implement various musical instrument parts or vibrating objects that can be assembled together through a high-level programming language based on Common Lisp.[61] Modalys inherited from an older environment called Mosaic [134] and benefits from decades of research done at IRCAM on these topics.

CORDIS-ANIMA [34] allows for the implementation of mass/spring physical models [60] using a graphical user interface.

Edgar Berdahl's Synth-A-Modeler [18] adopts a higher level approach by providing a graphical user interface to assemble musical instrument parts. It is based on a set of functions implemented in the FAUST programming language (see §1.5.3) and focuses on hardware integration for haptic feedback (see §1.1.7). There are some similarities between Synth-A-Modeler and the FAUST Physical Modeling Library presented in §6.2.

Other environments target specific classes of musical instruments. Digital Guitar Workshop [10] is a tool to help guitar luthier design new instruments by modeling their acoustical behavior. Harrison et al. [71] created a toolkit incorporated to the Sound Loom interface of the Composers Desktop Project[62] to model brass instruments using finite-difference schemes [24].

1.5.3 FAUST and Physical Modeling

The FAUST programming language has been used extensively to implement waveguide and modal physical models of musical instruments. The FAUST-STK [125] presented in §A is a collection of physical models implemented in FAUST and based on some of the algorithms of the "original" STK [43]. Additionally, Julius Smith describes how to implement a virtual electric guitar and its related effects in [176].

The implementation of such models in FAUST is eased by the wide range of functions available in the FAUST DSP libraries [123] and by the block-diagram/signal-oriented syntax of the language.

In §6, we introduce a series of tools to facilitate and generalize the design of physical models of musical instruments in FAUST. The goal of some of these tools is to blur the physical/virtual boundary towards the design of hybrid mobile instruments.

[60] Institut de Recherche et Coordination Acoustique/Musique: `https://www.ircam.fr/`
[61] `https://common-lisp.net/`
[62] `http://www.composersdesktop.com/`

1.6 3D Printing, Acoustics, and Lutherie

Digital fabrication has been used to make musical instruments for decades and is now an established process part of many luthier's toolkit. For example, CNC[63] machines are widely used by electric guitar makers to "sculpt" guitar bodies using 3D models.

A special case in the wide range of digital fabrication techniques is 3D printing. While 3D printers have been used since the 1980s [25], they only became a popular technique in recent years. This, combined with the significant decrease of the price of printers (typically under $3000), triggered a "3D printing mania" in the early 2010s. In a speech on February 12, 2013, former president Barack Obama said:

> "A once-shuttered warehouse is now a state-of-the art lab where new workers are mastering the 3D printing that has the potential to revolutionize the way we make almost everything [69]."

Both the acoustic and the digital musical instrument making communities were affected by this, and 3D printing has been used extensively in the past five years to make novel, traditional, acoustic, digital, etc. musical instruments [102, 68]. Nowadays, even though media attention significantly decreased [130], 3D printers are part of many workshops and are fully integrated to the tool-chain used by musical instrument makers.

While high-end 3D printers can be used to make full size traditional acoustic musical instruments (usually drawing media's attention), cheaper printers are often utilized to augment or modify existing instruments or make new ones from random objects. Fast prototyping and iterative design are at the heart of this new approach to lutherie and musical instrument making.

1.6.1 Printing Musical Instruments

Despite some technical limitations, 3D printers have been used to print musical instruments parts or complete instruments for several years. 3D printing techniques significantly evolved during the last decade, allowing for the printing of a wide range of materials with a high level of precision. It is now even possible to print wood grain and change the physical properties of a material in different locations [45]. However, these features are only available on high-end printers, at a prohibitive cost for most musical instrument designers/makers. Additionally, lutherie and traditional acoustic musical instrument making require extremely advanced technical skills and a high level of intuition that can only be acquired after years of practice and apprenticeship [84]. Such skills are currently hard to translate to digital fabrication.

> "We should not forget that 3D printing technologies, or any other novel fabrication technologies, cannot compete with the traditional ones in the process of fabricating traditional

[63] Computer Numerical Control

> instruments. However, the new technologies have the potential to change instrument design, and to open a door for new acoustic experiments and musical possibilities. As such, digital instruments, acoustic 'traditional' instruments, and acoustic 'experimental' instruments can coexist, and can perhaps merge into hybrid instruments, integrating different qualities and different fabrication technologies [211]."

Despite these limitations, 3D printing has been used to make successful instruments and is an ideal tool to prototype novel musical instruments. This section gives a non-exhaustive overview of these digitally-fabricated instruments.

String instruments are particularly well represented in this world. It is now a common fabrication technique for the making of electric guitar bodies since those have little impact on the quality of the generated sound. Some guitar makers [142] such as Olaf Diegel[64] specialized in this type of instruments. A similar approach is taken by Laurent Bernadac who makes and sells a 3D printed electric violin: the 3DVarius.[65]

More "adventurous" projects aim at acoustic string instruments. Scott Summit's acoustic guitar is a famous example of such instrument [191]. The associated patent [181] gives plethora of technical details on its making and design. As the founder of one of the first 3D prosthesis company, Summit had access to very high-end printers normally used for medical applications, allowing him to print the body of his guitar as a single part with a high level of precision.

The Hovalin[66] is an "open source violin," 3D printable on low-end printers. Since the bed of such printers is relatively small, the instrument has to be printed in several sections that can be glued together. Similarly, ukulele are popular 3D printable instruments,[67][68] probably because of their size and also the fact that the quality of their making usually matters less than for other types of string instruments.

A wide range of existing and novel wind instruments have been 3D printed in the last decade. Amit Zoran made a full transverse flute [211] where he printed each part of the instrument independently on a high-end printer and assembled it by hand. He gives interesting insights on the acoustic simulation of 3D models of musical instruments prior to printing that are reused in §6.

3D printing has been used to explore unusual temperament on traditional wind instruments. For example, Nicholas Bailey et al. made a microtonal clarinet [11]. Similarly, Terumi Narushima et al. made a set of microtonal flutes and also prototyped a wide range of new designs for this instrument such as a double helix flute [47].

Other projects around wind instruments are open source and can be be printed on low-end

[64] http://www.oddguitars.com/
[65] https://www.3d-varius.com/
[66] http://www.hovalabs.com/hova-instruments/hovalin/
[67] https://3dprint.com/63412/3d-printed-electric-ukulele/
[68] http://www.3ders.org/articles/20130221-3d-printing-musical-instrument-ukulele.html

printers such as Aito Esteve Alvarado's baroque recorder"[69] or Dan Olson's trumpet.[70]

Finally, a few instrument makers designed completely novel instruments that would be hard to make without the use of 3D printing. Eric Goldemberg and Veronica Zalcberg created a wide range of such instruments as part of their Monad Studio project.[71]

1.6.2 Modifying/Augmenting Existing Objects and Musical Instruments

One of the main power of 3D printing lies in the ability to fast-prototype elements that were either hard or impossible to make in a small workshop in the past. As a result, the idea of making small augmentations to extend existing musical instruments or make new ones from random objects rose up. This idea is investigated and theorized by John Granzow in his PhD thesis [68].

> "The scrap yard becomes like the archeological dig, prone to what we might playfully call the Martian perspective, a forestalled recognition of things and their functions, as if we, or these things come from another world. Whether the vision is Martian or future-archeoacoustical, the exercise remains the same: To resist preconceptions and allow the object to float into imaginaries of altered function. Part of this process is a mental sketching around the object, projecting alternative assemblies upon it. This mental sketching now transitions quickly to physical form when the observer has access to CAD and additive manufacturing toolchains [68]."

This concept gave birth to a wide range of new musical instruments such as Granzow's Javalele, a string instrument made from a portafilter found in a scrap yard and resembling a small banjo.

Other people explored this idea such as Matt Pearson who made a wide range of musical instruments based on gourds augmented through 3D printing.[72] Strassel Guitar[73] makes electric guitars with modular 3D printed pickups that can be used to change the sound of the instrument. A similar concept is exploited by Amit Zoran with his Chameleon Guitar [213], where the 3D printed front plate of a guitar sound board can be changed to obtain different sounds. Through this type of instrument, Zoran [212] partially introduced the idea of hybrid instrument presented in §6.

CAD modeling allows for the creation of parametric models that can be modified using high level parameters. This potential has been exploited to allow amateur musical instrument makers to print custom versions of instrument parts. For example, Austin Peppel developed a toolkit allowing the user to change the properties of a violin bow through a parametric CAD model,[74] facilitating the making of a wide range of bows for various uses and applications. Similarly, Morton Underwood developed a framework to design 3D printable musical instrument mouthpieces.[75] This idea had

[69] https://3dprint.com/88386/3d-printed-baroque-recorder/
[70] https://3dprint.com/41360/3d-printed-trumpet-project/
[71] http://www.monadstudio.com/
[72] http://www.mattpearsonworkshop.com/
[73] http://www.strassellguitars.com/
[74] http://www.austinpeppel.com/3D-Printable-Customizable-Musicians-Bow/
[75] http://www.mortonunderwood.co.uk/production/3d-printed-mouthpieces/

been extended by John Granzow et al. as well [98].

1.6.3 Other Uses

3D printing has been used for other types of projects in the world of musical instruments design and audio in general. In acoustics, it is used by architects to print miniature models of concert halls. That's the case of the Walt Disney Concert Hall that was designed in partnership with famous acoustician Yasuhisa Toyota. As a privileged tool for designers, it makes it possible to explore different shapes for speakers such as the Dardanus or passive amplifiers[76] similar to the ones presented in §1.4.1. Similarly, a few companies such as Normal[77] make 3D printed customized earphones adapted to sepecific ears.

There also exists more experimental projects such as Stephen Barrass' digital fabrication of acoustic sonifications where dataforms with acoustic properties that provide useful information are created [13].

With 3D printing and physical modeling (see 1.5), the boundary between the physical/acoustical and the virtual/digital worlds becomes more blurry, making it possible to approach musical instrument design in a multidimensional way where instrument parts can be either physical or virtual (with some limitations). §6 develops this concept and provides various tools to create such instruments.

[76] http://3dprintingindustry.com/
[77] http://newnrml.com/

Chapter 2

Genesis

The concepts and ideas presented in this dissertation partly originated from the various work presented in §1, but also from the design of a series of musical instruments and art installations that we developed in the course of the past five years. While some of them just leverage the idea of hybrid instrument developed in §6, others combine it with the concept of mobile device augmentation introduced in §4 and §5.

This chapter gives an overview of these instruments and installations in a chronological order and links them to the following chapters of this dissertation.

2.1 Towards the BladeAxe

2.1.1 The Féraillophone

The Féraillophone is the first instrument that we made that used physical/acoustical elements to drive virtual/digital ones. It is based on a compact disc and a laser disc (see Figure 2.1) that can both be used as exciters to drive the virtual portion of the instrument. Contact microphones placed on the two discs pick up sound excitations that are then digitized, and fed directly into a series of physical models to excite them. This technique is borrowed from some instruments presented in §1.2.2 such as the Korg Wavedrum. By combining the randomness of "real-world" excitations to the infinite possibilities of physically-informed virtual elements, very expressive sounds can be produced. Models range from a simple Karplus-Strong [89] to a more advanced non-linear feedback delay network similar to the one described in [178] and capable of generating various types of percussion sounds (e.g., gongs, cymbals, etc.). A companion interface can be used to control some of the parameters of the models.

The Féraillophone has been used in various performances including in *Study for Féraillophone*, a musical piece specifically written for this instrument. More technical details about this piece and

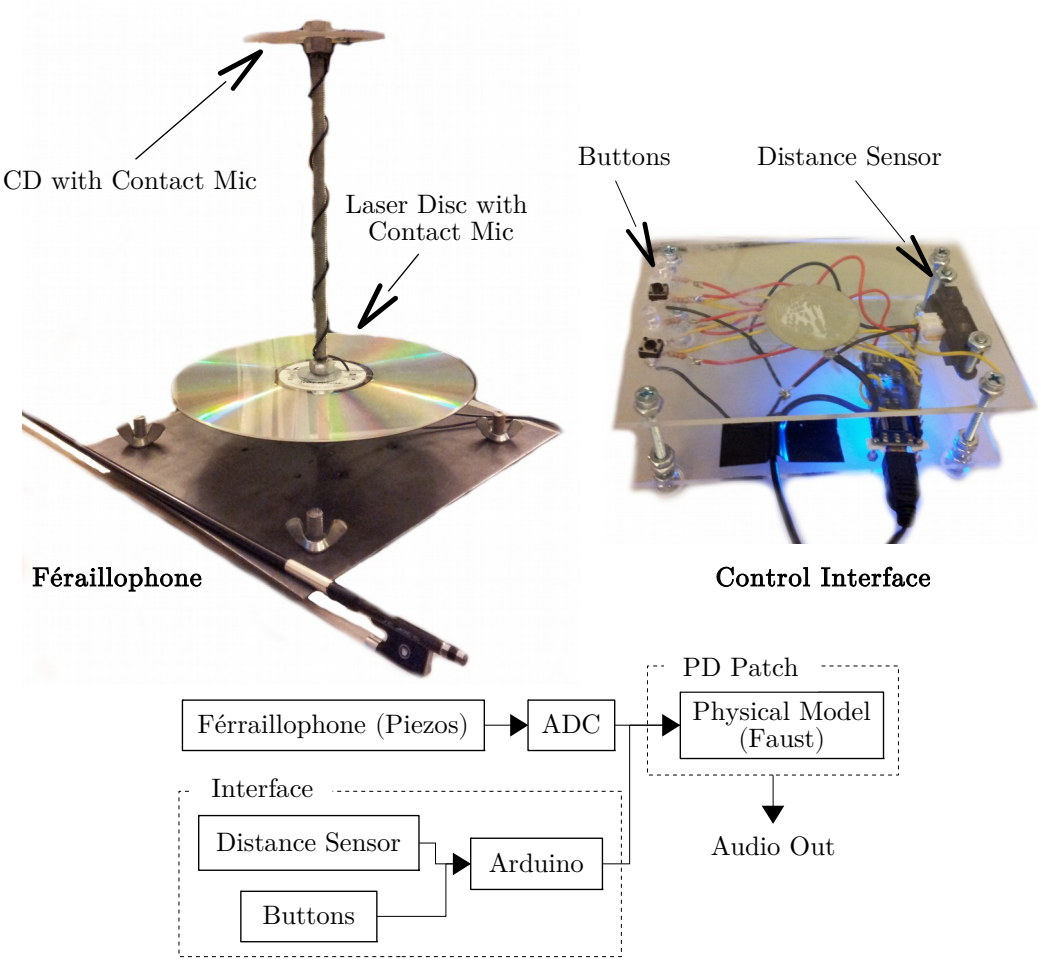


Figure 2.1: The FÉRAILLOPHONE, Its Companion Interface, and Overview of the Implementation of the System.

the design and implementation of this instrument are available on the corresponding webpage.[1]

2.1.2 The HybridScreen

The HybridScreen uses the same principles as the Féraillophone presented in §2.1.1. It is based on a screen with a transparent acrylic sheet mounted on it (see Figure 2.2). Textures were manually engraved on the acrylic using various tools (dremel, sand paper, etc.). Each texture generates a different sound or rhythmic pattern. A contact microphone glued to the acrylic sheet picks up the sounds happening on it. They are then digitized and used to drive a series of simple modal physical models [5] from the Faust-STK, [125] ranging from Tibetan bowls to metal bars, etc. The screen provides visual feedback to the performer by displaying a 3D sphere whose color and size change in function of the amplitude and the spectrum of the generated sound. A distance sensor can be used to control some of the parameters of the models (e.g., duration of the resonance, pitch, etc.).

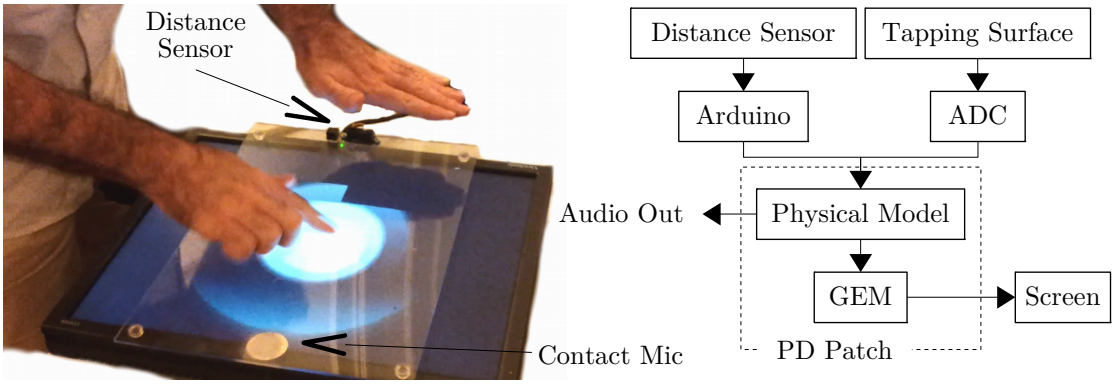


Figure 2.2: Overview of the HybridScreen.

More technical details about the HybridScreen as well as videos of performances using this instrument can be found on the corresponding webpage.[2]

2.1.3 The BlackBox

The BlackBox [117] is a sound installation leveraging the same principles as the Féraillophone and the HybridScreen presented in §2.1.1 and §2.1.2, respectively. A cube hung from a geodesic dome (see Figure 2.3) is used as a multi-modal interface to control various physical models. Sounds happening on the faces of the cube are picked-up, digitized, and used to drive the physical models that are similar to the ones used by the Féraillophone (see §2.1.1). Resulting sounds are played

[1] https://ccrma.stanford.edu/~rmichon/feraille/
[2] https://ccrma.stanford.edu/~rmichon/hybridScreen/

on a 4.1 sound system mounted inside the dome. Each face of the cube provides a different rhythmic pattern to play with. A built-in accelerometer is used to control some of the parameters of the physical models (e.g., duration of the resonance, pitch, etc.). Visual feedback is provided through a set of RGB LEDs placed inside the cube.

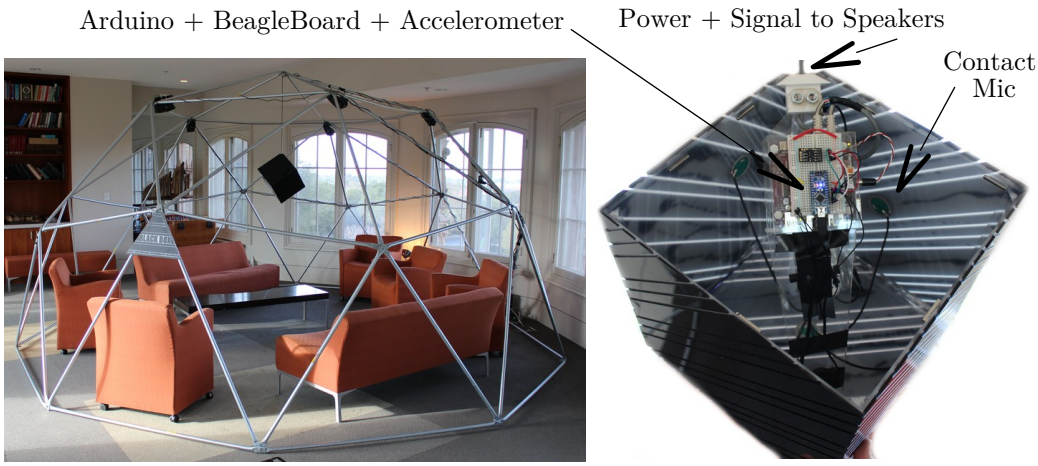


Figure 2.3: The BlackBox Installation in the CCRMA Lounge and Detailed View of the System Inside the Cube.

Additional technical details and demo videos can be found on the corresponding project webpage.[3]

2.1.4 The Chanforgnophone

The Chanforgnophone (see Figure 2.4) is a musical instrument based on a large steel cube with strings mounted on its faces. A set of contact microphones pick up the sounds happening on the instrument at different locations and feed them into a physical model. The model is a 3D mesh implementing a cube whose dimension is the same as the one of its real-world counterpart. The physical cube can be driven with a large transducer to create complex (and very unstable) feedback behaviors. A set of springs mounted inside the cube can be used to process the sound. Finally, an array of distance sensors can be used to control various parameters of the system.

The main idea is to have a hybrid instrument where physical elements all have a virtual equivalent. Connections between physical and virtual parts are made through contact microphones whose signals are digitized and are used to drive virtual elements. Inversely, virtual elements are connected to physical ones using transducers. Physical connection positions are respected both for virtual and physical elements. This concept is extended and generalized in §6.

[3] https://ccrma.stanford.edu/~rmichon/blackbox/

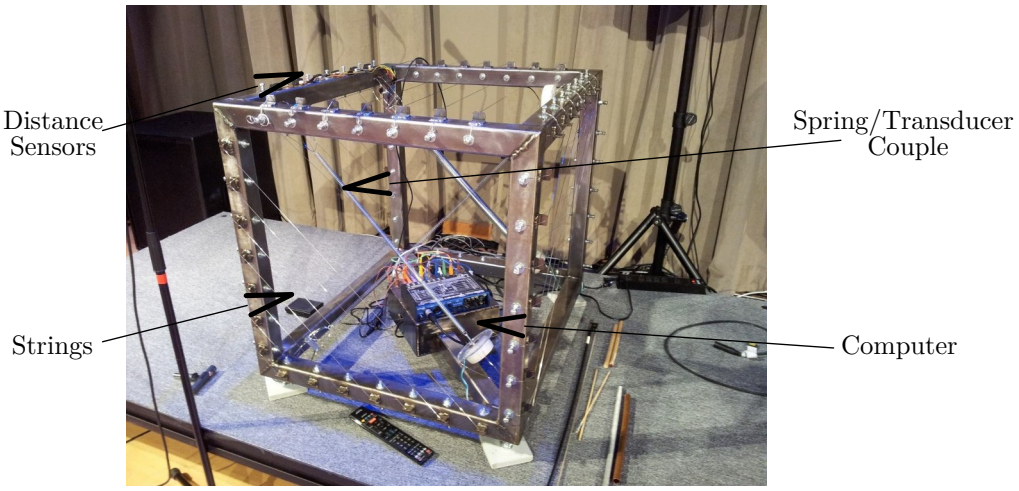


Figure 2.4: Overview of the CHANFORGNOPHONE.

More technical details about the CHANFORGNOPHONE as well as the description of *Bruits pour Chanforgnophone*, a musical piece written for this instrument can be found on the corresponding project webpage.[4]

2.1.5 Augmented iPads

After experimenting with the idea of driving physical models with real-world acoustic excitations (see §2.1.1, §2.1.2, §2.1.3, and §2.1.4), we started trying to integrate this concept to mobile device-based instruments. One of our goal was to make these instruments standalone and compact. As part of this project, we designed a series of prototypes that are briefly presented in this section.

The first one (see Figure 2.5) uses 3D printed plates hosting various textures, rhythmic patterns, etc., to drive a wide range of physical models. Each plate was designed like a "playground" for the performer's fingers. Plates are mounted on a custom iPad cover in order to be placed on the back of the device. Small contact microphones on the plates capture sounds happening on them and can be connected to the iPad through the headphone jack. The performer holds the instrument with his two hands and can use the build-in accelerometer and his thumbs on the touchscreen to control some of the parameters of the model.

Our second prototype (see Figure 2.6) uses a transparent polycarbonate film placed on the touchscreen of the iPad to add tangible textures to it. The layer is thin enough to not impact the quality of the capacitive touch sensing and was designed to be as seamless as possible. A small contact microphone placed on the frame holding this layer on the screen picks up sounds and feed them into various physical models. The touchscreen can be used to change some of the properties

[4] `https://ccrma.stanford.edu/~rmichon/chanforgnophone/`

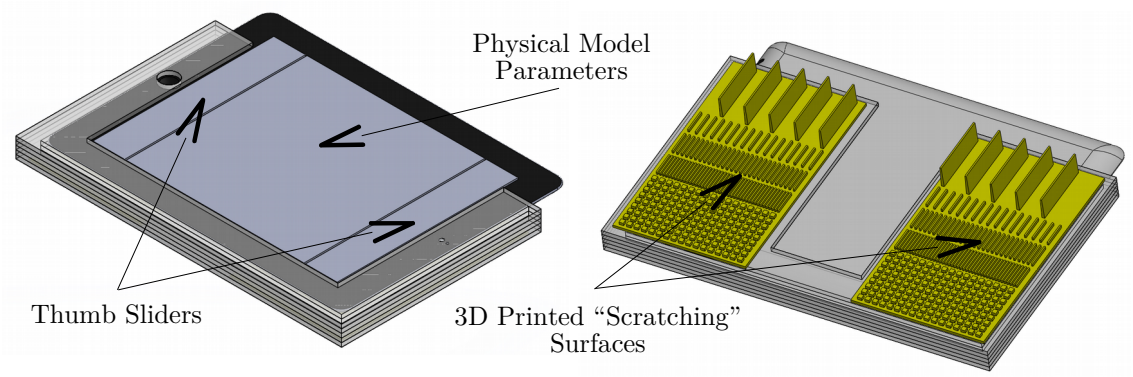


Figure 2.5: Portable Augmented iPad.

of the model. The main idea behind this instrument is to allow the performer to trigger the sound and control some of its parameters with a single gesture.

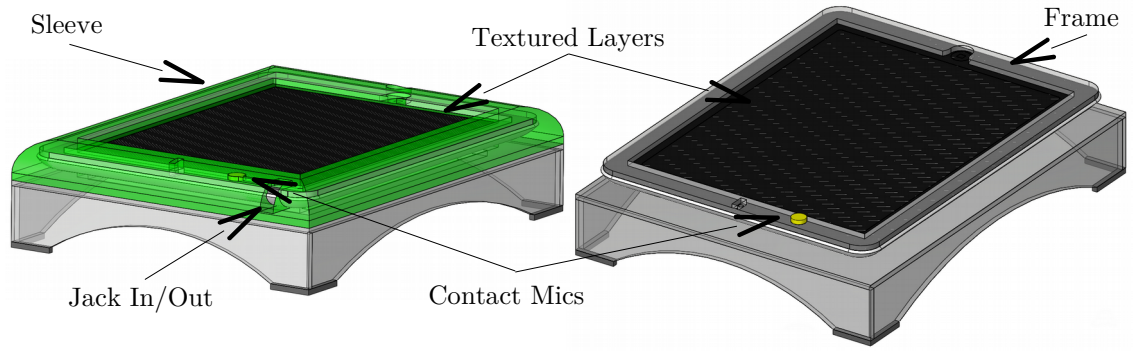


Figure 2.6: iPad Augmented With a Texture Layer on Its Touchscreen.

The apps of both instruments were made from scratch and programmed in C++ and Objective-C for the interface, the audio engine, etc., and in FAUST for the DSP portion. §3.3 introduces a tool to facilitate the design of this type of app.

Similarly, the hardware portion of these instruments (i.e., the 3D printed textured plates, the iPad case, etc.) were made from scratch in SolidWorks.[5] A framework facilitating the design of mobile device augmentations is presented in §4.1.

[5] http://www.solidworks.com/

2.2 The BladeAxe1: a Hybrid Guitar Physical Model Controller

After the various experiments presented in §2.1.1, §2.1.2, §2.1.3, §2.1.5, and §2.1.4, we tried to design a guitar physical model controller where virtual strings are controlled by "real-world" acoustical excitations. Our primary goal was to create a "physically coherent" instrument, following the ideas and principles presented in §1.1 and §1.2. During this iterative process, we went through several versions of the system where various neck designs were prototyped, multiple techniques and materials were tried to capture plucking gestures, etc.

This section is based on a paper[6] that we published at the 2014 International Computer Music Conference[7] on *A Hybrid Guitar Physical Model Controller: The* BladeAxe [128]. It gives an overview of the design of the first version of the BladeAxe: the BladeAxe1.

2.2.1 Plucking System

Hardware

The plucking system of the BladeAxe1 is based on six independent piezoelectric films: one for each string. We chose to use SEN-09196[8] because of their reasonable cost (about three dollars each), their simplicity, and also because they are very common on the market and thus easy to replace.

Each piezo is glued to the middle of separate polycarbonate sheets (see Figure 2.7). The height and the thickness of these sheets is chosen to approximate the elasticity of a guitar string. Each sheet is placed in parallel and is 3/8" apart from its neighbors (corresponding to the distance between the strings on a "standard" guitar).

Piezoelectric films are very sensitive to electromagnetic fields like the one created by the human body. Thus, they were shielded with grounded copper tape to prevent disturbances in their signal. Two additional bands of copper tape are used on each side of the piezoelectric film as capacitive touch sensors to detect if the skin of the performer is touching the piece of polycarbonate. This system is used to damp the virtual strings when they are not plucked and to detect the plucking position of the player (§2.2.2 provides more details on the way this system works).

Each piezo is connected to an independent preamplifier that also takes care of canceling DC (when a constant pressure is applied to a piezoelectric system, it creates a continuous current). Also, each preamplifier contains a lowpass filter with cut-off frequency at 10 kHz. Piezoelectric films have a very strong resonance peak at around 16kHz[9] and it is crucial to eliminate this part of the spectrum of the signal before feeding it into a waveguide.

[6] Some sections and figures of this paper were copied verbartim here.
[7] http://icmc14-smc14.eu/
[8] http://dlnmh9ip6v2uc.cloudfront.net/datasheets/Sensors/ForceFlex/LDT_Series.pdf
[9] http://en.wikipedia.org/wiki/File:Piezoelectric_sensor_frequency_response.svg

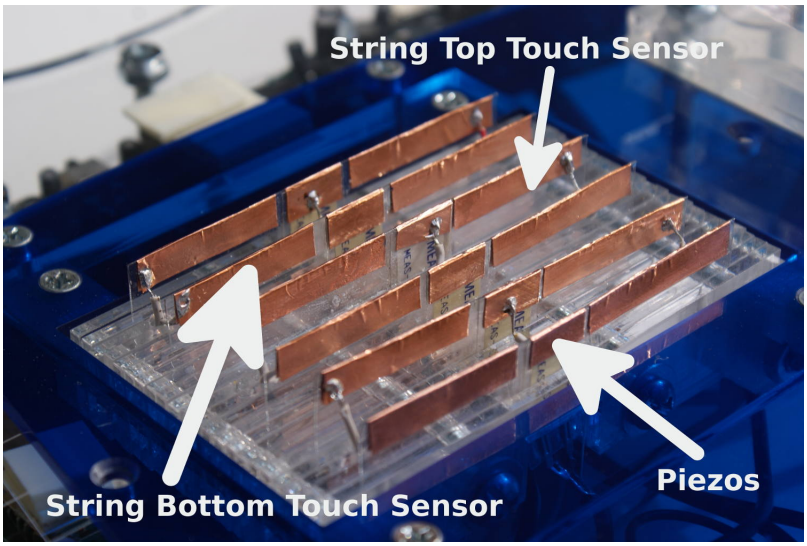


Figure 2.7: The Plucking System of the BladeAxe1.

Finally, the output of each preamplifier is independently digitized and sent to the laptop to be fed into the physical model. The sampling rate of the system is 32KHz (the analog signals of the piezos are bandlimited to 10KHz by the preamp's lowpass filters).

The plucking latency is fully determined by the Analog to Digital Converter (ADC) latency which is about ten milliseconds on the BladeAxe1. This is fine for most cases but it could be improved by using a better ADC.

Pluck Sound Analysis

The nature of the impulses created by the finger when plucking one of the polycarbonate sheets can vary a lot as a function of the pluck type. This is what makes the sound of the BladeAxe1 so natural. For example, the impulse will be different if the pluck is carried out with a nail, the skin of the finger, a bow, etc., just like on a "traditional" guitar.

The excitation signal also naturally embeds the pluck position along the edge of the blade: for example, Figure 2.8 shows more energy below 1KHz and less energy above 4KHz when one of the blade is plucked at the middle rather than at one end. The pluck position uniformly affects the spectra of the virtual strings (see §2.2.2), for example the relative amplitude of the fundamental (typically below 1KHz) versus the higher harmonics, as shown in Figure 2.9: the virtual strings sound like they are plucked at different positions.

As the two sides of the polycarbonate sheet sound the same, the capacitive touch sensors are used to determine if the impulse was created closer to the bridge or to the nut. A digital filter is applied to the impulse as a function of this parameter (more details on this point are provided in

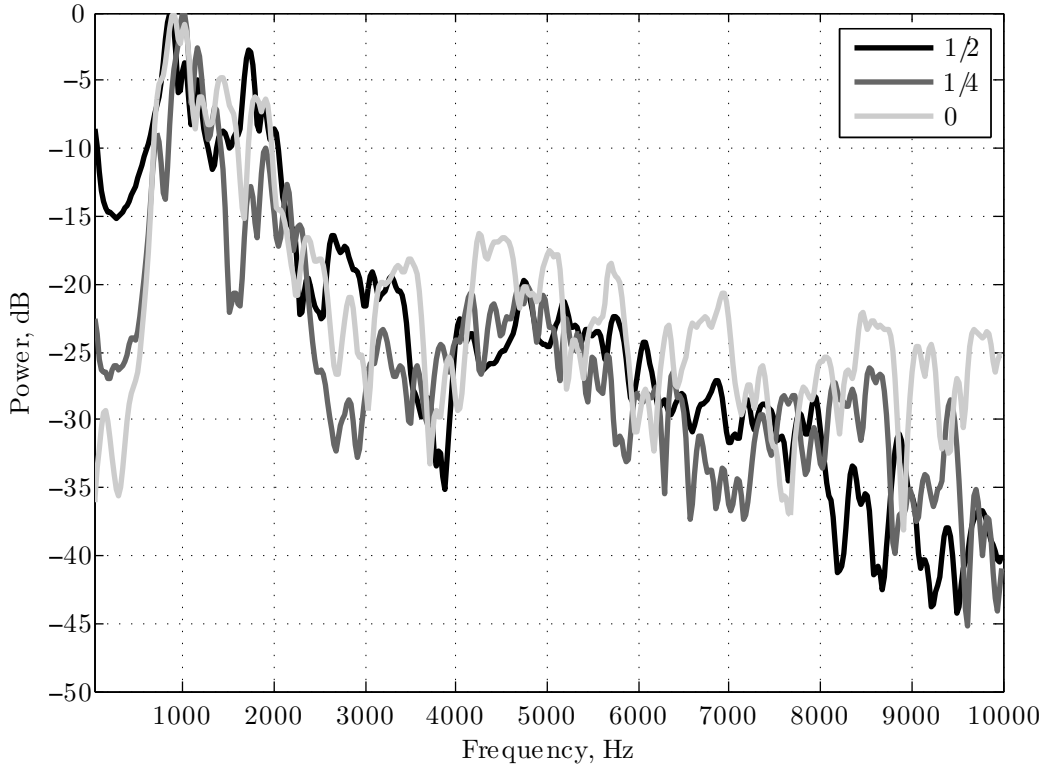


Figure 2.8: Frequency Responses of One of the Blades When Plucked at Different Locations With a Pick Where 0 Is the Bottom of the Blade (Towards the Bridge) and 1/2 the Middle.

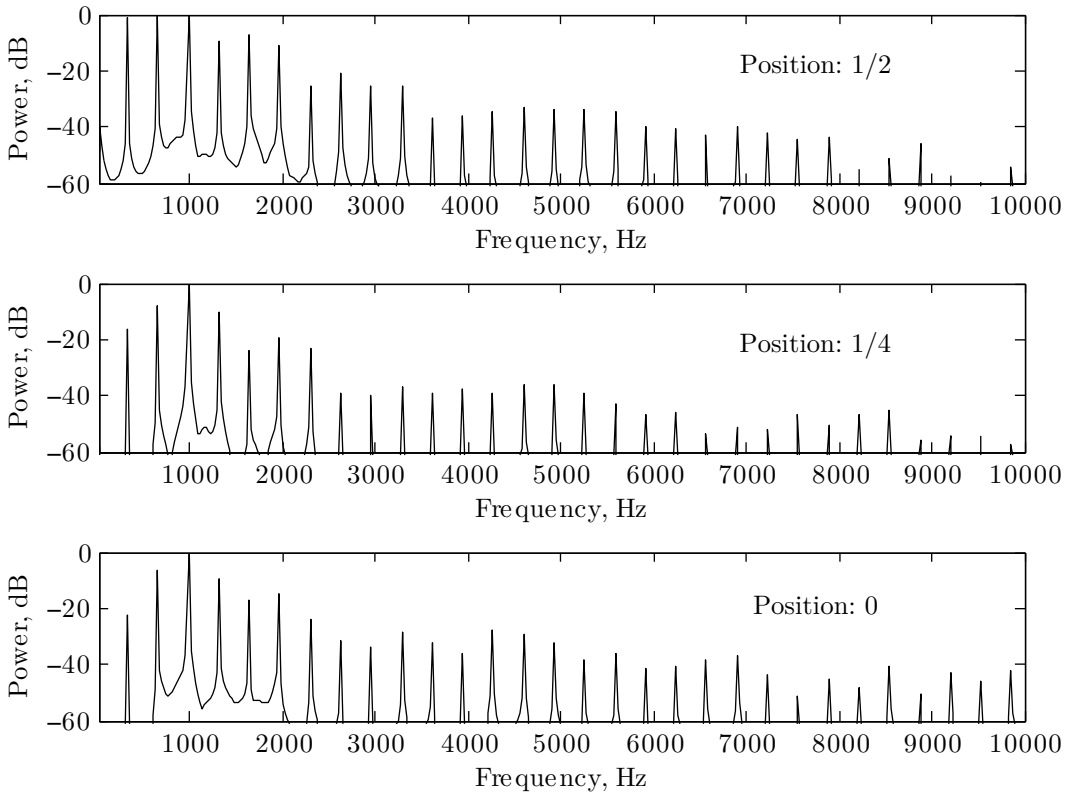


Figure 2.9: Frequency Responses of a Virtual String When Excited by the Signals From Figure 2.8.

§2.2.2).

2.2.2 Physical Model

The guitar physical model of the BladeAxe1 is based on *Free Axe* [176]. It has six parallel virtual strings connected to a series of effects such as an amplifier and a speaker model, a distortion, a chorus, a flanger, etc. The model is implemented in Faust and is running on a laptop as a standalone ALSA[10] application.

The Arduino Due retrieving the data of the different BladeAxe1 controllers placed on the neck and on the plucking system (capacitive touch sensors) runs a custom firmware that sends serial OSC[11] messages to the physical model software. Serial USB is extremely fast so the different BladeAxe1 controllers have almost no latency. Also, the use of OSC with serial makes the system very portable and "plug and play".

The digitized signals of the piezoelectric preamps are first lowpass filtered at 10KHz to make sure that the preamp analog lowpass filters removed the peak around 16KHz. Next, a noise gate lets the signal drive the virtual string only if it reaches a certain level. Indeed, our preamplifiers tend to generate a little bit of noise (less than -55dB). This system prevents the noise from making the virtual string vibrate if it's not plucked.

After this step, the signal goes through a pick position filter (a simple feed-forward comb filter) [82]. The coefficients of this filter are computed as a function of the pluck position detected by the capacitive touch sensors on the polycarbonate sheets.

Finally, the signal is fed into the virtual string whose different parameters are controlled by the neck. An overview of this process can be seen in Figure 2.10.

2.2.3 Neck

Designing a neck for the BladeAxe1 offering a level of control similar to that of a real guitar fretboard proved more complicated than we expected, and we constructed several prototypes before finding a satisfactory solution.

Guitar Neck Taxonomy

On a real guitar, the neck enables changing the length of the strings in order to modify the frequencies of their fundamental modes of vibration—in other words, their pitch. For that, the guitarist can press the strings against different frets, shifting the position of "the nut termination" of the string.

A standard guitar neck contains twenty-four frets placed one semitone apart from each other covering two octaves on each string. Frets are not equidistant, but rather get closer as we go up

[10] Advanced Linux Sound Architecture
[11] Open Sound Control

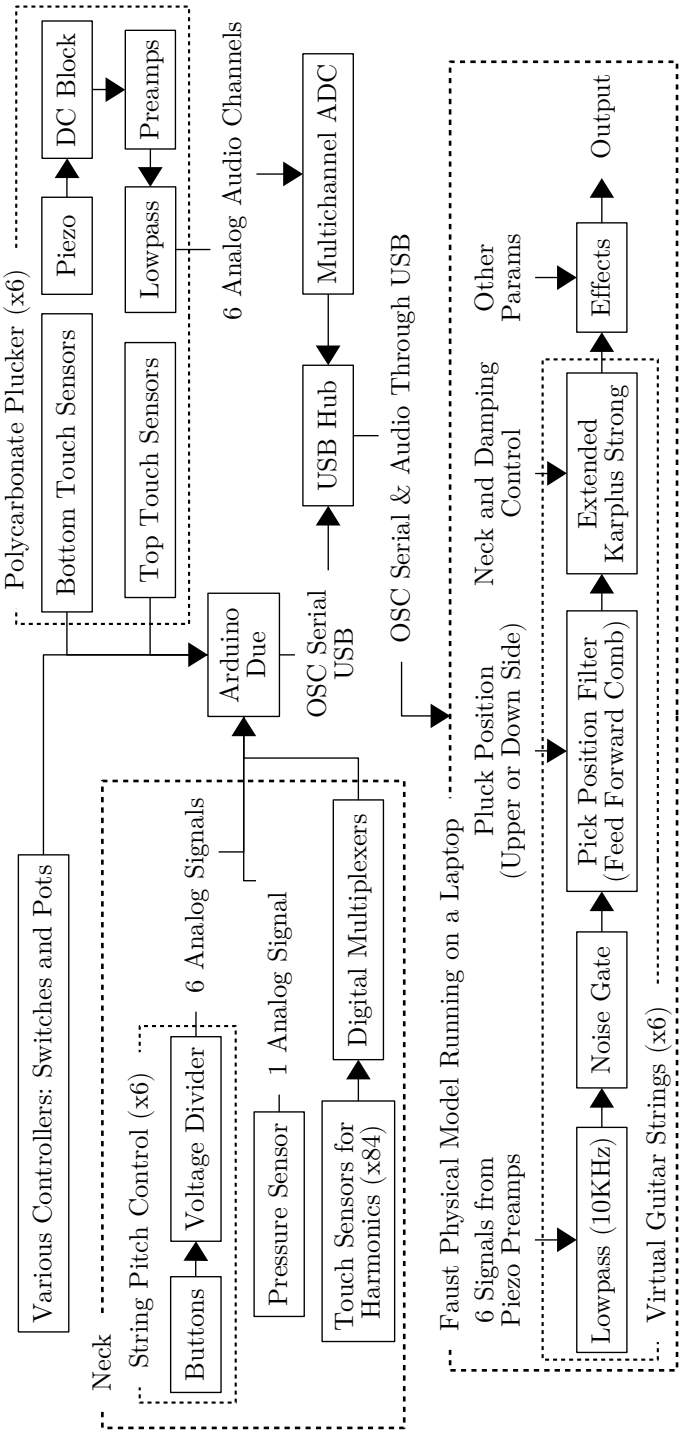


Figure 2.10: Overview of the Different Components of the BladeAxe1.

the neck. This feature has two important consequences on the technique used by a guitarist. First, one has to remember intuitively that frets are not equally spaced. But more importantly, it is much easier to play chords near the bottom of the neck where the fret-spacing is larger.

These are the main parameters that we tried to take into account to design the BladeAxe1 neck. More information on the physics and design of guitar necks can be found in [59].

Guitar Neck Features

Let's try to establish a non-exhaustive list of the general features offered by a standard guitar neck:

- In the absence of bending, the pitch of each string is changed by semitones.
- The pitch of a string can be bent up by two semitones or more by sliding the string along a fret with the finger stopping the string. This technique can be used also to add vibrato.
- The amount of pressure applied to a string by a finger enables the player to switch between two modes of vibration: if the string is just touched but not pressed against a fret, the fundamental will be canceled and only the harmonics will be heard. If the string is pressed against the fret, the termination of the string becomes stiffer and the fundamental can be heard.

While designing our different neck prototypes, we tried to find the best solutions to implement these features. Dozens of technologies usable to create a controller that would meet our needs and expectations to design a good guitar virtual neck are available on the market. Their quality and price can vary a lot. Our goal was to build a precise controller that would provide sensations close to those offered by a traditional guitar neck and keep it cost effective.

Ideally, a guitar neck based on a technology similar to the one of the LinnStrument (see §1.1.2) would have been excellent. Unfortunately, this system greatly exceeds our budget, due in part to the fact that a customized version of it would be required.

Soft Pot Based Prototype

Our first prototype (see Figure 2.11) was inspired by the technique used in the GXtar [91], where a soft pot is used to detect the position of the finger on the neck. The main advantage of using soft pots is that they provide a continuous control. Unfortunately, they are not precise enough to create a good vibrato. They also present many other handicaps:

- Large soft pots (500mm) can sometime have unpredictable behaviors which makes them almost impossible to use in a concert context.
- Commercial large soft pots are too wide to be placed in parallel to make a six-string virtual guitar neck and "handmade" soft pots are hard to make.
- Commercial large soft pots are relatively expensive sensors (about $30 per unit).

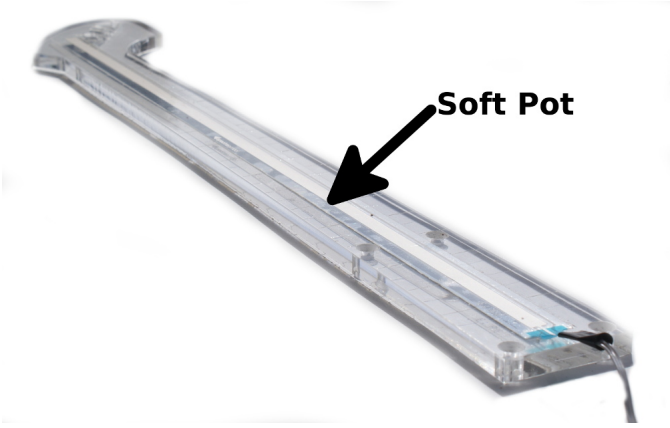


Figure 2.11: First Neck Prototype for the BladeAxe1 Based on a Soft Pot.

Simple Buttons Based Prototypes

Most commercial guitar MIDI controllers have button based necks (see §1.1.4). Buttons have many advantages: they are very cheap, reliable and they provide haptic feedback to the performer when they are engaged which is a very important feature. Indeed, we found out that it is very hard to know if the pitch of the string is changed properly without getting a physical feedback which discouraged us from building a neck based on capacitive touch sensors.

We considered using pressure sensitive buttons (that are much more expensive than discrete buttons) but we realized that these wouldn't improve the control a lot. Yet, we wanted the performer to be able to bend the pitch of the strings and add vibrato. Therefore, we added a pressure sensor placed between two silicon sheets at the bottom of the neck (see Figure 2.15) to detect the amount of force applied by the player to the buttons when pressing them.

The pressure sensor can also be used to control other parameters like an audio effect or to add artificial vibrato to the strings (in this case, we control the amplitude of the vibrato by pushing harder on the neck).

Our button system is based on six parallel rows of fourteen push buttons. We decided that fourteen semitones per strings (7/6 octave) were enough to play a big part of the electric guitar repertoire. Unlike on a real neck, buttons are equally spaced. Indeed this enabled us to greatly simplify the design (we didn't have to use a custom PCB) and we thought that guitar players should be able to adapt to a "linear" neck. Moreover, most commercial MIDI guitars have equally spaced buttons.

On the electronic side, each row of buttons representing a string acts as a voltage divider. Each button outputs a different voltage that is measured on one of the analog inputs of the Arduino (see Figure 2.10). Priority is given to the lowest buttons in order to enable the player to do "hammer-ons"

or to play bar-chords.

We spent a lot of time trying different kinds of buttons. Our first design was based on a silicon sheet that we placed over the electronic push buttons (see Figure 2.12 for the silicon buttons and Figure 2.14 for the electronic buttons board). The texture of silicon was very nice because it felt a little bit like skin. However, this solution presented many drawbacks. For example pushing very hard on a button could sometimes trigger neighboring buttons because of mechanical coupling. Another big issue with this design was that it was very hard to do slides. Finally, our silicon buttons were a little bit too sensitive and it was hard in the case of some chords to know if a finger was touching the string or not.

Figure 2.12: BladeAxe1 Neck Based on Silicon Buttons.

Despite the fact that we really liked the idea of having silicon buttons, we decided to adopt a more robust and durable solution based on laser-cut acrylic buttons. These worked in a way similar to the silicon one by placing them above the electronic push-buttons board in order to reuse it (see Figure 2.13). While these acrylic buttons worked much better it was still relatively hard to slide between them because of their square shape. We also found that it was difficult to distinguish them because they all had the same color. This is what led us to the final version of the BladeAxe neck.

Final Version

The final version of the BladeAxe1 neck is very similar to the one described in the last paragraph of the previous section. Buttons have a round shape in order to allow players to slide their fingers along "virtual strings". Each fret has a different color to easily distinguish it. Of course, the pressure sensor is still present at the bottom of the neck and can be used to add more expressivity to the play.

Very thin capacitive touch sensors were added at the top of each button in order to detect if the

Figure 2.13: First Version of the BladeAxe1 Laser Cut Acrylic Buttons.

performer is touching the button without pressing it. In this case, the physical model switches to a "harmonic mode", just like on a "real" guitar when a string is touched but not pushed against a fret.

This final version of the neck uses seven analog inputs on the Arduino Due: six for the strings and one for the pressure sensor.

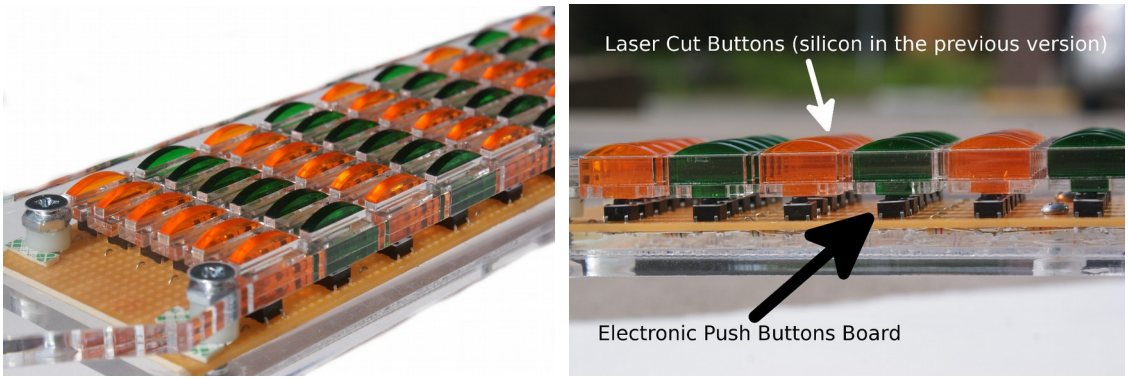


Figure 2.14: Laser Cut Acrylic Buttons as They Appear in the Final Version of the BladeAxe1 Neck.

2.3 The BladeAxe2: Augmenting the iPad

While the BladeAxe1 presented in §2.2 was perfectly usable, we were not fully satisfied with it. First, we didn't like the fact that it had to be connected to a laptop through a six audio channels interface in order to work, impacting its overall coherence (see §1.1) and "standaloneness."

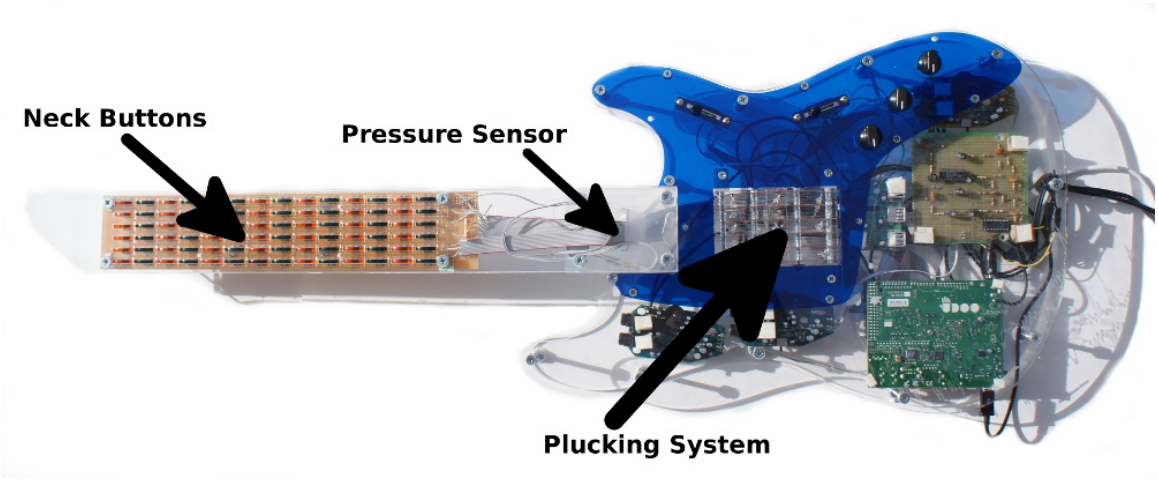


Figure 2.15: Top View of the BladeAxe1.

Additionally, even though our latest neck design (§2.2.3) was quite playable, it didn't compete with a traditional guitar neck.

In this section, we present the BladeAxe2 where these problems were partially solved by using an iPad both as a controller and to implement the virtual part of the instrument. After presenting an intermediate version of the BladeAxe, we'll describe our final design. We will also introduce a similar instrument: the PlateAxe.

This section is based on a paper[12] that we published at NIME-16[13] on *Augmenting the iPad: the* BladeAxe [127].

2.3.1 Towards the BladeAxe2

On our way to the BladeAxe2, we made an intermediate version using the iPad and borrowing a lot of elements to the BladeAxe1. This instrument is presented in this subsection.

As we wanted to keep the same plucking system as the BladeAxe1, we used the iPad as the neck of the guitar (see Figure 2.16). This implied major design modifications and also a totally different approach to the way this instrument was controlled. Indeed, while the various necks presented in §2.2 prioritized guitar playing skill transfer, this new version of the BladeAxe cannot be controlled in the same way as a traditional guitar because of the shape of the touchscreen. We also mounted an acrylic plate with laser engraved textures to the front face of the BladeAxe, with its own attached piezo disc, allowing the performer to drive the virtual strings through a variety of interactions. Each texture creates a different sound excitation and implies a specific gesture: rubbing, scratching,

[12] Some sections and figures of this paper were copied verbatim here.
[13] http://nime2016.org/

tapping, etc. (see Figure 2.19).

We wanted this instrument to be fully standalone and self-powered. Running the guitar physical model on the iPad forced us to do a series of optimizations detailed in the next section.

The most technically challenging problem that we had to solve with this instrument was to find a way to send the seven independent audio channels of the plucking system (the six blades plus the textured plate) to the iPad. We could have used a custom ADC just like we did for the BladeAxe1, but this would have required the use of an external power supply such as a battery, greatly complicating the design of the instrument. Thus, in order to use the built-in stereo ADC of the iPad, we had to find a way to differentiate the six blades; we used the capacitive touch sensors of the plucking system (see §2.2.1) to detect which blade was plucked to route the excitation to the corresponding virtual string (see Figure 2.17). While this system worked fine if the performer was playing slow, it was hard to control in the frame of a piece involving fast playing.

Another challenge was to find a way to connect the Arduino retrieving the touch sensing data from the plucking system to the iPad. Indeed, iOS devices let other devices connect to them only if they are approved by Apple. There are a couple of exceptions to that including MIDI controllers. We used Dimitri Diakopoulos' work [51] as a basis to replace the serial USB driver of the *Arduino* with a MIDI USB driver making it usable with the iPad. Indeed, while a series of microcontrollers such as the Teensy[14] (see §5.2) now provide built-in MIDI support, they didn't exist when this version of the BladeAxe was built. A proximity sensor was also embedded in the body of the BladeAxe (see Figure 2.16) to control some of the effects applied to the model such as a wah pedal.

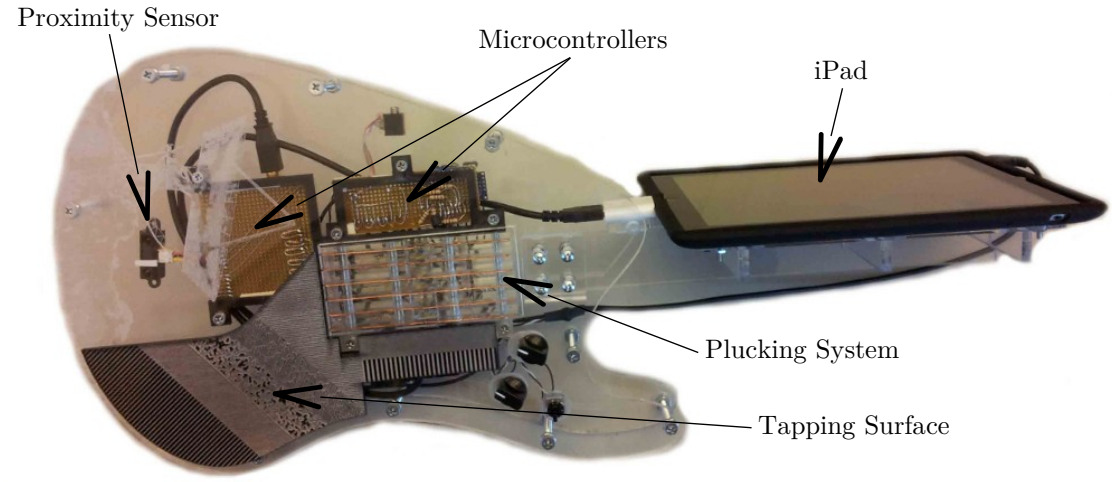


Figure 2.16: Intermediate Version of the BladeAxe2 Using an iPad.

[14] https://www.pjrc.com/teensy/

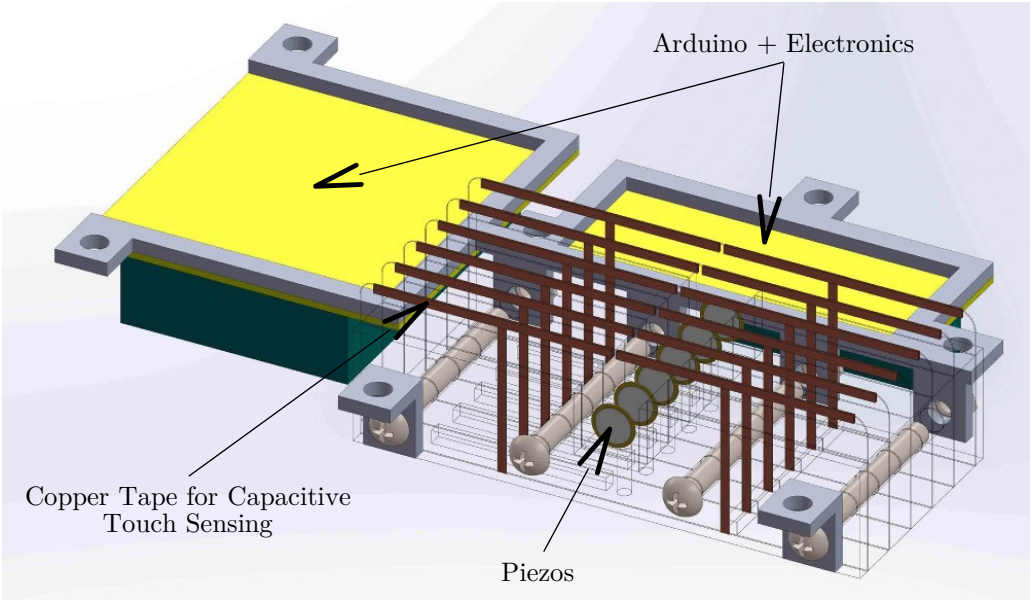


Figure 2.17: Plucking System of the Intermediate Version of the BladeAxe2 Presented in Figure 2.16.

2.3.2 Final Version

The final version of the BladeAxe2 can be seen as a simplified version of the instrument presented in the previous section. The way the performer interacts with it was totally rethought and the number of blades was reduced from six to two (see Figure 2.18) allowing us to use a simple USB stereo ADC/DAC. We didn't use the built-in ADC/DAC of the iPad since it only has a mono input. The redesigned tapping/scratching surface hides the ADC/DAC. A switch mounted on the front face of the instrument makes it possible to route the signal from the two blades of the plucking system or from the tapping surface to any of the two input channels of the ADC. The USB ADC/DAC is directly connected to the iPad through the lightning connector since there is no microcontroller in this version of the BladeAxe2. The chestnut wood body of the instrument was CNC machined and laser engraved (see Figure 2.18). The built-in iPad sleeve is mounted on a rotating axis in order for the performer to be able to adjust its inclination. It is fully powered by an iPad Air 2 and can be used for more than eight hours without charging the battery. The DAC's single stereo 1/4" jack output with adjusted impedance makes it compatible with any guitar amp, pedal, etc.

Several pieces specifically composed for this instrument can be listened to on the BladeAxe webpage[15].

[15] https://ccrma.stanford.edu/~rmichon/bladeaxe/

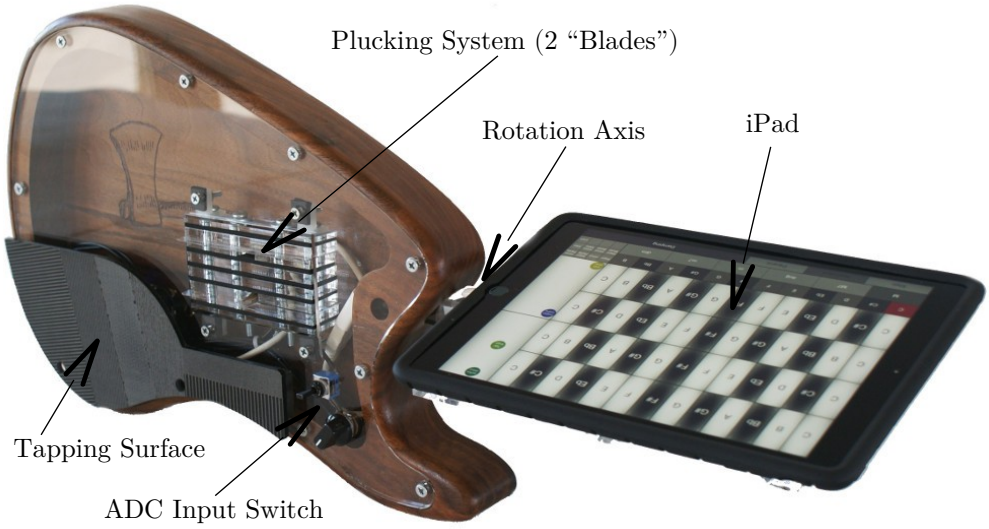


Figure 2.18: Final Version of the BladeAxe2.

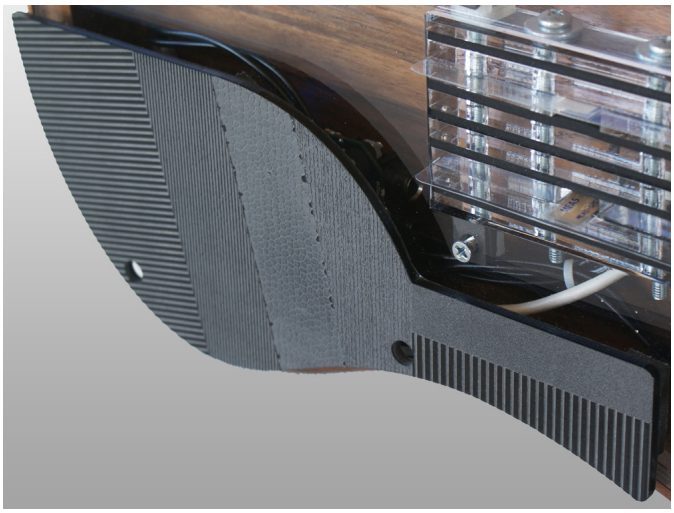


Figure 2.19: Textured Plate of the BladeAxe2.

2.3.3 Control

While the overall shape of the final BladeAxe2 is quite similar to its previous version presented in §2.3.1, the way it is controlled/performed is very different, mainly because of two versus six blades. Indeed, since we were using an iPad as the neck to control parameters such as the pitch of the strings, we thought that the paradigm where strings can be driven independently from the plucking system became obsolete. Instead, the performer is now able to drive any string with a single blade. Pitch is controlled using a chromatic keyboard interface on the screen of the iPad (see Figure 2.20). If one key is pressed, only one virtual string is used. If several keys are touched, several virtual strings are allocated and the excitation from the single blade is routed to these strings.

The BladeAxe2 guitar physical model gives access to a pool of ten virtual strings. When the performer presses a key on the screen of the iPad, an available string is allocated and its input is activated. If a glissando is performed or if the performer plays an interval smaller than a major third, the same string is used, otherwise, a new string is selected. If several keys are pressed at the same time, then the excitation from the blade is sent to all the activated strings.

The chromatic keyboard of the BladeAxe2 covers four octaves and is arranged as "an S" (see Figure 2.20) to facilitate slides across several octaves. Thus the lowest key is the one on the top right corner of the interface and the highest one is in the bottom right corner. While this keyboard allows vibrato and sliding between keys, it also "rounds" the pitch of a note when the finger of the performer is not moving (see §3.3.4). This is very important to make sure that the instrument is not out of tune. The different keys of the keyboard can be highlighted and locked in a specific scale by using the scale selector at the bottom right area of the screen.

The second blade can be used to strum a set of six virtual strings. When an excitation is created, it is progressively sent to the strings with different short delays corresponding to the amount time it would take for the exciter (finger, pick, etc.) to go from one string to another. The duration of this "inter-string-delay" is calculated according to the excitation's Random Mean Square (RMS) amplitude: the louder the excitation, the faster the strum. Chords can be selected on a table located at the bottom of the interface (see Figure 2.20) by choosing their root and their type.

Some parameters related to the expressivity of the physical model such as the decay time (T60), brightness, detuning, and material of the strings can be controlled using the "movable dots" located on the right side of the screen. A button located on the bottom of the screen can be used to damp the strings. Finally, each parameter of the physical model can be accessed independently in a parameters menu that can also be used to save presets. More details about the different parameters of the physical models of the BladeAxe2 are given in the next section.

Many of the features of the BladeAxe2 iPad interface inspired the SmartKeyboard system presented in §3.3.

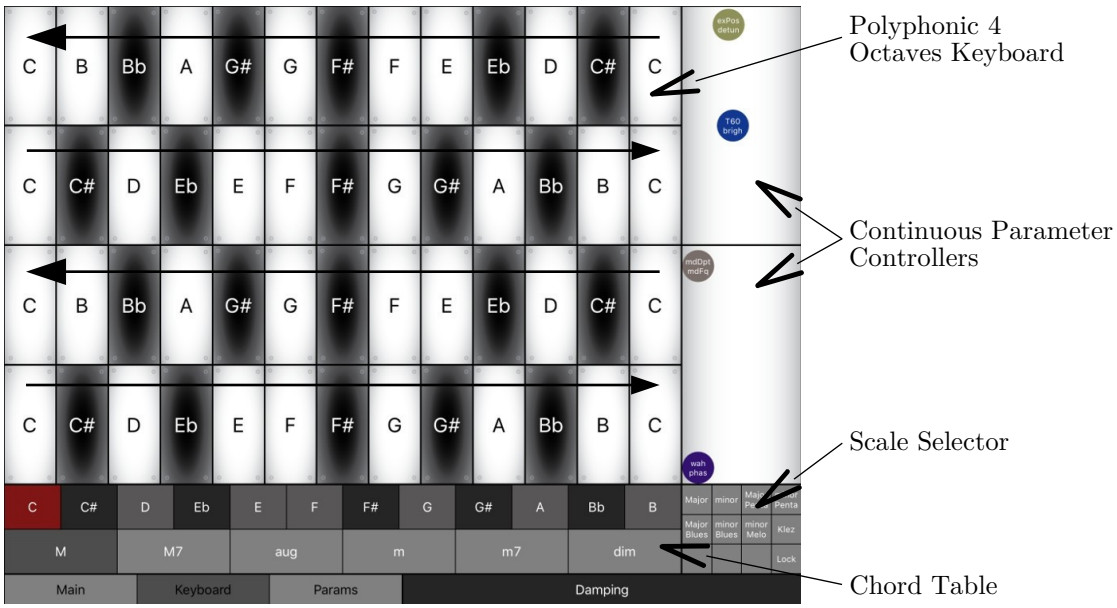


Figure 2.20: User Interface of the iPad App of the BladeAxe2.

2.3.4 Physical Model

The electric guitar physical model of the BladeAxe2 is similar to the one of the BladeAxe1 described in §2.2.2 and is fully implemented in Faust. It contains a set of ten virtual strings that can be controlled from the iOS layer. They are connected to a series of effects that are directly taken from the Faust libraries (guitar amp simulator, flanger, phaser, chorus, echo, reverb, distortion, wah pedal, etc.). The virtual strings can be both excited with a signal coming from the plucking system of the BladeAxe2 and with a synthesized excitation. The signal coming from the plucking system is lowpass filtered in function of the pitch of the string (lower notes require a lower cutoff frequency than higher notes). Each string waveguide implements a set of two coupled strings using two parallel delay lines [176]. Their length can be offset to create harmonics. The amount of offset can be controlled dynamically from the GUI by using the *dots interface* (see §2.3.3). The length of the delay lines can also be modulated by a sine oscillator to create very expressive effects. The frequency of the modulation and its amplitude can be controlled from the *dots interface* as well.

The pitch of the virtual strings can be changed using a "slide mode" or a "discontinuous mode". In slide mode, the frequency values are interpolated to create a smooth continuous change. The discontinuous mode can be used to abruptly change the pitch of a string without creating a click. To do that, two parallel delay lines are used. When the pitch of a string is changed, the length of the second delay line is altered and a fast crossfade is carried out to switch from the first delay line to the second one creating a very natural transition.

The model and its associated effects were compiled as a C++ code using the vectorization option of the FAUST compiler. This is very important as this allows us to run all these elements in real-time by utilizing the iPad Air 2's multi-core processor.

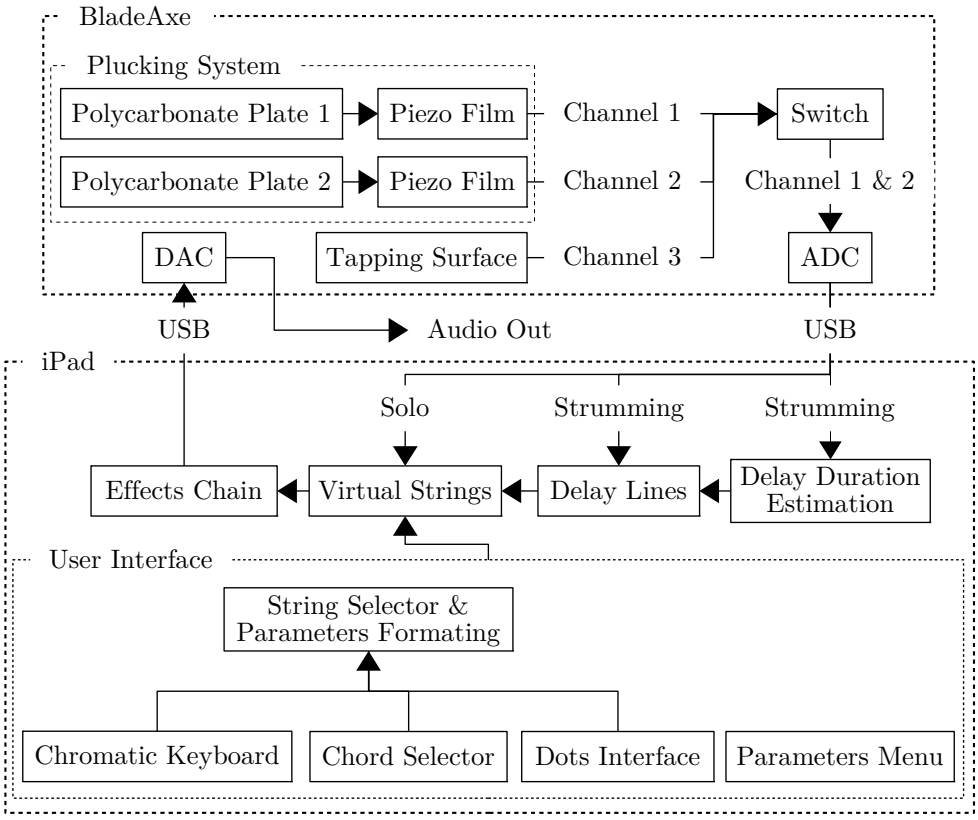


Figure 2.21: Overview of the Implementation of the BLADEAXE2.

2.3.5 The PlateAxe

For our most recent instrument, we wanted to create an interface compatible with the iPad app of the BLADEAXE2 but having a different form factor. Thus the PLATEAXE is intended to be a percussion instrument providing the same kind of laser engraved tapping surfaces as the BLADEAXE2 and a small plucking interface where a circular polycarbonate disc with variable diameter can be used to generate the sound (see Figure 2.22). A series of switches and knobs can be used to route the signals of these elements to the different input channels of the BLADEAXE2 app. Thanks to its non-uniform diameter, the polycarbonate disc creates excitations with different spectra depending on where it is plucked.

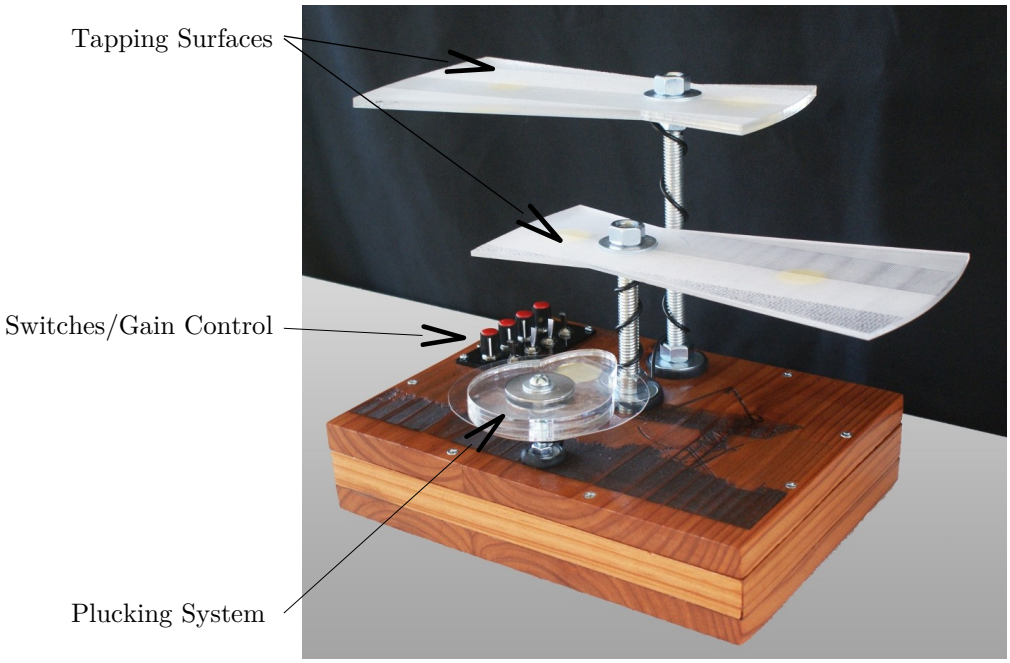


Figure 2.22: The PlateAxe.

2.3.6 Discussion

Our first approach (see §2.2) consisted of creating an interface as close as possible from the one provided by the original instrument. While we think that we partially achieved this goal with "the right hand" and the plucking system, we struggled more with the neck. Finding a design offering the same kind of interactions and sensations as an actual guitar neck prove to be hard. Also, by getting too close to the original instrument, we realized that we lost part of its novelty and that it would never be as good as its purely acoustical counterpart. With the iPad version of the BladeAxe, we think that we found a good compromise between expressivity, skill transfer and novelty.

The ability to control several virtual strings with a single blade made this instrument very intuitive to play. We believe that we could even remove the strumming blade and transfer its functionality to the touchscreen interface, allowing us to strum and play independent notes with the same blade.

While the textured plate (see Figure 2.19) significantly increases the expressive potential of the BladeAxe, we think that it could be improved by expanding the diversity of sounds it can produce, perhaps via 3D printing. Indeed, although the current textures all sound different, their spectral content are nonetheless very similar. Varying the shape and the thickness of the plate could help solve this issue.

The iPad provides a level of stability and robustness competing with professional keyboard synthesizers. After dozens of performances, the BladeAxe never crashed or ran out of power. Even though the combined latency of the touchscreen and the DAC is about 35 ms (this value is relatively constant), we find it not to be an issue especially after some practice.

Finally, one of the main limitations of using the iPad is sweat! Indeed, it often happens during a frenzied performance that the fingers of the performer get clammy or even worse, that a drop of sweat ends up on the touchscreen preventing it from working properly.

Chapter 3

MobileFaust: Facilitating Musical Apps Design and Skill Transfer

> "Digital lutherie is in many respects very similar to music creation. It involves a great deal of different know-how and many technical and technological issues. At the same time, like in music, there are no inviolable laws. That is to say that digital lutherie should not be considered as a science nor an engineering technology, but as a sort of craftsmanship that sometimes may produce a work of art, no less than music." (Sergi Jordà [84])

With that in mind, making musical apps for mobile devices involves the use and mastery of various technologies, standards, programming languages, and techniques ranging from low level C++ programming for real-time DSP to advanced interface design. This adds up to the variety of the platforms (e.g., iOS, Android, etc.) and of their associated tools (e.g., Xcode, Android Studio, etc.), standards, and languages (e.g., JAVA, C++, Objective-C, etc.).

While there exists a few tools to facilitate the design of musical apps such as libpd, [28] Mobile CSOUND, [97] and more recently JUCE[1] and SuperPowered,[2] none of them provide a comprehensive cross-platform environment for musical touchscreen interface design, high level DSP programming, turnkey instrument physical model prototyping, built-in sensors handling and mapping, MIDI and OSC compatibility, etc.

In this chapter, we introduce a series of tools around the Faust programming language facilitating the creation of musical apps. The use of musical instrument physical models in this context and in that of "hybrid mobile instruments" (see §6) is emphasized. Similarly, allowing for the design of interfaces implementing skill transfer from existing musical instruments is one of our main focus

[1] https://www.juce.com/
[2] http://superpowered.com/

here.

Some context around FAUST and mobile development is given in the first section by presenting two early command line tools to convert FAUST code into fully working Android and iOS applications: `faust2android` and `faust2ios`. The user interface of apps generated using this system corresponds to the standard UI specifications provided in the FAUST code and is made out of sliders, buttons, groups, etc. Next, `faust2api`, a tool to generate audio engines with FAUST featuring polyphony, built-in sensors mapping, MIDI and OSC support, etc. for a wide range of platforms including Android and iOS is presented. Finally, `faust2smartkeyb`, an environment based on `faust2api` allowing for the design of advanced musical apps focused on skills transfer for Android and iOS is introduced. Various examples of apps leveraging different sets of existing skills are presented. All of them are based on physical models from the FAUST Physical Modeling Library (see §6.2) and compatible with some of the augmentations described in §4 and §5.

3.1 Early Tools: `faust2android` and `faust2ios`

`faust2api` (see §3.2) and `faust2smartkeyb` (see §3.3) constitute the core of our framework to facilitate the development of musical mobile apps for live performance. Both of them were inspired by earlier systems used to make mobile apps with FAUST: `faust2ios` and `faust2android`. This section gives of an overview of their features and implementation and demonstrates how they led to our current system.

3.1.1 First Faust App Generator: `faust2ios`

Pushed by the interest around mobile music in the early 2010s (see §1.3), developers at GRAME worked on a tool to convert FAUST programs into ready-to-use iOS applications: `faust2ios`. As for any other FAUST "architecture," the user interface of such apps is based on the UI description provided in the FAUST code, and is thus typically made out of sliders, knobs, buttons, groups, etc.

Figure 3.1 presents the screenshot of sfCapture,[3] an app made with `faust2ios` as part of the *SmartFaust* project[4] (see §1.3.2).

3.1.2 Android and Real Time Signal Processing in the Early 2010s

At the time when `faust2ios` was implemented, Android had been left behind, mostly because of audio latency issues and the complexity of the architecture of apps involving real-time DSP on this platform (see §3.1.3). Google started addressing these problems in 2013 when Jelly Bean 4.2[5] was released. This convinced us to implement an equivalent of `faust2ios` on Android: `faust2android`

[3] `https://itunes.apple.com/us/app/sfcapture/id799532659?mt=8`
[4] `http://www.grame.fr/prod/smartfaust/`
[5] `http://developer.android.com/about/versions/jelly-bean.html`

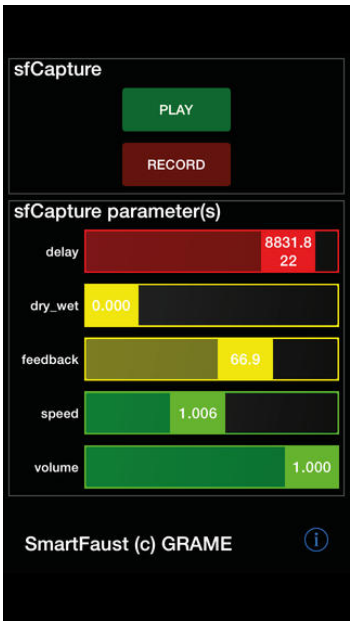


Figure 3.1: Screen-shot of sfCapture, an App Made with `faust2ios`.

[118]. [6]

This tool is still functional and part of the FAUST distribution. While its current version was partly rewritten to be based on `faust2api` (see §3.1.7), the following sections present its old implementation that was made "from scratch" in order to illustrate the evolution of this type of early system towards `faust2smartkeyb` (see §3.3).

Audio Latency

Until recently, Android had always been infamous in the audio developer community for its very high latency for audio playback and recording (larger than 300ms). However, with the release of Jelly Bean 4.2, Google made a step toward latency reduction. Victor Lazzarini reported in a post from December 2012[7] that he was able to achieve a "round-trip latency" of 100ms and a "touch-to-sound latency" of 120ms. We obtained similar results at the time with `faust2android` on a Nexus 7 (105ms for the "round-trip latency" and 130ms for the "touch-to-sound latency"). In both cases, these performances greatly surpassed the one of the previous versions of Android. These figures are completely obsolete nowadays as this is shown in §3.2.3.

[6] Some sections and figures of this paper were copied verbatim here.
[7] `http://audioprograming.wordpress.com/category/android/`

C or JAVA?

Android applications are mainly programmed in JAVA and the Android SDK provides an API for real-time audio recording and playback. Thus, signal processing classes can be directly implemented in JAVA which greatly simplifies the overall architecture of the app. Moreover, FAUST2 can generate JAVA code instead of C++.

Several tests where various FAUST generated JAVA snippets code were "manually" embedded in an Android app were carried out at the time on both a Samsung Galaxy S2 and a Google Nexus 7. While results varied greatly between the two devices (for example, we were not able to record and playback audio simultaneously in real-time on the Galaxy S2), they were very deceiving because of the instability of the process and the audio latency that was greater than 200ms.

In his post from March 2012, Victor Lazzarini describes a technique to do Android audio streaming with OpenSL ES[8] and Android's Native Development Toolkit (NDK).[9] After several tests, this technique proved to be far more stable than the "full JAVA" one and was used to build `faust2android`.

3.1.3 Real-time Audio With `faust2android`

As mentioned in the previous section, the Android NDK makes possible the use of functions written in C or C++ in a JAVA app by wrapping them as a shared library using SWIG[10] that creates the elements to interface these two programming languages.

In an app generated by `faust2android`, the different tasks are shared between C++ and JAVA as presented in Table 3.1.

JAVA	**C++**
- Android application - dynamic user interface - send the values of the different DSP parameters at every audio frame	- DSP (process one audio frame) - information about the DSP parameters and the user interface - audio resources management

Table 3.1: Building Blocks of a `faust2android` App.

Accessing and managing audio resources was carried out using Victor Lazzarini's simple but very useful API that makes available OpenSL ES on Android for real-time audio recording and playback. As a result, Android apps generated by `faust2android` operate audio streaming and signal processing natively which is far more efficient than if these tasks were done directly in JAVA.

[8] `http://www.khronos.org/opensles/`
[9] `http://http://developer.android.com/tools/sdk/ndk/`
[10] Simplified Wrapper and Interface Generator: `http://www.swig.org/`

3.1.4 Generating Code

Unlike other FAUST architectures, `faust2android` can't generate a single file containing all the elements needed by the C compiler to create an object. Indeed, as mentioned before, the generated apps are based on JAVA, C++, etc. files which doesn't make the task easier.

`faust2android` uses a simple bash script to carry out the different tasks that will turn a FAUST program into an Android application. It first calls the FAUST compiler that generates C++ code. This code is then embedded into an architecture file that interfaces it with a template Android app whose content is dynamically changed according to the user interface specifications contained in the C++ code produced by FAUST.

Finally, the Android cross compiler is called by the script to generate the binary file of the app. A simple option allows the user to load the app on the default Android device connected to the computer that execute the script. Another option creates an Android Studio[11] project in the current directory if the user wishes to "manually" modify the content of the app.

An overview of the first version of the implementation of `faust2android` is given in Figure 3.2.

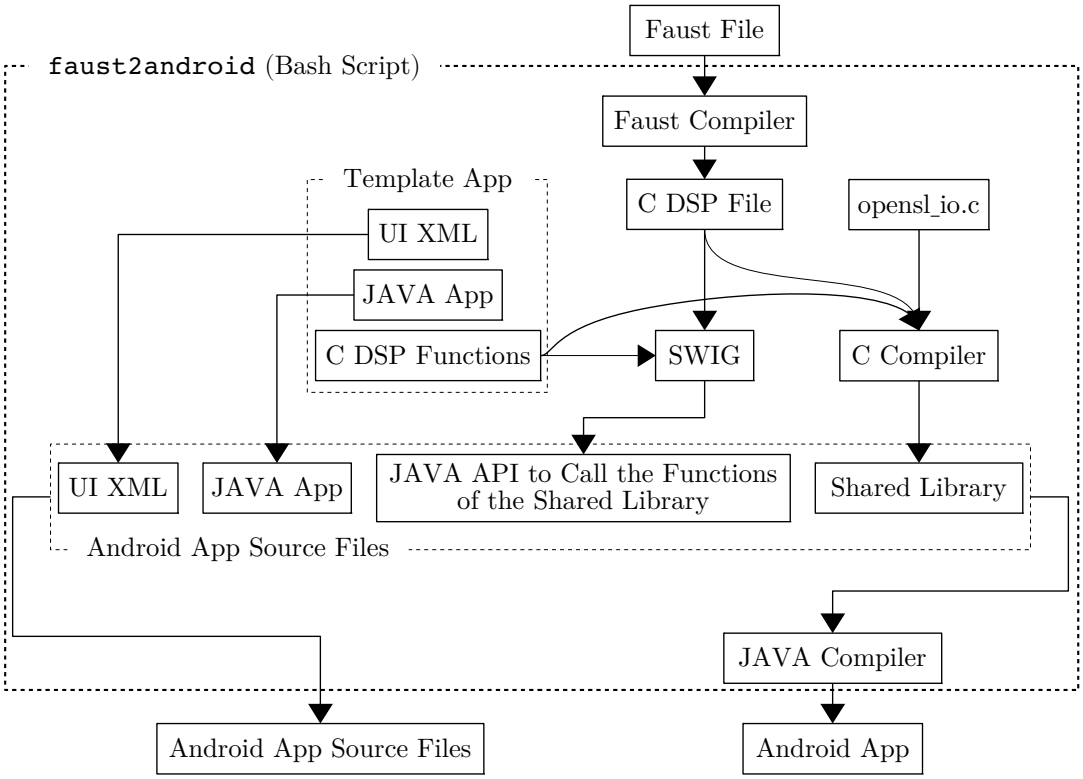


Figure 3.2: `faust2android` Overview.

[11] `https://developer.android.com/studio/`

3.1.5 Simple User Interface

Although the diversity of the standard user interface widgets provided with the Android SDK is rather limited, it is currently used to build the different parameter controllers of an app generated by `faust2android`. All the standard Faust UI elements are available: horizontal and vertical groups, horizontal and vertical sliders, numerical entries, knobs, checkboxes, buttons, drop-down menus, radio buttons, bargraphs, etc. Some examples are shown in Figure 3.3.

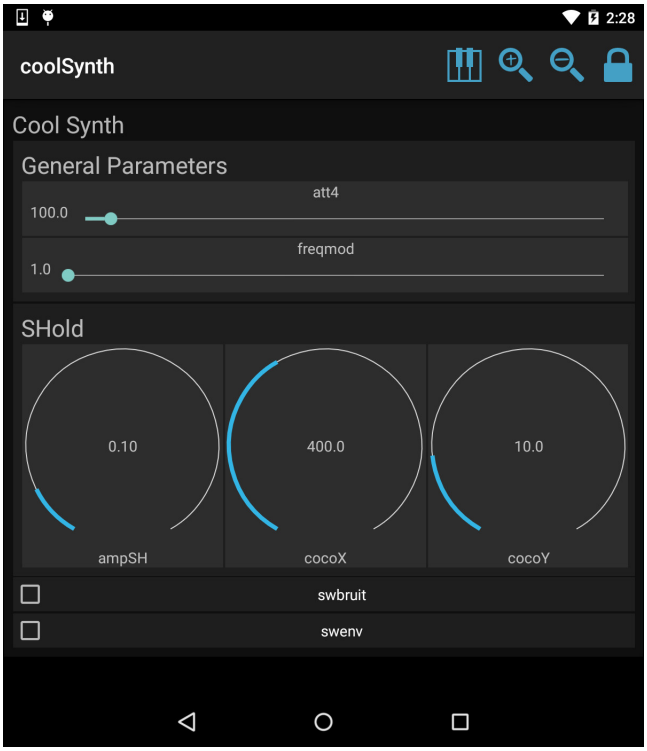


Figure 3.3: Example of Interface Generated by `faust2android` Containing Groups, Sliders, Knobs and Checkboxes.

3.1.6 Using Built-In Sensors

The accelerometer can be used to control some elements of the user interface. Assignments are made in the "Accelerometer Parameters" panel that can be opened by holding the label of a parameter for more than one second (see Figure 3.4).

From here, the mapping of an accelerometer to a parameter can be configured precisely to create complex linear and non-linear behaviors. For instance, the user can choose which axis will control the parameter (x, y, or z), its motion orientation, and sensitivity. All these parameters can be configured from the FAUST code using metadata as well (see [119] for more details on this).

Raw data from the accelerometers are passed directly to the FAUST audio process. Filtering can be carried out in FAUST which is better suited for that kind of task than JAVA.

Finally, the accelerometer parameters window is only accessible if the app is unlocked by touching the "lock" icon on the top right corner of the screen (see Figure 3.3). Apps can be locked to prevent users from opening a configuration window or rotating the screen during a performance.

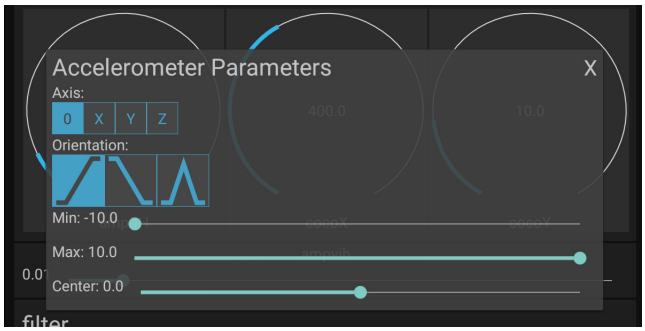


Figure 3.4: Accelerometer Configuration Panel of an Application Generated by `faust2android`.

3.1.7 Keyboard and Multitouch Interface

In 2015, a new version of `faust2android` leveraging some of the functionalities of `faust2api` (see §3.2) was released [129].[12] It introduced a series of new features that are presented in the following sections.

Standard FAUST UI elements are not particularly well adapted to live performance on a touch-screen. Thus, `faust2android` allows for the assignment of more interactive interfaces to the FAUST process. Note that this topic is further investigated in §3.4 and that more advanced solutions to this problem are available in `faust2smartkeyb` (see §3.3).

Three different metadata items can be added to the top-level group of a FAUST program to make it more controllable. The `[style:keyboard]` metadata item specifies that the `freq`, `gain`, and `gate` parameters in the FAUST code should be assigned to a piano keyboard that can be opened by touching the "keyboard icon" in the top right corner of the app. Also, these three parameters will be automatically removed from the main interface for controlling the other parameters.

The following example program illustrates a simple usage:

```
import("stdfaust.lib");
s = button("gate");
g = hslider("gain",0.1,0,1,0.001);
f = hslider("freq",100,20,10000,1);
```

[12] Multiple sections and figures of this paper were copied verbatim here.

```
process = vgroup("[style:keyboard]",s*g*os.sawtooth(f));
```

This interface uses the polyphonic capabilities of `faust2api` and allows up to eight voices of polyphony. Touching a key on the keyboard determines the reference pitch of the note but sliding the finger across the X axis of the screen allows the user to continuously control it. The Y axis determines the gain of the note. If a MIDI keyboard is plugged into the Android device, it will be able to control the keyboard interface (see §3.1.8).

The `[style:multi]` metadata item will create a simple interface in which parameters are represented by moveable dots on the screen. Each dot can have two parameters assigned to it, corresponding to X and Y screen coordinates. This interface can also be opened by touching the keyboard icon on the top right corner of the screen. Parameters are linked to the interface via `[multi:x]` metadata where `x` is the ID of the parameter in the interface. For example, the FAUST program

```
import("stdfaust.lib");
freq = hslider("freq[multi:0]",440,50,2000,0.1);
process = hgroup("[style:multi]",os.osc(freq));
```

creates an app in which the frequency parameter of a sine oscillator is controlled by the X axis of the dot in the multitouch interface. Parameters that have the accelerometer assigned to them (see §3.1.6) will continue to be driven by the accelerometer in the interface.

Finally, the `[style:multikeyboard]` metadata combines the keyboard and multitouch interface into one (see Figure 3.5).

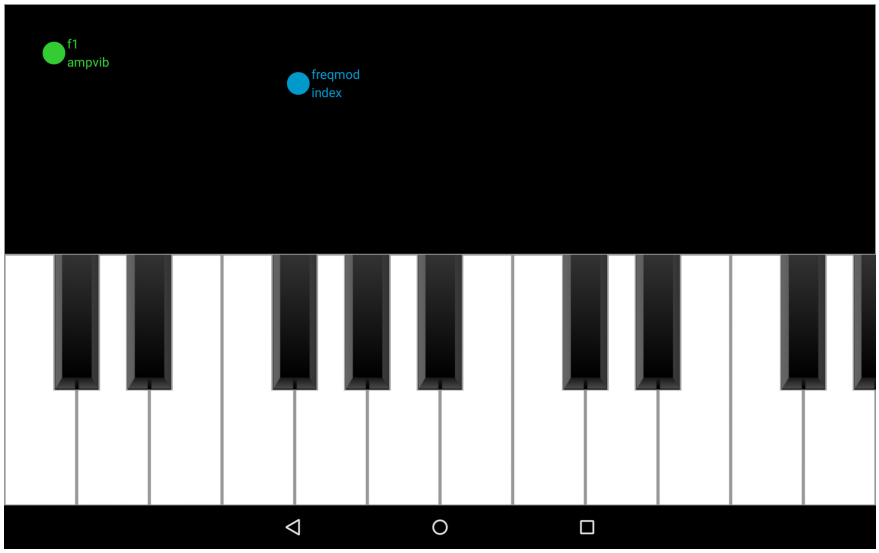


Figure 3.5: Example of a `MultiKeyboard` Interface in an `faust2android` application.

3.1.8 OSC and MIDI Support

OSC support is enabled by default for all the parameters of applications generated by faust2android. The OSC address of a parameter corresponds to the path to this parameter in the FAUST code. For example, the OSC address of the `freq` parameter of the FAUST code

```
freq = hslider("freq",440,50,2000,0.1);
process = hgroup("Main",os.osc(freq));
```

will be `/Main/freq`.

Similarly, MIDI support is also enabled by default in apps generated by `faust2android` through `faust2api` (more details about this are provided in §3.2.1).

3.1.9 Audio IO Configuration

Android applications generated with `faust2android` automatically choose the best sampling rate and buffer size as a function of the device that is running them (for Nexus[13] devices only). Indeed, it was explained during the *Google I/O 2013 conference on High Performance Audio*[14] that Android phones and tablets achieve better audio latency performance if they run with a specific buffer size and sampling rate (see Table 3.2). Users may override these default values in the settings menu of the app.

Device	Sampling Rate	Buffer Size
Nexus S	44100	880
Galaxy Nexus	44100	144
Nexus 4	44100	240
Nexus 7	44100	512
Nexus 10	44100	256
Others	44100	512

Table 3.2: Preferred Buffer Sizes and Sampling Rates for Various Android Devices.

3.1.10 Easy App Generation

While it is relatively simple to use `faust2android`, it requires the programmer to have an important number of dependencies installed (Android SDK and NDK, etc.). However, FAUSTLIVE [50] and the FAUST ONLINE COMPILER [122] make the process of turning FAUST code into an Android application very simple. Indeed, when the user chooses to compile a FAUST program as an Android app, a QR code pointing to the generated app package is displayed that can be scanned by the device where the user want the app to be installed.

[13] `http://www.google.com/nexus/`
[14] `http://youtu.be/d3kfEeMZ65c/`

faust2android has been for a wide range of musical productions and projects such as SmartFaust, GeekBagatelles, SmartLand, etc. (see §1.3.2).

3.2 Towards a Generic System: faust2api

faust2api is a tool to generate ready-to-use DSP engines using FAUST for a wide range of platforms. While available features might vary slightly from one architecture to another, they are the same on Android and iOS:

- polyphony and MIDI support,
- audio effects chains,
- built-in sensors support,
- low latency audio,
- etc.

In this section, we first give an overview of how faust2api works. Then, technical details on the implementation of this system are provided. Finally, we evaluate it and present future directions for this project.

3.2.1 Overview

Basics

At its highest level, faust2api is a command line program taking a FAUST source file 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:

```
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 §3.2.2), but the format of the API itself remains very similar (see Table 3.3. 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 the programmer 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 as 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 §3.2.2). Otherwise, the same steps can be used to work with the `DspFaust` object.

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 [67] 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);
```

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);
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 (see §3.3.3).

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.

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 us 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.

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 [119]. This reference also provides a series of tutorials on how to use `faust2api`.

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

Table 3.3: Overview of the API Functions.

3.2.2 Implementation

`faust2api` takes advantage of the modularity on the FAUST architecture system to generate its custom DSP engines [100]. 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 us to interact with the native (C++) block. The DSP C++ code is the same for all platforms (see Figure 3.6) and is wrapped into an object implementing the polyphonic synthesizer followed by the effects chain (assuming that the `-nvoices` 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.

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`, [162] 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 `propagate[Acc/Gyr]()` methods.

Android

Android applications are primarily written in JAVA. However, despite the fact that the FAUST compiler can generate JAVA code, it is not a good choice for real-time audio signal processing. 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 3.6) which is based on OpenSL ES.[15] 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).[16]

[15] `https://www.khronos.org/opensles/`
[16] `https://developer.android.com/ndk/index.html`

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.

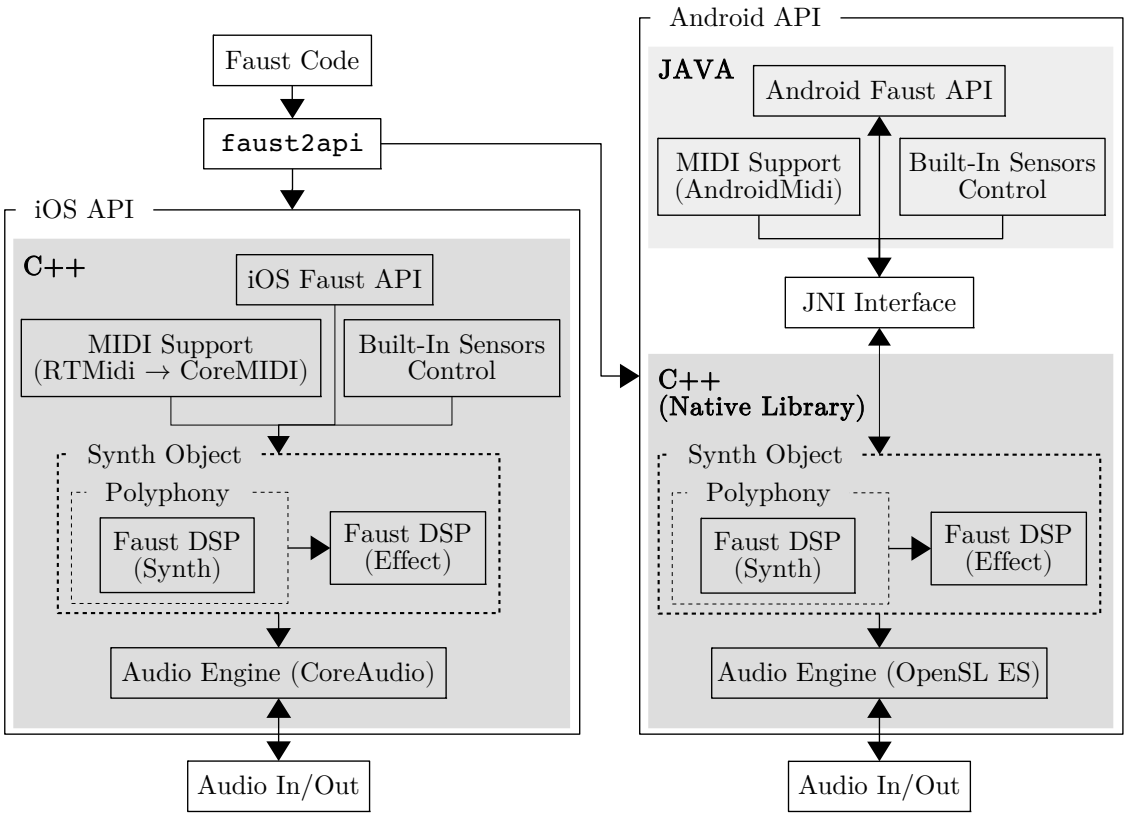


Figure 3.6: Overview of DSP Engines Generated with `faust2api`.

3.2.3 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.[17] The "touch-to-sound" latency is the time it takes to generate a sound after a touch event was registered on the touchscreen of the device. The "round-trip" latency is the time it takes to process an analog signal

[17] `https://source.android.com/devices/audio/latency_measurements.html`

recorded by the built-in microphone or acquired by the line input.

Table 3.5 shows that a "reasonable" latency can only be achieved with the latest version of `Android`, which confirms the measurements made by Google.[18] Unfortunately, such performances can only be attained on a few supported device, and configured with a specific sampling rate and buffer length.

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 3.4: 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 3.5: Audio latency for different Android devices using `faust2api`.

3.2.4 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 like the LinnStrument or the ROLI Seaboard (see §1.1.2).

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.

[18] `https://source.android.com/devices/audio/latency_measurements.html#measurements`

3.3 faust2smartkeyb

In §3.1.7, we showed that user interfaces better adapted to musical applications (e.g., piano keyboards, x/y controllers, etc.) can replace the standard UI of a FAUST object in apps generated by faust2android. However, they are far from providing a generic solution to capture musical gestures on a touchscreen and to allow for musical skill transfer. In this section, we introduce faust2smartkeyb,[19] a tool to generate Android and iOS apps using FAUST where an extended number of musical interfaces and behaviors can be designed and implemented directly from FAUST code. First, we describe the implementation of the system. Next, we demonstrate how to use it to implement a wide range of behaviors and mappings. Finally, §3.4 presents a series of examples where physical models from the FAUST Physical Modeling Library (see §6.2) are turned into standalone instruments using faust2smartkeyb and implement various types of instrumental skills. faust2smartkeyb is used as the main tool in our framework to implement the software portion of our hybrid mobile instruments (see §6).

3.3.1 Apps Generation and General Implementation

faust2smartkeyb works the same way as most FAUST targets/"architectures" [67] and can be called using the faust2smartkeyb command-line tool:

```
faust2smartkeyb [options] faustFile.dsp
```

where faustFile.dsp is a FAUST file declaring a SMARTKEYBOARD interface (see §3.3.2) and [options] is a set of options allowing us to configure general parameters of the generated app (see Table 3.6).

Option	Description
-android	Generate an Android app
-ios	Generate an iOS app
-effect	Specify a FAUST effect file
-install	Install the app on the device (Android only)
-nvoices	Specify the number of polyphony voices of the DSP engine
-reuse	Reuse an existing app project (only update what was changed)
-source	Generate the source code of the app

Table 3.6: Selected faust2smartkeyb Options.

The only required option is the app type (-android or -ios). Unless specified otherwise (e.g., using the -source option), faust2smartkeyb will compile the app directly in the terminal and upload it on any Android device connected to the computer if the -install option is provided.

[19] faust2smartkeyb is now part of the FAUST distribution. Additional information and documentation about this tool can be found on this webpage: https://ccrma.stanford.edu/~rmichon/smartKeyboard/.

If `-source` is used, an Xcode[20] or an Android Studio[21] project is generated, depending on the selected app type.

`faust2smartkeyb` is based on `faust2api` (see §3.2) and takes advantage of most of the features of this system. It provides polyphony, MIDI, and OSC support and allows SmartKeyboard interfaces to interact with the DSP portion of the app at a very high level (see Figure 3.7).

`faust2smartkeyb` inherits some of `faust2api`'s options. For example, an external audio effect Faust file can be specified using `-effect`. This is very useful to save computation when implementing a polyphonic synthesizer (see §3.2). Similarly, `-nvoices` can be used to override the default maximum number of polyphony voices (twelve) of the DSP engine generated by `faust2api`.

The DSP engine generated by `faust2api` is transferred to a template Xcode or Android Studio project (see Figure 3.7) and contains the SmartKeyboard declaration (see §3.3.2). The interface of the app, which is implemented in JAVA on Android and in Objective-C on iOS, is built from this declaration. While OSC support is built-in in the DSP engine and works both on iOS and Android, MIDI support is only available on iOS thanks to Rt-MIDI (see §3.2). On Android, raw MIDI messages are retrieved in the JAVA portion of the app and "pushed" to the DSP engine. MIDI is only supported since Android-23 so `faust2smartkeyb` apps wont have MIDI support on older Android versions.

3.3.2 Architecture of a Simple `faust2smartkeyb` Program

The SmartKeyboard interface can be declared in a Faust file using the `SmartKeyboard{}` metadata:

```
declare interface "SmartKeyboard{
  // configuration keys
}";
```

It is based on the idea that a wide range of touchscreen musical interface can be implemented as a set of keyboards with different key numbers (like a table with columns and cells, essentially). Various interfaces ranging from drum pads, isomorphic keyboards, (x, y) controllers, wind instruments fingerings, etc. can be implemented using this paradigm. The position of fingers in the interface can be continuously tracked and transmitted to the DSP engine both as high level parameters formatted by the system (e.g., frequency, note on/off, gain, etc.) or low level parameters (e.g., (x, y) position, key and keyboard ID, etc.). These parameters are declared in the Faust code using default parameter names (see Table 3.7 for a summary).

By default, the screen interface is a polyphonic chromatic keyboard with thirteen keys whose lowest key is a C5 (MIDI note number 60). A set of key/value pairs can be used to override the default look and behavior of the interface (see Table 3.8). Code Listing 3.1 presents the Faust

[20] `https://developer.apple.com/xcode/`
[21] `https://developer.android.com/studio/`

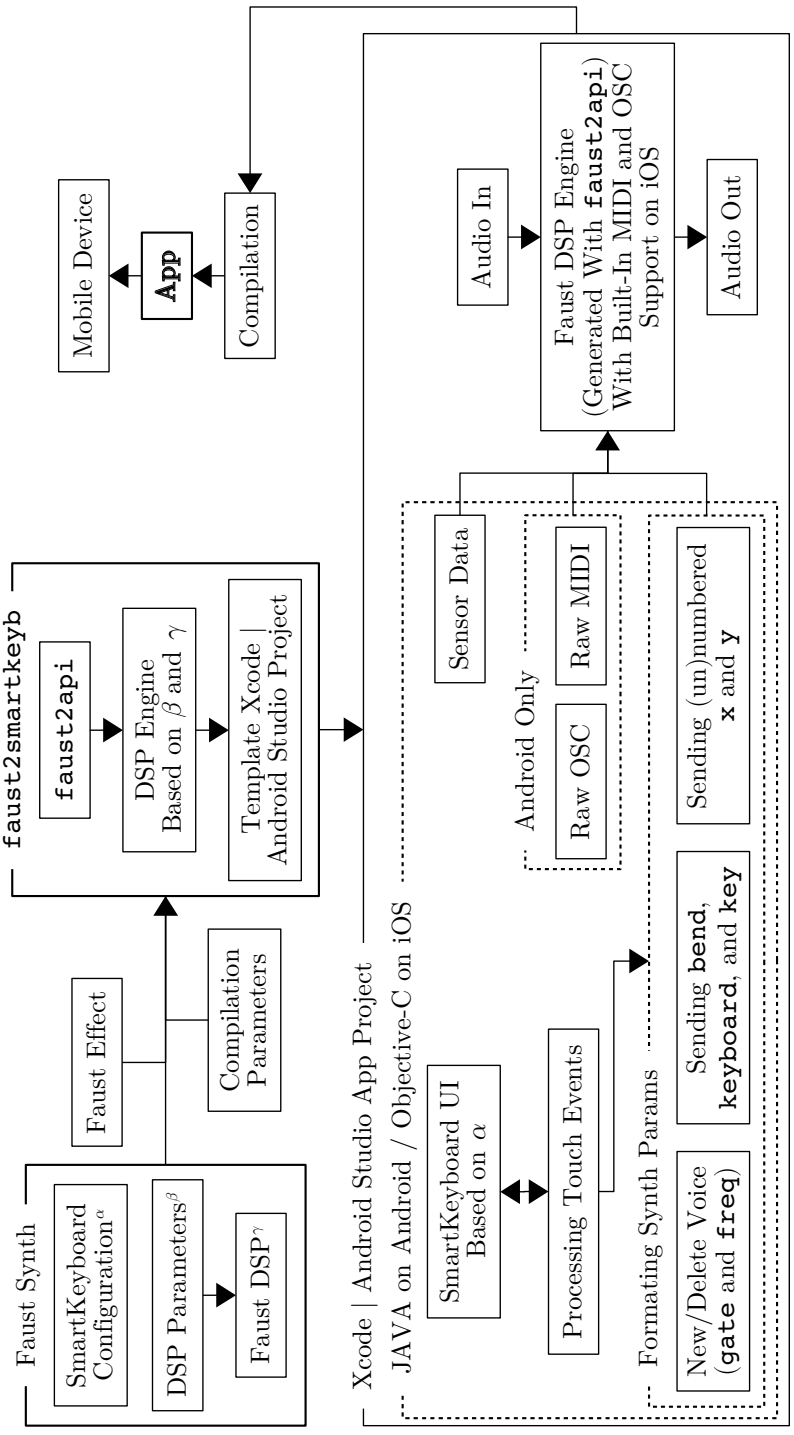


Figure 3.7: Overview of `faust2smartkeyb`.

Parameter Name	Description
freq	Base frequency (if any) of the current note
bend	Deviation from freq as a ratio (1 = no deviation) for continuous pitch control
gate	Note on (1) / Note off (0), typically changes with freq
key	Current key ID
keyboard	Current keyboard ID
kbMfingers	Number of fingers on a specific keyboard M
kbMkNstatus	Status of the current key N in keyboard M
kbMkNx	Normalized (0-1) x position of a finger in key N in keyboard M
kbMkNy	Normalized (0-1) y position of a finger in key N in keyboard M
x	Normalized (0-1) x position of the finger in any key
y	Normalized (0-1) y position of the finger in any key
xN	Normalized (0-1) x position of finger N in a key
yN	Normalized (0-1) y position of finger N in a key

Table 3.7: SmartKeyboard Standard Parameters Overview.

code of a simple app where two identical keyboards can be used to control a simple synthesizer based on a band-limited sawtooth wave oscillator and a simple exponential envelope generator. Since MIDI support is enabled by default in apps generated by faust2smartkeyb and that the SmartKeyboard standard parameters are the same as the one used for MIDI in Faust, this app is also controllable by any MIDI keyboard connected to the device running it. A screen-shot of the interface of the app generated from Code Listing 3.1 can be seen in Figure 3.8.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2'
}";
import("stdfaust.lib");
f = nentry("freq",200,40,2000,0.01);
g = nentry("gain",1,0,1,0.01);
t = button("gate");
envelope = t*g : si.smoo;
process = os.sawtooth(f)*envelope <: _,_;
```

Listing 3.1: Simple SmartKeyboard Faust App.

3.3.3 Preparing a Faust Program for Continuous Pitch Control

In faust2smartkeyb programs, pitch is handled using the freq and bend standard parameters (see Table 3.7). The behavior of the formatting of these parameters can be configured using some of the keys presented in Table 3.8.

Key	Description
`Inter-Keyboard Slide`	Enables slide between keyboards
`Keyboard N - Key M - Label`	Specify text in a specific key and keyboard
`Keyboard N - Lowest Key`	MIDI key number of the lowest key on a specific keyboard
`Keyboard N - Number of Keys`	Number of keys of a specific keyboard
`Keyboard N - Orientation`	Orientation (left to right or right to left) of a specific keyboard
`Keyboard N - Piano Keyboard`	Activate piano keyboard mode (black keys) on a specific keyboard
`Keyboard N - Root Position`	Position of the root on a specific keyboard
`Keyboard N - Scale`	Specify the scale of a specific keyboard
`Keyboard N - Send Freq`	Send `freq` and `bend` from a specific keyboard
`Keyboard N - Send Key X`	Activates the `kbMkNx` standard parameter
`Keyboard N - Send Key Y`	Activates the `kbMkNy` standard parameter
`Keyboard N - Send Key Status`	Activates the `kbMkNstatus` standard parameter
`Keyboard N - Send Numbered X`	Activates the `xN` standard parameter
`Keyboard N - Send Numbered Y`	Activates the `xY` standard parameter
`Keyboard N - Send X`	Activates the `x` standard parameter
`Keyboard N - Send Y`	Activates the `y` standard parameter
`Keyboard N - Show Labels`	Show key labels on a specific keyboard
`Keyboard N - Static Mode`	Fix key appearance on a specific keyboard
`Number of Keyboards`	Number of keyboards in the interface
`Max Fingers`	Maximum number of fingers allowed in the interface
`Max Keyboard Polyphony`	Maximum keyboards polyphony voices
`Mono Mode`	Mode when keyboards are monophonic
`Rounding Cycles`	Number of cycles of pitch rounding
`Rounding Mode`	Pitch rounding mode
`Rounding Smooth`	Smoothness of pitch rounding
`Rounding Threshold`	Pitch rounding threshold
`Rounding Update Speed`	Pitch rounding update speed
`Send Current Key`	Activates the `key` standard parameter
`Send Current Keyboard`	Activates the `keyboard` standard parameter
`Send Fingers Count`	Activates the `kbMfingers` standard parameter
`Send Sensors`	Send sensor values

Table 3.8: `faust2smartkeyb` Keys Overview.

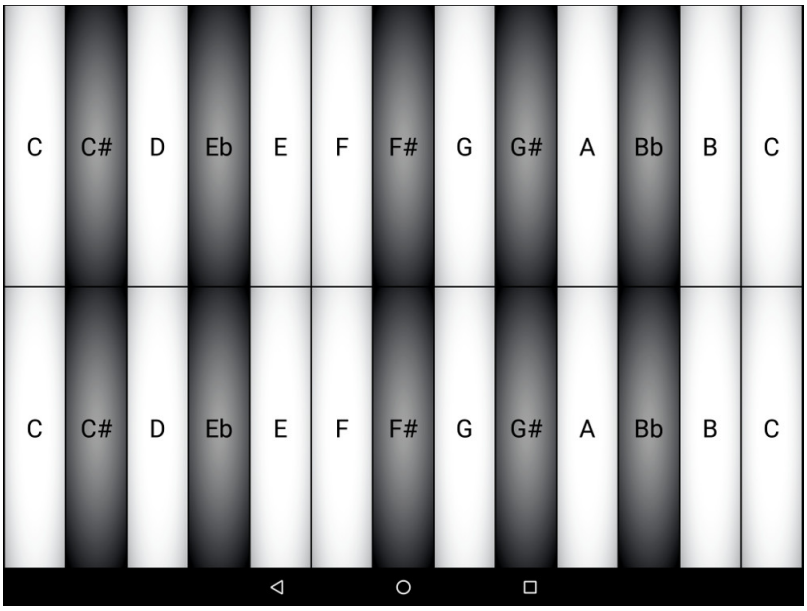


Figure 3.8: Simple SmartKeyboard Interface.

`freq` gives the "reference frequency" of a note and is tied to the `gate` parameter. Every time `gate` goes from 0 to 1 (which correlates with a new note event), the value of `freq` is updated. `freq` always corresponds to an integer MIDI pitch number which implies that its value is always quantized to the nearest semitone.

Pitch can be continuously updated by using the `bend` standard parameter. `bend` is a ratio that should be multiplied to `freq`. E.g.:

```
f = nentry("freq",200,40,2000,0.01);
bend = nentry("bend",1,0,10,0.01) : si.polySmooth(t,0.999,1);
freq = f*bend;
```

The state of polyphonic voices is conserved in memory until the app is ended. Thus, the value of `bend` might jump from one value to another when a new voice is activated. `polySmooth()` is used here to smooth the value of `bend` to prevent clicks, only after the voice started. This suppresses any potential "sweep" that might occur if the value of `bend` changes abruptly at the beginning of a note.

3.3.4 Configuring Continuous Pitch Control

The `Rounding Mode` configuration key has a significant impact on the behavior of `freq`, `bend`, and `gate`.

When `Rounding Mode = 0`, pitch is fully "quantized," and the value of `bend` is always 1. Additionally, a new note is triggered every time a finger slides to a new key, impacting the value of `freq` and `gate`.

When `Rounding Mode = 1`, continuous pitch control is activated, and the value of `bend` is constantly updated in function the position of the finger on the screen. New note events updating the value of `freq` and `gate` are only triggered when fingers start touching the screen. While this mode might be useful in some cases, it is hard to use when playing tonal music as any new note might be "out of tune."

When `Rounding Mode = 2`, "pitch rounding" is activated and the value of `bend` is rounded to match the nearest quantized semitone when the finger is not moving on the screen. This allows generated sounds to be "in tune" without preventing slides, vibratos, etc. While the design of such a system has been previously studied, [147] we decided to implement our own algorithm for this (see Figure 3.9). `touchDiff` is the distance on the screen between two touch events for a specific finger. This value is smoothed (`sTouchDiff`) using a unity-dc-gain one pole lowpass filter in a separate thread running at a rate defined by configuration key `Rounding Update Speed`. `Rounding Smooth` corresponds to the pole of the lowpass filter used for smoothing (0.9 by default). A separate thread is needed since the callback of touch events is only called when events are received. If `sTouchDiff` is greater than `Rounding Threshold` during a certain number of cycles defined by `Rounding Cycles`, then rounding is deactivated and the value of `bend` corresponds to the exact position of the finger on the screen. If rounding is activated, the value of `bend` is rounded to match the nearest pitch of the chromatic scale.

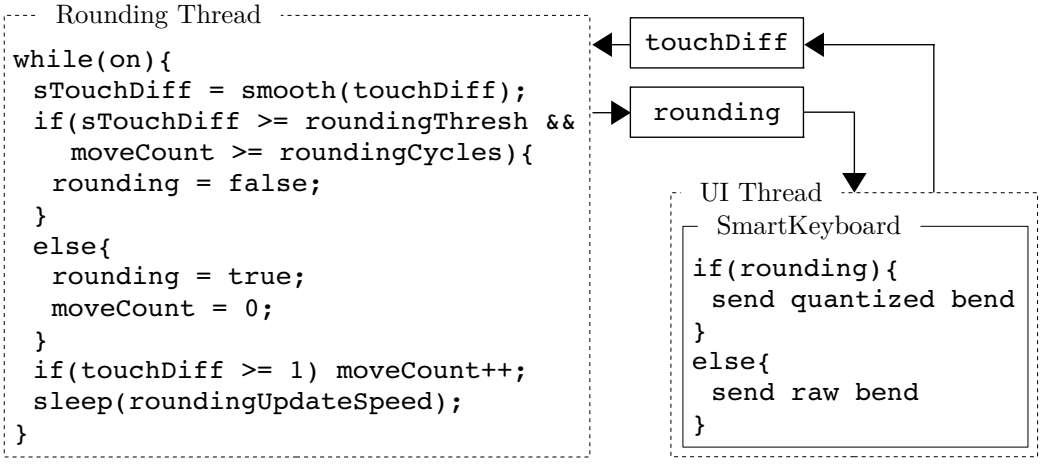


Figure 3.9: SmartKeyboard Pitch Rounding *Pseudo Code* Algorithm.

3.3.5 Using Specific Scales

A wide range of musical scales (see Table 3.9), all compatible with the system described in §3.3.4, can be used with the SmartKeyboard interface and configured using the `Keyboard N - Scale` key (see Table 3.8). When other scales than the chromatic scale are used, keys on the keyboard all have the same color.

Scale ID	Scale Name
0	Chromatic
1	Major
2	Minor
3	Harmonic Minor
4	Dorian
5	South-East Asian
6	Minor Pentatonic
7	Minor Blues
8	Japanese
9	Major Pentatonic
10	Major Blues
11	Mixolydian
12	Klezmer

Table 3.9: SmartKeyboard Scales Configurable With the `Keyboard N - Scale` Key.

Custom scales and temperaments can be implemented using the `Keyboard N - Scale` configuration key. It allows us to specify a series of intervals to be repeated along the keyboard (not necessarily at the octave). Intervals are provided as semitones and can have a decimal value. For example, the chromatic scale can be implemented as:

```
Keyboard N - Scale = {1}
```

Similarly, the standard equal-tempered major scale can be specified as:

```
Keyboard N - Scale = {2,2,1,2,2,2,1}
```

A 5-limit just intoned major scale (rounded to the nearest 0.01 cents) could be:

```
Keyboard N - Scale = {2.0391,1.8243,1.1173,2.0391,2.0391,1.8243,1.1173}
```

Equal-tempered Bohlen-Pierce (dividing 3:1 into 13 equal intervals) would be:

```
Keyboard N - Scale = {146.304230835802}
```

Alternatively, custom scales and pitch mappings can be implemented directly from the Faust code using some of the lower level standard parameters returned by the SmartKeyboard interface (e.g., `x`, `y`, `key`, `keyboard`, etc.).

3.3.6 Handling Polyphony and Monophony

By default, the DSP engine generated by `faust2api` has twelve polyphony voices. This parameter can be overridden using the `-nvoices` option when executing the `faust2smartkeyb` command. This system works independently from the monophonic/polyphonic configuration of the SmartKeyboard interface. Indeed, even when a keyboard is monophonic, a polyphonic synthesizer might still be needed to leave time for the release of an envelope generator, for example.

The `Max Keyboard Polyphony` key defines the maximum number of voices of polyphony of a SmartKeyboard interface. Polyphony is tied to fingers present on the screen, in other words, one finger corresponds to one voice. If `Max Keyboard Polyphony = 1`, then the interface becomes "monophonic." The monophonic behavior of the system is configured using the `Mono Mode` key (see Table 3.10). Each mode might be useful for a specific context. For example, *Mode 3* might be great to use keyboards in the interface as independent guitar strings, etc. More examples of this type of use are provided in §3.4.

Monophonic Mode	Description
`0`	Focus stays on the same finger even if other fingers touch the interface.
`1`	Focus always goes to the latest finger to touch the interface (voice stealing). When the focused finger leaves the interface, focus is transfered to the closest finger.
`2`	Sames as `1`, but the voice is terminated when the focused finger leaves the interface.
`3`	Sames as `1`, but focus is given to new fingers only if their pitch is higher than the current note.
`4`	Sames as `2`, but focus is given to new fingers only if their pitch is lower than the current note.

Table 3.10: Different Monophonic Modes Configured Using the `Mono Mode` Key in SmartKeyboard Interfaces.

3.3.7 Other Modes

In some cases, both the monophonic and the polyphonic paradigms are not adapted. For example, when implementing an instrument based on a physical model, it might be necessary to use a single voice and constantly run it. This might be the case of a virtual wind instrument where notes are "triggered" by some of the continuous parameters of the embouchure (see §3.4.4) and not by discrete events such as the one created by a key. This type of system can be implemented by setting the `Max Keyboard Polyphony` key to zero. In that case, the first available voice is triggered and run until the app is killed. Adding new fingers on the screen will have no impact on that and the

`gate` parameter won't be sent to the DSP engine. `freq` will keep being sent unless the `Keyboard N - Send Freq` is set to zero. Since this parameter is keyboard specific, some keyboards in the interface might be used for pitch control while others might be used for other types of applications (e.g., X/Y controller, etc.). Various examples of this type of use are presented in §3.4.

It might be useful in some cases to number the standard `x` and `y` parameters as a function of the fingers present on the screen. This can be easily accomplished by setting the `Keyboard N - Count Fingers` key to one. In that case, the first finger to touch the screen will send the `x0` and `y0` standard parameters to the DSP engine, the second finger `x1` and `y1`, and so on.

This section just gave an overview of some of the features of `faust2smartkeyb`. More details about this tool can be found in its documentation [22] as well as in the corresponding online tutorials.[23]

3.4 Skill Transfer and Screen Interface: `faust2smartkeyb` Apps Examples

Implementation of skill transfer is one of the primary goals of `faust2smartkeyb`. It is a crucial factor in making a successful DMI as it can help accelerate its learning and make it quickly usable by a large number of performers (see §1.1). A wide range of screen controllers mimicking the interface of existing instruments can be implemented using the SmartKeyboard interface.

This section presents a few examples where traditional acoustic instruments served as models and were turned into digital version running on mobile devices using physical models from the Faust Physical Modeling Library (see §6.2) and `faust2smartkeyb`. We demonstrate that in most cases, the implementation of such instruments can be approached in two different ways. The first one consists of only specifying a single element of an instrument (e.g., one string of a guitar or a violin, membrane of a drum, etc.) and then use the polyphonic features of `faust2smartkeyb` to implement the ability of the instrument to generate several sounds simultaneously. In the other approach, the instrument is modeled in its whole (e.g., four strings for a violin, six strings in a guitar, etc.) and the mapping between the interface and the model is handled directly in the Faust code.

While the goal of this section is not to be exhaustive, it should provide enough material to demonstrate how to implement most traditional musical instruments and more.

3.4.1 Plucked Strings Instruments: the Guitar

Piano Keyboard Paradigm

Plucked string instruments such as the guitar, the banjo, etc. are relatively close to struck string instruments (e.g., the piano, etc.) as they are excited by punctual events (unlike bowed strings

[22] `https://ccrma.stanford.edu/~rmichon/smartKeyboard/`
[23] `https://ccrma.stanford.edu/~rmichon/faustTutorials/`

or wind instruments, where energy must be constantly introduced in the system for it to produce any sound). For this reason, controlling these types of instrument with a "piano keyboard like" interface makes a lot of sense as the performer expect sound to be heard when a key is pressed. A good commercial example of such DMI is GeoShred (see §1.1.4), where a new pluck is triggered every time a finger touches a virtual string on the touch screen (this behavior might slightly change depending on the configuration of the interface).

Listing 3.2 presents a `faust2smartkeyb` code implementing an instrument working in a similar way as GeoShred. Six parallel keyboards are used to represent six parallel strings. They are all monophonic and implement "voice stealing" with priority to higher pitches which means that the current note is terminated when a new finger touches the same keyboard only if the pitch of the note to trigger is higher than the current one (like on a physical electric guitar string). Even though keyboards are monophonic, the overall instrument is polyphonic and several strings can be excited at the same time, taking advantage of the voice allocation system of `faust2smartkeyb`.

Keyboards are placed one fourth apart from each other, in a similar way as on a guitar neck, in order to facilitate skills transfer for guitar players. Finally, slides and vibratos can be carried out on the same string just by continuously moving the finger along the virtual keyboard.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'6','Max Keyboard Polyphony':'1',
  'Mono Mode':'3','Rounding Mode':'2',
  'Keyboard 0 - Number of Keys':'13',
  [...same for all other keyboards...]
  'Keyboard 0 - Lowest Key':'72','Keyboard 1 - Lowest Key':'67',
  'Keyboard 2 - Lowest Key':'62','Keyboard 3 - Lowest Key':'57',
  'Keyboard 4 - Lowest Key':'52','Keyboard 5 - Lowest Key':'47'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
f = hslider("freq",300,50,2000,0.01);
bend = hslider("bend[midi:pitchwheel]",1,0,10,0.01) :
  si.polySmooth(gate,0.999,1);
gain = hslider("gain",1,0,1,0.01);
s = hslider("sustain[midi:ctrl 64]",0,0,1,1); // for sustain pedal
t = button("gate");

// MODEL PARAMETERS
```

```
gate = t+s : min(1);
freq = f*bend : max(60); // min freq is 60 Hz
stringLength = freq : pm.f2l;
pluckPosition = 0.8;
mute = gate : si.polySmooth(gate,0.999,1);

process = pm.elecGuitar(stringLength,pluckPosition,mute,gain,gate)
  <: _,_;
```

Listing 3.2: `faust2smartkeyb` App Implementing an Electric Guitar With an Isomorphic Keyboard.

The electric guitar string physical model is implemented in the FAUST Physical Modeling Library as `elecGuitar()`. The effect chain is declared in a separate file in order to use the `-effect` option when using `faust2smartkeyb` and involves a distortion and a reverb:

```
process = par(i,2,ef.cubicnl(0.8,0)) : dm.zita_rev1;
```

The pitch of the virtual string is controlled by the combination of the `freq` and `bend` standard parameters. Strings are progressively muted when the finger leaves the string. In other words, they only resonate if the associated finger remains on the screen.

External Plucking Paradigm

Even though the paradigm presented previously works well with plucked string instruments, it differs from that of a real guitar because of the lack of an independent interface for exciting the different strings. Listing 3.3 presents a `faust2smartkeyb` app where virtual strings are excited through a separate keyboard on the touch-screen. This keyboard could be easily substituted by an external active controller (see §5).

The interface contains seven keyboards: six implementing the different strings of the guitar (and tuned the same way as on this instrument: E, A, D, G, B, E) and one used as the interface to trigger the virtual strings. `Max Keyboard Polyphony` is set to zero so that a single voice is computed when the app is launched. Indeed, unlike the previous example, the six strings of the instrument are all implemented in the same process, thus only one voice is necessary. The `freq` and `bend` standard parameters of the first six keyboards are retrieved and used to control the pitch of the six independent strings.

The seventh keyboard is configured to have six keys (one for each string). We want a specific string to be excited when a finger touches the corresponding key. Since this system should react both to *touch* and *move* events, both event types `1` and `4` are considered when formating the value of `kb6kstatus`. As mentioned previously, this keyboard could (and probably should) be replaced by an external active controller (see §5) and is only here as a proof of concept. Similarly, the pluck

position is currently controlled using the y axis of the accelerometer but this parameter could also be associated to a potential external active controller.

The acoustic guitar physical model used in this example is implemented in the FAUST Physical Modeling Library (see §6.2) as `nylonGuitarModel()`. Here, six models (one for each string) are computed in parallel.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'7','Max Keyboard Polyphony':'0',
  'Rounding Mode':'2',
  'Keyboard 0 - Number of Keys':'14',
  [...same for other keyboards 1, 2, 3, 4, and 5...]
  'Keyboard 6 - Number of Keys':'6',
  'Keyboard 0 - Lowest Key':'52','Keyboard 1 - Lowest Key':'57',
  'Keyboard 2 - Lowest Key':'62','Keyboard 3 - Lowest Key':'67',
  'Keyboard 4 - Lowest Key':'71','Keyboard 5 - Lowest Key':'76',
  'Keyboard 0 - Send Keyboard Freq':'1',
  [...same for all other keybaords...],
  'Keyboard 6 - Piano Keyboard':'0',
  'Keyboard 6 - Send Key Status':'1',
  'Keyboard 6 - Key 0 - Label':'S0','Keyboard 6 - Key 1 - Label':'S1',
  'Keyboard 6 - Key 2 - Label':'S2','Keyboard 6 - Key 3 - Label':'S3',
  'Keyboard 6 - Key 4 - Label':'S4','Keyboard 6 - Key 5 - Label':'S5'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kbfreq(0) = hslider("kb0freq",164.8,20,10000,0.01);
kbbend(0) = hslider("kb0bend",1,0,10,0.01);
[...same for other keyboards until kb5...]
kb6kstatus(0) = hslider("kb6k0status",0,0,1,1) <: ==(1) | ==(4) : int;
kb6kstatus(1) = hslider("kb6k1status",0,0,1,1) <: ==(1) | ==(4) : int;
[...same for all other keys of kb6...]

// MODEL PARAMETERS
sl(i) = kbfreq(i)*kbbend(i) : pm.f2l : si.smoo; // strings length
pluckPosition =
    hslider("pluckPosition[acc: 1 0 -10 0 10]",0.5,0,1,0.01) : si.smoo;
```

```
// ASSEMBLING MODELS
nStrings = 6; // number of strings
guitar = par(i,nStrings,kb6kstatus(i) : ba.impulsify :
    pm.nylonGuitarModel(sl(i),pluckPosition)) :> _;

process = guitar <: _,_;
```

Listing 3.3: `faust2smartkeyb` App Implementing an Acoustic Guitar With an Independent Plucking Interface.

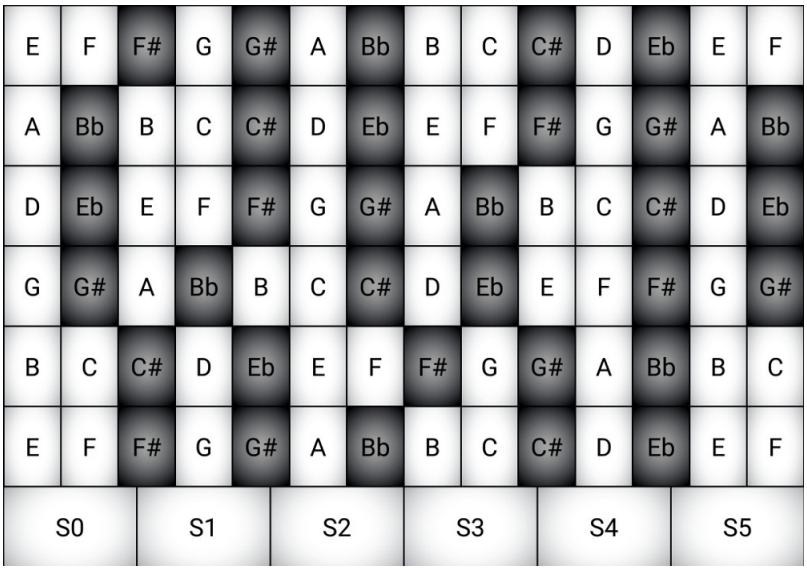


Figure 3.10: Screen-shot of the Interface of the App Generated From the Code Presented in Listing 3.3.

3.4.2 Bowed Strings Instruments: the Violin

Unlike plucked string instruments (see §3.4.1), bowed string instruments must be constantly excited to generate sound. Thus, parameters linked to bowing (i.e., bow pressure, bow velocity, etc.) must be continuously controlled. The `faust2smartkeyb` code presented in Listing 3.4 is a violin app where each string is represented by one keyboard in the interface (in a similar way than the guitar presented in §3.4.1). An independent interface can be used to control the bow pressure and velocity. This interface is common to all strings that are activated when they are touched on the screen. Just like for the acoustic guitar app presented in §3.4.1, this interface could be substituted by an external one (see §5).

The SmartKeyboard configuration declares 5 keyboards (4 strings and one control surface for bowing). "String keyboards" are tuned like on a violin (G, D, A, E) and are configured to be monophonic and implement "pitch stealing" when a higher pitch is selected (see §3.4.1). Bow velocity is computed by measuring the displacement of the finger touching the 5th keyboard (`bowVel`). Bow pressure just corresponds to the y position of the finger on this keyboard. Strings are activated when at least one finger is touching the corresponding keyboard (`as(i)`).

The app doesn't take advantage of the polyphony support of `faust2smartkeyb` and a single voice is constantly ran after the app is launched (`Max Keyboard Polyphony = 0`). Four virtual strings based on a simple violin string model (`violinModel()`) implemented in the Faust Physical Modeling Library are declared in parallel and activated in function of events happening on the screen.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'5','Max Keyboard Polyphony':'0',
  'Rounding Mode':'2','Send Fingers Count':'1',
  'Keyboard 0 - Number of Keys':'19',
  [...same for next 3 keyboards...]
  'Keyboard 4 - Number of Keys':'1',
  'Keyboard 0 - Lowest Key':'55','Keyboard 1 - Lowest Key':'62',
  'Keyboard 2 - Lowest Key':'69','Keyboard 3 - Lowest Key':'76',
  'Keyboard 0 - Send Keyboard Freq':'1',
  [...same for next 3 keyboards...]
  'Keyboard 4 - Send Freq':'0','Keyboard 4 - Send Key X':'1',
  'Keyboard 4 - Send Key Y':'1','Keyboard 4 - Static Mode':'1',
  'Keyboard 4 - Key 0 - Label':'Bow'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kbfreq(0) = hslider("kb0freq",220,20,10000,0.01);
kbbend(0) = hslider("kb0bend",1,0,10,0.01);
[...same for the 3 next keyboards...]
kb4k0x = hslider("kb4k0x",0,0,1,1) : si.smoo;
kb4k0y = hslider("kb4k0y",0,0,1,1) : si.smoo;
kbfingers(0) = hslider("kb0fingers",0,0,10,1) : int;
[...same for the 3 next keyboards...]

// MODEL PARAMETERS
```

```
// strings lengths
sl(i) = kbfreq(i)*kbbend(i) : pm.f2l : si.smoo;
// string active only if fingers are touching the keyboard
as(i) = kbfingers(i)>0;
bowPress = kb4k0y; // could also be controlled by an external controller
// finger displacement on screen
bowVel = kb4k0x-kb4k0x' : abs : *(8000) : min(1) : si.smoo;
bowPos = 0.7; // could be controlled by an external controller

// ASSEMBLING MODELS
// essentially 4 parallel violin strings
model = par(i,4,pm.violinModel(sl(i),bowPress,bowVel*as(i),bowPos))
  :> _;

process = model <: _,_;
```

Listing 3.4: `faust2smartkeyb` App Implementing a Violin With an Independent Interface for Bowing.

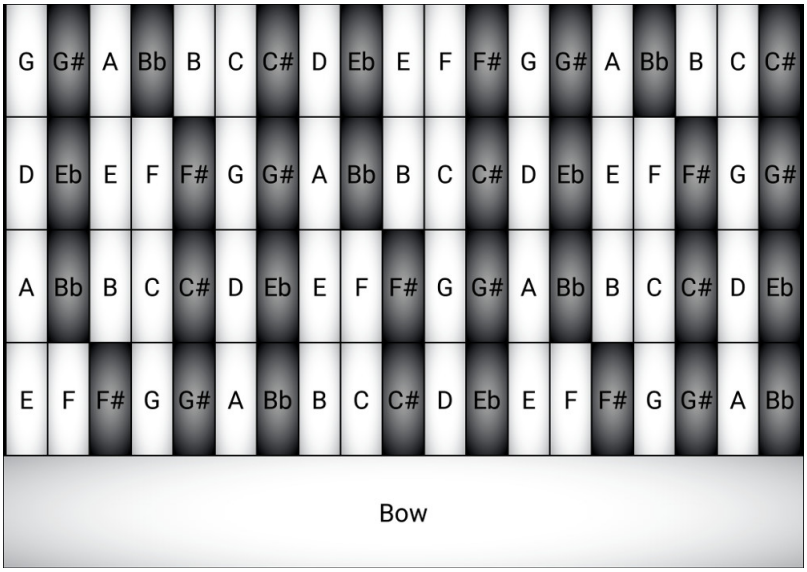


Figure 3.11: Screen-shot of the Interface of the App Generated From the Code Presented in Listing 3.4.

Alternatively, the bowing interface could be removed and the bow velocity could be calculated based on the displacement on the y axis of a finger on a keyboard, allowing one to excite the string

and control its pitch with a single finger. However, concentrating so many parameters on a single gesture tends to limit the affordances of the instrument. The code presented in Listing 3.4 could be easily modified to implement this behavior.

3.4.3 Percussion Instruments: Polyphonic Keyboard and Independent Instruments Paradigms

Just like plucked string instruments (see §3.4.1), percussion instruments can be implemented using `faust2smartkeyb` either as polyphonic instruments or as a constantly running synthesizer implementing multiple instruments in parallel. In the first case (see the djembes example below), a new voice is allocated every time the instrument is stroke. A voice might implement several models and choose one of them in function of the pad/key being touched. Another option is to use a single scalable model whose properties will change every time a voice is started. In the second case (see the bells example below), a single voice implementing several models in parallel is initiated when the app is launched and models are excited in function of the pad/key touched in the interface. The following subsections provide examples of these two paradigms.

Set of Djembes: Example of Polyphonic Keyboard Paradigm for Percussion Instruments

The code presented in Listing 3.5 implements a SmartKeyboard app where three pads can be used to play three djembes of different sizes. A single model whose fundamental frequency is adjusted in function of the virtual pad being stroke is used. This app takes advantage of the polyphony system of `faust2smartkeyb` and a new voice is instantiated every time a new strike happens on the touchscreen.

The interface is made out of two polyphonic keyboards (one with two keys and one with one key). The (x, y) position of the finger on the keys/pads are retrieved and used to compute the excitation position (`exPos`) on the model. The fundamental frequency (`rootFreq`) of the model is selected in function of the pad being touched. The djembe physical model used in this program is implemented in the Faust Physical Modeling Library.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2','Keyboard 0 - Number of Keys':'2',
  'Keyboard 1 - Number of Keys':'1','Keyboard 0 - Static Mode':'1',
  'Keyboard 1 - Static Mode':'1','Keyboard 0 - Send X':'1',
  'Keyboard 0 - Send Y':'1','Keyboard 1 - Send X':'1',
  'Keyboard 1 - Send Y':'1','Keyboard 0 - Piano Keyboard':'0',
  'Keyboard 1 - Piano Keyboard':'0','Keyboard 0 - Key 0 - Label':'High',
  'Keyboard 0 - Key 1 - Label':'Mid','Keyboard 1 - Key 0 - Label':'Low'
```

```
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
gate = button("gate");
x = hslider("x",1,0,1,0.001);
y = hslider("y",1,0,1,0.001);
keyboard = hslider("keyboard",0,0,1,1) : int;
key = hslider("key",0,0,1,1) : int;

djembeInstrument = pm.djembe(rootFreq,exPos,strikeSharpness,gain,gate)
with{
  bFreq = 60; // frequency of the lowest djembe
  padID = 2-(keyboard*2+key); // retrieving pad ID (0-2)
  rootFreq = bFreq*(padID+1); // djembe root freq
  exPos = min((x*2-1 : abs),(y*2-1 : abs)); // excitation position
  strikeSharpness = 0.5;
  gain = 2;
};

process = djembeInstrument <: _,_;
```

Listing 3.5: `faust2smartkeyb` App Implementing a Set of Djembes.

A similar approach could be used to map keys/pads to completely different models by declaring them in the same FAUST code (i.e., voice in this case) and activating them in function the key being touched.

Set of Bells: Examples of Independent Instrument Paradigm for Percussion Instruments

The code presented in Listing 3.6 implements a SMARTKEYBOARD app where four different bells are associated to four different pads on the touchscreen. The strike position on each pad is used to control the excitation position on the corresponding virtual bell.

The SMARTKEYBOARD interface is made out of two keyboards of two keys. A single voice is instantiated whenever the app is launched (`Max Keyboard Polyphony = 0`). Four bell physical models from the FAUST Physical Modeling Library are ran in parallel (see §6.2 and §B). The status of each key in the interface is retrieved and used to trigger the excitation for each bell independently.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2','Max Keyboard Polyphony':'0',
  'Keyboard 0 - Number of Keys':'2','Keyboard 1 - Number of Keys':'2',
  'Keyboard 0 - Send Freq':'0','Keyboard 1 - Send Freq':'0',
  'Keyboard 0 - Piano Keyboard':'0','Keyboard 1 - Piano Keyboard':'0',
  'Keyboard 0 - Send Key Status':'1','Keyboard 1 - Send Key Status':'1',
  'Keyboard 0 - Send X':'1','Keyboard 0 - Send Y':'1',
  'Keyboard 1 - Send X':'1','Keyboard 1 - Send Y':'1',
  'Keyboard 0 - Key 0 - Label':'English',
  'Keyboard 0 - Key 1 - Label':'French',
  'Keyboard 1 - Key 0 - Label':'German',
  'Keyboard 1 - Key 1 - Label':'Russian'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kb0k0status = hslider("kb0k0status",0,0,1,1) : min(1) : int;
kb0k1status = hslider("kb0k1status",0,0,1,1) : min(1) : int;
kb1k0status = hslider("kb1k0status",0,0,1,1) : min(1) : int;
kb1k1status = hslider("kb1k1status",0,0,1,1) : min(1) : int;
x = hslider("x",1,0,1,0.001);
y = hslider("y",1,0,1,0.001);

// MODEL PARAMETERS
strikeCutoff = 6500; strikeSharpness = 0.5;
strikeGain = 1; nModes = 10; // synthesize 10 modes out of 50
t60 = 30; // resonance duration is 30s
nExPos = 7; // number of excitation positions
exPos = min((x*2-1 : abs),(y*2-1 : abs))*(nExPos-1) : int;

// ASSEMBLING MODELS
bells =
    (kb0k0status : pm.strikeModel(10,strikeCutoff,strikeSharpness,
       strikeGain) : pm.englishBellModel(nModes,exPos,t60,1,3)) +
    (kb0k1status : pm.strikeModel(10,strikeCutoff,strikeSharpness,
       strikeGain) : pm.frenchBellModel(nModes,exPos,t60,1,3)) +
```

```
    (kb1k0status : pm.strikeModel(10,strikeCutoff,strikeSharpness,
       strikeGain) : pm.germanBellModel(nModes,exPos,t60,1,2.5)) +
    (kb1k1status : pm.strikeModel(10,strikeCutoff,strikeSharpness,
       strikeGain) : pm.russianBellModel(nModes,exPos,t60,1,3))
  :> *(0.2);

process = bells <: _,_;
```

Listing 3.6: `faust2smartkeyb` App Implementing a Set of Bells.

This approach is often better suited for physical-model-based percussion instruments as it is much closer to how acoustic musical instrument work. Indeed, unlike the djembe examples, all bell models are constantly ran here and no concept of polyphony is used.

3.4.4 Wind Instruments: Key Combinations and Continuous Control

As for instruments from the previous categories treated in this section, wind instruments can be implemented with `faust2smartkeyb` using either the "polyphonic keyboard" or the "full model" paradigm. This second case is particularly relevant for wind instruments that are often monophonic and where pitch is usually selected by combining several keys (unlike a piano keyboard where one key corresponds to one pitch). The `faust2smartkeyb` code presented in Listing 3.7 implements a clarinet app which is meant to be run on a small screen device (i.e., a smart-phone). The device is expected to be held with two hands with thumbs underneath and all other fingers on the screen. The instrument is played by blowing onto the built-in microphone which is used to control breath pressure. Different buttons on the screen interface represent the keys of the instrument. The y axis of the built-in accelerometer controls the "bell opening" parameter which acts as a mute on the instrument.

The screen interface is made out of two keyboards of four and five keys, respectively. The highest key on both keyboards can be used to switch between octaves (see Figure 3.12). The key on the first keyboard switches octaves up (`octaveShiftUp`) and the key on the second keyboard octaves down (`octaveShiftDown`). These keys are meant to be touched by the "baby finger" of both hands. Other keys reproduce a simplified version of clarinet fingerings presented in Figure 3.12. This mapping was designed to leverage existing skills while adapting them to what can be implemented on a touchscreen. This type of behavior is created by retrieving the status of all keys in the interface by using the `kbMkNstatus` standard parameter and comparing them to expected fingers combinations. The length of the tube of the clarinet physical model is modulated in function of all these elements. The model is part of the FAUST Physical Modeling Library. The `pressure` parameter is computed by using an envelope follower (`an.amp_follower_ud()`) on the signal of the built-in microphone of the device. Better results can be achieved by using the passive mouthpiece presented in §4.2.1 or

by using an external active breath controller.

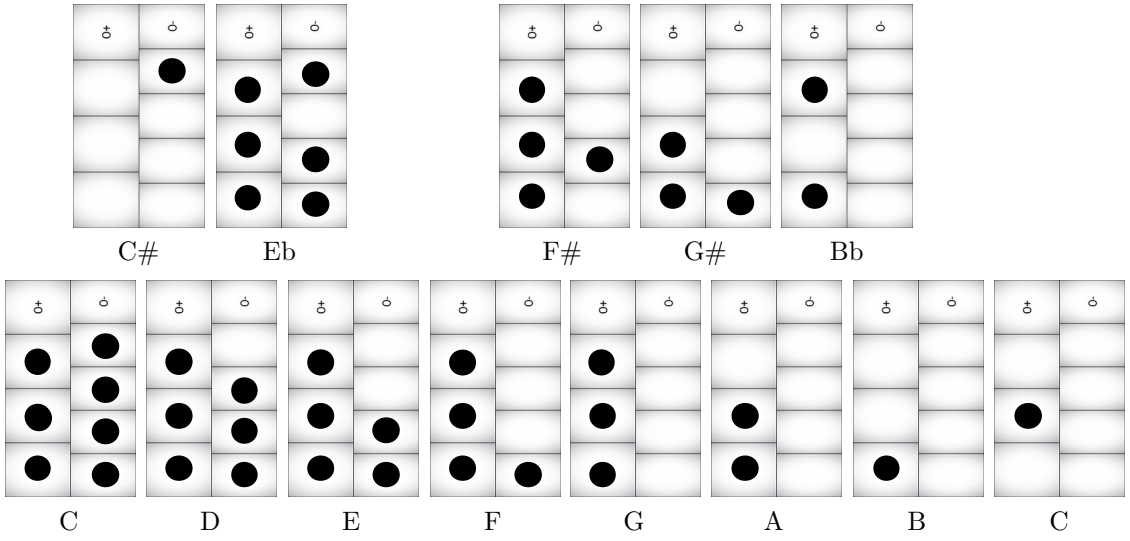


Figure 3.12: Fingers Mapping of the Interface of the App Generated From the Code Presented in Listing 3.7.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2','Max Keyboard Polyphony':'0',
  'Keyboard 0 - Number of Keys':'4','Keyboard 1 - Number of Keys':'5',
  'Keyboard 0 - Send Freq':'0','Keyboard 1 - Send Freq':'0',
  'Keyboard 0 - Piano Keyboard':'0','Keyboard 1 - Piano Keyboard':'0',
  'Keyboard 0 - Send Key Status':'1','Keyboard 1 - Send Key Status':'1',
  'Keyboard 0 - Key 3 - Label':'O+','Keyboard 1 - Key 4 - Label':'O-'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kb0k0status = hslider("kb0k0status",0,0,1,1) : min(1) : int;
kb0k1status = hslider("kb0k1status",0,0,1,1) : min(1) : int;
[...same for all other keys of all keyboards...]

// MODEL PARAMETERS
bellOpening =
```

```
  hslider("bellOpening[acc: 1 1 -10 0 10]",0.5,0.3,0.7,0.01) : si.smoo;
basePitch = 73; // C#4
pitchShift = // calculate pitch shift in function of "keys" combination
  ((kb0k0status == 0) & (kb0k1status == 1) & (kb0k2status == 0) &
    (kb1k0status == 0) & (kb1k1status == 0) & (kb1k2status == 0) &
    (kb1k3status == 0))*(-1) + // C
  [...same for other notes of the chromatic scale...]
  ((kb0k0status == 1) & (kb0k1status == 1) & (kb0k2status == 1) &
    (kb1k0status == 1) & (kb1k1status == 1) & (kb1k2status == 1) &
    (kb1k3status == 1))*(-13); // C
octaveShiftUp = +(kb0k3status : ba.impulsify)~_; // counting up
octaveShiftDown = +(kb1k4status : ba.impulsify)~_; // counting down
octaveShift = (octaveShiftUp-octaveShiftDown)*(12);
tubeLength =
  basePitch+pitchShift+octaveShift : ba.midikey2hz : pm.f2l : si.smoo;
reedStiffness = 0.5;

model(pressure) =
  pm.clarinetModel(tubeLength,pressure,reedStiffness,bellOpening);

// pressure is estimated from mic signal
process = an.amp_follower_ud(0.02,0.02)*0.7 : model <: _,_;
```

Listing 3.7: `faust2smartkeyb` App Implementing a Clarinet.

Even though `faust2smartkeyb` has been tested and evaluated within several workshops (see §4.4 and §5.4), it is such a large project that there probably remains bugs to be fixed. Additionally, despite the fact that we haven't found a touchscreen interface for live music performance that can't be implemented with this system yet, many cases probably haven't been tested (or thought of) and there definitely exists rooms for improvements.

Chapter 4

Passively Augmenting Mobile Devices

"Simplicity is the ultimate sophistication." (Leonardo Da Vinci)

In §1.4.1, we gave an overview of how some of the limitations of mobile devices can be overstepped by "enhancing" them with specialized passive augmentations. These usually leverage existing features of the device such as its built-in sensors, speakers, etc., but can also be purely aesthetic, or facilitate specific gestures.

In this chapter based on a paper [124][1] we published at the *2017 New Interfaces for Musical Expression Conference*, [2] we try to generalize the concept of "passively augmented mobile device" and we provide a framework to design this kind of instrument. We focus on "passive augmentations" leveraging existing components of hand-held mobile devices in a very lightweight, non-invasive way (as opposed to "active augmentation" presented in §5 that require the use of electronic components). We introduce Mobile3D, an OpenScad[3] library to help design mobile device augmentations using DIY digital fabrication techniques such as 3D printing and laser cutting. We give an exhaustive overview of the taxonomy of the various types of passive augmentations that can be implemented on mobile devices through a series of examples and we demonstrate how they leverage existing components on the device. Finally, we evaluate our framework and propose future directions for this type of research.

4.1 Mobile 3D

Mobile3D is an OpenScad library facilitating the design of mobile device augmentations. OpenScad is an open-source Computer Assisted Design (CAD) software using a high level functional programming language to specify the shape of any object. It supports fully parametric parts, permitting

[1] Some sections and figures of this paper were copied verbatim here.
[2] http://www.nime2017.org/
[3] http://www.openscad.org/

users to rapidly adapt geometries to the variety of devices available on the market.

MOBILE3D is organized in different files that are all based on a single library containing generic standard elements (`basics.scad`) ranging from simple useful shapes to more advanced augmentations such as the ones presented in the following sections. A series of device-specific files adapt the elements of `basics.scad` and are also available for the iPhone 5, 6, and 6 Plus and for the iPod Touch. For example, a generic horn usable as a passive amplifier for the built-in speaker of a mobile device can be simply created with the following call in OpenScad:

```
include <basics.scad>
SmallPassiveAmp();
```

The corresponding 3D object can be seen in Figure 4.1.

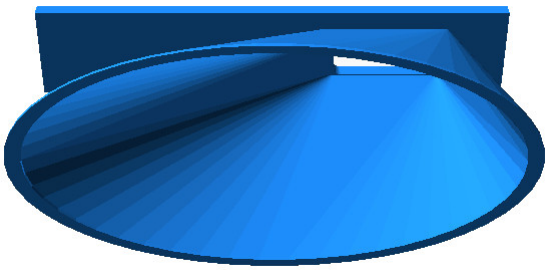


Figure 4.1: CAD Model of a Generic Passive Amplifier for the Built-In Speakers of a Mobile Device.

To generate the same object specifically for the iPhone 5, the following code can be written:

```
include <iPhone5.scad>
iPhone5_SmallPassiveAmp();
```

Finally, the shape of an object can be easily modified either by providing parameters as arguments to the corresponding function, or by overriding them globally before the function is called. If this approach is chosen, all the parts called in the OpenScad code will be updated, which can be very convenient in some cases. For example, the radius (expressed in millimeters here) of `iPhone5_SmallPassiveAmp()` can be modified locally by writing:

```
include <iPhone5.scad>
iPhone5_SmallPassiveAmp(hornRadius=40);
```

or globally by writing:

```
include <iPhone5.scad>
iPhone5_SmallPassiveAmp_HornRadius = 40;
iPhone5_SmallPassiveAmp();
```

MOBILE3D is based on two fundamental elements that can be used to quickly attach any prosthetic to the device: the top and bottom *holders* (see Figure 4.2). They were designed to be 3D printed using elastomeric material such as *NinjaFlex*®[4] in order to easily install and remove the device without damaging it. They also help reducing printing duration, which is often a major issue during prototyping. These two holders glued to a laser-cut plastic plate form a sturdy case (as shown in Figure 4.2), whereas completely printing this part would take much more time.

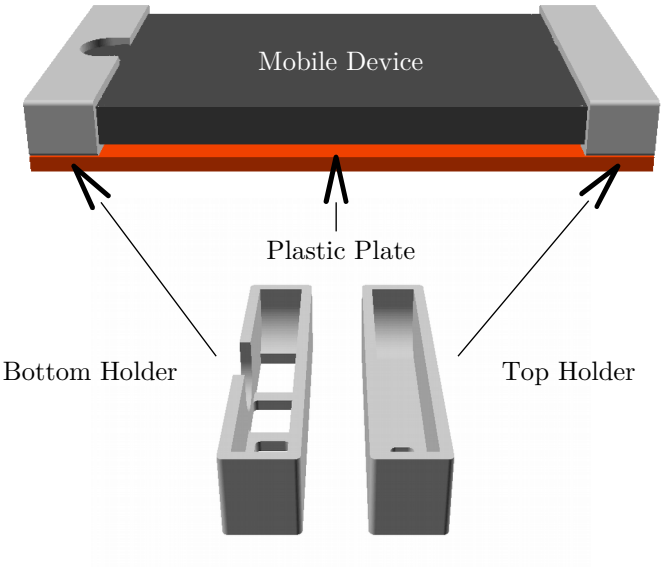


Figure 4.2: CAD Model of a Simple iPhone 5 Case Made From 3D-Printed Holders and a Laser-Cut Plastic Plate.

A wide range of elements can be easily added to the basic mobile phone case presented in Figure 4.2 by using adhesives. Some of them will be presented in greater details in the following sections and are part of MOBILE3D.

Figure 4.3 presents an example of an iPhone 5 augmented with a passive amplifier similar to the one presented above. The bottom holder and the horn were printed separately and glued together, but they could also have been printed as one piece. In this example, the bottom and top holders were printed with PLA,[5] which is a hard plastic, and they were mounted on the plate using Velcro®. This is an alternative solution to using *NinjaFlex*® that can be useful when augmenting the mobile device with large appendixes requiring a stronger support.

The passive amplifier presented in Figure 4.3 was made by overriding the default parameters of the `iPhone5_SmallPassiveAmp()` function:

[4] `https://ninjatek.com/`
[5] PolyLactic Acid.

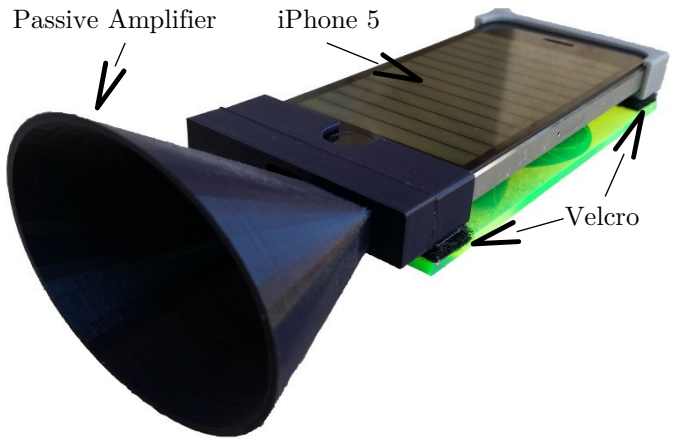


Figure 4.3: iPhone 5 Augmented With a Horn Used as Passive Amplifier on Its Built-In Speaker (Instrument by Erin Meadows).

```
include <lib/iPhone5.scad>
iPhone5_SmallPassiveAmp_HornLength = 40;
iPhone5_SmallPassiveAmp_HornRadius = 40;
iPhone5_SmallPassiveAmp_HornDeformationFactor = 0.7;
iPhone5_SmallPassiveAmp();
```

An exhaustive list of all the elements available in MOBILE3D can be found on the project webpage.[6]

4.2 Leveraging Built-In Sensors and Elements

Mobile devices host a wide range of built-in sensors and elements that can be used to control sound synthesizers (see §1.3). While the variety of available sensors and elements differs from one device to another, most smart-phones have at least a touchscreen, a loudspeaker, a microphone, and some type of motion sensor (accelerometer, gyroscope, etc.). In this section, we'll focus on these four elements and we'll demonstrate how they can be "augmented" for specific musical applications.

4.2.1 Microphone

While the built-in microphone of a mobile device can simply serve as a source for any kind of sound process (e.g., audio effect, physical model, etc.), it can also be used as a versatile, high rate sensor [132]. In this section, we demonstrate how it can be augmented for different kinds of uses.

[6] https://ccrma.stanford.edu/~rmichon/mobile3D

Amplitude-Detection-Based Augmentations

One of the first concrete uses of the built-in microphone of a mobile device to control some sound synthesis process was done with Smule's Ocarina (see §1.3.2). There, the microphone serves as a blow sensor by measuring the gain of the signal created when blowing on it to control the gain of an ocarina sound synthesizer.

MOBILE3D contains an object that can be used to leverage this principle when placed in front of the microphone (see Figure 4.4). It essentially allows the performer to blow into a mouthpiece mounted on the device. The air-flow is directed through a small aperture inside the pipe, creating a sound that can be recorded by the microphone and analyzed in the app using standard amplitude tracking techniques. The air-flow is then sent outside of the pipe, preventing it from ever being in direct contact with the microphone.

The acquired signal is much cleaner than when the performer blows directly onto the mic, allowing us to generate precise control data. Additionally, condensation never accumulates on the mic which can help extend the duration of its life, etc.

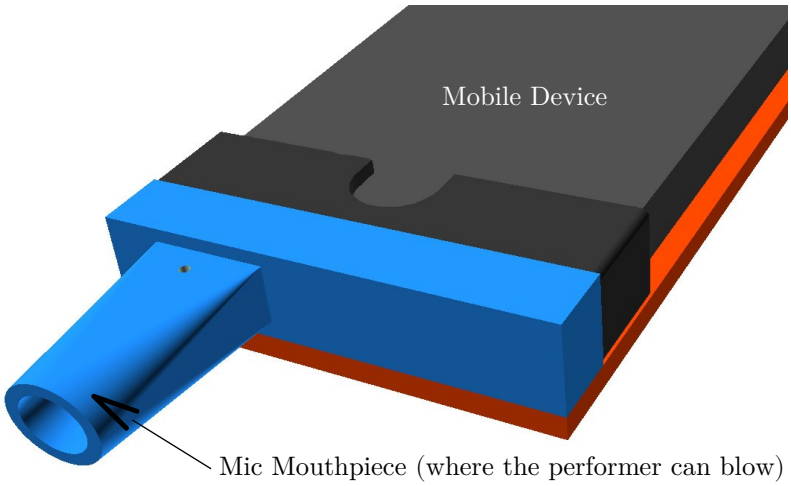


Figure 4.4: Mouthpiece for Mobile Device Built-In Mic.

Frequency-Detection-Based Augmentations

The built-in microphone of mobile devices has already been used as a data acquisition system to implement various kinds of sensors using frequency analysis techniques [95]. MOBILE3D contains an object using similar principles that can be used to control some of the parameters of a synthesizer running on a mobile device. It is based on a conical tube (see Figure 4.5) where dozens of small tines of different length and diameter are placed inside it. These tines get thicker towards the end

of the tube and their length varies linearly around it. When the performer blows inside the tube, the resulting airflow hits the nails, creating sounds with varying harmonic content. By directing the airflow towards different locations inside the tube, the performer can generate various kind of sounds that can be recognized in the app using frequency analysis techniques. The intensity and the position of the airflow around the tube can be measured by keeping track of the spectral centroid of the generated sound, and used to control synthesis parameters.

The same approach can be used with an infinite number of augmentations with different shapes. While our basic spectral-centroid-based analysis technique only allows us to extract two continuous parameters from the generated signal, it should be possible to get more of them using more advanced techniques such as those used with MOGEES (see §1.2.2).

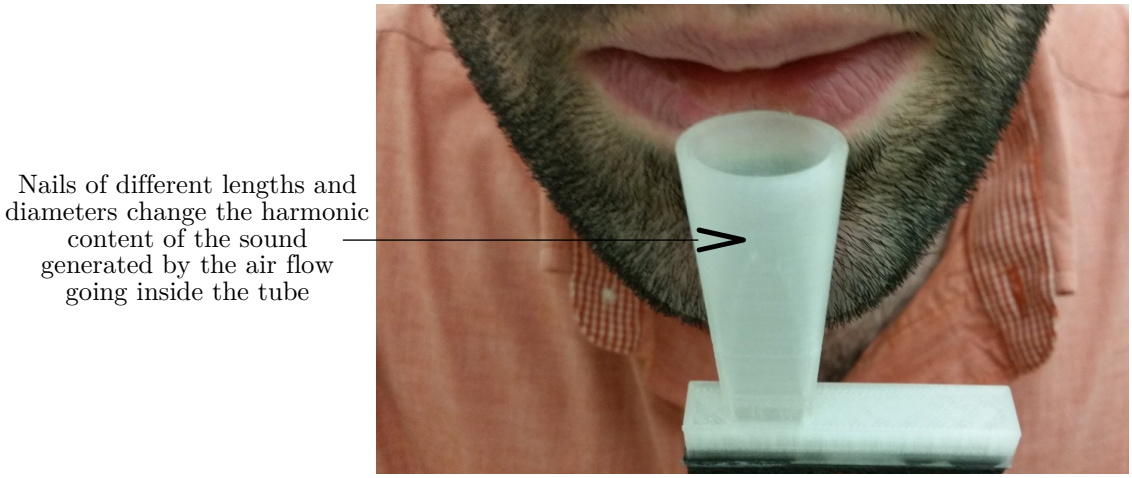


Figure 4.5: Frequency-Based Blow Sensor for Mobile Device Built-In Microphone.

4.2.2 Speaker

Even though their quality and power has significantly increased during the last decade, mobile device built-in speakers are generally only good for speech, not music. This is mostly due to their small size and the lack of a proper resonance chamber to boost bass, resulting in a very curvy frequency response and a lack of power.

There exists a wide range of passive amplifiers on the market to boost the sound generated by the built-in speakers of mobile devices, also attempting to flatten their frequency response (see §1.4.1). These passive amplifiers can be seen as resonators driven by the speaker. In this section, we present various kinds of resonators that can be connected to the built-in speaker of mobile devices to amplify and/or modify their sound.

Passive Amplifiers and Resonators

MOBILE3D contains multiple passive amplifiers of various kinds that can be used to boost the loudness of the built-in speaker of mobile devices (e.g., see Figure 4.3). Some of them were designed to maximize their effect on the generated sound [59]. Their shape can vary greatly and will usually be determined by the type of the instrument. For example, if the instrument requires the performer to make fast movements, a small passive amplifier will be preferred to a large one, etc. Similarly, the orientation of the output of the amplifier will often be determined by the way the performer holds the instrument, etc. These are design decisions that are left up to the instrument designer.

3D printed musical instrument resonators (e.g., guitar body, etc.) can be seen as a special case of passive amplifiers. MOBILE3D contains a few examples of such resonators that can be driven by the device's built-in speakers. While they don't offer any significant advantage over "standard" passive amplifiers like the one presented in the previous paragraph, they are aesthetically interesting and perfectly translate the idea of hybrid mobile instrument developed in §6.

Dynamic Resonators

Another way to use the signal generated by the built-in speakers of mobile devices is to modify it using dynamic resonators. For example, in the instrument presented in Figure 4.6, the performer's hand can filter the generated sound to create a *wah* effect. This can be very expressive, especially if the signal has a dense spectral content. This instrument is featured in the teaser video [4] of the workshop presented in §4.4.

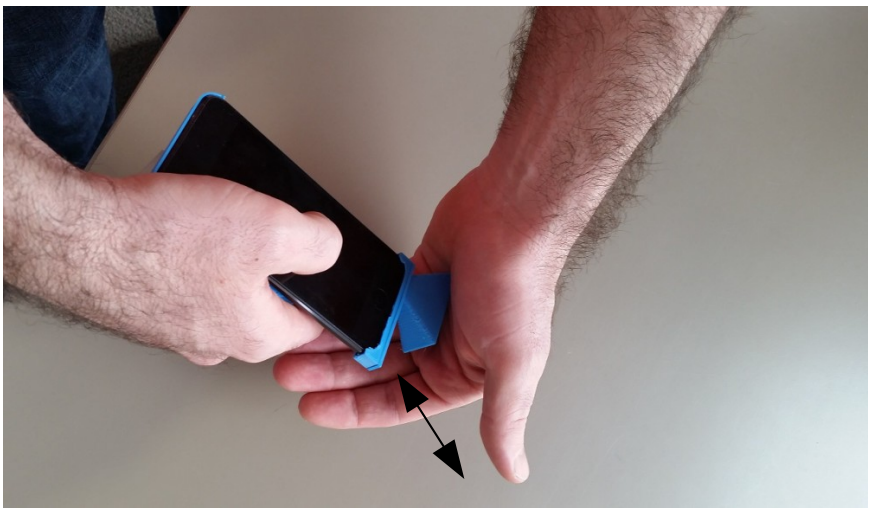


Figure 4.6: Hand Resonator for Mobile Device Built-In Speaker.

Similarly, the sound generated by the built-in speaker is sent to the mouth of the performer in

the instrument presented in Figure 4.7. The sound is therefore both modulated acoustically and through the embedded synthesis and touch-screen. The same result can obviously be achieved by directly applying the mouth of the performer to the speaker, but the augmentation presented in Figure 4.7 increases the effect of the oral cavity on the sound through a passive wave guide.

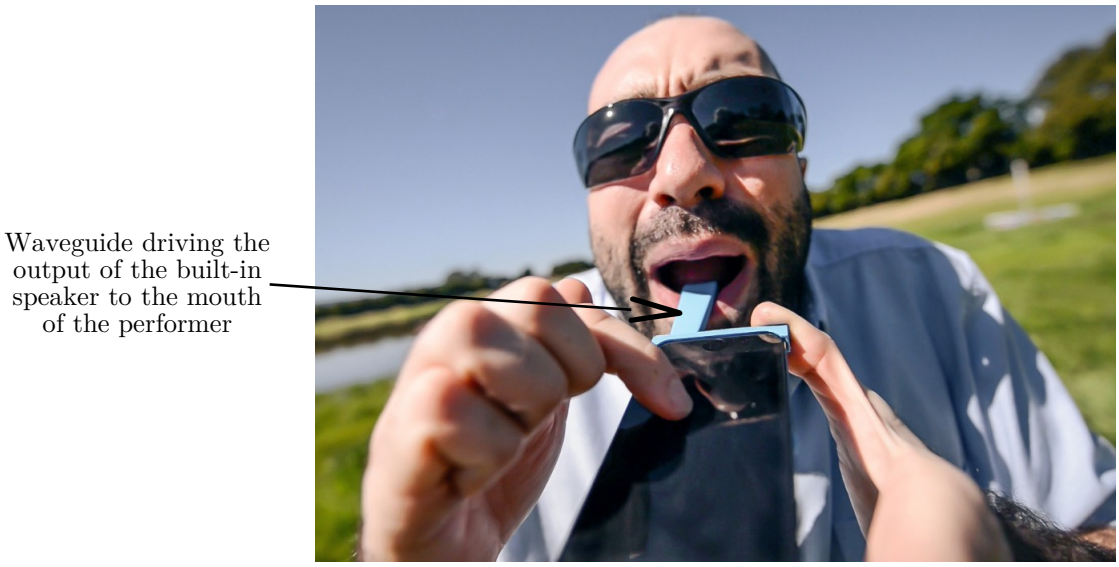


Figure 4.7: Mouth Resonator for Mobile Device Built-In Speaker.

4.2.3 Motion Sensors

Most mobile devices have at least one kind of built-in motion sensor (e.g., accelerometer, gyroscope, etc.). They are perfect to continuously control the parameters of sound synthesizer and have been used as such since the beginning of mobile music (see §1.3.2).

Augmentations can be made to mobile devices to direct and optimize the use of this type of sensor. This kind of augmentation can be classified in two main categories:

- augmentations to create specific kinds of movements (spin, swing, shake, etc.),
- augmentations related to how the device is held.

Figure 4.8 presents a "sound toy" where a mobile device can be spun like a top. This creates a slight "Leslie effect", increased by the passive amplifier. Additionally, the accelerometer and gyroscope data are used to control the synthesizer running on the device. This instrument is featured in the teaser video [4] of the workshop presented in §4.4.

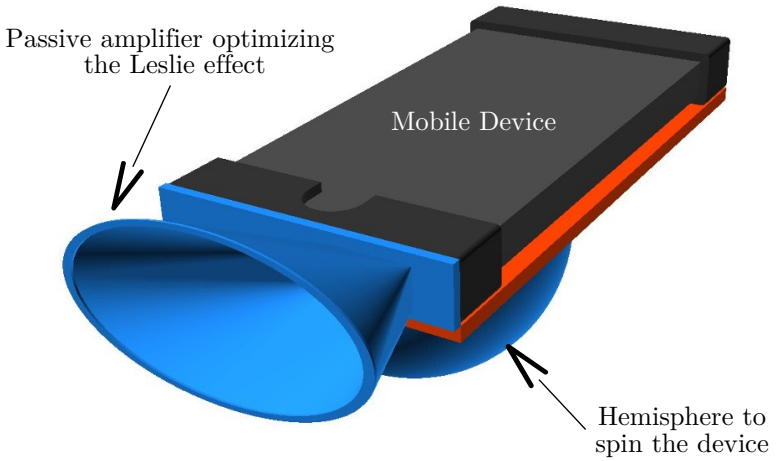


Figure 4.8: Mobile-Device-Based Top Creating a "Leslie" Effect When Spun.

Another example of motion-sensor-based augmentation is presented in Figure 4.13 and described with more details in §4.4. It features a smart-phone mounted on a bike wheel [3] where, once again, the gyroscope and accelerometer data are used to control the parameters of a synthesizer running on the device. Similarly, a "rolling smart-phone" is presented in Figure 4.12 and described in §4.4. Mobile3D contains a series of templates and functions to make this kind of augmentation.

Augmentations leveraging built-in sensors related to how the device is held are presented in more detail in §4.3.

4.2.4 Other Sensors

Most mobile devices host built-in sensors that exceed the ones presented in the previous sections and are not supported yet in Mobile3D. For example, built-in cameras can be used as very versatile sensors, [132] and a wide range of passive augmentations could be applied to them to "customize" their use for musical ends. We plan to support more sensors in Mobile3D in the future.

4.3 Holding Mobile Devices

Mobile devices were designed to be held in a specific way, mostly so that they can be used conveniently both as a phone and to use the touch-screen (see §1.3.2). Passive augmentations can be designed to hold mobile devices in different ways to help carry out specific musical gestures, better leveraging the potential of the touch-screen and of built-in sensors.

More generally, this type of augmentation is targeted towards making mobile-device-based musical instruments more engaging and easier to play.

In this section, we give a brief overview of the different types of augmentations that can be made with MOBILE3D to hold mobile devices in different ways.

4.3.1 Wind Instrument Paradigm

One of the first attempts to hold a smart-phone as a wind instrument was Smule's Ocarina (see §1.3.2), where the screen interface was designed to be similar to a traditional ocarina. The idea of holding a smart-phone as such is quite appealing since all fingers (beside the thumbs) of both hands perfectly fit on the screen (thumbs can be placed on the other side of the device to hold it). However, this position is impractical since at least one finger has to be on the screen in order to hold the device securely. The simple augmentation presented in Figure 4.9 solves this problem by adding "handles" on both sides of the device so that it can be held using the palm of the two hands, leaving all fingers (including the thumbs) free to carry out any action. Several functions and templates are available in MOBILE3D to design these types of augmentations.

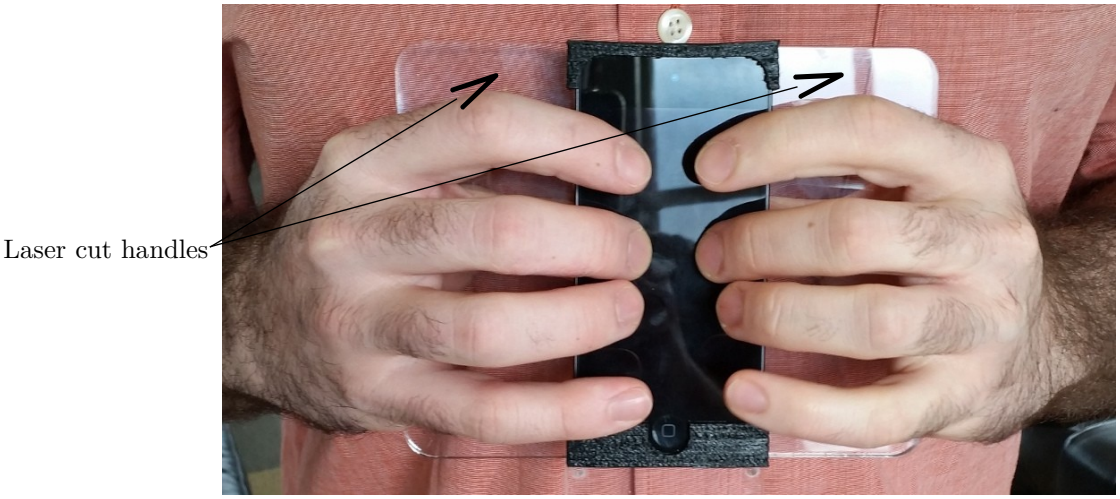


Figure 4.9: Smart-Phone Augmented to be Held as a Wind Instrument.

4.3.2 Holding the Device With One Hand

MOBILE3D contains several functions and templates to hold mobile devices with one hand, leaving at least four fingers available to perform on the touch-screen. This way to hold the device opens up a wide range of options to fully take advantage of the built-in motion sensors and easily execute

free movements. Additionally, the performer can decide to use two devices in this case (one for each hand).

The instrument presented in Figure 4.10 uses one of MOBILE3D's ring holders to hold the device with only the thumb. Similarly, Figure 4.11 features an instrument held in one hand using a laser-cut plastic handle mounted on the device.

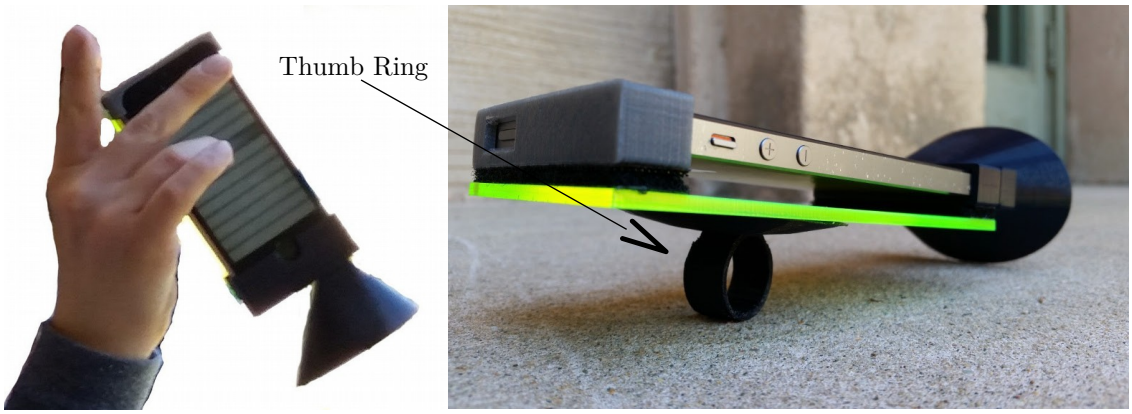


Figure 4.10: Thumb-Held Mobile-Device-Based Musical Instrument (by Erin Meadows).

4.3.3 Other Holding Options

There are obviously many other options to hold mobile-devices to carry out specific musical gestures. For example, one might hold the device in one hand and perform it with the other, etc. In any case, we believe that MOBILE3D provides enough options to cover the design needs for most musical instruments.

4.4 More Examples and Evaluation

MOBILE3D and its corresponding framework were evaluated through a series of two workshops during which participants learned how to make basic musical smart-phone apps using `faust2smartkeyb` (see §3.3) and how to use MOBILE3D to design mobile device augmentations. They were free to make any musical instrument or sound toy for their final project.

The first workshop (*The Composed Instrument Workshop: Intersections of 3D Printing and Digital Audio for Mobile Platforms*) happened during the summer of 2016 at CCRMA [3, 2]. Some of the instruments made by it participants can be seen in Figures 4.3, 4.12, 4.13, and 4.14.

The second workshop (*Augmented Smartphone Workshop*) happened in March 2017 at Aalborg University in Copenhagen (Denmark). Some of the instruments made by its participants can be

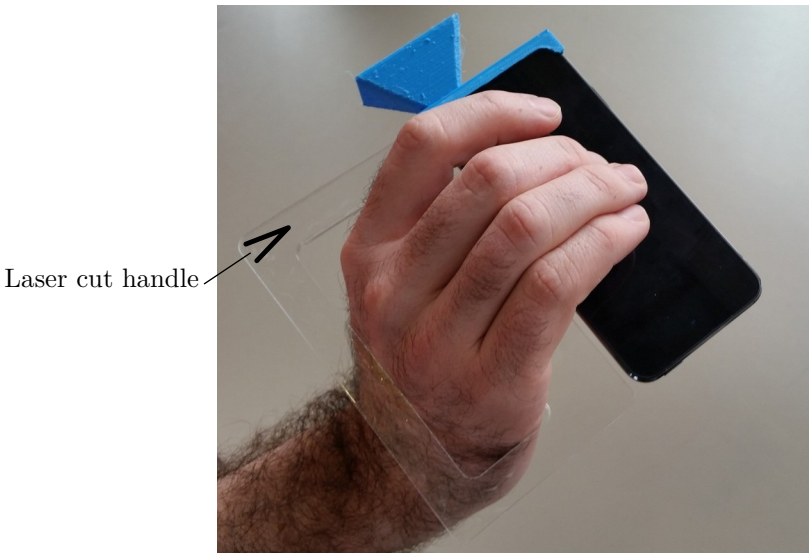


Figure 4.11: Single-Hand-Held Musical Instrument Based Using a Laser-Cut Plastic Handle.

seen in Figures 4.15 and 4.16.

In only one week, participants mastered all these techniques and designed and implemented very original instrument ideas. This helped us debug and improve MOBILE3D with new objects and features.

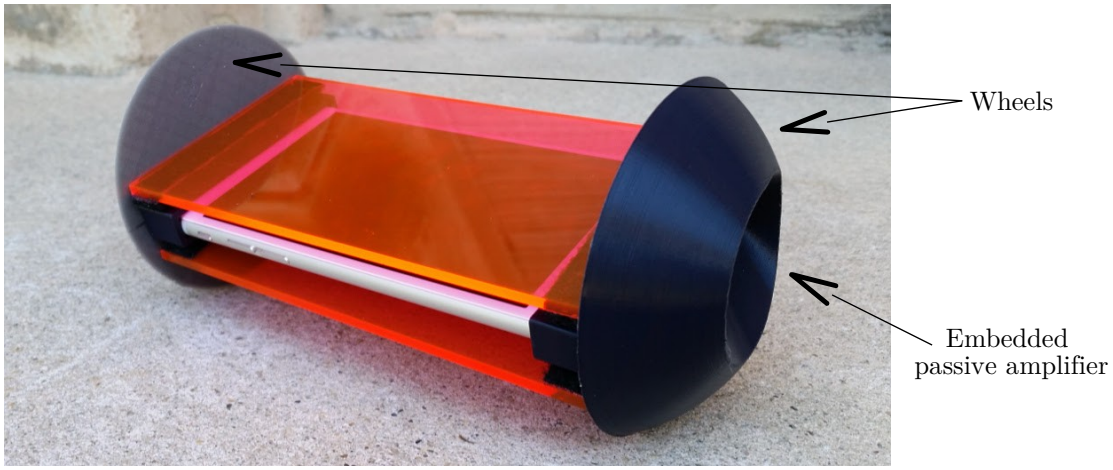
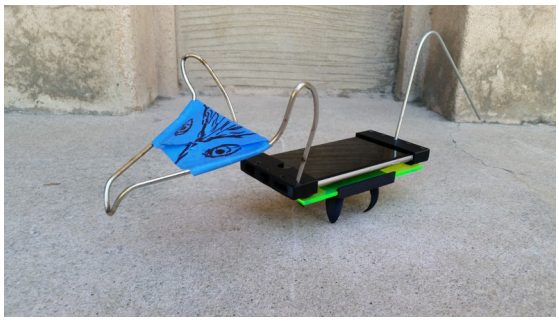


Figure 4.12: Rolling Mobile Phone With Phasing Effect (Instrument by Revital Hollander).

Figure 4.13: Mobile Device Mounted on a Bike Wheel (Instrument by Patricia Robinson).



Instrument by Mark Hertensteiner

Instrument by Noa Hollander

Instrument by Chuck Cooper

The Crew

Figure 4.14: Other Instruments from the *2016 Composed Instrument Workshop*.

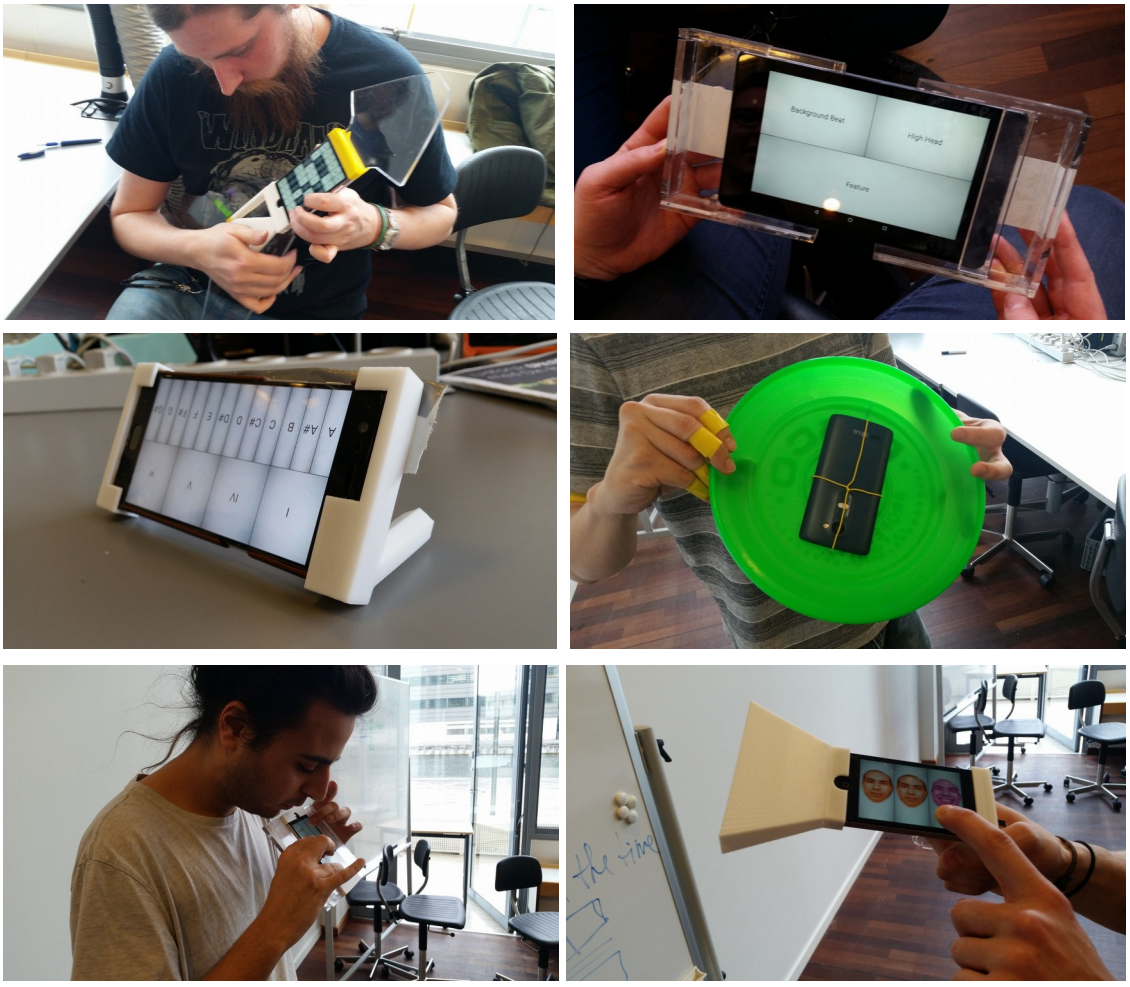


Figure 4.15: Instruments From the *2017 Copenhagen Augmented Smart-Phone Workshop* (1).

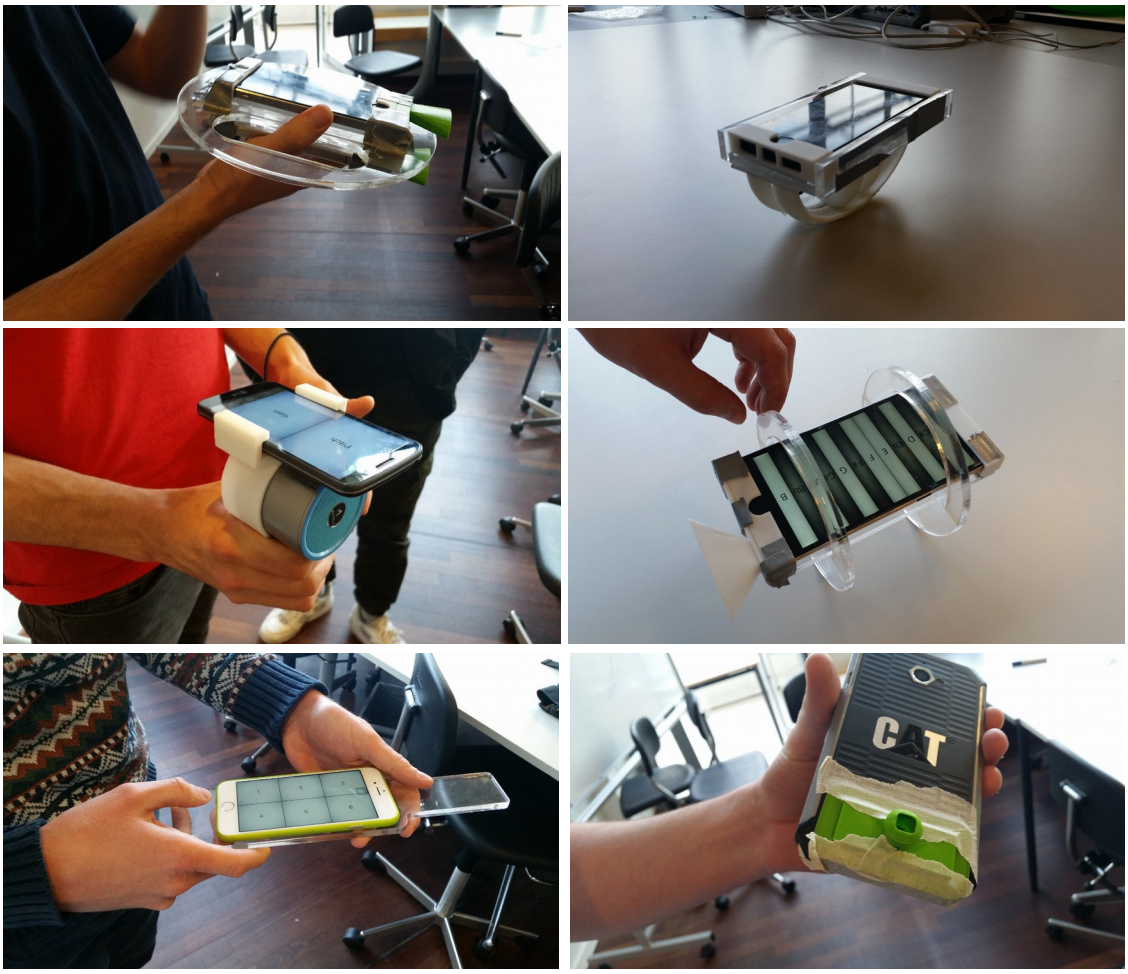


Figure 4.16: Instruments From the *2017 Copenhagen Augmented Smart-Phone Workshop* (2).

Chapter 5

Actively Augmenting Mobile Devices With Sensors

> "It's easy to play any musical instrument: all you have to do is touch the right key at the right time and the instrument will play itself." (Johann Sebastian Bach)

Some of the limitations of mobile devices when used as musical instruments can be overstepped by "augmenting" them with passive prosthetics (see §4). By being relatively non-invasive and lightweight, they contribute to the overall coherence/unity of the instrument. However, their simplicity can sometimes be a limitation as they remain tied to what built-in active elements of the device (e.g., touchscreen, microphone, speaker, etc.) can offer. Inversely, active sensor augmentations can take any form and can be used to implement almost anything that mobile devices don't have. While their level of complexity can be more or less infinite, we encourage an incremental approach where instrument designers should first take advantage of elements already available on mobile devices, and then use active sensor augmentations parsimoniously to implement what they could not have done otherwise.

In this section, we provide a framework and a method to make active sensor augmentations for mobile devices. The special case of active acoustic augmentations is treated in §6.

Unlike passive augmentations, the scope of active sensor augmentations is huge and any musical controller could probably fit in this category. Therefore, we will only consider the tools to carry out this task and leave design or aesthetic considerations up to digital luthiers.

First, we present NUANCE, an iPad-based instrument where a set of sensors are used to add force sensitivity to the touchscreen of the device. It can be seen as a first step towards the framework presented in the following sections. Next, we introduce different strategies to transmit sensor data to mobile devices. We then use the conclusions from this study to build our active sensors augmentation framework. Finally, we present a set of examples of musical instruments using active sensor

augmentations and that were designed in the frame of the *2017 Mobile Synth CCRMA Summer Workshop*.[1]

5.1 NUANCE: Adding Force Detection to the iPad

Before generalizing the concept of active sensor augmentation for mobile devices, and in continuity with the BLADEAXE2 presented in §2.3, we worked on NUANCE: an instrument sending sensors data to an iPad. NUANCE was designed before `faust2smartkeyb` (see §3.3), and its software implementation presented in the corresponding SMC-16 paper [126][2] slightly differs from the one introduced in this section. NUANCE served as a prototype for some of the work on transmitting sensors data to mobile devices presented in §5.2 which is why it is provided as an introduction to the current chapter.

The lack of force sensitivity on touchscreens is a known issue in the world of mobile music (see §1.3.2). It narrows the range of possible interactions, and makes performance with specific classes of instruments such as percussion and plucked string instruments less intuitive. For example, striking force must be substituted by some other dimension such as the y coordinate in a strike area.

Park et al. addressed this issue a few years ago and proposed a solution using the built-in accelerometer of the device and a foam padding [146]. While this solution is very self-contained as it uses only the built-in sensors of the device, it presents several limitations diminishing the range of practical applications (e.g., no multi-touch support, sensitivity to table/support vibrations, no automatic re-calibration, limited sampling rate, etc.).

Some of the most recent generations of devices such as the iPhone 6 provide basic multi-touch force detection on the screen ("3D Touch").[3] This feature has already been exploited by some companies such as ROLI with its Noise app (see §1.3.2) to create expressive musical instruments. Unfortunately, this technology is not yet available on larger screen devices (tablets, etc.) that provide a better interface to control certain type of instruments such as percussion [153]. Instead, tablet manufacturers currently favor the use of force sensitive pencils[4] [5] that provide a simpler solution to this problem. Also, the "3D Touch" technology of the iPhone 6 has some limitations. While it can provide very accurate data in the case of a continuous touch event ("after-touch"), its usability for deriving the *velocity* of a strike gesture on the screen is very limited. This makes it practically unusable to control percussion or plucked string instruments where the attack is very sharp.

In this section, we introduce NUANCE: a device adding high quality multi-touch low-latency force detection to the iPad touch screen, fast and accurate enough to be suitable for deriving striking velocity. NUANCE is based on four force sensitive sensors placed on each corner at the back of

[1] `https://ccrma.stanford.edu/~rmichon/mobileSynth/`
[2] Some sections and figures of this paper were copied verbartim here.
[3] `http://www.apple.com/iphone-6s/3d-touch/`
[4] `http://www.apple.com/apple-pencil/`
[5] `https://www.microsoft.com/surface/en-us/accessories/pen/`

the device. It communicates with the iPad using its *audio jack* connector through a purely analog system streaming the sensor data as an audio signal. This ensures a fast data rate (up to the audio bandwidth of the iPad, nominally 20 kHz) as well as a high sample resolution (bit depth).

After describing the hardware implementation of NUANCE, we demonstrate how it can be used to design musical instruments. We then provide a series of examples, evaluate its performance and discuss future applications and improvements.

5.1.1 Hardware

The case of NUANCE is made out of wood (plywood and birch) and black laser cut acrylic (see Figure 5.1). The current version was designed for the *iPad Air 2*[6] but it is also compatible with the *9.7 in iPad Pro.* [7]

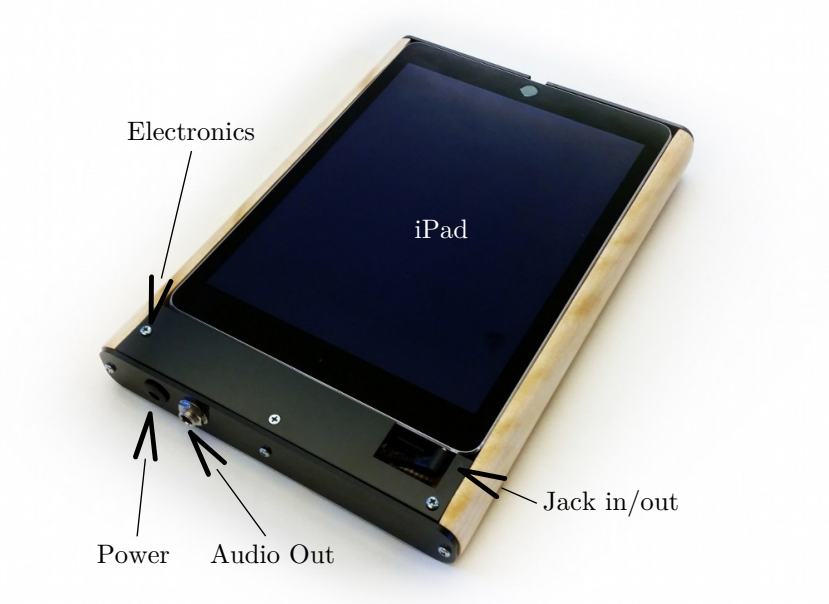


Figure 5.1: Global View of NUANCE.

An FSR (Force Sensitive Resistors)[8] is placed under each corner of the iPad (see Figure 5.2) [73]. The FSRs are covered with a thin (1/4 in) piece of foam whose rigidity was chosen to offer a good compromise between responsiveness and damping [153]. The foam is used to cushion the strikes of the performer and also to give some slack to the iPad during continuous push gestures (after-touch).

[6] http://www.apple.com/ipad-air-2/
[7] http://www.apple.com/ipad-pro/
[8] The FSRs used for the device are *Interlink Electronics FSR 400*: http://www.interlinkelectronics.com/FSR400.php

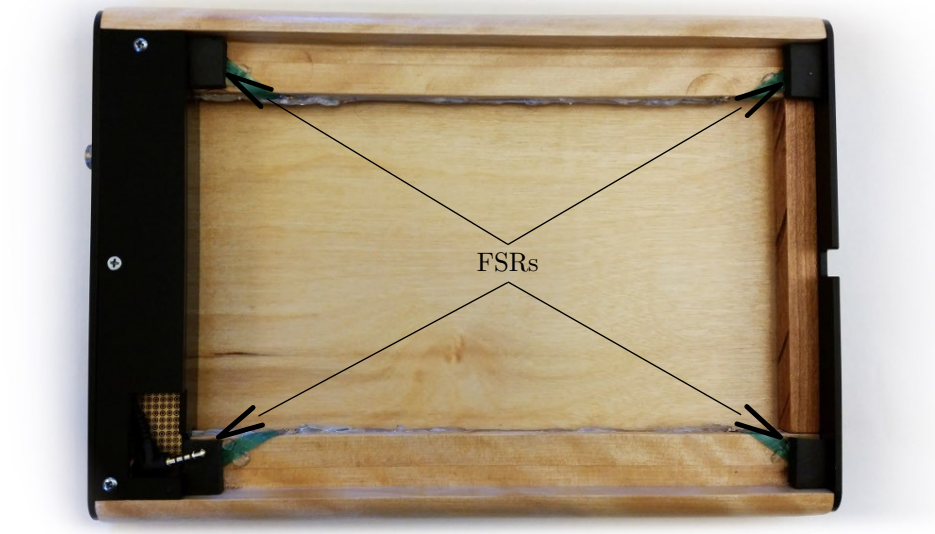


Figure 5.2: Top View of NUANCE Without the iPad.

The signals from the different FSRs is sent to the iPad using amplitude modulation (AM). The use of this technique to send sensors data to mobile devices is further investigated in §5.2. Each force signal controls the gain of its own analog oscillator. The oscillators are very simple based on a 555 timer (see Figure 5.3). This kind of circuit doesn't generate a pure sine wave but it is straightforward to efficiently isolate each carrier wave during the demodulation process described in §5.1.2 (reducing the dynamic range of the signal is not a problem since it is pretty large, thanks to the audio ADC).

The frequency of each oscillator is different and is controlled by $R1$ and $C1$. Frequencies (2, 6, 10 and 14 kHz) are spread across the bandwidth of the line input of the iPad (assuming that the sampling rate of the target app is at least 44.1 kHz). Since we're not carrying an audio signal, and since the sharpest attack we could achieve by tapping the screen was longer than 10ms (corresponding to a bandwidth less than 100Hz), we don't have to worry about sidebands. Moreover, the demodulation technique used on the iPad (see §5.1.2) significantly reduces the risk of sidebands contaminating neighboring signals.

The FSRs were calibrated to output their maximum value when a weight of approximately 400 grams is applied on the touchscreen. This was set empirically to provide the best feeling to our taste but this value (that should remain reasonable to not damage the screen) can be easily adjusted by a small potentiometer mounted on the circuit board. Since the iPad itself applies some weight to the FSRs, the minimum value of the range is constantly adjusted on the software side (see §5.1.2).

The output of the different oscillators is mixed and sent to the iPad using the line input pin of the *audio jack* input.

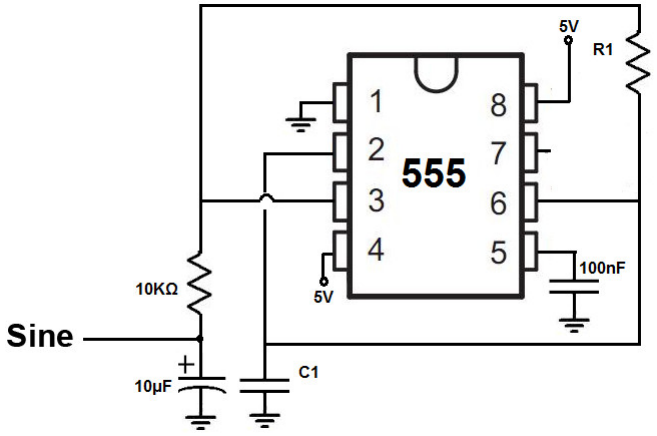


Figure 5.3: Circuit Diagram of One of the Simple Sine Oscillators Used in NUANCE.

The system is powered with an external 5 V power supply and connects to the iPad using a 3.5 mm (1/8 inch) headset (four-pole) audio jack. The output signal is routed to a 1/4 inch stereo audio jack mounted on the side of NUANCE.

The circuits were chosen to be as simple as possible to reduce the cost of NUANCE to less than $30. Variations of the sine oscillators (stability of the frequency, purity of the sine wave, etc.) are easily compensated on the software side.

5.1.2 Software

The amount of force applied to each FSR of NUANCE is carried by different sine waves to the iPad using its single audio analog input. Four band-pass filters isolate the signal of each sine wave (see Figure 5.4). Their bandwidth is big enough to accommodate the variations of the frequencies of the simple sine oscillators described in §5.1.1. The amplitude of the output signal of each filter is extracted and corresponds to the force measured by each FSR of NUANCE.

The software implementation of the different apps compatible with NUANCE that are presented in §5.1.3 is implemented using `faust2smartkeyb` (see §3.3). The force information extraction system described in the previous paragraph is implemented as a single FAUST function that is executed in the same audio callback function as the synthesizer that it is controlling. In other words, the force data from the FSRs are acquired and are controlling the synthesizer at the audio rate.

Touch events (including the (x, y) position on the screen) are compared with the data provided by the FSRs to associate a force signal to a specific touch event on the screen (see Figure 5.4). If the touch event was just initiated, the force is converted into a velocity proportional to the instantaneous

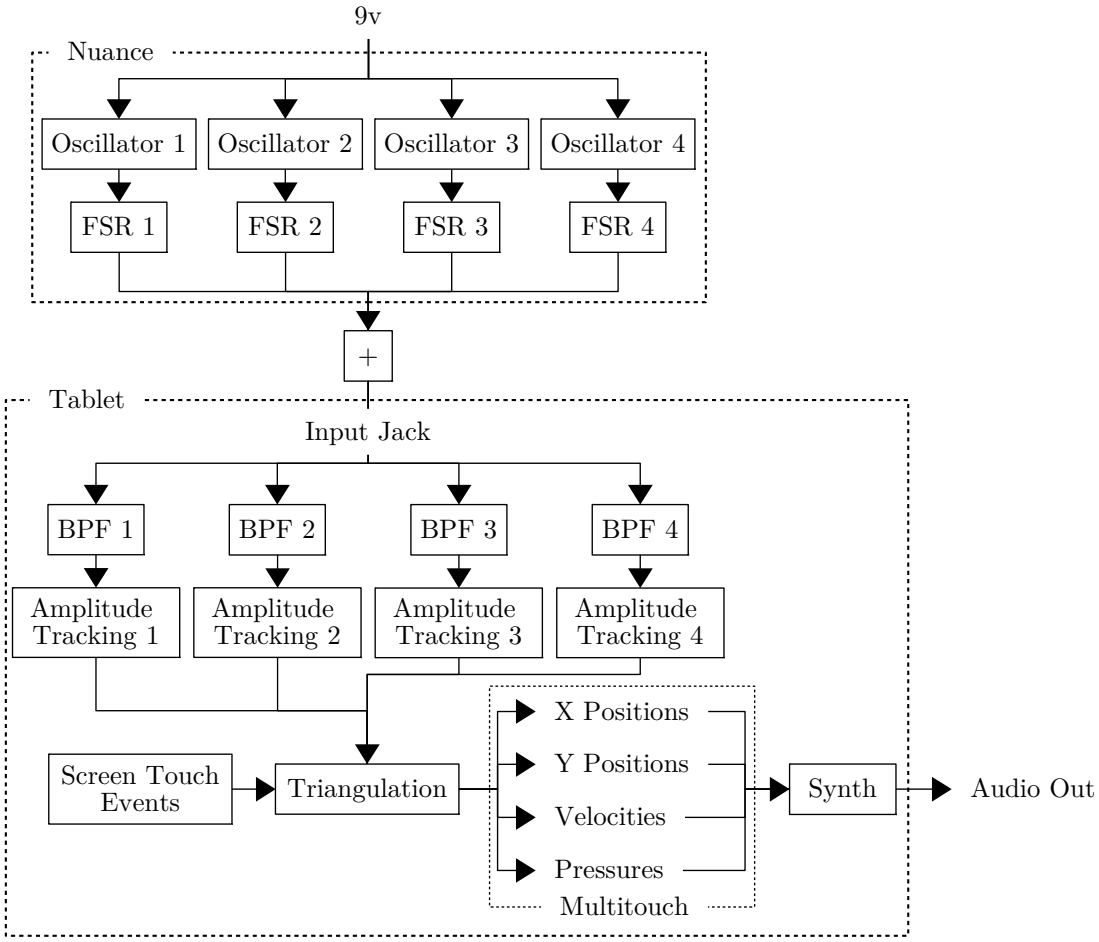


Figure 5.4: Overview of NUANCE.

force at the beginning of the touch. If the touch event persists, then the force is converted into a series of after-touch events.

As mentioned in §5.1.1, the iPad itself applies a certain amount of weight to the FSRs. Depending on its position in the case or the level of inclination of the table where NUANCE is installed, this value can vary a little bit. To make sure that the range of the FSRs is always accurate, it is readjusted when there are no fingers touching the screen. If there are several simultaneous active touches (multi-touch) on the screen, a simple triangulation algorithm compares the force level at each FSR with the (x, y) position of the touch on the screen to associate a velocity or an after-touch event to it. Obviously, the more simultaneous touches on the screen, the harder it becomes for the system to differentiate independent forces. We find that the system is very accurate in the case of two simultaneous touches, but the force distribution tends to become more uniform if more touches are engaged. However, we find that this is not an issue for many types of percussion and plucked-string instrument control.

5.1.3 Examples

While it would be quite easy to write an app to use NUANCE as a MIDI controller, we liked the idea of creating standalone musical instruments taking full advantage of the possibilities offered by this system (see §1.2). For this reason, the different apps that we created and that are compatible with NUANCE target specific instruments.

While NUANCE would work well with a wide range of screen interfaces (piano keyboards, isomorphic keyboards, etc.), we mostly focused on percussion instruments so far. The different apps that we created implement one or several drums represented by rectangular regions on the touch screen. The app presented in Figure 5.5 has three different zones, each controlling a different drum. Each offers a physical representation of the virtual instrument (striking on the edges sounds different than striking at the middle, etc.). The drum synthesizers are all based on modal physical models [5] which strengthens the link between the physical and virtual parts of the instrument, increasing its overall coherence. The physical models were implemented using the FAUST Physical Modeling Toolkit (see §6).

The multi-touch capabilities of NUANCE allow the performer to simultaneously strike two drums with different velocities. The after-touch information can also be used to interact with the resonance time (T60) of the virtual drums. This results in a highly expressive instrument.[9]

5.1.4 Evaluation/Discussion

The "most standard" way to connect an external music controller to the iPad is by using MIDI through the lightening connector. While this solution works well for most basic applications (e.g., a

[9] `https://ccrma.stanford.edu/~rmichon/nuance/` presents a series of demo videos of NUANCE.

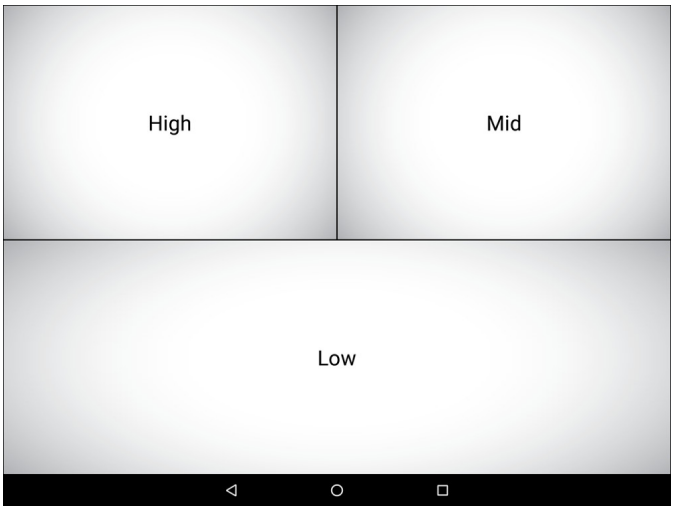


Figure 5.5: Screenshot of One of the Percussion Apps Made With `faust2smartkeyb` and Compatible With NUANCE.

MIDI keyboard triggering events in a synthesizer), the limited bandwidth and bit depth, as well as the jitter in the latency of MIDI, can be problematic for applications requiring a high rate of data and precise synchronization (see §5.2) [104].

The idea of using the line-input of the *audio jack* plug of mobile devices to send data to it has already been exploited a lot. Various commercial products such as credit-card readers, like Square, use this technique, but this idea has also been used by the DIY community to send sensor data. For example, [182] is a simple modem that uses the *audio jack* connector of *Android* and *iOS* devices to transmit digital data. Its main limitation is its very small bandwidth (30 bytes/sec). [193] uses a different paradigm where the analog signals are multiplexed and sent one after the other.

Our approach described in the three previous sections is less versatile and more "low-tech" but it allows for the streaming of the signals from four sensors to the iPad using the audio bandwidth. The consistency and rate of the information remain constant, greatly simplifying its synchronization with the sound synthesizer. It is also much cheaper and easier to build from scratch.

Its main disadvantage is that the demodulation technique (see §5.1.2) that it uses on the mobile device to retrieve the sensor data is rather computationally expensive. However the extended power of modern mobile devices compensates for this, and running the various band-pass filters, amplitude trackers, and the triangulation function presented in Figure 5.4 only takes 4% of the resources of the CPU on an iPad Air 2.

A more general limitation of our system is the fact that force can't be accurately measured on more than two independent simultaneous touches on the screen. We have not found this to be an issue in musical-performance applications to date. Perhaps only a built-in technology, such as 3D

Touch, or a force sensitive glove can efficiently resolve this issue. The fact that the most recent versions of iPhones address this problem leads us to think that larger devices such as the iPad will follow this trend too in future models. As mentioned in the introduction, it seems that tablet manufacturers prefer to settle for a force-sensitive pencil for now, which does not provide multi-touch force. Also, while the technology used by Apple iPhone 6s is fully multi-touch, it is not as fast as the method presented here, and it can't be used for example to accurately detect the velocity of striking gestures.

5.2 Transmitting Sensor Data to Mobile Devices

Despite the wide range of options and technologies available, transmitting sensor data to mobile devices to control real-time sound synthesis can be a hard task. Finding a good balance between the complexity of the system, good latency, portability, and universality can be hard. This section gives a brief overview of selected techniques to send sensor data to mobile devices. They are evaluated in the context of mobile device augmentation in order to provide the basis for our framework presented in §5.3.

5.2.1 Digital Transmission

Hardware

Sending sensor data digitally to a mobile device will likely imply the use of a microcontroller. This type of electronic component has been spread and generalized by the Arduino,[10] that greatly simplifies their use thanks to various high level hardware and software features (e.g., IDE[11], high level programming language, USB support, etc.).

Micro-computers from the "Arduino family" are usually powered through their USB port. While many types of low-energy sensors can be powered directly by the microcontroller itself, sensors requiring more power will generally require an external power supply. Therefore, lightweight augmentations based on a microcontroller and a few sensors can be connected to mobile devices, that are also used to provide power through their built-in battery. A wide range of simple standalone musical instruments can be created using this approach.

Wired Transmission

There exists dozens of models of mobile devices made by different manufacturers. While they all offer the possibility to connect USB devices to them, the format of the connectors used for this purpose greatly varies between smart-phones/tablets, which tends to complexify the design of "universal"

[10] `https://www.arduino.cc/`
[11] Integrated Development Environment

guest devices/active sensor augmentations. Additionally, some operating systems such as iOS have strict hardware and software requirements for external devices. For instance, the only standard openly recognized and allowed by Apple for transmitting real-time data through USB is MIDI (see Figure 5.6). While MIDI is well-established, extremely robust, and has been used for decades as a protocol for transmitting musical data in real-time, it has some well-known limitations (e.g., sampling rate, resolution/bit depth, etc.) that can be problematic in some cases.

Other proprietary/custom standards can be used, but external devices using them must be approved by Apple, making their development out of reach for the DIY community.

To summarize, for a custom external device to work with most smart-phones and tablets, it will have to use the MIDI standard and have replaceable USB-compatible connectors (e.g., lightening, Micro-USB, etc.).

Turning microcontrollers into MIDI devices has been greatly simplified by the Teensy Board,[12] which allows for the easily replacement of its USB serial driver by a USB MIDI one directly from its programming interface (Teensyduino[13]). Previous techniques consisted of replacing the driver "by hand" using a DFU programmer [51].

Wireless Transmission

An alternative, more generic solution to digitally transmit sensor data to mobile devices is to use a wireless connection between the microcontroller and the mobile device (see Figure 5.6). The two main technologies usable for this purpose and available on most mobile devices are WIFI and Bluetooth. WIFI allows for the use of the OSC (Open Sound Control) standard, [204] which is widely spread and is better than MIDI in many respects. On the other hand, MIDI can be transmitted over bluetooth with relatively low latency and can be seen as "the standard way" to implement external musical controllers on mobile devices.

However, both solutions greatly complexify the overall design of active sensor augmentations. Indeed, bluetooth and WIFI transmitters can be expensive, take additional space, and are very energy consuming, implying the use of powerful batteries also taking space, potentially impacting the overall coherence of the instrument (see §1.1). To us, these disadvantages exceed the benefits of using such technologies in the frame of this work, which is why we opted for a wired solution in §5.3.

5.2.2 Analog Transmission

Analog transmission of sensors data can offer several advantages over digital transmission. Cost is one of them since simple analog circuits can be designed to carry out this type of task, preventing the use of a microcontroller which is an expensive component. For example, NUANCE (see §5.1) uses

[12] https://www.pjrc.com/teensy/
[13] https://www.pjrc.com/teensy/teensyduino.html

a series of cheap 555 timers to carry the signal of four force sensitive resistors (FSR) to an iPad (see 5.1.4).

When transmitted as an audio signal, sensor data can be sent directly through the microphone input of the headphone jack present on most mobile devices (see Figure 5.6). Unlike digital USB inputs, their format doesn't vary between devices providing a generic way to connect to them.

Several techniques to "encode" sensor data as audio signals such as amplitude and frequency modulation [116, 201, 135] or multiplexing [193] have been used in the context of mobile devices. They all provide a relatively generic way to transmit data at a higher rate than MIDI and with a better resolution.

Other analog transmission techniques consist of "hacking" some of the built-in sensors of the mobile device such as the magnetometer. The MagGetz project [79] makes a good use of this technique.

Finally, wireless analog transmission techniques could probably be used as well but they don't present any significant advantage over wired connections in the context of mobile device active sensor augmentations.

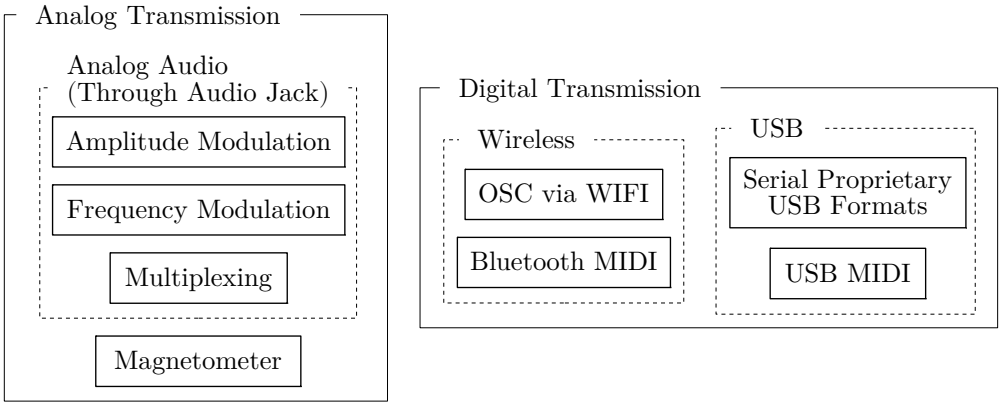


Figure 5.6: Selected Real-Time Sensor Data Transmission Techniques for Active Sensor Mobile Device Augmentations.

5.3 Active Sensors Augmentation Framework

In our view, mobile device augmentations should supplement existing built-in sensors (e.g., touchscreen, motion sensors, etc.) and remain as lightweight and confined as possible. Indeed, there's often not much to add to a mobile device to turn it into a truly expressive musical instrument. NUANCE is a good example of that since it adds a whole new level of expressivity to the touchscreen, simply by using a few sensors. On the other hand, unlike passive augmentations, active sensor

augmentations can be used to add an infinite number of features.

In this section, we introduce a framework for designing active sensor mobile device augmentations supplementing sensors already available on the device. This allows us to keep our augmentations lightweight and powered by the device, preserving the standalone aspect and partly the coherence of the instrument.

To keep our augmentations simple, we propose to use a wired solution for transmitting sensor data to the mobile device, which also allows for the augmentation to be powered. Augmentations requiring an external power supply (e.g., battery) are discarded and are not considered in the frame of this work.

MIDI is a standard universal way to transmit real-time musical (and non-musical) control data to mobile devices, so we opted for this solution. Teensys such the Teensy 3.2[14] are micro-controllers providing built-in USB MIDI support, making them particularly well suited to be used in our framework.

Teensyduino (see §5.2.1) (Teensy's IDE), comes with a high level library part of `Bounce.h` for sending MIDI over USB. The code presented in Listing 5.1 demonstrates how to use this library to send sensor values on a MIDI "Continuous Controller" (CC).

```
#include <Bounce.h>

void setup() {
}

void loop() {
  int sensorValue = analogRead(A0);
  int midiCC = 10; // must match the faust configuration
  int midiValue = sensorValue*127/1024; // value between 0-127
  int midiChannel = 0;
  usbMIDI.sendControlChange(midiCC,midiValue,midiChannel); // send!
  delay(30); // wait for 30ms
}
```

Listing 5.1: Simple Teensy Code Sending Sensor Data in MIDI Format Over USB.

Once uploaded to the microcontroller, the Teensy board can be connected via USB to any MIDI-compatible mobile device (iOS and Android) to control selected parameters of a `faust2smartkeyb` app (see §3.3). This will require the use of a USB adapter, depending on the type of USB plug available on the device. MIDI is enabled by default in `faust2smartkeyb` apps and parameters in the FAUST code can be mapped to a specific MIDI CC by using a metadata (see §3.2.1):

[14] `https://www.pjrc.com/store/teensy32.html`

```
frequency = nentry("frequency[midi:ctrl 10]",1000,20,2000,0.01);
```

Here, the `frequency` parameter will be controlled by MIDI messages coming from MIDI CC 10 and mapped to the minimum (20Hz for MIDI CC 10 = 0) and maximum (2000Hz for MIDI CC 10 = 127) values defined in the `nentry` declaration. Therefore, if this parameter was controlling the frequency of an oscillator and that the Teensy board running the code presented in Listing 5.1 was connected to the mobile device running the corresponding `faust2smartkeyb` app, the sensor connected to the `A0` pin of the Teensy would be able to control the frequency of the generated sound.

Other types of MIDI messages (e.g., `sendNoteOn()`) can be sent to a `faust2smartkeyb` app using the same technique.

Most of the parameters controlled by elements on the touchscreen or by built-in sensors of the apps presented in §3.4 could be substituted by external sensors or custom interfaces using the technique described above.

5.4 Examples and Evaluation: *CCRMA Mobile Synth Summer Workshop*

The framework presented in §5.3 was evaluated within a two weeks workshop at CCRMA at the end of June 2017. [15] It was done in continuity with the FAUST *Workshop* taught the previous years[16] and the *Composed Instrument Workshop* presented in §4.4. During the first week (*Mobile App Development for Sound Synthesis and Processing in Faust*), participants learned how to use FAUST through `faust2smartkeyb` and made a wide range of musical apps. During the second week (*3D Printing and Musical Interface Design for Smart-phone Augmentation*), they designed various passive (see §4) and active sensors augmentations using the framework presented in §5.3. They were encouraged to first use elements available on the device (e.g., built-in sensors, touchscreen, etc.) and then think about what was missing to their instrument to make it more expressive and controllable.

This section presents selected works from students of the workshop.

5.4.1 *Bouncy-Phone* by Casey Kim

Casey Kim designed *Bouncy-Phone*, an instrument where a 3D printed spring is "sandwiched" between an iPhone and an acrylic plate hosting a set of photo-resistors (see Figure 5.7). The interface on the touchscreen implements two parallel piano keyboards controlling the pitch of a monophonic synthesizer. The instrument is played by blowing onto the built-in microphone, in a similar way than Ocarina (see §1.3.2). The x axis of the accelerometer is mapped to the frequency of a lowpass

[15] `https://ccrma.stanford.edu/~rmichon/mobileSynth/`: this webpage contains more details about the different instruments presented in the following subsections.

[16] `https://ccrma.stanford.edu/~rmichon/faustWorkshops/2016/`

filter applied to the generated sound. The spring is used to better control the position of the device in space in order to finely tune the frequency of the filter. The shades created by the two hands of the performer between the phone and the acrylic plate are used to control the parameters of various audio effects.

Figure 5.7: *Bouncy-Phone* by Casey Kim.

5.4.2 *Something Else* by Edmond Howser

Edmond Howser designed *Something Else*, an instrument running a set of virtual strings based on physical models from the FAUST Physical Modeling Library (see §6.2). The touchscreen of an iPhone can be used to trigger sound excitations of different pitches. A set of three photoresistors were placed in 3D printed cavities (see Figure 5.8) that can be covered by the fingers of the performer to progressively block the light, allowing for a precise control of the parameters associated to them. These sensors were mapped to the parameters of a set of audio effects applied to the sounds generated by the string physical models. The instrument is meant to be held as a trumpet with three fingers on top of it (one per photoresistor) and fingers from the other hand on the side, on the touchscreen.

5.4.3 *Mobile Hang* by Marit Brademann

Mobile Hang is an instrument based on an iPhone designed by Marit Brademann. A 3D printed prosthetic is mounted on the back of the mobile device (see Figure 5.9). It hosts a Teensy board as well as a set of force sensitive resistors that can be used to trigger a wide range of percussion sounds based on modal physical models of the FAUST Physical Modeling Library (see §6.2) with different velocities. A large hole placed in the back of the tapping surface allows the performer to hold the instrument with the thumb of his right hand. The left hand is then free to interact with the

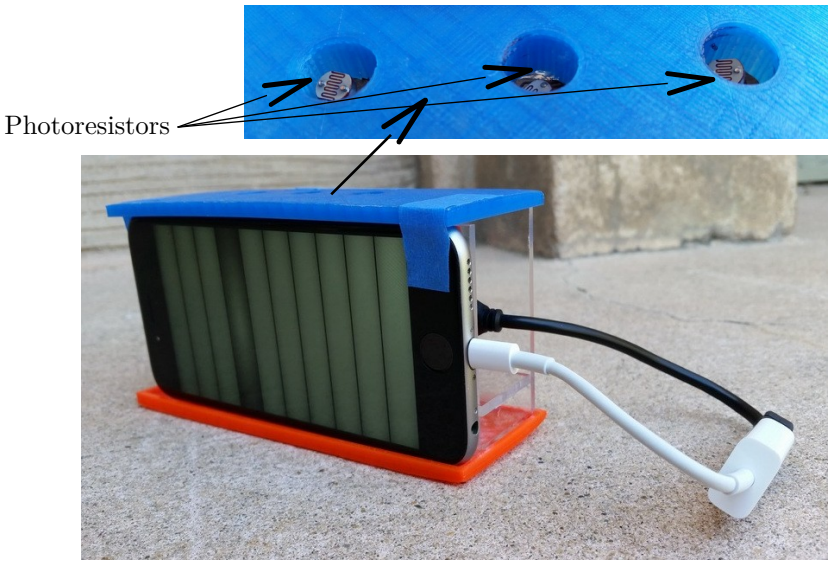


Figure 5.8: *Something Else* by Edmond Howser.

different (x, y) controllers on the touchscreen controlling the parameters of various effects applied to the generated sounds. *Mobile Hang* also takes advantage of the built-in accelerometer of the device to control additional parameters.

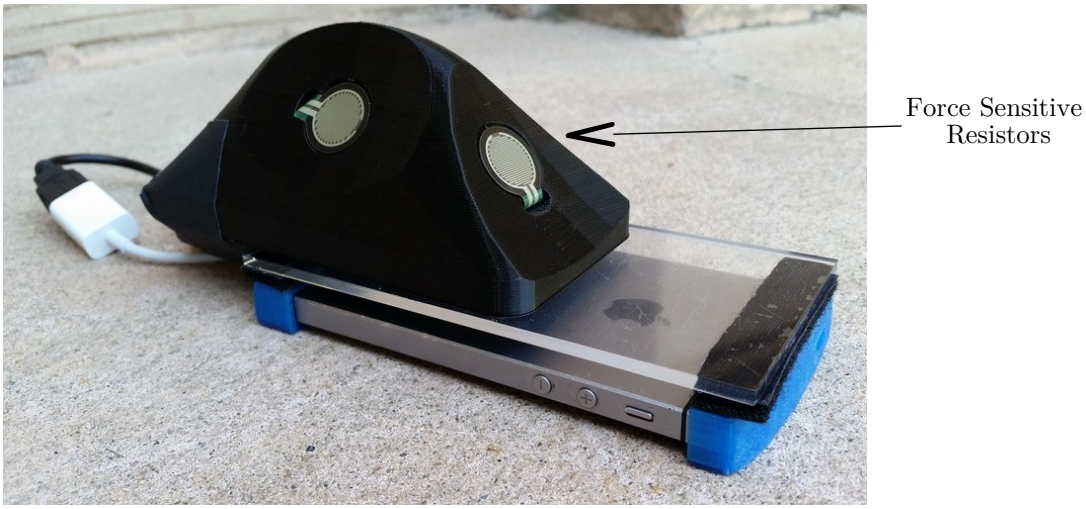


Figure 5.9: *Mobile Hang* by Marit Brademann.

Chapter 6

Developing the Hybrid Mobile Instrument

> "There are no hard distinctions between what is real and what is unreal, nor between what is true and what is false. A thing is not necessarily either true or false; it can be both true and false." (Harold Pinter)

In a world where everything tends to become virtual, musical instruments are no exception. Virtual/digital instruments present a wide range of advantages over acoustic ones (see §1.1) and are broadly used nowadays. Physical-model-based virtual instruments (see §1.5) constitute a special branch of this family by being tied to the real/physical world. Expectations for "physically informed" virtual instruments are usually greater than for other types of instruments of this family, probably because of our ability to precisely analyze and process physical phenomenons.

Current technologies allow one to blur the boundary between the physical/acoustical and the virtual/digital world. Transforming a physical object into its virtual approximation can be done easily using various techniques (see §1.5). On the other hand, recent progress in digital fabrication, with 3D printing in particular (see §1.6), allows us to materialize 3D virtual objects. Even though 3D printed acoustic instruments such as Scott Summit's guitar (see §1.6.1) don't compete yet with "traditionally made" ones, their quality keeps increasing and they remain perfectly usable.

This chapter generalizes some of the concepts introduced by instruments presented in §2.1, §2.2, and §2.3, where sound excitations made by physical objects are used to drive physical-model-based virtual elements. It allows instrument designers to arbitrarily choose the nature (physical or virtual) of the different parts of their creations.

This chapter introduces a series of tools completing the framework presented in this thesis to approach musical instrument design in a multimodal way where physical parts can be "virtualized" and vice versa. First, we give an overview of our framework to design mobile hybrid instruments.

Then we introduce the FAUST Physical Modeling Library, "the core" of our framework, that can be used to implement a wide range of physical models of musical instruments to be run on a mobile device (e.g., using `faust2smartkeyb`). Finally, `mesh2faust`, a tool to generate FAUST physical models using 3D objects is presented.

6.1 Hybrid Instrument Framework Overview

6.1.1 From Physical to Virtual

In §1.5, we gave an overview of different physical modeling techniques that can be used to make virtual versions of physical objects designed to generate sound (i.e., musical instruments). The framework presented in this chapter is a bit more limiting and focuses on specific modeling techniques that are more flexible, computationally cheap, and easy to use in the context of hybrid instrument design.

Various linearizable acoustical physical objects can be easily turned into model physical models (see §1.5.1) using their impulse response [5]. Pierre-Amaury Grumiaux et al. implemented `ir2faust` [70], a command-line tool taking an impulse response in audio format and generating the corresponding FAUST physical model compatible with the FAUST Physical Modeling Library presented in §6.2. This technique is commonly used to make signal models of musical instrument parts (e.g., acoustic resonators such as violin and guitar bodies, etc.).

Modal physical models can also be generated by carrying out a finite element analysis (FEM) on a 3D volumetric mesh. Meshes can be made "from scratch" or using a 3D scanner, allowing musical instrument designers to make virtual parts using a CAD model. In §6.3, we introduce `mesh2faust`, a tool to generate FAUST modal physical models from 3D volumetric meshes. While this technique is more flexible and allows us to model elements "from scratch," generated models are usually not as accurate as the one deduced from the impulse response of a physical object that faithfully reproduce its harmonic content.

Even though it is tempting to model an instrument in its whole using its complete graphical representation, better results are usually obtained using a modular approach where each part of the instrument (e.g., strings, bridge, body, etc.) are modeled as single entities. The FAUST Physical Modeling Library introduced in §6.2 implements a wide range of ready-to-use musical instrument parts. Missing elements can then be easily created using `mesh2faust` or `ir2faust`. Various examples of such models are presented in §6.2.2 and §6.3.5.

6.1.2 From Virtual to Physical

§1.6 gives an overview of various digital fabrication techniques, with 3D printing in particular, that can be used to materialize virtual representation of musical instrument parts under certain

conditions. In other words, most elements provided to `mesh2faust` (see §6.3) can be printed and turned into physical objects.

6.1.3 Connecting Virtual and Physical Elements

Standard hardware for digitizing mechanical acoustic waves and vice versa can be used to connect the physical and virtual elements of a hybrid instrument (see Figure 6.1). Piezos (contact microphones) can capture mechanical waves on solid surfaces (e.g., guitar body, string, etc.) and microphones mechanical air waves (e.g., in a tube, etc.). Captured signals can be digitized using an analog to digital converter (ADC). Inversely, digital audio signals can be converted to analog signals using a digital to analog converter (DAC) and then to mechanical waves with a transducer (for solid surfaces) or a speaker (for the air).

In some cases, a unidirectional connection is sufficient as waves travel in only one direction and are not (or almost not) reflected. This is the case of the BladeAxe instrument series presented in §2.2 and §2.3 where sound excitations (i.e., plucks) are picked up using piezos and transmitted to virtual strings. This type of system remains simple and works relatively well as the latency of the DAC or the ADC doesn't impact the characteristics the generated sound.

On the other hand, a bidirectional connection (see §6.2.1) might be necessary in other cases. Indeed, reflection waves play a crucial role in the production of sound in some musical instruments such as woodwinds. For examples, connecting a physical clarinet mouthpiece to a virtual bore (see §E) will require the use of a bidirectional connection in order for the frequency of vibration of the reed to be coupled to the tube it is connected to. This type of connection extends beyond the instrument to the performer that constantly adjusts its various parameters in function of the generated sound [42]. However, implementing this type of system can be very challenging (see §E) as the DAC and the ADC will add latency, which in the case of the previous example will artificially increase the length of the virtual bore. Thus, using low latency DACs and ADCs is crucial when implementing this type of systems sometimes involving the use of active control techniques [61, 115, 114]. §E presents an example of such a hybrid instrument.

More generally, the use of high-end components with a flat frequency response is very important when implementing any kind of hybrid instruments. Also, hardware can become very invasive in some cases, and it is the musical instrument designer's responsibility to find the right between all these parameters.

6.1.4 Adapting This Framework to Mobile Devices

Beyond this theoretical modularity (keeping in mind that audio latency can be a limiting factor in some cases) where any part of mobile hybrid instruments can either be physical or virtual, some design "templates" are more efficient than others. Here, we give some guidelines/rules to restrain the scope of our framework to optimize its results when making mobile hybrid instruments.

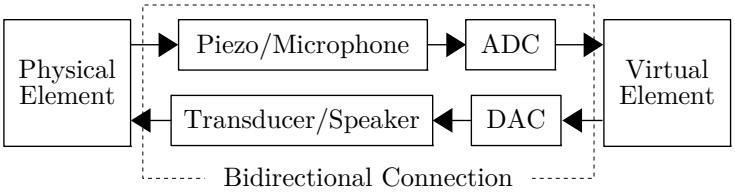


Figure 6.1: Bidirectional Connection Between Virtual and Physical Elements of a Hybrid Instrument.

In the context of augmented mobile instruments where standalone aspects and lightness are key factors, the number of physical/acoustical elements of hybrid instruments must be scaled down compared to what is possible with a desktop-based system. Indeed, transducers are large and heavy components requiring the use of an amplifier, which itself needs a large power source other than the mobile device battery, etc. Similarly, multichannel ADCs and DACs can take a fair amount of space and will likely need to be powered with an external battery/power supply.

Even though applications generated with `faust2smartkeyb` (see §3.3) are fully compatible with external USB ADC/DACs, we believe that restraining hybrid mobile instruments to their built-in ADC/DACs helps preserve their compactness and playability.

Beyond the aesthetic and philosophical implications of hybrid instruments (which are of great interest but are not the object of this thesis), their practical goal is to leverage the benefits of physical and virtual elements to combine them. In practice, the digital world is more flexible and allows us to model/approximate many physical elements. However, even with advanced sensor technologies, it often fails to capture the intimacy (see §1.1) between a performer and an acoustic instrument allowing us to directly interact with its sound generation unit (e.g., plucked strings, hand drum, etc.) [6].

Thus, a key factor in the success of hybrid mobile instruments lies in the use of a physical/acoustical element as the direct interface for the performer, enabling passive haptic feedback and taking advantage of the randomness and unpredictability of acoustical elements (see §1.2). In other words, even though it is possible to combine any acoustical element with any digital one, we encourage instrument designers to use acoustical excitations to drive virtual elements (see Figure 6.2), implementing the concept of "acoustically driven hybrid instruments" presented in §1.2. While the single analog input available on most mobile devices allows for the connection of one acoustical element, having access to more independent analog inputs would significantly expend the scope of the type of instruments implementable with our framework. This remains one of its main limitation.

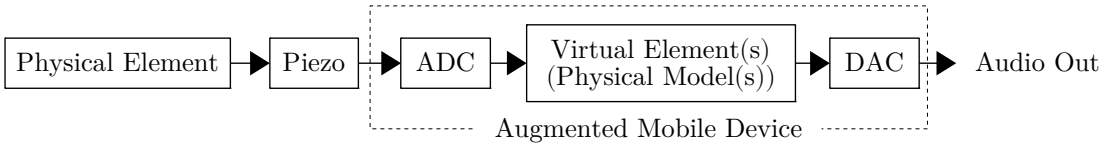


Figure 6.2: "Typical" Acoustically Driven Mobile Hybrid Instrument Model.

6.2 FAUST Physical Modeling Library

FAUST is particularly well suited for implementing digital waveguide and modal physical models (see §1.5.3). While the FAUST-STK presented in §A offers a diverse collection of physical models, it is just a "translation"/reimplementation (with some new models) of the original STK [43] in FAUST. Even though FAUST-STK models share many functions through the `instrument.lib` library, they are still implemented as standalone objects, and virtual instrument parts can't be reused.

In this section, we introduce the FAUST Physical Modeling Library: `physmodels.lib`,[1] an environment to create physical models of musical instruments in a modular way. Low and high level elements can be combined to implement existing or completely novel instruments.

6.2.1 Bidirectional Block-Diagram Algebra

In the physical/acoustical world, waves propagate in multiple dimensions and directions across the different parts of musical instruments. Thus, coupling between the constituting elements of an instrument sometimes plays an important role in its general acoustical behavior (see §6.1.3 and §E). Coupling might be limited and even neglected when designing models of some instruments such as the electric guitar where energy is transmitted from the string to the pickup in a unidirectional way [176]. On the other hand, coupling is crucial for other types of instruments such as woodwinds (e.g., the clarinet), where the frequency of vibration of the reed(s) is determined by the length of the tube it is connected to (see §E).

Coupling can be implemented by creating bidirectional connections between the different elements of a model. The block-diagram algebra of FAUST allows us to connect blocks in a unidirectional way (from left to right) and feedback signals (from right to left) can be implemented using the tilde (~) diagram composition operation:

```
process = (A : B) ~ (C : D) ;
```

where `A`, `B`, `C`, and `D` are hypothetical functions with a single argument and a single output. The resulting FAUST-generated block diagram can be seen in Figure 6.3.

In this case, the `D/A` and the `C/B` couples can be seen as bidirectional blocks/functions that could implement some musical instrument part. However, the FAUST semantics doesn't allow them

[1] `physmodels.lib` is now a standard library and is part of the FAUST distribution.

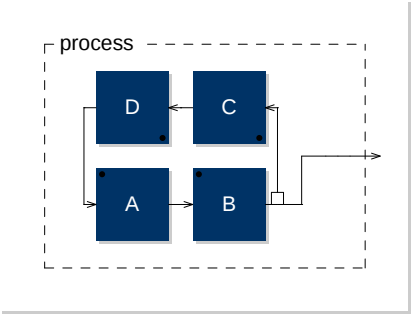


Figure 6.3: Bidirectional Construction in FAUST Using the Tilde Diagram Composition Operation.

to be specified as such from the code, preventing the implementation of "bidirectional functions." Since this feature is required to create a library of physical modeling elements, we had to implement it.

Bidirectional blocks in the FAUST Physical Modeling Library all have three inputs and outputs. Thus, an empty block can be expressed as:

```
emptyBlock = _,_,_;
```

The first input and output correspond to left-going waves (e.g., `C` and `D` in Figure 6.3), the second input and output to right-going waves (e.g., `A` and `B` in Figure 6.3), and the third input and output can be used to carry any signal to the end of the algorithm. As we'll see in §6.2.2, this can be useful when picking up the sound at the middle of a virtual string, for example.

Bidirectional blocks are connected to each other using the `chain` primitive which is part of `physmodels.lib`. For example, an open waveguide (no terminations) expressed as:

```
waveguide(nMax,n) = par(i,2,de.fdelay4(nMax,n)),_;
```

where `nMax` is the maximum length of the waveguide and `n` its current length, could be connected to our `emptyBlock`:

```
foo = chain(emptyBlock : waveguide(256,n) : emptyBlock) ;
```

Note the use of `fdelay4` in `waveguide`, which is a fourth order fractional delay line [1, 189].

The FAUST compiler is not able yet to generate the block diagram corresponding to the previous expression in an organized bidirectional way (see §6.4). However, a "hand-made" diagram can be seen in Figure 6.4.

The placement of elements in a `chain` matters and corresponds to their order in the physical/acoustical world. For example, for a set of hypothetical functions implementing the different parts of a violin, we could write:

```
violin = chain(nuts : string : bridge : body);
```

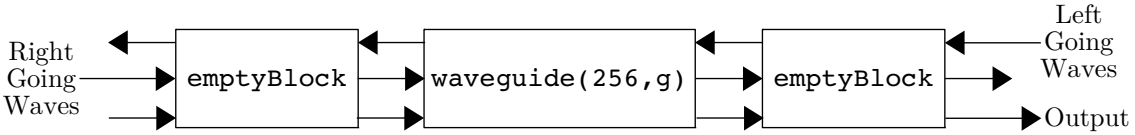


Figure 6.4: Bidirectional Construction in FAUST Using the `chain` Primitive.

The main limitation of this system is that it introduces a one sample delay in both directions for each block in the `chain` due to the internal use of ~ [67]. This has to be taken into account when implementing certain types of elements such as a string or a tube.

Terminations can be added on both sides of a chain using `lTermination(A,B)` for a left-side termination and `rTerminations(B,C)` for a right-side termination where `B` can be any bidirectional block, including a `chain`, and `A` and `C` are functions that can be put between left and right-going signals (see Figure 6.5).

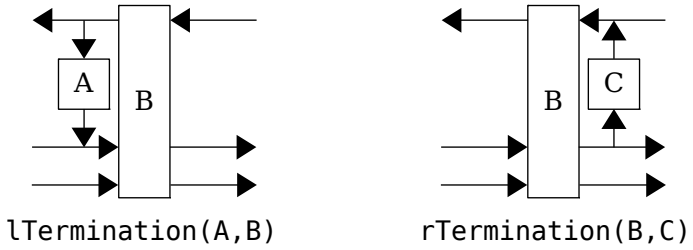


Figure 6.5: `lTermination(A,B)` and `rTermination(B,C)` in the FAUST Physical Modeling Library.

A signal `x` can be fed anywhere in a `chain` by using the `in(x)` primitive. Similarly, left and right-going waves can be summed and extracted from a chain using the `out` primitive (see Code Listing 6.1).

Finally, a chain of blocks `A` can be "terminated" using `endChain(A)` which essentially removes the three inputs and the first two outputs of `A`.

Assembling a simple waveguide string model with "ideal" rigid terminations is simple using this framework:

```
string(length,pluckPosition,excitation) = endChain(wg)
with{
  maxStringLength = 3; // in meters
  lengthTuning = 0.08; // adjusted "by hand"
  tunedLength = length-lengthTuning;
```

```
  nUp = tunedLength*pluckPosition; // upper string segment length
  nDown = tunedLength*(1-pluckPosition); // lower string segment length
  lTerm = lTermination(*(-1),basicBlock); // phase inversion
  rTerm = rTermination(basicBlock,*(-1)); // phase inversion
  stringSegment(maxLength,length) = waveguide(nMax,n)
  with{
    nMax = maxLength : l2s; // meters to samples
    n = length : l2s/2; // meters to samples
  };
  wg = chain(lTerm : stringSegment(maxStringLength,nUp) :
    in(excitation) : out : stringSegment(maxStringLength,nDown) :
    rTerm); // waveguide chain
};
```

Listing 6.1: "Ideal" String Model With Rigid Terminations.

In this case, since `in` and `out` are placed next to each other in the `chain`, the position of excitation and the position of the pickup are the same as well.

6.2.2 Assembling High Level Parts: Violin Example

The FAUST Physical Modeling Library contains a wide range of ready-to-use instrument parts and pre-assembled models. An overview of the content of the library is provided in §C and is also available in the FAUST libraries documentation [1]. Detailing the implementation of each function of the library would be interesting, however this section focuses on one of its models: `violinModel` (see Code Listing 6.2) which implements a simple bowed string connected to a body through a bridge.

```
violinModel(stringLength,bowPressure,bowVelocity,bowPosition) =
endChain(modelChain)
with{
  stringTuning = 0.08;
    stringL = stringLength-stringTuning;
    modelChain = chain(
        violinNuts :
        violinBowedString(stringL,bowPressure,bowVelocity,bowPosition) :
        violinBridge : violinBody : out
    );
```

```
};
```

Listing 6.2: `violinModel`: a Simple Violin Physical Model From the FAUST Physical Modeling Library.

`violinModel` assembles various high-level functions implementing violin parts. `violinNuts` is a termination applying a light low-pass filter on the reflected signal. `violinBowedString` is made out of two open string segments allowing us to choose the bowing position. The bow nonlinearity is implemented using a table. `violinBridge` implements the "right termination" as well as the reflectance and the transmittance filters [177]. Finally, `violinBody` is a simple violin body modal model.

In addition to its various models and parts, the FAUST Physical Modeling Library also implements a series of ready-to-use models hosting their own user interface (see §C). The corresponding functions end with the `_ui` suffix. For example:

```
process = pm.violin_ui;
```

is a complete FAUST program adding a simple user interface to control the violin model presented in Code Listing 6.2.

While `[...]_ui` functions associate continuous UI elements (e.g., knobs, sliders, etc.) to the parameters of a model, functions ending with the `_ui_midi` prefix automatically format the parameters linked the FAUST MIDI parameters (i.e., frequency, gain, and note-on/off) using envelope generators. Thus, such functions are ready to be controlled by a MIDI keyboard.

Nonlinear behaviors play an important role in some instruments (e.g., gongs, cymbals, etc.). While waveguide models and modal synthesis are naturally linear, nonlinearities can be introduced using nonlinear allpass ladder filters [178]. `allpassNL` implements such a filter in the FAUST Physical Modeling Library.

Some of the physical models of the FAUST-STK (see §A) were ported to the FAUST Physical Modeling Library and are available through various functions summarized in Table 6.1.

FAUST-STK Model	**Corresponding FPML Functions**
`bowed.dsp`	`violin` / `violin_ui` / `violin_ui_midi`
`brass.dsp`	`brassModel` / `brass_ui` / `brass_ui_midi`
`clarinet.dsp`	`clarinetModel` / `clarinet_ui` / `clarinet_ui_midi`
`fluteStk.dsp`	`fluteModel` / `flute_ui` / `flute_ui_midi`

Table 6.1: FAUST-STK Models and Their Corresponding Function Re-Implementations in the FAUST Physical Modeling Library.

6.3 `mesh2faust`: a FAUST Modal Physical Model Generator

While a wide range of models of existing or novel instruments can be built using any of the functions implementing high-level musical instrument parts in the FAUST Physical Modeling Library (e.g., strings, tubes, bow, bodies, bridges, mouthpieces, etc.), being able to use new custom parts is important as well. We believe that the best way to specify an object to model is through its 3D graphical representation (see §1.5.2).

In this section, we present `mesh2faust`, [121][2] an open-source modal physical model generator for the FAUST programming language. `mesh2faust` takes a volumetric mesh of a 3D object as its main argument, carries out a finite element analysis, and generates the corresponding FAUST modal physical model. A wide range of parameters can be configured to fine-tune the analysis as well as the behavior of the generated object.

Models generated by `mesh2faust` can be easily integrated to larger systems implemented with the FAUST Physical Modeling Library (see §6.2). For example, a waveguide bowed string could be connected to a violin bridge and body modeled with `mesh2faust`, etc. While this system doesn't allow us to model "anything" and has a few limitations (e.g., nonlinearities are ignored, "air modes" are not modeled, etc.), it remains a simple way to generate versatile models and instrument parts.

First, we present a brief review of the theory behind finite-element and modal synthesis. Next, we describe the implementation of `mesh2faust` and present a complete open-source framework to model 3D objects and turn them into physical models of musical instruments.

6.3.1 Theory: FEM

The Finite Element Method (FEM) is a frequently used technique for modeling the dynamic deformation of an object and synthesizing the sound emitted by the object after an excitation. The method consists of meshing the object in small elements defined by several nodes (depending on the desired element type), and then solving the equations of motion for each of the nodes.
The linear deformation equation with no damping can be written as follows:

$$\mathbf{M}\,\ddot{\mathbf{x}}(\mathbf{t}) + \mathbf{K}\,\mathbf{x}(\mathbf{t}) = \mathbf{f}(\mathbf{t}) \tag{6.1}$$

where $\mathbf{x}(\mathbf{t}) \in \mathbb{R}^{3n}$ corresponds to the vector of displacements at all the nodes, and $\mathbf{M}$ and $\mathbf{K}$ represent respectively the mass and stiffness matrices determined by the object properties.
A first attempt to solve Equation 6.1 is to assume that the solutions of the corresponding homogeneous equation ($\mathbf{f}(\mathbf{t}) = \mathbf{0}$) are of the form $u_i(t) = \mathbf{U}_i e^{j\omega_i t}$ where $\mathbf{U}_i \in \mathbb{R}^{3n}$ and $\omega_i \in \mathbb{R}$. The substitution of those potential solutions into the homogeneous equation of Equation 6.1 defines the

[2] `mesh2faust` has been introduced at the ICMC-17 conference. Some sections and figures of this paper were copied verbatim here.

commonly called generalized eigenvalue problem:

$$\mathbf{KU} = \mathbf{\Lambda MU} \tag{6.2}$$

where $\mathbf{\Lambda}$ is a diagonal matrix containing the eigenvalues $\lambda_i = \omega_i^2$ of Equation 6.1, and $\mathbf{U}$ is the modal matrix containing the eigenvectors $\mathbf{U}_i$ of Equation 6.1. By solving this problem, both eigenvalues and eigenvectors of the system will be obtained.
Now, the system in Equation 6.1 can be decoupled by using the transformation $\mathbf{x} = \mathbf{Uq}$ and Equation 6.1 can be rewritten as

$$\ddot{\mathbf{q}} + \mathbf{\Lambda q} = \mathbf{U^T f}. \tag{6.3}$$

Thus, the solutions of the decoupled homogeneous modal form $\ddot{\mathbf{q}} + \mathbf{\Lambda q} = \mathbf{0}$ can be implemented using a parallel bank of modes of the form

$$q_i = a_i \sin(2\pi f_i + \theta_i), \tag{6.4}$$

where a_i is the excited amplitude of the ith mode, f_i is its frequency, and θ_i its initial phase. The excitation force $\mathbf{f(t)}$ is taken to be an impulse at time $t = 0$, and our simulation will start at time 0. To maximize the initial attack without creating an amplitude discontinuity at time 0, we choose our initial phases as $\theta_i = 0$. The excited mode amplitudes a_i depend on the location of the object excitation, and the frequency of the ith mode depends on the object geometry and material properties according to

$$f_i = \frac{1}{2\pi}\sqrt{\lambda_i}\,, \tag{6.5}$$

where λ_i denotes the ith eigenvalue obtained from solving Equation 6.2.

Since the damping matrix was omitted in the above formulation, the modes in Equation 6.4 are missing an important factor for sound synthesis which is the exponential decay. Exponential decays have not been estimated by FEM, as will be explained in the following section.

6.3.2 FAUST Modal Physical Model

Modal synthesis [5] consists of implementing each mode of a linear system as an exponentially decaying sine wave (see §1.5). Each mode can then be configured with its frequency, gain, and resonance duration (T60). Sine waves with an exponential decay are typically implemented using a sine wave oscillator with an exponential envelope or with a resonant bandpass filter [177]. The second option offers more flexibility since any signal can be fed into the model to excite it. This feature is important to be able to create modules compatible with FPML, which is why modal physical models generated by `mesh2faust` use this approach.

Our mode filters are implemented as a biquad section having transfer function

$$H(z) = g\frac{1 - z^{-2}}{1 + \alpha_1 z^{-1} + \alpha_2 z^{-2}} \tag{6.6}$$

with

$$\begin{aligned} \alpha_1 &= -2\tau\cos\omega \\ \alpha_2 &= \tau^2 \\ \omega &= \frac{2\pi f}{f_s} \\ \tau &= 0.001^{\frac{1}{t_{60}}} \end{aligned}$$

having the following parameters:

- g: the mode gain
- f: the mode frequency
- t_{60}: the mode T60

The constrained form of the transfer-function numerator $1 - z^{-2} = (1 + z^{-1})(1 - z^{-1})$ enforces a zero of transmission at both DC and half the sampling rate, thereby giving a bandpass characteristic appropriate for a resonant mode. The denominator parameters similarly enforce a complex-conjugate pole pair with angles $\pm\omega$ and radius τ.

The corresponding FAUST function (`modeFilter`) is used to implement our FAUST "modal model" which takes the position of excitation and the excitation signal as its two arguments:

```
modalModel(exPos) = _ <: par(i,nModes,modeFilter(modesFreqs(i),modesT60s
    (i),modesGains(exPos,i))) :> /(nModes)
```

The FAUST-generated[3] block diagram associated with this function can be seen in Figure 6.6. `modesFreqs(i)`, `modesT60s(i)`, and `modesGains(exPos,i)` are arrays that are formatted by `mesh2faust` (see §6.3.3). This model is compatible with all the decoupled excitation functions of the FAUST Physical Modeling Library such as hammer, pluck, impulse, etc. (see §6.2).

6.3.3 `mesh2faust`

`mesh2faust` is implemented in C++ and works as a UNIX command line application taking a volumetric mesh as its main argument and outputting a FAUST modal physical model. Various parameters can be configured using a wide range of flags that are presented in this section.[4]

[3] This diagram was generated using the `faust2svg` tool.

[4] A complete list of the `mesh2faust` options is available in its online documentation: `https://github.com/grame-cncm/faust/blob/master-dev/tools/physicalModeling/mesh2faust/README.md`

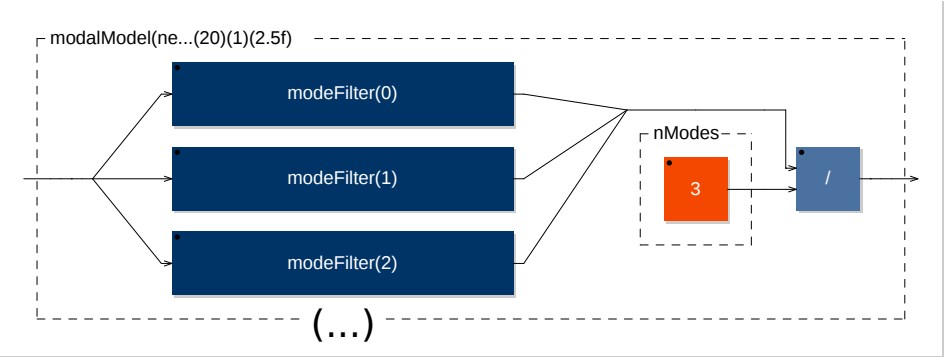


Figure 6.6: Block Diagram of a FAUST Modal Model Implementing Three Modes.

`mesh2faust` relies on the Vega FEM Library[5] [62] to carry out the finite element analysis needed to compute mode parameters. The provided volumetric mesh must be saved as an "object file" (`.obj`) and its dimensions should be in meters. The volumetric mesh must first be converted into a 3D tetrahedral mesh (see Figure 6.7). This is easily done by Vega which implements its own tetrahedral mesher [170]. Material properties (Young Modulus in N/m^2, Poisson's Ratio, and Density in kg/m^3) are applied to the model during this step and can be configured by the user using the `--material` flag.

Next, the corresponding mass and stiffness matrices are generated and fed to the Vega eigen solver. The number of modes to be computed during this step can be configured by the user using the `--nfemmodes` flag and must be smaller than the number of vertices in the volumetric mesh. The result of this operation is a list of eigenvalues and eigenvectors that are ordered linearly starting from the lowest mode.

As mentioned in §6.3.1, the modes frequencies can be easily calculated from eigenvalues (see Equation 6.5), and the mode gains can be computed from the matrix of eigenvectors and the excitation force.

Before the mode gains and frequencies are integrated to the FAUST physical model, they are selected based on a series of user-defined parameters:

- `--nsynthmodes`: number of modes to synthesize
- `--minmode`: lowest mode frequency
- `--maxmode`: highest mode frequency
- `--cb`: mode selection by critical bands
- `--expos`: list of "excitable" vertices

[5] `http://run.usc.edu/vega/`

- `--lmexpos`: number of excitation position

`--minmode` and `--maxmode` allow us to define the frequency range of the modes to synthesize. If the number of modes within this range is smaller than `--nsynthmodes`, this parameter will be adapted accordingly. Note that `--nsynthmodes` can be different than `--nfemmodes` since some modes might be discarded at the bottom of the spectrum depending on the value of `--minmode`.

If the number of modes in the range defined by `--minmode` and `--maxmode` is greater than `--nsynthmodes`, synthesized modes will be selected by frequency, starting from the lowest mode. `--cb` allows us to change this behavior by selecting modes by critical bands. In this case, the frequency range defined by `--minmode` and `--maxmode` will be split into `--nsynthmodes` critical bands and the loudest mode for each of them will be selected. This feature is very useful if the model has lots of modes.

By default, the number of excitation positions in the generated model is the same as the number of vertices in the provided volumetric mesh. If this mesh has a high density, the amount of data to integrate to the model might become a problem. For example, for a mesh with 3E4 vertices and 200 modes to synthesize, the modes gains matrix will have a size of 3E4x200 which corresponds to 6E6 floating point values to be hard-coded in the FAUST physical model source code! Thus, it might be helpful to optimize the model by limiting its number of excitation positions by using `--lmexpos` . In that case, positions are "randomly" selected, however, specific vertices can be selected using `--expos`. Vertex IDs can be easily retrieved using a mesh visualizer such as MeshLab[6] (see §6.3.4).

After this, the resulting FAUST modal physical model (see §6.3.2) is generated and placed in a FAUST library file (`.lib`).

Currently, `mesh2faust` doesn't compute the damping matrix of the system. Thus, while mode T60s cannot be estimated, they can be optionally empirically computed as a function of the frequency and the gain of the modes relatively to the fundamental. This solution is temporary and we hope to add damping matrix support to our system in the future.

6.3.4 Complete Open Source Solution to Finite Element Analysis

CAD and FEM tools are widely used in industry for different types of applications. Most of these tools are proprietary and their cost is often prohibitive for personal applications. In this section, we briefly describe a completely open-source (OS) framework/tool chain allowing us to quickly design 3D models from scratch and turn them into FAUST physical models using `mesh2faust`.

OpenSCAD is an open-source CAD program in which shapes are specified using a high-level functional programming language (see §4.1). While it allows us to design complex 3D objects by combining or differentiating simple 3D elements (e.g., cubes, spheres, cylinders, etc.), it can also linearly or rotationally extrude 2D shapes specified as a polygon (expressed as a set of 2D Cartesian

[6] `http://www.meshlab.net/`

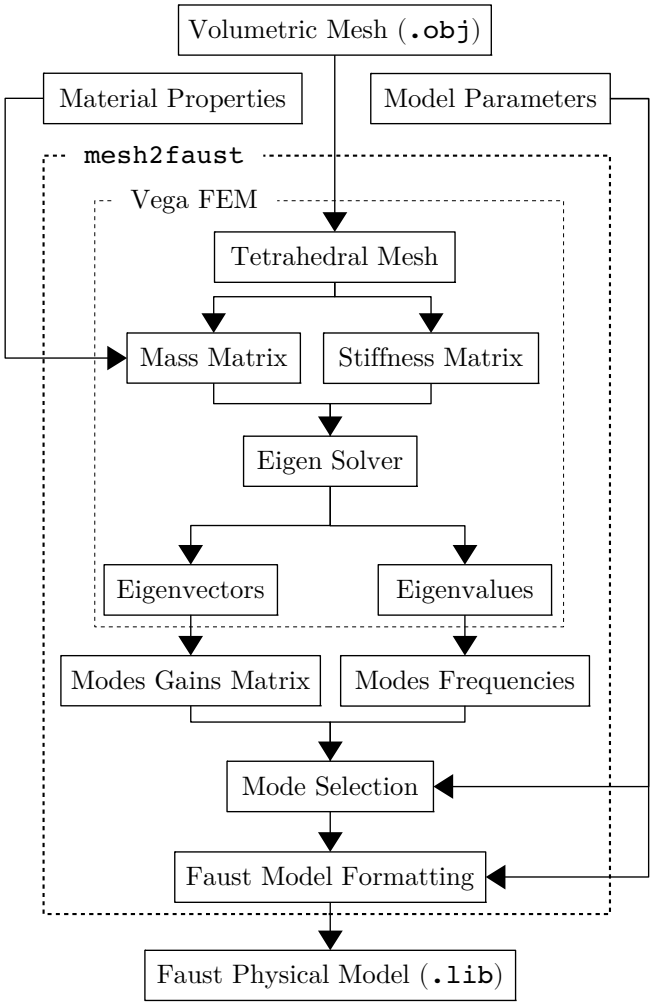


Figure 6.7: Overview of the `mesh2faust` Implementation.

coordinates). Thus, `mesh2faust` comes with a modified version of Daniel Newman's Inkscape to OpenSCAD converter[7] allowing us to export Inkscape[8] 2D paths to OpenSCAD. This is very useful to create more complex 3D shapes (see Figure 6.8) such as the one presented in §6.3.5. Various parameters such as the number of points (resolution) in the generated polygon can be configured, etc.

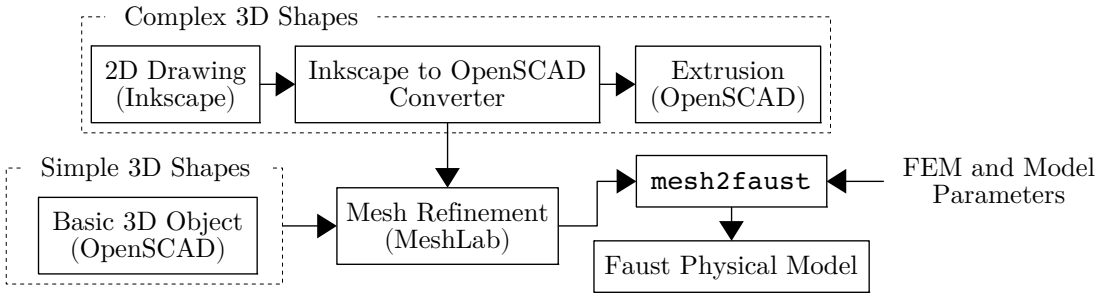


Figure 6.8: Open Source Framework to Make FAUST Modal Physical Models From Scratch.

6.3.5 Example: Marimba Physical Model Using FPML and `mesh2faust`

This section demonstrates how a simple marimba physical model can be made using `mesh2faust` and FPML.[9] The idea is to use a 3D CAD model of a marimba bar, generate the corresponding modal model, and then connect it to a tube model implemented in FPML.

A simple marimba bar 3D model was made by extruding a marimba bar cross section (see Figure 6.9) using the Inkscape to OpenSCAD tool presented in §6.3.4. The resulting CAD model was then turned into a volumetric mesh by importing it to MeshLab and by uniformly re-sampling it to have approximately 4500 vertices (more details about this type of operation are provided in §B). The mesh produced during this step (`marimbaBar.obj` in the following code listing) was processed by `mesh2faust` using the following command:

```
mesh2faust --infile marimbaBar.obj --nsynthmodes 50 --nfemmodes 200
  --maxmode 15000 --expos 2831 3208 3624 3975 4403 --freqcontrol
  --material 1.3E9 0.33 720 --name marimbaBarModel
```

The material parameters are that of rosewood which is traditionally used to make marimba bars. The number of modes is limited to 50 and various excitation positions were selected to be uniformly spaced across the horizontal axis of the bar. `frequency control mode` is activated to be able

[7] `http://www.thingiverse.com/thing:25036/`

[8] `https://inkscape.org/`

[9] An extended version of this example with more technical details is also available here: `https://ccrma.stanford.edu/~rmichon/faustTutorials/#making-custom-elements-using-mesh2faust`

to transpose the modes of the generated model in function of the fundamental frequency making the model more generic.

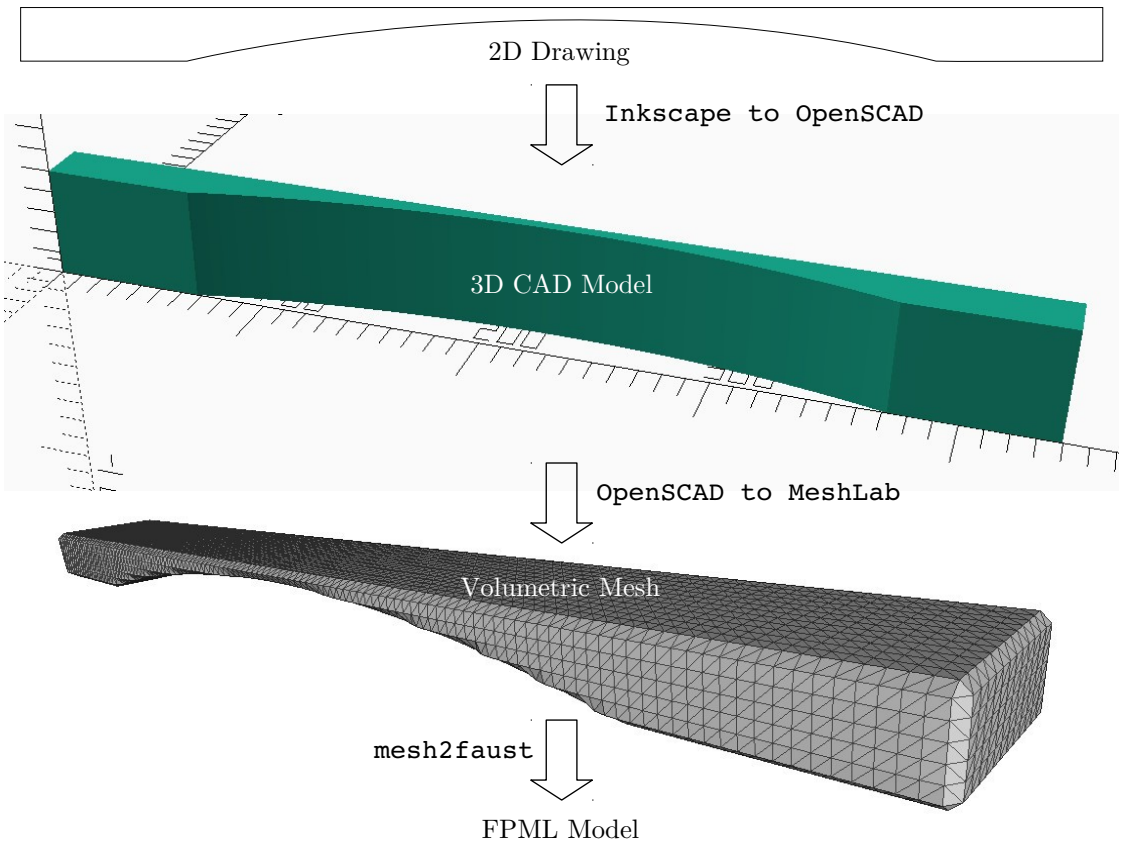


Figure 6.9: Marimba Bar Model – Steps From a 2D Drawing to a FAUST Modal Model.

A simple marimba resonator was assembled using FPML and is presented in Code Listing 6.3. It is made out of an open tube where two simple lowpass filters placed at its extremities are used to model the wave reflections. The model is excited on one side of the tube and sound is picked-up on the other side.

```
marimbaResTube(tubeLength,excitation) = endChain(tubeChain)
with{
  lengthTuning = 0.04; tunedLength = tubeLength-lengthTuning;
  absorption = 0.99; lowpassPole = 0.95;
  endTubeReflection = si.smooth(lowpassPole)*absorption;
  tubeChain = chain(
    in(excitation) : terminations(endTubeReflection,
```

```
    openTube(maxLength,tunedLength),
    endTubeReflection) : out
  );
};
```

Listing 6.3: Simple Marimba Resonator Tube Implemented With FPML.

Code Listing 6.4 demonstrates how `marimbaBarModel` can be simply connected to the marimba resonator. A unidirectional connection can be used in this case since waves are only transmitted from the bar to the resonator.

```
marimbaModel(freq,exPos) =
  marimbaBarModel(freq,exPos,maxT60,T60Decay,T60Slope) :
  marimbaResTube(resTubeLength)
with{
  resTubeLength = freq : f2l;
  maxT60 = 0.1; T60Decay = 1; T60Slope = 5;
};
```

Listing 6.4: Simple Marimba Physical Model.

This model is now part of the FAUST Physical Modeling Library. More examples of models created using this technique can be found in §B.

6.4 Discussion and Future Directions

The framework presented in this section remains limited by several factors. Audio latency introduced by ADCs and DACs prevents in some cases the implementation of cohesive bidirectional chains between physical and virtual elements. Audio latency reduction has been an ongoing research topic for many years and more work has to be done in this direction. This problem is exacerbated by the use of mobile devices at the heart of these systems that are far from being specialized for this specific type of application (i.e., operating system optimizations inducing extra latency, number of analog inputs and outputs, etc.). On the other hand, we believe that despite the compromises that they entail, mobile devices remain a versatile, and yet easy to customize platform well suited to implement hybrid instruments (e.g., the BLADEAXE).

The FAUST Physical Modeling Library is far from being exhaustive and many models and instruments could be added to it. We believe that `mesh2faust` (see §6.3) will help enlarge the set of functions available in this system.

The framework presented in §6.2.1 allows us to assemble the different parts of instrument models in a simple way by introducing a bidirectional block diagram algebra to FAUST. While it provides

a high level approach to physical modeling, FAUST is not able to generate the corresponding block diagram in a structured way. This would be a nice feature to add.

Similarly, we would like to extend the idea of being able to make multi-dimensional block diagrams in FAUST by adding new primitives to the language. This idea is further developed in §D.

More generally, we hope to make more instruments using this framework and use them on stage for live performance.

Conclusion

> "Musical interface construction proceeds as more art than science, and possibly this is the only way that it can be done." (Perry Cook [41])

By combining physical and virtual elements, hybrid instruments are "physically coherent" and mutualized by nature and allow instrument designers to play to the strengths of both acoustical and digital elements. Current technologies and techniques allow us to blur the boundary between the physical and the virtual world enabling musical instrument designers to treat instrument parts in a multidimensional way. The FAUST Physical Modeling Library presented in §6.2 facilitates the design of such instruments by providing a way to approach physical modeling of musical instruments at a very high level.

Mobile devices combined with physical passive or active augmentations are well suited to implement hybrid instruments. Their built-in sensors, standalone aspect, and computational capabilities are the core elements required to implement the virtual portion of hybrid instruments. `faust2smartkeyb` facilitates the design of mobile apps using elements from the FAUST Physical Modeling Library, implementing skill transfer, and serving as the glue between the various parts of the instrument. Mobile devices might limit the scope of hybrid instruments by scaling down the number of connections between acoustical and digital elements because of technical limitations. However, we demonstrated that a wide range of instruments can still be implemented using this type of system.

The framework presented in this thesis is a toolkit for musical instrument designers. By facilitating skill transfer, it can help accelerate the learning process of instruments made with it. However, musical instruments remain tools controlled by the performer. Having a well designed instrument leveraging some of the concepts presented here doesn't mean that it will systematically play beautiful music and generate pleasing sounds: this is mostly up to the performer.

We believe that mobile hybrid instruments presented in this thesis help reconcile the haptic, the physical, and the virtual, partially solving some of the flaws of DMIs depicted by Perry Cook [42].

We recently finished releasing the various elements of the framework presented in this thesis and we hope to see the development of more hybrid mobile instruments in the future.

Summary of Contributions

Chapter 1 Contributions

In Chapter 1, we reviewed the literature relevant to this thesis and we gave an overview of the different instruments that inspired this work. We demonstrated that by being modular, digital musical instruments are often demutualized. We showed that augmented and hybrid instruments partially solve this problem by being partly based on physical elements or existing acoustic instruments. An overview of the field of mobile music was given. We emphasized the fact that mobile devices constitute a suitable platform to implement standalone, versatile DMIs, and that various types of skill transfer can be implemented on touchscreen interfaces. We gave an overview of the different physical modeling techniques used by the various tools presented in the following chapters. Finally, we demonstrated how digital fabrication, with 3D printing in particular, has been used to make acoustic and electronic musical instruments.

Chapter 2 Contributions

Chapter 2 presented various instruments and art installations exploring the idea of exciting physical models using physical/acoustical elements. The BladeAxe, which is one of them, is based on an iPad and provided the basis for the framework presented in the following chapters. These instruments were evaluated in the frame of various performances that were also briefly described.

Chapter 3 Contributions

The main contribution of Chapter 3 is MobileFaust, a set of tools facilitating the design of musical mobile apps targeting live performance. `faust2android` and `faust2ios` that predated MobileFaust were first presented. We showed how `faust2api` can be used to generate advanced DSP engines for a wide range of platforms from a simple Faust code and how it was used as the basis of `faust2smartkeyb`. We gave an overview of `faust2smartkeyb`'s features and we demonstrated how it can be used to implement a wide range of instruments leveraging performers skills. Finally, we highlighted the role of `faust2smartkeyb` in our framework as the "glue" between the different constituting elements of hybrid mobile instruments.

Chapter 4 Contributions

Chapter 4 demonstrated how mobile devices can be passively augmented to enhance or suggest specific interactions with some of their built-in elements towards musical instrument design. An exhaustive overview of this type of augmentation was provided. A framework implemented through

MobileFaust, an OpenScad library to facilitate the design of musical mobile device passive augmentations was introduced. Finally, a method approaching in a global way the design of musical instruments based on passively augmented mobile devices and apps generated with `faust2smartkeyb` was presented and evaluated through the results of a series of workshops on this topic.

Chapter 5 Contributions

Chapter 5's main contribution is a framework to actively augment mobile devices with sensors. We first presented Nuance, a device adding pressure sensitivity to the iPad touchscreen, increasing its expressive potential as a musical instrument. Multiple use examples of this system implementing various types of instruments were presented and used to evaluate it. An exhaustive overview of communication techniques to transmit real-time sensor data to mobile devices for the control of virtual instruments was provided. We then gave a brief description of our framework which uses micro-controllers to transmit MIDI data to the device through USB. We demonstrated that successful augmentations are simple, non-invasive, and should be complimentary of the built-in sensors and features of the device. Finally, we evaluated our framework through a workshop on this topic.

Chapter 6 Contributions

Chapter 6 developed the concept of hybrid mobile instrument. A framework to design this type of instruments was first provided. It demonstrated how musical instrument parts can be either physical/acoustical or virtual/digital. As part of this, an overview of the techniques to connect physical/acoustical to virtual/digital (and vice versa) was provided. It also showed how this type of approach can be scaled down to be implemented on mobile device. The Faust Physical Modeling Library, a tool to implement physical models of musical instruments using high and low level elements in a modular way was introduced. We demonstrated how to use it to implement various types of instruments. We showed how to extend the potential of this tool using `mesh2faust`, a system allowing us to convert graphical 3D models to acoustic Faust modal physical models. Finally, we evaluated our framework by making a marimba physical model from scratch using the Faust Physical Modeling Library and `mesh2faust`.

Future Work

The scope of the different elements presented in this dissertation is relatively broad, therefore there are many potential directions for future works and improvements.

Mobile technologies evolve fast and there's a trend towards the development of apps as web applications. This solve numerous issues such as maintaining systems working on multiple platforms (e.g., Android, iOS, etc.), openly distributing applications, etc. While web technologies don't compete yet

with native apps when it comes to real-time audio signal processing, significant progress has been made in recent years. The future release of Audio Worklets in the Web Audio API, combined with the power of WebAssembly, should help solve these problems in a near future. The FAUST compiler is already capable of generating WebAssembly code, so creating a web version of `faust2smartkeyb` should be relatively easy when these systems will be available in all major web browser.

While the FAUST Physical Modeling Library provides a convenient way to assemble bidirectional blocks in FAUST, diagrams generated by `faust -svg`, for example, don't reflect well this type of construction. This should be improved. A more advanced solution to this problem would be to implement the multidimensional extended FAUST block-diagram algebra presented in §D. We believe that such system would greatly simplify the design of physical models of musical instruments.

Similarly, we would like to facilitate the use of linear algebra in FAUST. Indeed, various physical modeling techniques (e.g., finite difference scheme, wave digital filters, etc.) rely on the use of matrix operations that are currently extremely hard to implement in Faust.

Beyond that, and more generally, a lot of work remains to be done in the field of physical modeling, especially in the frame of 3D printing. Most luthiers using this digital fabrication technique wished at some point they would be able to accurately predict the sound of an object before it is materialized.

While the finite element method used by `mesh2faust` allows us to specify models in a very high level way, simply by providing their 3D graphical representation, the type of materials they are made of, etc., it remains limited and generated models are only approximations of their real world counterparts. More simply, for now, we plan to add damping matrix support to `mesh2faust`, which will allow us to automatically compute the resonance duration of the modes of the generated modal models. We also plan to keep expanding the scope of the FAUST Physical Modeling Library.

Finally, we believe that the field of active control of wind instruments should be further explored (see §E). This would allow us to extend our concept of hybrid lutherie to wind instruments, for example.

We plan to keep teaching workshops of these topics and hope to see more hybrid mobile instruments in the computer music community and beyond in the future.

Appendices

Appendix A

FAUST-STK

This appendix is based on a modified version of [125].[1] It presents the FAUST-STK, a set of virtual musical instruments programmed in the FAUST programming language. Most of them are based on physical models inspired from the algorithms implemented in the Synthesis ToolKit (STK) [43] and the program SynthBuilder [150]. The FAUST-STK was the main inspiration for the FAUST Physical Modeling Library presented in §6.2.

The STK has been developed since 1996. It is a set of open source audio signal processing and algorithmic synthesis classes written in the C++ programming language that can be used in the development of music synthesis and audio processing software.

SynthBuilder was a program developed at Stanford's CCRMA in the nineties to implement sound synthesis based on physical models of musical instruments. Most of its algorithms use the waveguide synthesis technique but some of them are also based on modal synthesis [5].

An important part of our work consisted of improving and simplifying the models from these two sources in order to make them more efficient thanks to the FAUST semantic. All FAUST code in the FAUST-STK is commented, including frequent references to external bibliographical elements. Finally, many of the algorithms from the STK and SynthBuilder were upgraded with nonlinear allpass filters [178].

We discuss the different models of musical instruments implemented in the FAUST-STK, noting problems encountered and how they were resolved.

[1] Some sections and figures of this paper were copied verbatim here.

A.1 Waveguide Models

A.1.1 Wind Instruments

The algorithms used in the FAUST-STK are almost all based on instruments implemented in the Synthesis ToolKit and the program SynthBuilder. However, it is important to observe that some of them were slightly modified in order to adapt them to the FAUST semantic.

An attempt was made to use functions already defined in the default FAUST libraries to build our models. However, new support functions were written as needed in order to be able to use parameters from the STK classes and the SynthBuilder patches verbatim, without transformation or embedding within more general functions. The added functions were placed in a file called `faust-stk/instrument.lib`.

All the wind instruments implemented in the FAUST-STK are based on a similar architecture. Indeed, in most cases, the breath pressure that corresponds to the amplitude of the excitation is controlled by an envelope. The excitation is used to feed one or several waveguides that implement the body of the instrument. For example, in the case of a clarinet, the excitation corresponds to the reed that vibrates in the mouthpiece, and the body of the instrument is the bore and the bell. In Figure A.1, it is possible to see the block diagram of one of the two clarinet models that are implemented in the FAUST-STK. In that case, an ADSR[2] envelope that is embedded in the `breathPressure` box controls the breath pressure.

The other clarinet implemented in the FAUST-STK is a bit more complex as it has a tone hole model that makes it possible to change the pitch of the note being played in a more natural way. Indeed, in the algorithm shown in Figure A.1 and as in most of the basic waveguide models, the pitch is modulated by changing the length of the loop delay line which would correspond in "the real world" to changing dynamically the size of the clarinet's bore during the performance, as if it were a trombone (see §6.2).

The reed table employed with the two clarinets to excite the model was also used to create a very simple saxophone model that is even more comparable to a violin whose strings are excited by a reed.

Two models of flute are implemented in the FAUST-STK. The first one is based on the algorithm used in the Synthesis ToolKit that is a simplified version of [192]. The other model is showed in Figure A.2. It uses two loops instead of one.

A simple model of a brass instrument inspired from a class of the Synthesis ToolKit and with a mouthpiece based on the model described in [155] is implemented in the FAUST-STK. It can be used to emulate a wide range of instrument such as a French horn, a trumpet or even a trombone. Its algorithm can be seen in Figure A.3.

Finally, a tuned bottle in which it is possible to blow through the neck to make sound is also

[2] Attack - Decay - Sustain - Release

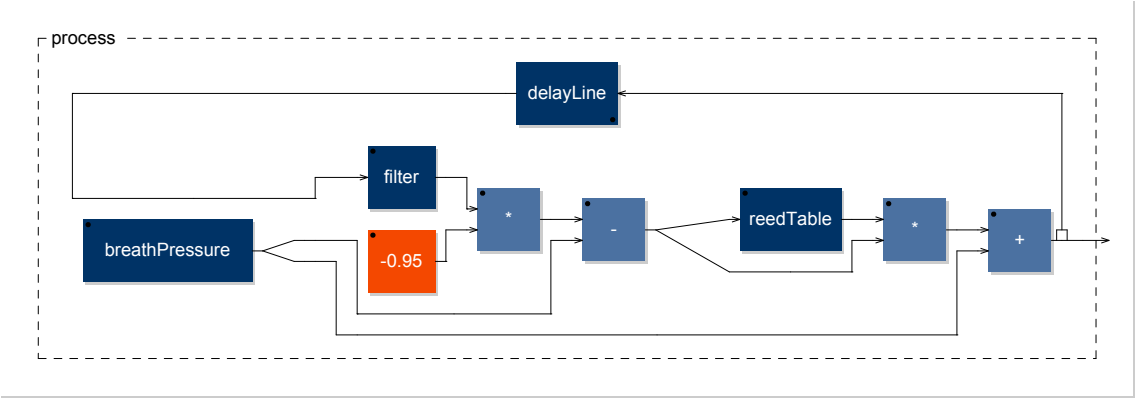


Figure A.1: `clarinet.dsp` Algorithm Drawn by FAUST Using `faust2svg`.

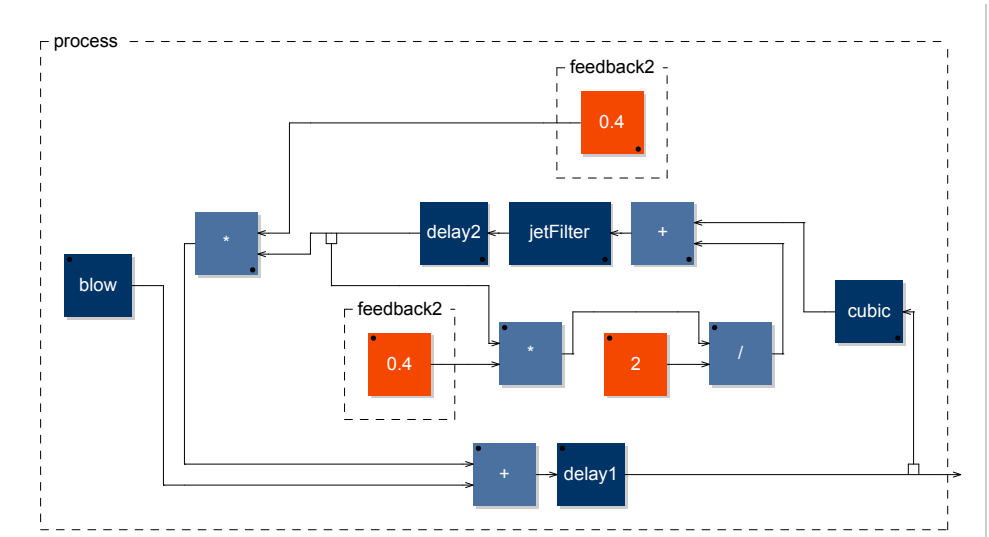


Figure A.2: `flute.dsp` Algorithm Drawn by FAUST Using `faust2svg`.

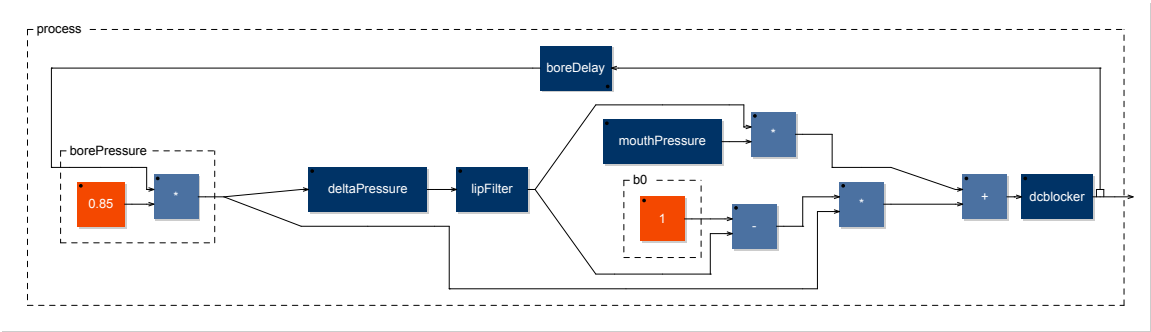


Figure A.3: `brass.dsp` Algorithm Drawn by FAUST Using `faust2svg`.

implemented in the FAUST-STK.

A.1.2 String Instruments

Some waveguide synthesis algorithms for plucked strings have previously been implemented in FAUST, [176] and elements of these ports appear in the old libraries `filter.lib` and `effect.lib` within the FAUST distribution. Going beyond these, the FAUST-STK includes models of stringed instruments from the STK such as a Sitar, bowed-string instrument, and SynthBuilder patches (running on a NeXT Computer) for an acoustic bass, piano, and harpsichord. Most of these models were furthermore extended with the new nonlinear allpass for spectral enrichment [178]. Further discussion regarding the nonlinear allpass and synthesis of keyboard instruments is given below in §A.2 and §A.5, respectively.

A.1.3 Percussion Instruments

Four objects in the FAUST-STK use the banded waveguide synthesis technique (described in [55]) to model the following percussion instruments:

- an iron plaque;
- a wooden plaque;
- a glass harmonica;
- a tibetan bowl.

Each of them can be excited with a bow or a hammer.

A.2 Using Nonlinear Passive Allpass Filter With Waveguide Models

Some of the instruments implemented in the FAUST-STK are using nonlinear passive allpass filters in order to generate nice natural and unnatural sound effects [178]. Nonlinear allpass filters can add interesting timbral evolution when inserted in waveguide synthesis/effects algorithms. The nonlinearities are generated by dynamically modulating the filter coefficients at every sample by some function of the input signal. For the instruments that use this kind of filter in the FAUST-STK, the user can decide whether the coefficients are modulated by the input signal or by a sine wave. In both cases, a "nonlinearity factor" parameter scales the range of the modulation of the filter coefficients. This parameter can be controlled by an envelope in order to make the modulated behavior more natural.

We adjust the length of the delay line of the instruments that use nonlinear allpass filters in function of the nonlinearity factor and of the order of the filter as follows:

$$DL = (SR/F) - FO \times NF$$

where DL is the delay length in samples, SR is the sampling rate, F is the pitch frequency, FO is the filter order and NF the nonlinearity factor (value between 0 and 1).

The nonlinear allpass filter can be placed anywhere in the waveguide loop, for example just before the feedback as showed in Figure A.4.

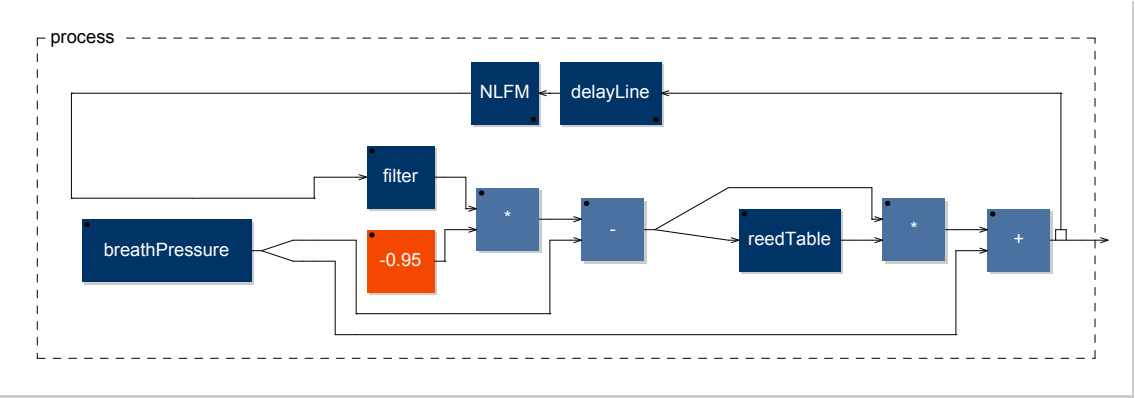


Figure A.4: Modified Version of `clarinet.dsp` That Uses a Nonlinear Allpass Filter in Its Feedback Loop.

Finally, it is interesting to mention that we were able to implement a frequency modulation synthesizer in the FAUST-STK by using this kind of filter on a sine wave signal. A related result is reported in [174].

A.3 Modal Models

A set of instruments using modal synthesis can be found in the FAUST-STK. They are all implemented in the same code (see Figure A.5) as they are based on the same algorithm.

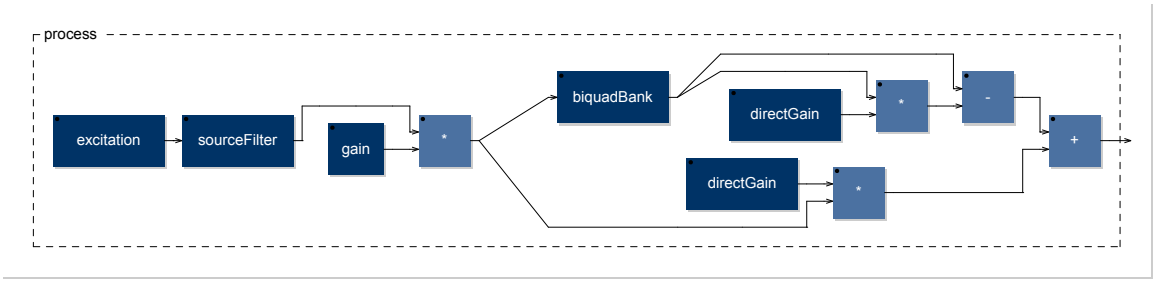


Figure A.5: `modalBar.dsp` Algorithm Drawn by FAUST Using `faust2svg`.

A.4 Voice Synthesis

A very simple voice synthesizer based on the algorithm from the Synthesis ToolKit is implemented in the FAUST-STK. It uses a low-pass-filtered impulse-train to excite a bank of 4 bandpass filters that shape the voice formants. The formant parameters are stored in a C++ function as a set of center frequencies, amplitudes, and bandwidths. This function is then called in the FAUST-STK code using the foreign function primitive. The thirty-two phonemes stored in this function are the same as in the Synthesis ToolKit.

A.5 Keyboards

A SynthBuilder patch implementing a commuted piano [175] was written in the late 1990s at Stanford's CCRMA. This patch was partly ported in 2006 by Stephen Sinclair at McGill University in the Synthesis ToolKit [171]. A big part of his work consisted of extracting parameter-values from the SynthBuilder patch and storing them in a set of C++ functions. We reused them to build our FAUST commuted piano version by using the foreign function mechanism [67].

In this piano model, the keyboard is split into two parts, each using a different algorithm: The tones below E6 use the commuted waveguide synthesis technique [177] while tones above or equal to E6 use modal synthesis (a series of biquad filters) to generate the sound (Figure A.6).

A commuted harpsichord has also been implemented in the FAUST-STK. It was inspired by another SynthBuilder patch that uses a very similar algorithm to the one described above.

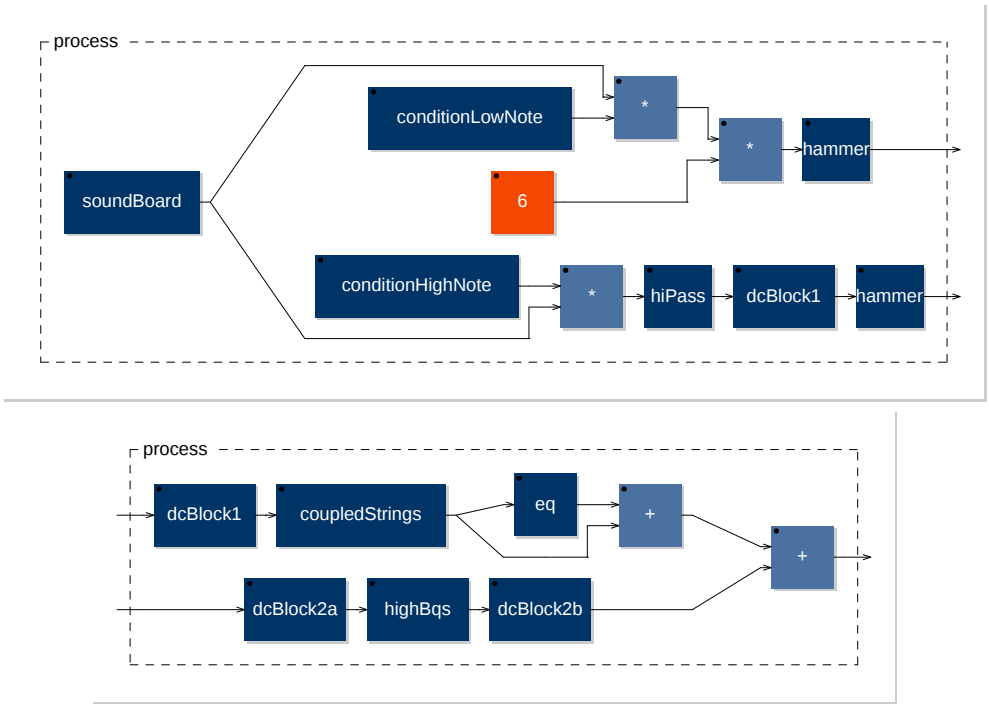


Figure A.6: Commuted Piano Algorithm Drawn by FAUST Using `faust2svg`.

A.6 Using a FAUST-STK Model With Gesture-Following Data

Parameter values are very important when dealing with physical modeling (see §1.5). Indeed, even if in most cases it is possible to produce nice sounds with static values for each parameter, the sound quality can be improved a lot by using dynamic values that can describe better the state of the model as a function of the note and the amplitude being played.

E. Maestre worked during his PhD on modeling the instrumental gesture for the violin [108] at the MTG.[3] With his help, it was possible to modify the algorithm of the bowed instrument from the STK in order to make it compatible with gesture data. The following changes were performed on the model:

- the ADSR used to control the bow velocity was removed;
- a "force" parameter that controls the slope of the bow table was added;
- a switch was added at the output of the bow table;
- we created a four-string violin where it is possible to modify the value of the parameters of each string independently;
- the simple body filter was replaced by a bank of biquad filters that impart a violin body response on the generated sound;
- an improved reflection filter also based on a bank of biquads is used.

The FAUST code was used to create a Pure Data plug-in. The gesture data for each physical parameter (note frequencies, bow position, bow velocity, bow force, and number of the string to be used) of the violin model were placed in separated text files that can be used in a PD patch. In the example shown in Figure A.7, the values are changed every 4.167 milliseconds. The gesture dataset used plays a traditional Spanish song called Muiñeira.

[3] Music Technology Group, University Pompeu Fabra, Barcelona (Spain).

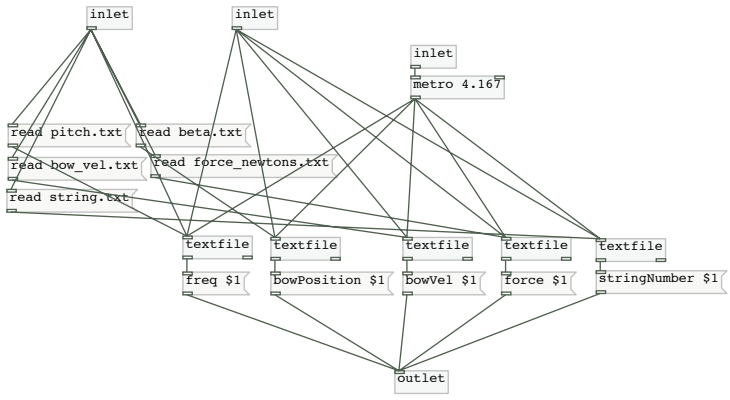


Figure A.7: Pure Data Sub-Patch Used to Send the Gesture Data for Muiñeira in the Faust Generated Plug-In.

Appendix B

Bell Modeling Using `mesh2faust`

Bells can be considered as quasi-linear systems and can be successfully synthesized using modal synthesis.[1] The acoustics of bells is well understood [158] and bell founders have been using FEM for decades to tune bells before making them [14].

In this section, we model a church bell after Rossing's elliptical arc approach using the framework described in the previous sections, and we compare the results of our system with the one published in his paper [158].

A Bezier curve was drawn on top of Rossing's church bell profile in Inkscape and was exported to OpenSCAD using the extension presented in §6.3.4. The radius of the bell was set to be 351mm, as in Rossing's paper. The result of this operation was a high definition mesh with about 25E4 vertices (see Figure B.1). This number was arbitrarily chosen to provide a good balance between performance and quality.

This high density mesh was restructured in MeshLab using the technique described in §6.3.4 and down-sampled to a lower definition mesh with 15E3 vertices (see Figure B.2).

This mesh was fed into `mesh2faust` with material parameters corresponding to bell metal [14] (Young's Modulus: 1.05E11 N/m^2, Poisson's Ratio: 0.33, and Density: 8600 kg/m^3). The results of the FEM modal analysis are presented in Table B.1 and plotted in Figure B.3.

Figure B.1 compares the theoretical "ideal" partial ratios to prime with the one computed by `mesh2faust` (the computed frequency of the undertone partial is 490.25 Hz). The modes naming conventions are the same as the one used by Rossing [158].

We can see that the FEM modes respect relatively well the theoretical mode hierarchy, resulting in very realistic synthesized sounds.

The same procedure was applied for a wide range of bells (e.g., carillon bells, hand bells, church bells from different countries, etc.) and the corresponding generated functions were added to the FAUST Physical Modeling Library. Web app synthesizers were created from these functions and are

[1] This appendix is partly based on [121]. Some sections and figures of this paper were copied verbatim here.

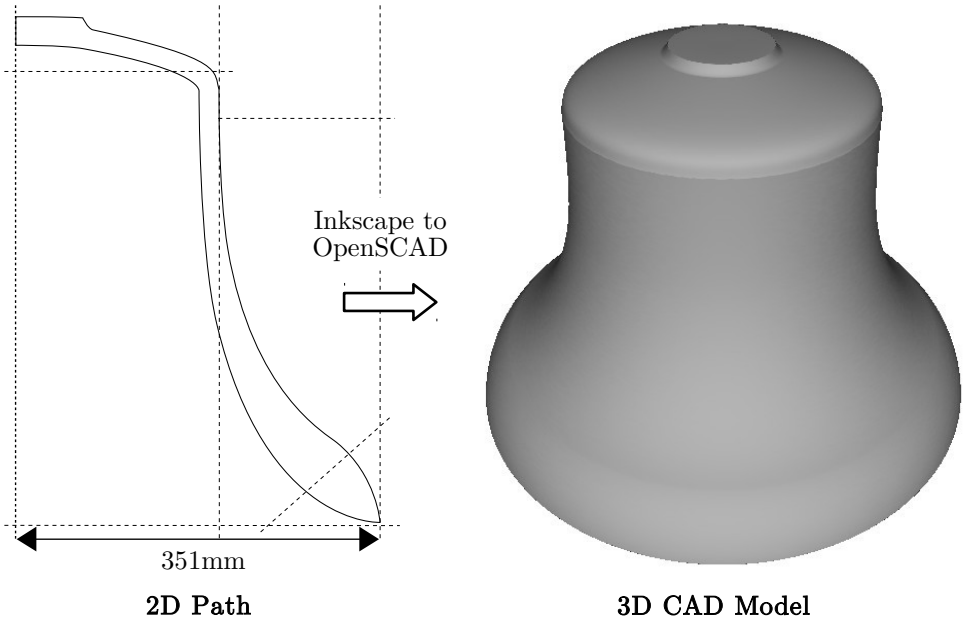


Figure B.1: Church Bell Cross Section and Corresponding CAD Model Modeled After Rossing's Elliptical Arc Approach.

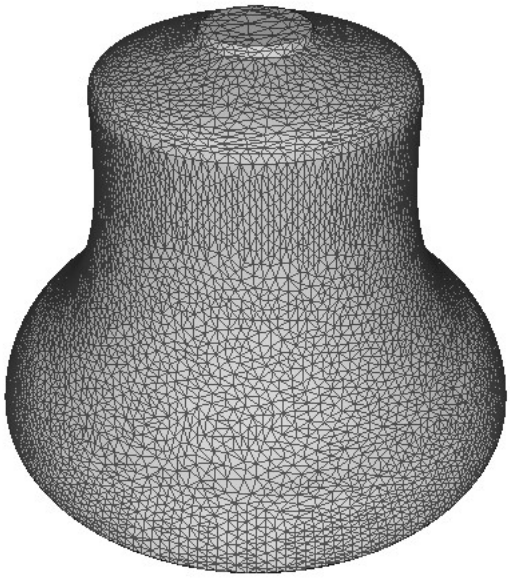


Figure B.2: Mesh Generated in MeshLab After Quadric Edge Collapse Decimation and Laplacian Smoothing.

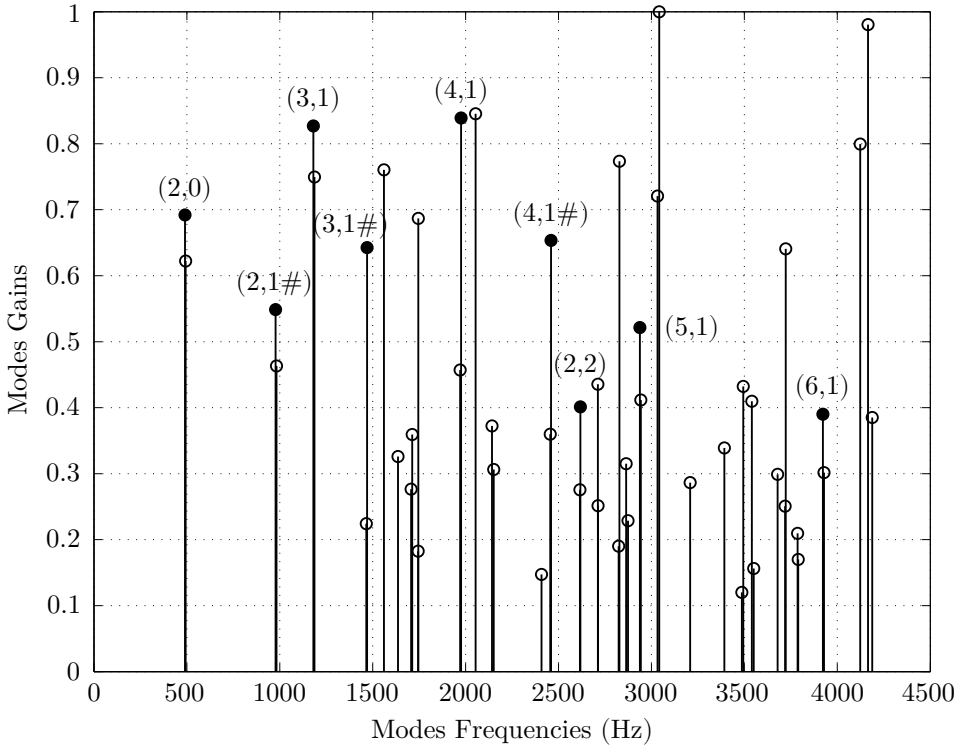


Figure B.3: First Fifty Modes Computed by MTF for the Bell Mesh Presented in Table B.1 For an Excitation Position Matching the Strike Position of the Clapper Inside the Bell.

Modes	Names of Partials	Theoretical Ratios	MTF Ratios
(2,0)	Hum, undertone	0.500	0.500
(2,1#)	Fundamental, prime	1.000	1.012
(3,1)	Tierce, minor third	1.200	1.208
(3,1#)	Quint, fifth	1.500	1.6
(4,1)	Nominal, octave	2.000	1.980
(4,1#)	Major Third, deciem	2.500	2.451
(2,2)	Fourth, undeciem	2.667	2.610
(5,1)	Twelfth, duodeciem	3.000	3.073
(6,1)	Upper octave	4.000	4.11

Table B.1: Comparison Between the Theoretical "Ideal" Mode Ratios to Prime With the Ones Computed by `mesh2faust` for the Bell Mesh Presented in Figure B.2.

available online.[2]

[2] `https://ccrma.stanford.edu/~rmichon/pmFaust/#bells/`

Appendix C

FPML Functions Listing

The FAUST Physical Modeling Library (`physmodels.lib`) is part of the FAUST distribution [1] and can be found in its `/libraries` folder. This appendix gives a listing of the functions of this library (as of Sept. 20, 2017).[2]

Some of the functions listed here were contributed by other people. In particular, Michael Olsen wrote most of the elements in the *Vocal Synthesis* section.

Global Variables	`rTerminations`
`speedOfSound`	`closeIns`
`maxLength`	`closeOuts`
Conversion Tools	`endChain`
`f2l`	**Basic Elements**
`l2f`	`waveguideN`
`l2s`	`waveguide`
Bidirectional Utilities	`bridgeFilter`
`basicBlock`	`modeFilter`
`chain`	**String Instruments**
`inLeftWave`	`stringSegment`
`inRightWave`	`openString`
`in`	`nylonString`
`outLeftWave`	`steelString`
`outRightWave`	`openStringPick`
`out`	`openStringPickUp`
`terminations`	`openStringPickDown`
`lTerminations`	`ksReflexionFilter`

Table C.1: FAUST Physical Modeling Library Functions (1).

[1] `https://github.com/grame-cncm/faust/`

[2] The detailed documentation of these functions can be found online: `http://faust.grame.fr/library.html#physmodels.lib/`

String Instruments (Continue)	clarinetMouthPiece
rStringRigidTermination	brassLips
lStringRigidTermination	fluteEmbouchure
elecGuitarBridge	openTube
elecGuitarNuts	reedTable
guitarBridge	fluteJetTable
guitarNuts	brassLipsTable
idealString	clarinetReed
ks	clarinetMouthPiece
ks_ui_MIDI	brassLips
elecGuitarModel	fluteEmbouchure
elecGuitar	wBell
elecGuitar_ui_MIDI	fluteHead
guitarBody	fluteFoot
guitarModel	clarinetModel
guitar	clarinetModel_ui
guitar_ui_MIDI	clarinet_ui
nylonGuitarModel	clarinet_ui_MIDI
nylonGuitar	brassModel
nylonGuitar_ui_MIDI	brassModel_ui
Bowed String Instruments	brass_ui
bowTable	brass_ui_MIDI
violinBowTable	fluteModel
bowInteraction	fluteModel_ui
violinBow	flute_ui
violinBowedString	flute_ui_MIDI
violinNuts	**Exciters**
violinBridge	impulseExcitation
violinBody	strikeModel
violinModel	strike
violinModel_ui	pluckString
violin_ui_MIDI	blower
Wind Instruments	blower_ui
openTube	**Modal Percussions**
reedTable	djembeModel
fluteJetTable	djembe
brassLipsTable	djembe_ui_MIDI
clarinetReed	marimbaBarModel

Table C.2: FAUST Physical Modeling Library Functions (2).

Modal Percussions (Continue)	germanBell_ui
marimbaResTube	russianBellModel
marimbaModel	russianBell
marimba	russianBell_ui
marimba_ui_MIDI	standardBellModel
churchBellModel	standardBell
churchBell	standardBell_ui
churchBell_ui	**Vocal Synthesis**
englishBellModel	formantFilter
englishBell	SFFormantModel
englishBell_ui	SFFormantModel_ui
frenchBellModel	SFFormantModel_ui_MIDI
frenchBell	**Misc Functions**
frenchBell_ui	allpassNL
germanBellModel	
germanBell	

Table C.3: FAUST Physical Modeling Library Functions (3).

Appendix D

Extending FAUST's Block-Diagram Algebra Towards Multidimensionality

In this appendix, we present an unimplemented extension of FAUST's block-diagram algebra aimed at providing a better support for physical modeling. This unpublished work has been done at GRAME[1] in Lyon in May 2016 in collaboration with Yann Orlarey.

While FAUST can currently be used to implement any waveguide or modal physical model, creating multidimensional block diagrams where signals can go in any direction is not optimal. The FAUST Physical Modeling Library partially solves this problem by providing a set of functions to implement bidirectional block diagrams (see §6.2.1). Even though this allows programmers to design physical models at a very high level simply by connecting instrument parts together, the resulting block-diagrams are difficult to read due to their inherent left to right orientation. For example, the following construction made with FPML will result in the diagram presented in Figure D.1:

```
wg(x1) = chain(waveguide : input(x1) : out : waveguide);
```

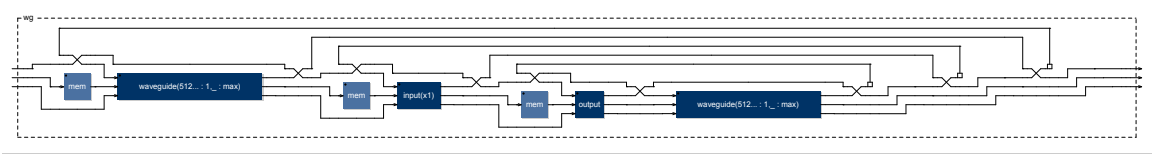


Figure D.1: FAUST-Generated Block Diagram of a Simple Physical Modeling Block Assembled Using FPML.

In order to better describe and represent bi-directional connections, the proposed extension will

[1] http://grame.fr/

allow inputs and outputs on all four sides of a diagram.[2] It will also introduce a new vertical composition operation, the possibility to rotate expressions and a series of new routing primitives.

D.1 Conventions

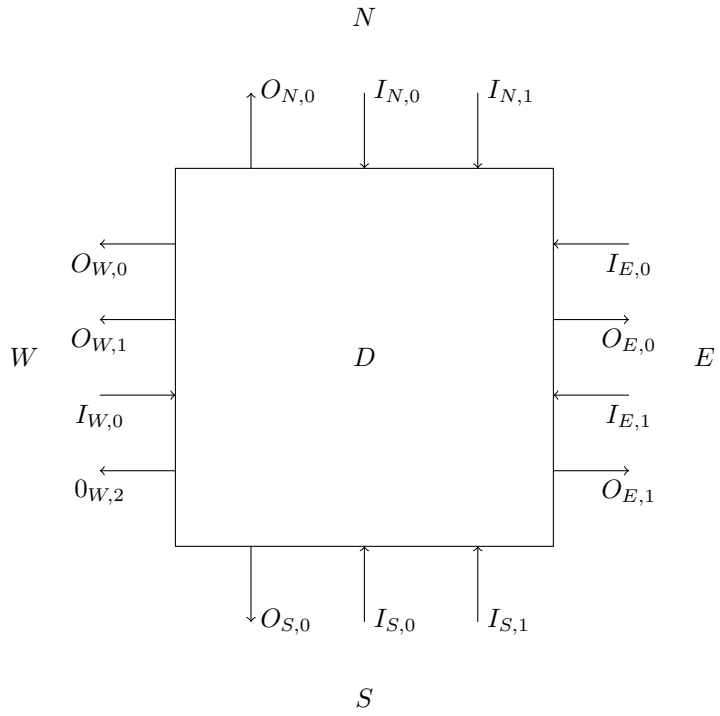


Figure D.2: Generic Extended Diagram Block.

We distinguish the sides as well as the inputs and outputs of a block in a diagram (see Figure D.2) by naming and numbering them, respectively.

Definition 1 (Side names)**.** The sides are named: *West*, *North*, *East*, *South* and are always enumerated in that order.

Definition 2 (Number of inputs and outputs)**.** We introduce 8 functions: $I_{W,N,E,S}$ and $O_{W,N,E,S}$ to denote the number of inputs and outputs of each side of a diagram. For example, in Figure D.2, we have: $I_W(D) = 1$, $O_W(D) = 3$, $I_N(D) = 2$, etc.

Definition 3 (Numbering of inputs and outputs)**.** Inputs and outputs of the North and South sides are numbered from left to right starting from 0. Inputs and outputs of the West and East sides are

[2]Here *(block-)diagram* must be understood as the graphical representation of a Faust expression.

numbered from top to down starting from 0. For example, $I_{E,0}(D)$ designates the first input on the east side of the diagram D, and $O_{W,2}(D)$ the third output of the west side.

Definition 4 (Topological types)**.** The *topological type* of a diagram describes its connectivity. It indicates the number of inputs and the number of outputs of the *West*, *North*, *East*, and *South* sides. For example, the diagram D of Figure D.2 is of type:

$$D : (1, 2, 2, 2) \to (3, 1, 2, 1)$$

For any diagram D, we have:

$$D : (I_W(D), I_N(D), I_E(D), I_S(D)) \to (O_W(D), O_N(D), O_E(D), O_S(D))$$

With this new convention, Faust primitive + (see Figure D.3) has type $(2, 0, 0, 0) \to (0, 0, 1, 0)$.

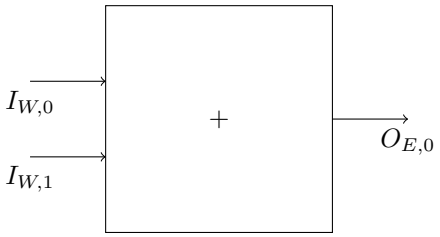


Figure D.3: The + Primitive.

D.2 Horizontal Composition

Sequential composition `A:B`, split composition `A<:B`, and merge composition `A:>B`, are subsumed under a more general *horizontal composition* operation also notated `A:B`. Horizontal composition connects the East side of `A` to the West side of `B`. The East outputs of `A` are connected to the West inputs of `B`, and the West outputs of `B` are connected to the East inputs of `A`. The number of inputs and outputs must be either the same or a multiple of each other. In this case, the split or merge rule apply. In other words, the *horizontal composition* `A:B` is possible if we have $n, m, p, q \in \mathbf{N}^*$ such that:

$$n = 1 \vee m = 1$$
$$p = 1 \vee q = 1$$
$$n.I_E(A) = m.O_W(B)$$
$$p.O_E(A) = q.I_W(B)$$

In the example presented in Figure D.4, we have:

$$\begin{aligned} I_E(A) &= 2 \\ O_W(B) &= 2 \\ O_E(A) &= 2 \\ I_W(B) &= 1 \end{aligned}$$

Thus, *horizontal composition* is possible. Moreover we have a merge of the East outputs of A into the West input of B because $O_E(A) = 2 \times I_E(B)$.

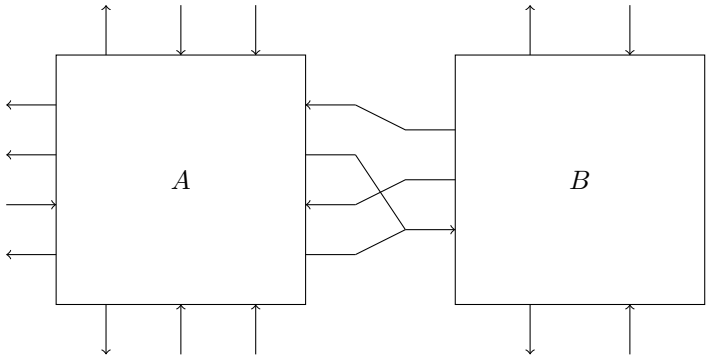


Figure D.4: Horizontal Composition With Implicit Merge: $O_E(A) = 2 \times I_W(B)$.

Horizontal composition `A:B` is such that:

$$\begin{aligned} I_W(A,B) &= I_W(A) \\ I_N(A,B) &= I_N(A) + I_N(B) \\ I_E(A,B) &= I_E(B) \\ I_S(A,B) &= I_S(A) + I_S(B) \end{aligned}$$

and

$$\begin{aligned} O_W(A,B) &= O_W(A) \\ O_N(A,B) &= O_N(A) + O_N(B) \\ O_E(A,B) &= O_E(B) \\ O_S(A,B) &= O_S(A) + O_S(B) \end{aligned}$$

D.3 Vertical Composition

Vertical composition `A||B`, is the equivalent of *horizontal composition* in the vertical direction. It connects the South side of `A` to the North side of `B`.

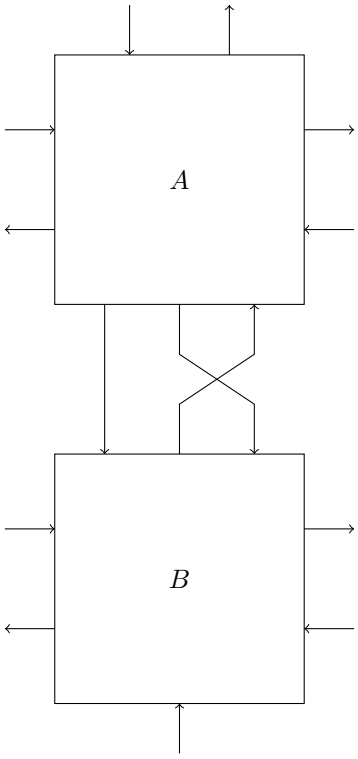


Figure D.5: Vertical Composition `A||B`.

The composition is possible if and only if we have $n, m, p, q \in \mathbf{N}^*$ such that:

$$n = 1 \vee m = 1$$
$$p = 1 \vee q = 1$$
$$n.I_S(A) = m.O_N(B)$$
$$p.O_S(A) = q.I_N(B)$$

Vertical composition `A||B` is such that:

$$
\begin{aligned}
I_W(A,B) &= I_W(A) + I_W(B) \\
I_N(A,B) &= I_N(A) \\
I_E(A,B) &= I_E(A) + I_E(B) \\
I_S(A,B) &= I_S(B)
\end{aligned}
$$

and

$$
\begin{aligned}
O_W(A,B) &= O_W(A) + O_W(B) \\
O_N(A,B) &= O_N(A) \\
O_E(A,B) &= O_E(A) + O_E(B) \\
O_S(A,B) &= O_S(B)
\end{aligned}
$$

D.4 Parallel Composition

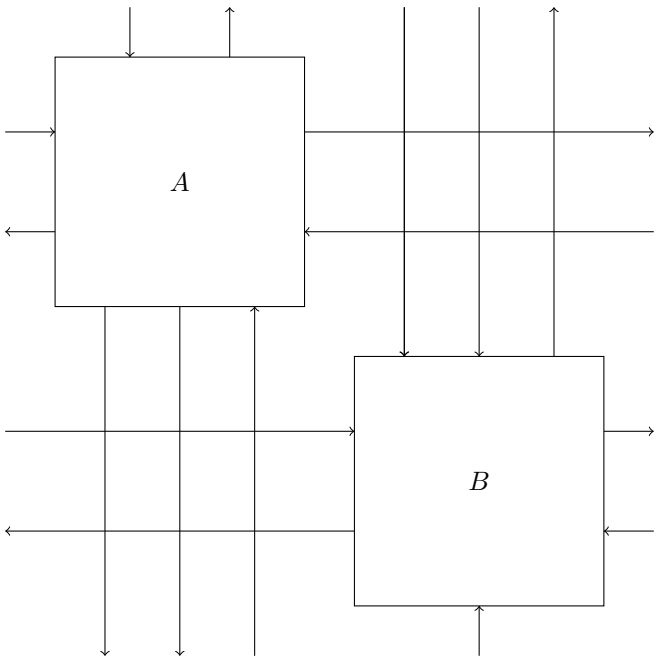


Figure D.6: Parallel Composition `A,B`.

Parallel composition `A,B` is always possible. It is such that:

$$I_W(A,B) = I_W(A) + I_W(B)$$
$$I_N(A,B) = I_N(A) + I_N(B)$$
$$I_E(A,B) = I_E(A) + I_E(B)$$
$$I_S(A,B) = I_S(A) + I_S(B)$$

and

$$O_W(A,B) = O_W(A) + O_W(B)$$
$$O_N(A,B) = O_N(A) + O_N(B)$$
$$O_E(A,B) = O_E(A) + O_E(B)$$
$$O_S(A,B) = O_S(A) + O_S(B)$$

D.5 Route Primitive

The `route()` primitive is used to describe small connection routes likes those in Figure D.7.

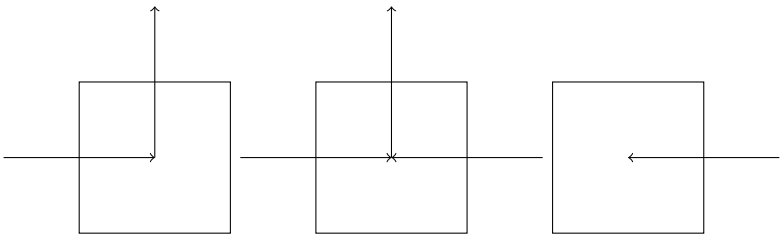


Figure D.7: `route("io..")`, `route("ioi.")` and `route(".i..")`.

Each route is defined with a 4-character string indicating if a side contains an input (letter "i"), an output (letter "o"), or nothing (letter "."). For example, `route("ioio")` correspond to Figure D.8 and is of type:

$$(1,0,1,0) \rightarrow (0,1,0,1)$$

Note that all the input signals are added together and the resulting signal is delivered to all outputs. A route with no inputs delivers a 0 signal to its outputs.

We have the following equivalences (see Figure D.9):

```
route("..o.")   = 0
route("i.o.")   = _
route("i...")   = !
```

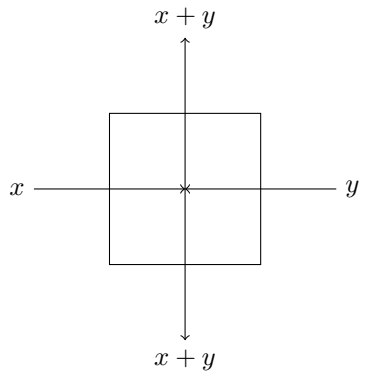


Figure D.8: `route("ioio")`.

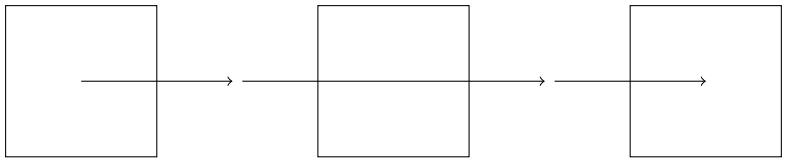


Figure D.9: `route("..o.")`, `route("i.o.")`, `route("i...")`.

D.6 Rotation

A diagram `D` can be rotated by -90° using the unary prefix operator `<*` or by +90° using the unary postfix operator `*>` (see Figure D.10).
If we have `D` with type $(a, b, c, d) \to (i, j, k, l)$ then `<* D` has type $(b, c, d, a) \to (j, k, l, i)$, and `D *>` has type $(d, a, b, c) \to (l, i, j, k)$.

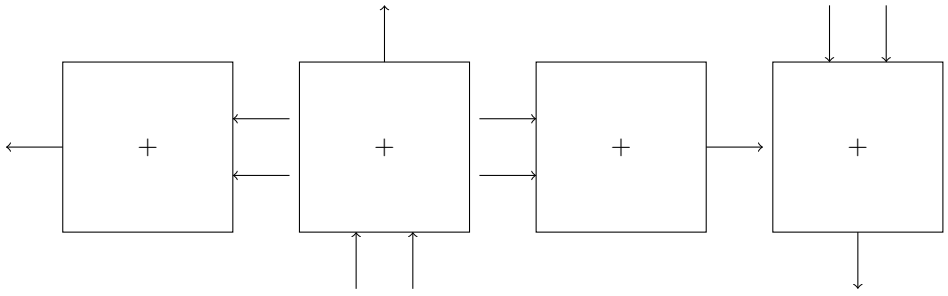


Figure D.10: `<*<*+`, `<*+`, `+`, and `+*>`.

We have the following equivalences:

```
                <*(A:B)   =   (<*B)||(<*A)
                <*(A||B)  =   (<*A):(<*B)

                (A:B)*>   =   (A*>)||(B*>)
                (A||B)*>  =   (B*>):(A*>)

          <*<*<*<*<*(A:B)  =  <*<*<*<*(<*B)||(<*A)
        <*<*<*<*(<*B)||(<*A)  =  <*<*<*((<*<*B):(<*<*A))
      <*<*<*((<*<*<*B):(<*<*<*A))  =  <*<*((<*<*<*<*A)||(<*<*<*<*B))
    <*<*((<*<*<*<*A)||(<*<*<*<*B))  =  (<*<*<*<*<*A):(<*<*<*<*<*B)
     (<*<*<*<*<*A):(<*<*<*<*<*B)  =  A:B
```

D.7 Examples

This section provides some examples illustrating the extended FAUST block-diagram algebra presented in the previous sections.

D.7.1 General Case: Feedback

A "typical" FAUST feedback loop

```
+~_
```

is equivalent to

```
feedback = route("..io") || route("iio.") : (_' : route("o..i")) ||
   route("ioo.");
```

While this is much more verbose, it is just presented as an example here and programmers are expected to keep using the ~ composition.

D.7.2 Physical Modeling

In this section, we reformulate some of the elements of the FAUST Physical Modeling Library (see §6.2) using the extended FAUST algebra.

A simple waveguide block (two parallel delay lines going in opposite directions) equivalent to `pm.waveguide()` can be implemented as:

```
waveguide(l) = <*<*(@(l)),@(l);
```

where `l` is the delays length.
A block equivalent to `pm.rTermination()` closing a bidirectional chain (e.g., `waveguide`) on its right side can be expressed with:

```
terminationUp(a) = route("o..i") || <*a || route("io..");
```

where `a` can be any function with one input and one output (e.g., a filter).
Similarly, a block equivalent to `pm.lTermination()` closing a bidirectional chain on its right side can be written as:

```
terminationDown(a) = route("..io") || a*> || route(".io");
```

where `a` can be any function with one input and one output (e.g., a filter).
Additional inputs and outputs can be easily added to the previous blocks:

```
terminationUpOutput(a) = route("o..i") || <*a || route("ioi.");
terminationDownInput(a) = route("i.io") || a*> || route(".io");
```

and simple physical elements can be created using them (see §6.2):

```
rigidTerminationUp = terminationUp(*(-1));
rigidTerminationUpOutput = terminationUpOutput(*(-1));
rigidTerminationDown = terminationDown(*(-1));
```

Finally, a simple "ideal string" model with rigid terminations equivalent to `pm.idealString` can be implemented using the previous elements as:

```
input = _,cross : route("o.ii") || route("io.o") || route("iio.");
idealString(freq,pos) = rigidTerminationDown : waveguide(n1) : input :
  waveguide(n2) : rigidTerminationUpOutput
with{
  l = SR/freq/2;
  n1 = l*pos;
  n2 = l*(1-pos);
};
```

D.7.3 Transformer-Normalized Digital Waveguide Oscillator

A particularly hard algorithm to implement with the current version of FAUST is that of the *tranformer-normalized digital waveguide oscillator* [177]. This is mostly due to the fact that it involves both feedback and signals crossing each others between the feed-forward and the feedback portion of the system. Such a filter can be easily expressed using the extended block-diagram algebra presented in this appendix:

```
wgOsc(g,c) = (route("..oi") || <*(_') || route(".oi."))
  : ((route("..oi") : route("i..o")) // upper feed forward
  || ( *(g) : route("ioo.") : route("i.oi")
```

```
   : *(c) : route("i.oo") : route("iio."))
|| (route("oii.") : route("ooi."))) : (route("i..o")
|| (_')*> || route("oi.."));
```

Appendix E

Hybrid Woodwind Instrument and Active Control

For most musical instruments, the excitation is the element that has the greatest number of parameters to control (see §2.1). The properties of the bore of a clarinet for example (but this also applies to most of the woodwind and brass instruments) can only be modified by tone holes, that are very discrete controllers (they can be opened or closed or half closed in some cases). On the other hand, the interactions between the mouthpiece and the player are extremely complex and difficult to simulate on a computer. This problem has been presented in §1.1 and various kind of solutions have been proposed.

In this appendix, we present an experimental example of a hybrid instrument involving the use of bidirectional virtual/physical connections (see §6.1.3).[1] A 3D printed mouthpiece combined with a system based on a piezo sensor and a loudspeaker are used to drive a simple physical model of a generic bore. We discuss the results of various experiments we carried out and try to provide solutions to the problems encountered.

E.1 General Concept

The use of acoustic excitations to drive virtual physical models has been described in §2.1, §2.2, and §2.3. In the case of percussion and plucked string instruments, the process is very simple as any kind of audio impulse can be fed into a waveguide to excite it (see §6.1.3).

While this technique can be theoretically applied to any physical model of musical instruments, things become complicated when the excitation signal partly determines the pitch of the sound

[1] This work is based on a series of of experiments conducted at CCRMA in 2012/2013 presented in an unpublished report written in collaboration with John Granzow [120]. Since this work predates the creation of the framework presented in this dissertation, it doesn't use it.

generated by the system. This is the case for woodwind and brass instruments for example where there is a coupling between the mechanism that produces the excitation and the size of the bore of the instrument [59]. In other words, the length of the bore determines the frequency of vibration of the reed on a saxophone and of the lips on a trumpet. This is due to traveling waves reflected by the end of the pipe of the instrument whose distance from the reed can be adjusted with the tone holes in the saxophone case and with the pistons on the trumpet. The goal of the project presented in this appendix is to leverage 3D printing technology to that of physical modeling to create a hybrid single reed instrument whose 3D printed mouthpiece is connected and coupled to a virtual computer modeled bore. The most challenging part of this work is to create a feedback system that reproduces the reflexion pulse created by the end of the bore and send it to the mouthpiece using a simple speaker. The advantages of such a system are multiple. It would make it possible to design hybrid instruments whose response was partly independent of its physical geometry. The player would have full control of the generated sound and experience some of the vibrotactile stimulation of a real instrument.

E.2 First Model and Experiments

E.2.1 3D Printed Mouthpiece and Feedback System

Figure E.1 depicts the computer model of a tenor saxophone mouthpiece fabricated for our experiment. It's design is based on that of a real mouthpiece. We tried the printed mouthpiece with a "real" saxophone and it competed with the original. The other object depicted in Figure E.1 connects to the mouthpiece and houses a speaker. Figure E.2 presents the printed version of these elements with a bamboo reed mounted on the mouthpiece and the speaker attached to the other section. We used this system to conduct most of the experiments that are presented in this report.

Originally, a microphone was placed inside the mouthpiece to pick up the audio excitation created by the reed. This was obviously a very bad solution as the microphone was too closed to the speaker, creating feedback. We first tried to solve this problem by replacing the microphone by a piezo film glued on the reed. However, the impedance between the reed and the piezo was too high and we were not able to get usable signals. The solution we adopted was to glue a very small piezo disc (whose much stiffer than a piezo film) on the reed. This enabled us to get high quality signals of the excitation without creating feedback with the speaker.

The position of the piezo on the reed was chosen so that it is as close as possible from the vibrating area without affecting the sound quality. As the piezo stays in a very humid environment, we had to find a way to protect it from becoming too wet. For that, we drilled a small hole in the reed with an identical diameter than the one of the piezo. Finally, we covered the piezo with a protection paste (see Figure E.3).

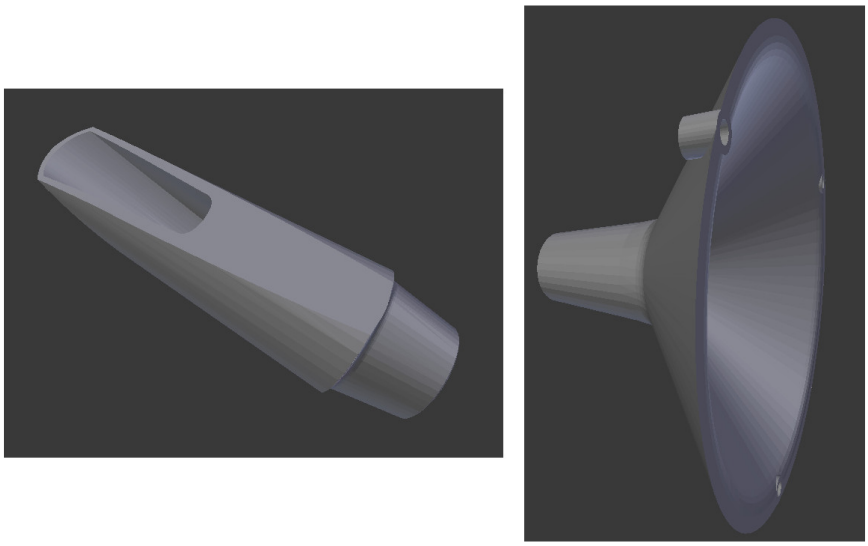


Figure E.1: 3D Model of Our First Mouthpiece Feedback System.

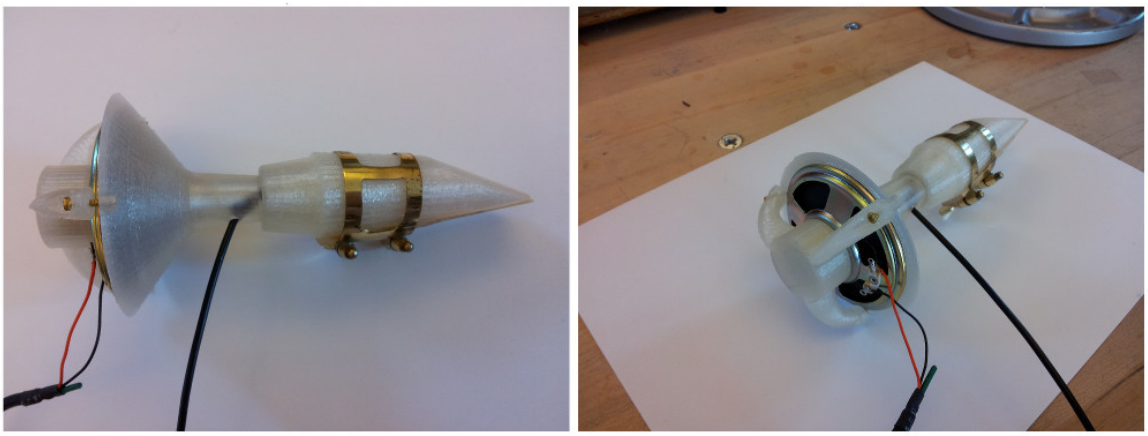


Figure E.2: 3D Printed Mouthpiece Feedback System.

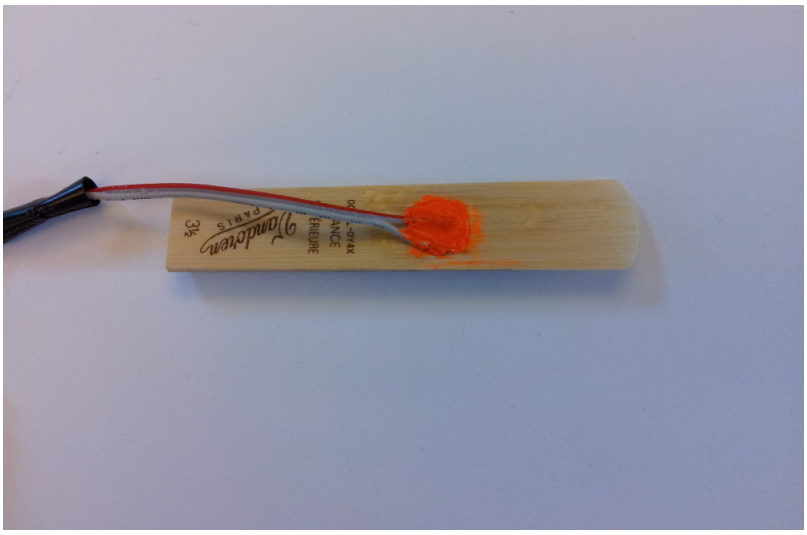


Figure E.3: Saxophone Reed With a Piezo Disc Glued on It.

E.2.2 Physical Model

The physical model we used for our experiments is based on a simple waveguide (see §1.5). Figure E.4 presents the top level block diagram of this algorithm. The input signal (`x`), for instance the signal generated by the piezo on the reed is fed into the system. The reflection filter is a simple one-zero filter. The output of this filter is retrieved and sent to the speaker placed in front of the mouthpiece (see Figure E.2). Before that, it is scaled and it goes through a delay line allowing us to control the phase of the signal (however, this proved to be totally useless as changing the phase had absolutely no effect on the behavior of the reed).

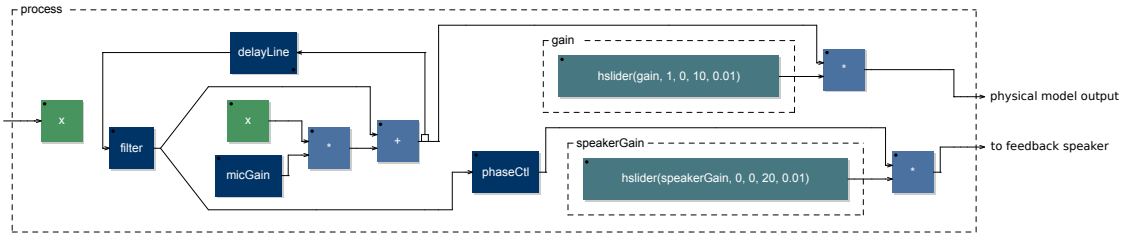


Figure E.4: Block Diagram of the FAUST Physical Model as Drawn by `faust -svg` Implementing the Virtual Portion of Our Hybrid Woodwind Instrument.

E.2.3 First Experiment

In order to achieve the best latency with the audio interface we used,[2] we had to use a very high sampling rate of 192kHz with buffers of 128 samples and 2 periods per buffer. Unfortunately, we were never able to use the PD-externals compiled from the FAUST code at such a high sampling rate. The trick we used to compensate for this was to compile our FAUST code as jack applications that we controlled from PD[3] sending OSC messages. We chose PureData to carry out this task because it is easy to quickly create controllers in this environment.

For our first experiment, we tried the system with different lengths for the virtual bore, using a very high amplitude mouthpiece feedback signal. While we were not able to change the frequency of vibration of the reed, the saxophone player found it very hard in some cases to hold the pitch and felt really disturbed by the behavior of the reed when the length of the bore was set such that it approached the natural frequency of vibration of the reed.

After these primarily experiments, we figured out that the latency introduced by our audio interface, even though it was very small (around 2ms with the configuration described at the beginning of this section), was one of the problems in the system. Indeed, a latency of $2ms$ in an environment where speed of sound is $340m/s$ corresponds to a delay of $68cm$ which is huge in the case of musical instruments. Also, we thought it would be interesting to do more basic experiments like sending a square wave signal into the mouth piece. The implementation and the results of these experiments are presented in the two following sections.

E.3 Square Wave Experiments

The waveform of an audio signal created by a clarinet is similar to a square wave. For this reason, we thought it would be interesting to try to send a square wave audio signal at different amplitudes and frequencies in the mouthpiece while blowing in it. In our first experiment, we tried to send a high amplitude signal where the frequency of the square wave was the same as the natural frequency of vibration of the reed (665Hz) and slowly increased it to 1000Hz. We recorded the signal from the piezo on the reed and plotted the spectrogram (see Figure E.5).

We can see that after 3 seconds, the frequency of vibrations of the reed starts to be modulated, creating a vibrato effect that evolves into some kind of frequency modulation behavior after 12 seconds. Therefore, somehow the square wave acts as a modulating signal on the reed which can be compared to a carrier in this case.

Another experiment where a square wave signal with a constant frequency of 670Hz and an increasing gain is sent to the mouthpiece. The spectrogram of the signal recorded on the reed is plotted in Figure E.6 along with the waveform of the square wave. We can see that increasing the

[2] We used a Roland UA-101 for all our experiments.
[3] PureData: `http://puredata.info/`.

gain of the square wave broadens the range of the modulation, exactly like a frequency modulation synthesizer when the index of modulation is changed.

These results prove that the frequency of vibration of the reed can be modulated by the signal coming from a speaker if the amplitude of the wave it creates is big enough.

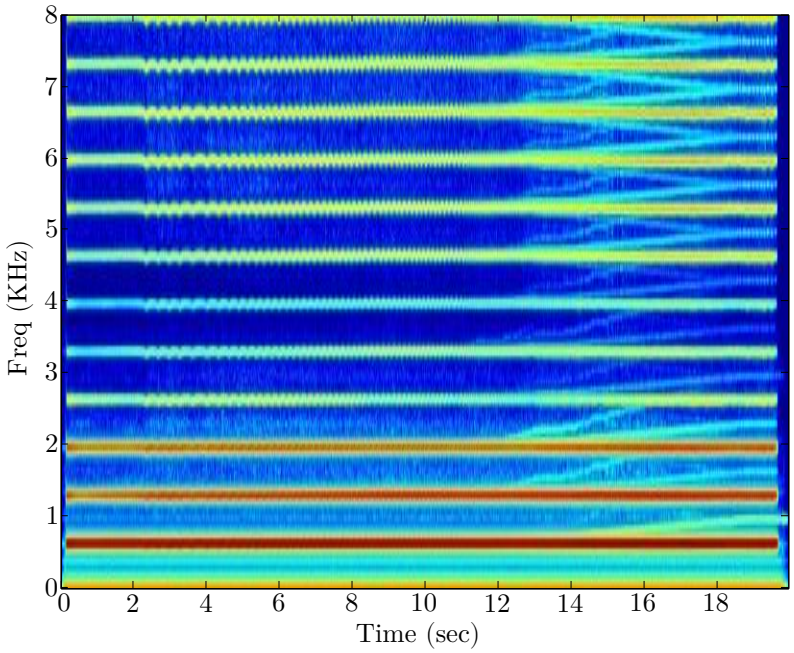


Figure E.5: Spectrogram of the Signal Measured on the Reed When "Reflecting" a Square Wave With Frequency Evolving From $665Hz$ to $1kHz$.

E.4 Limited "Zero-Latency" System

One way to achieve $0ms$ latency in our system was to create a model where the latency of the audio interface is used as the delay line of the waveguide. A diagram of this model can be seen in Figure E.7. We basically take the signal from the reed and use the same filter as the one used in the model depicted in Figure E.4 and send it back to the speaker. Some delay can be added to increase the size of the simulated bore whose minimum length is defined by the latency of the audio interface ($65cm/2 = 32.5cm$).

We tried to use this model by slowly increasing the length of the virtual bore starting at $32cm$ (the minimum length we can achieve with this configuration) and ending at $132cm$. The spectrogram of the signal recorded from the reed can be seen in Figure E.8.

We can see that as the length of the virtual bore is increased, the frequency of the reed is shifted

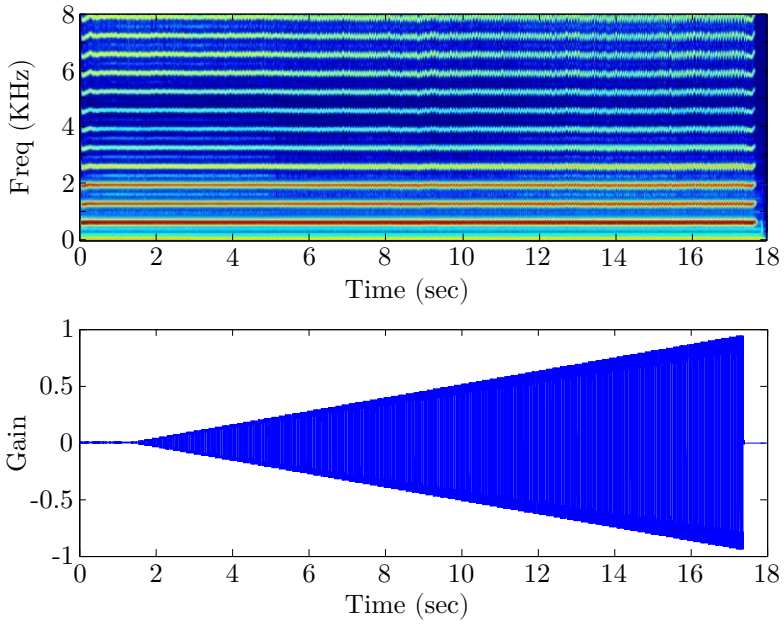


Figure E.6: Spectrogram of the Signal Measured on the Reed When "Reflecting" a Square Wave With Constant Frequency ($670Hz$) and Increasing Amplitude.

down by a semitone and comes back to its original state every time the minimum size of the bore is doubled (every time 32cm is added). This proves that despite the fact that our system is not able to control the frequency of vibration of the reed, it can shift it a little bit.

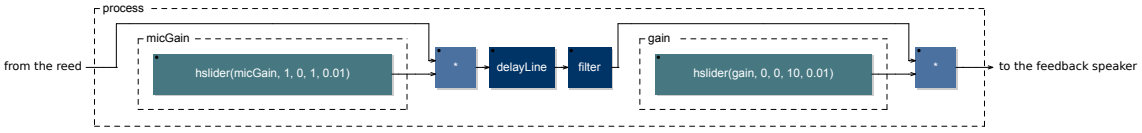


Figure E.7: Block Diagram of the FAUST Physical Model as Drawn by `faust -svg` Implementing the "Zero-Latency" Bore Model.

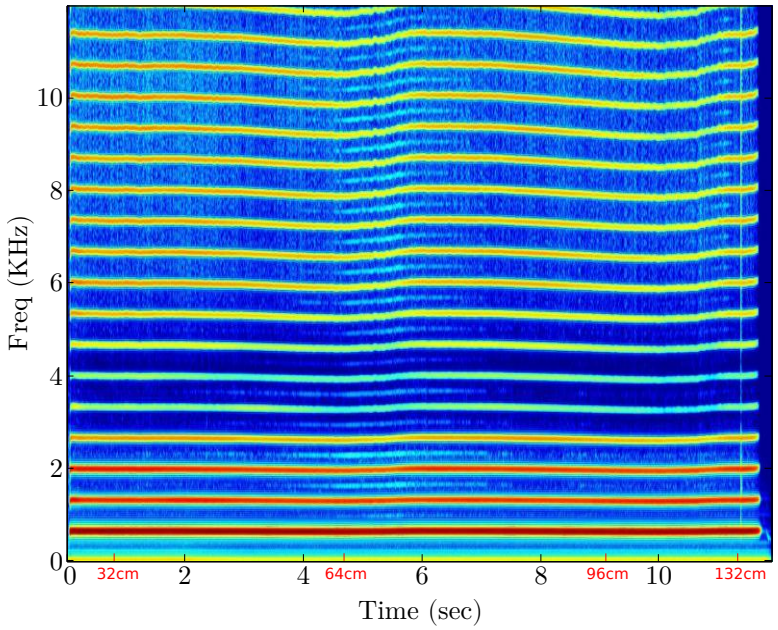


Figure E.8: Spectrogram of the Signal Measured on the Reed When Increasing the Length of the Virtual Bore of Our "Zero-Latency" System From 32cm to 132cm.

E.5 Additional Experiments and Future Directions

E.5.1 Further Reducing Latency

While the trick we presented in the previous chapter to get rid of latency issues works, it is not optimal because it imposes a minimal length to the virtual bore. Moreover, it increases the unpredictability of the model as the addition of the feedback signal with the reed signal is not carried out on the computer but in the real world.

The only solution to this problem is to reduce the latency of the system (ADC -> computing -> DAC), potentially using specialized hardware.

E.5.2 Improving the Mouthpiece Feedback System

We think that our system is limited by the fact that the "real" reflected wave created by the mouthpiece and the small pipe that links it to the speaker chamber arrives earlier and is stronger than the virtual reflection wave generated by the speaker. Thus, we designed a new system (see Figure E.9) where we will try to drive the wave created by the reed in a pipe. An acoustical damping material will be placed at the end of the pipe to cancel this wave in order to prevent it from being reflected.

The speaker will be placed at the middle of the pipe and thus will be able to send the virtual reflection before the "remains" of the damped wave reach the reed. We hope this solution will work or provide new clues for the next step.

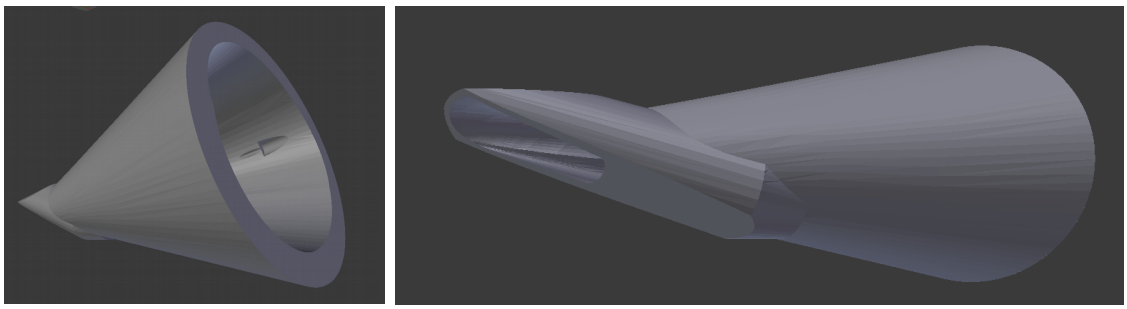


Figure E.9: Future Mouthpiece Feedback System.

Bibliography

[1] Faust libraries documentation. On-line. http://faust.grame.fr/library.html.

[2] CCRMA 2016 composed instrument workshop: Intersections of 3D printing and digital audio for mobile platforms. Web-Page, 2016. URL: https://ccrma.stanford.edu/~rmichon/composedInstrumentWorkshop/.

[3] CCRMA composed instrument workshop 2016 – final projects. On-line – YouTube Video, July 2016. https://www.youtube.com/watch?v=YOWMh66Etck.

[4] CCRMA summer workshop - intersections of 3D printing and mobile audio - teaser video. On-line – YouTube Video, March 2016. https://www.youtube.com/watch?v=dGBDrmvG4Yk.

[5] Jean-Marie Adrien. The missing link: Modal synthesis. In *Representations of Musical Signals*, chapter The Missing Link: Modal Synthesis, pages 269–298. MIT Press, Cambridge, USA, 1991.

[6] Roberto Mario Aimi. *Hybrid Percussion: Extending Physical Instruments Using Sampled Acoustics*. PhD thesis, Massachusetts Institute of Technology, USA, 2007.

[7] Ercan Altinsoy and Sebastian Merchel. Electrotactile feedback for handheld devices with touch screen and simulation of roughness. *IEEE Transactions on Haptics*, 5(1):6–13, January 2012.

[8] The Ampeg Company, Elkhart, Indiana. *Lyricon Wind Synthesizer Driver – Owner's Manual*, 1978.

[9] Daniel Arfib, Jean-Michel Couturier, and Loic Kessous. Expressiveness and digital musical instrument design. *Journal of New Music Research*, 34(1):125–136, 2005.

[10] Rolf Bader, Jan Richter, Malte Münster, and Florian Pfeifle. *Digital Guitar Workshop Manual*. Hamburg University, 2014.

[11] Nicholas J Bailey, Theo Cremel, and Alex South. Using acoustic modelling to design and print a microtonal clarinet. In *Proceedings of the 9th Conference on Interdisciplinary Musicology – CIM14*, Berlin, Germany, 2014.

[12] Bruce Banter. Touch screens and touch surfaces are enriched by haptic force-feedback. *Information Display*, 26(3):26–30, 2010.

[13] Stephen Barrass. Digital fabrication of acoustic sonifications. *Journal of the Audio Engineering Society*, 60(9):709–715, September 2012.

[14] Dariusz Bartocha and Czeslaw Baron. Influence of tin bronze melting and pouring parameters on its properties and bells' tone. *Archives of Foundry Engineering*, 16(4):17–22, 2016.

[15] Marc Battier. *Les Musiques électroacoustiques et l'environnement informatique*. PhD thesis, University of Paris X, Nanterre, France, 1981.

[16] Frauke Behrendt. *Handymusik. Klangkunst und 'mobile devices'*. Electronic Publication, Osnabrück, 2004.

[17] Edgar Berdahl. *Application of Feedback Control to Musical Instrument Design*. PhD thesis, Stanford University, USA, 2009.

[18] Edgar Berdahl. An introduction to the Synth-A-Modeler compiler: Modular and open-source sound synthesis using physical models. In *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[19] Edgar Berdahl and Alexandros Kontogeorgakopoulos. The firefader: Simple, open-source, and reconfigurable haptic force feedback for musicians. *Computer Music Journal*, 37(1):23–34, Spring 2013.

[20] Edgar Berdahl and Julius Orion Smith. A tangible virtual vibrating string. In *Eighth International Conference on New Interfaces for Musical Expression (NIME-08)*, Genova, Italy, June 2008.

[21] Edgar Berdahl, Hans-Christoph Steiner, and Collin Oldham. Practical hardware and algorithms for creating haptic musical instruments. In *Proceedings of the Internation Conference on New Interfaces for Musical Expression (NIME-08)*, Genova, Italy, 2008.

[22] Frédéric Bevilacqua, Nicolas Rasamimanana, Emmanuel Fléty, Serge Lemouton, and Florence Baschet. The augmented violin project: research, composition and performance report. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Paris, France, June 2006.

[23] Frédéric Bevilacqua, Norbert Schnell, Nicolas Rasamimanana, Bruno Zamborlin, and Fabrice Guédy. *Musical Robots and Interactive Multimodal Systems*, chapter Online Gesture Analysis and Control of Audio Processing, pages 127 – 142. Springer Berlin Heidelberg, 2011.

[24] Stefan Bilbao. *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*. John Wiley and Sons, Chichester, UK, 2009.

[25] Jane Bird. Exploring the 3d printing opportunity. Financial Times, August 2012.

[26] David Birnbaum and Marcelo M. Wanderley. A systematic approach to musical vibrotactile feedback. In *Proceedings of the Iternational Computer Music Conference (ICMC-07)*, Copenhagen, Denmark, 2007.

[27] Richard Boulanger and Max Mathews. The 1997 mathews radio-baton & improvisation modes. In *Proceedings of the 1997 International Computer Music Conference (ICMC-97)*, Thessaloniki, Greece, 1997.

[28] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. Embedding PureData with libpd. In *Proceedings of the Pure Data Convention*, Weinmar, Germany, 2011.

[29] Cynthia Bruyns. Modal synthesis for arbitrarily shaped objects. *Computer Music Journal*, 30(3):22–37, Autumn 2006.

[30] Nicholas J. Bryan, Jorge Herrera, Jieun Oh, and Ge Wang. Momu: A mobile music toolkit. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, June 2010.

[31] Matthew Burtner. The metasaxophone: Concept, implementation, and mapping strategies for a new computer music instrument. *Organised Sound*, 7(2):201–213, 2002.

[32] Bill Buxton. Multi-touch systems that i have known and loved. On-line, June 2014. `http://www.billbuxton.com/multitouchOverview.html`.

[33] Claude Cadoz, Annie Luciani, and Jean-Loup Florens. Synthèse musicale par simulation des mécanismes instrumentaux, transducteurs gestuels rétroactifs pour l'étude du jeu instrumental. *Revue d'acoustique*, (59):279–292, 1981.

[34] Claude Cadoz, Annie Luciani, and Jean Loup Florens. Cordis-anima: A modeling and simulation system for sound and image synthesis: The general formalism. *Computer Music Journal*, 17(1):19–29, Spring 1993.

[35] René Caussé, Joel Bensoam, and Nicholas Ellis. Modalys, a physical modeling synthesizer: More than twenty years of researches, developments, and musical uses. *Journal of the Acoustical Society of America*, 130(4), 2011.

[36] Chris Chafe. Tactile audio feedback. In *Proceedings of International Computer Music Conference (ICMC93)*, Waseda University, Japan, 1993.

[37] Chris Chafe. Case studies of physical models in music composition. In *Proceedings of the 18th International Congress on Acoustics*, Kyoto, Japan, 2004.

[38] Liwei Chan, Stefanie Müller, Anne Roudaut, and Patrick Baudisch. Capstones and zebrawidgets: Sensing stacks of building blocks, dials and sliders on capacitive touch screens. In *Proceedings of the Conference for Human-Computer Interaction (CHI)*, Austin, Texas, May 2012.

[39] Mike Collicutt, Carmine Casciato, and Marcelo M. Wanderley. From real to virtual: A comparison of input devices for percussion tasks. In *Proceedings of the International Conference on New Interfaces for Musical Interaction (NIME09)*, Pittsburgh, USA, June 2009.

[40] Perry Cook. A meta-wind-instrument physical model, and a meta-controller for real-time performance control. In *Proceedings of the International Computer Music Conference (ICMC92)*, San Jose State Univeristy, USA 1992.

[41] Perry Cook. Principles for designing computer music controllers. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME'01)*, Seattle, USA, April 2001.

[42] Perry Cook. Remutualizing the instrument: Co-design of synthesis algorithms and controllers. In *Proceedings of the Stockholm Music Acoustics Conference (SMAC-03)*, Stockholm, Sweden, August 2003.

[43] Perry Cook and Gary Scavone. The Synthesis Toolkit (stk). In *Proceedings of the International Computer Music Conference (ICMC-99)*, Beijing, China, 1999.

[44] Adrien Cornelissen. Murmurate, the concert with smartphones as speakers. Digital|arti - Online Journal, 2005. `http://media.digitalarti.com/blog/digitalarti_mag/murmurate_the_concert_with_smartphones_as_speakers`.

[45] David Correa, Athina Papadopoulou, Christophe Guberan, Nynika Jhaveri, Steffen Reichert, Achim Menges, and Skylar Tibbits. 3d-printed wood: Programming hygroscopic material transformations. *3D Printing and Additive Manufacturing*, 2(3):106–116, 2015.

[46] Alain Crevoisier and Pietro Polotti. Tangible acoustic interfaces and their applications for the design of new musical instruments. In *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression (NIME05)*, Vancouver, Canada, May 2005.

[47] Matthew Dabin, Terumi Narushima, Stephen T. Beirne, Christian H. Ritz, and Kraig Grady. 3d modelling and printing of microtonal flutes. In *Proceedings of the 16th International Conference on New Interfaces for Musical Expression (NIME-16)*, Brisbane, Australia, 2016.

[48] Nicolas D'Alessandro and Thierry Dutoit. Handsketch bi-manual controller - investigation on expressive control issues of an augmented tablet. In *Proceedings of the 2007 Conference on New Interfaces for Musical Expression (NIME07)*, New York, USA, June 2007.

[49] Philip L. Davidson and Jefferson Y. Han. Synthesis and control on large scale multi-touch sensing displays. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME06)*, Paris, France, 2006.

[50] Sarah Denoux, Stéphane Letz, Yann Orlarey, and Dominique Fober. Faustlive: Just-in-time faust compiler... and much more. In *Proceedings of the Linux Audio Conference (LAC-12)*, Karlsruhe, Germany, April 2014.

[51] Dimitri Diakopoulos and Ajay Kapur. HIDUINO: A firmware for building driverless usb-midi devices using the arduino microcontroller. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME'11)*, Oslo, Norway, June 2011.

[52] Christopher Dobrian and Daniel Koppelman. The 'e' in nime: Musical expression with new computer interfaces. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME06)*, Paris, France, 2006.

[53] Alice Eldridge and Chris Kiefer. The Self-resonating Feedback Cello: Interfacing gestural and generative processes in improvised performance. In *Proceedings of the New Interfaces for Musical Expression Conference (NIME-17)*, Copenhagen, Denmark, 2017.

[54] Nicholas Ellis, Joël Bensoam, and René Caussé. Modalys demonstration. In *Proceedings of the 2005 International Computer Music Conference (ICMC05)*, Barcelona, Spain, 2005.

[55] Georg Essl and Perry Cook. Banded waveguides: Towards physical modeling of bowed bar percussion instruments. In *Proceedings of the International Computer Music Conference (ICMC-99)*, pages 321—-324, Beijing, China, 1999.

[56] Georg Essl and Michael Rohs. SHAMUS – a sensor-based integrated mobile phone instrument. In *Proceedings of the International Computer Music Conference (ICMC-07)*, Copenhagen, Denmark, 2007.

[57] Georg Essl and Michael Rohs. Interactivity for mobile music-making. *Organised Sound*, 14(2):197–207, 2009.

[58] Georg Essl, Ge Wang, and Michael Rohs. Developments and challenges turning mobile phones into generic music performance platforms. In *Proceedings of the Mobile Music Workshop*, Vienna, Austria, May 2008.

[59] Neville H. Fletcher and Thomas D. Rossing. *The Physics of Musical Instruments, 2nd Edition*. Springer Verlag, 1998.

[60] Jean-Loup Florens and Claude Cadoz. Modular modelisation and simulation of the instrument. In *Proceedings of the International Computer Music Conference (ICMC-90)*, Glasgow, UK, 1990.

[61] Christopher Fuller, Sharon Elliott, and Philip Nelson. *Active control of vibration*. Academic Press, 1996.

[62] Sin Fun Shing, Daniel Schroeder, and Jernej Barbič. Vega: Nonlinear fem deformable object simulator. *Computer Graphics Forum*, 32(1):36–48, February 2013.

[63] Lalya Gaye, Lars Erik Holmquist, Frauke Behrendt, and Atau Tanaka. 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 2006.

[64] Günter Geiger. PDa: Real time signal processing and sound generation on handheld devices. In *Proceedings of the International Computer Music Conference (ICMC-03)*, Singapore, 2003.

[65] Günter Geiger. 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, 2006.

[66] Camille Goudeseune. A violin controller for real-time audio synthesis. Technical report, Integrated Systems Laborator, University of Illinois, 2001.

[67] GRAME – Centre National de Création Musicale, Lyon, France. *FAUST Quick Reference*, June 2017.

[68] John Granzow. *Additive Manufacturing for Musical Applications*. PhD thesis, Stanford University, June 2017.

[69] Doug Gross. Obama's speech highlights rise of 3-d printing. On-line – CNN, February 13 2013. `http://www.cnn.com/2013/02/13/tech/innovation/obama-3d-printing/`.

[70] Pierre-Amaury Grumiaux, Romain Michon, Emilio Gallego Arias, and Pierre Jouvelot. Impulse-response and cad-model-based physical modeling in faust. In *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, May 2017.

[71] Reginald Langford Harrison, Stefan Bilbao, James Perry, and Trevor Wishart. An environment for physical modeling of articulated brass instruments. *Computer Music Journal*, 39(4):80–95, Winter 2015.

[72] Christian Heinrichs and Andrew McPherson. A hybrid keyboard-guitar interface using capacitive touch sensing and physical modeling. In *Proceedings of the 9th Sound and Music Computing Conference (SMC)*, Copenhagen, Denmark, 2012.

[73] Seongkook Heo and Geehyuk Lee. Force gestures: Augmenting touch screen gestures with normal and tangential forces. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST-11)*, Santa Barbara, California, 2011.

[74] John Hollis. Synthaxe website. On-line. `http://www.hollis.co.uk/john/synthaxe.html`.

[75] Peter Holstlin. Sounds of future past: The lyricon. *Red Bull Music Academy Daily*, 2015.

[76] Damon Holzborn. *Building Mobile Instruments for Improvised Musical Performance*. PhD thesis, Columbia University, USA, 2013.

[77] David M. Howard and Stuart Rimell. Gesture-tactile control physical modeling music synthesis. In *Proceedings of the Stockholm Music Acoustics Conference (SMAC-03)*, Stockholm, Sweden, August 2003.

[78] David M. Howard and Stuart Rimell. Real-time gesture-controlled physical modelling music synthesis with tactile feedback. *EURASIP Journal on Applied Signal Processing*, 7:1001–1006, 2004.

[79] Sungjae Hwang, Myungwook Ahn, and Kwang yun Wohn. Maggetz: Customizable passive tangible controllers on and around conventional mobile devices. In *Proceedings of the Symposium on User Interface Software and Technology*, St Andrews, United Kingdom, October 2013.

[80] Jonathan Impett. A meta-trumpet(er). In *Proccedings of the International Computer Music Conference (ICMC94)*, Danish Institute of Electroacoustic Music, Denmark, 1994.

[81] David A. Jaffe and Andrew Schloss. A virtual piano concerto – the coupling of the mathews/boie radio drum and yamaha disklavier grand piano. In *Proceedings of the International Computer Music Conference (ICMC-94)*, Danish Institute of Electroacoustic Music, Denmark, 1994.

[82] David A. Jaffe and Julius Orion Smith. Extensions of the karplus-strong plucked-string algorithm. *Computer Music Journal*, 7(2):56–69, 1983.

[83] Sergi Jordà, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. The reactable: Exploring the synergy between live music performance and tabletop tangible interfaces. In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, pages 139–146, Baton Rouge, Louisiana, 2007.

[84] Sergi Jordà. *Digital Lutherie Crafting Musical Computers for New Musics' Performance and Improvisation*. PhD thesis, Universitat Pompeu Fabra, Spain, 2005.

[85] Ajay Kapur, Georg Essl, Philip L. Davidson, and Perry Cook. The electronic tabla controller. In *Proceedings of the 2002 Conference on New Instruments for Musical Expression (NIME-02)*, Dublin, Ireland, May 2002.

[86] Matti Karjalainen, Teemu Mäki-Patola, Aki Kanerva, and Antti Huovilainen. Virtual air guitar. *Journal of the Audio Engineering Society*, 54(10):964–980, October 2006.

[87] Matti Karjalainen and Julius Orion Smith. Body modeling techniques for string instrument synthesis. In *Proceedings of the International Computer Music Conference (ICMC-96)*, Hong Kong, August 1996.

[88] 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, Autumn 1998.

[89] Kevin Karplus and Alex Strong. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55, Summer 1983.

[90] Sukandar Kartadinata. The gluiph: a nucleus for integrated instruments. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, Montreal, Canada, 2003.

[91] Loic Kessous, Julien Castet, and Daniel Arfib. 'gxtar', an interface using guitar techniques. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME06)*, Paris, France, June 2006.

[92] Juraj Kojs, Stefania Serafin, and Chris Chafe. Cyberinstruments via physical modeling synthesis: Compositional applications. *Leonardo Music Journal*, 17:61–66, 2007.

[93] Sven Kratz, Tilo Westermann, Michael Rohs, and Georg Essl. Capwidgets: Tangible widgets versus multi-touch controls on mobile devices. In *Proceedings of the conference for Human-Computer Interaction*, Vancouver, Canada, May 2011.

[94] Roland Lamb and Andrew Robertson. Seaboard: a new piano keyboard-related interface combining discrete and continuous control. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, 2011.

[95] Gierad Laput, Eric Brockmeyer, Scott Hudson, and Chris Harrison. Acoustruments: Passive, acoustically-driven, interactive controls for handheld devices. In *Proceedings of the Conference for Human-Computer Interaction (CHI)*, Seoul, Republic of Korea, April 2015.

[96] Guillaume Largillier. Developing the first commercial product that uses multi-touch technology. *Information Display*, 23(12):14–18, 2007.

[97] Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. The mobile Csound platform. In *Proceedings of the International Conference on Computer Music (ICMC-12)*, Ljubljana, Slovenia, September 2012.

[98] Sasha Leitman and John Granzow. Music maker: 3d printing and acoustics curriculum. In *Proceedings of the New Interfaces for Musical Expression Conference (NIME-16)*, Brisbane, Australia, July 2016.

[99] James Leonard and Claude Cadoz. Physical modelling concepts for a collection of multisensory virtual musical instruments. In *Proceedings of the Conference on New Interfaces for Musical (NIME15)*, Baton Rouge, USA, May 2015.

[100] Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon. 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, 2017.

[101] Rong-Hao Liang, Liwei Chan, Hung-Yu Tseng, Han-Chih Kuo, Da-Yuan Huang, De-Nian Yang, and Bing-Yu Chen. Gaussbricks: Magnetic building blocks for constructive tangible interactions on portable displays. In *Proceedings of the Conference for Human-Computer Interaction (CHI)*, Toronto, Ontario, May 2014.

[102] Hod Lipson and Melba Kurman. *Fabricated: The New World of 3D Printing.* Wiley, 2013.

[103] Henning Lohner. The upic system: A user's report. *Computer Music Journal*, 10(4):42–49, Winter 1986.

[104] Gareth Loy. Musicians make a standard: The midi phenomenon. *Computer Music Journal*, 9(4):8–26, Winter 1985.

[105] Otso Lähdeoja. An approach to instrument augmentation : the electric guitar. In *Proceedings of the 2008 Conference on New Interfaces for Musical Expression (NIME08)*, Genova, Italy, 2008.

[106] Tod Machover. Hyperinstruments: A progress report 1987–1991. Technical report, MIT Media Lab, 1992.

[107] Laura Maes, Godfried-Willem Raes, and Troy Rogers. The man and machine robot orchestra at logos. *Computer Music Journal*, 35(4):22–48, Winter 2011.

[108] Esteban Maestre. *Modeling Instrumental Gestures: An Analysis/Synthesis Framework for Violin Bowing*. PhD thesis, Universitat Pompeu Fabra, Spain, 2009.

[109] Thor Magnusson. Of epistemic tools: musical instruments as cognitive extensions. *Organised Sound*, 14(2):168–176, August 2009.

[110] Mark T. Marshall and Marcelo M. Wanderley. Vibrotactile feedback in digital musical instruments. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME06)*, Paris, France, June 2006.

[111] Max Mathews. Electronic violin: A research tool. *Journal of the Violin Society of America*, 8(1):71–88, 1984.

[112] Steven Maupin, David Gerhard, and Brett Park. Isomorphic tesselations for musical keyboards. In *Proceedings of the Sound and Music Computing Conference (SMC-11)*, Padova, Italy, July 2011.

[113] Sandor Mehes, Maarten van Walstijn, and Paul Stapleton. Virtual-acoustic instrument design: Exploring the parameter space of a string-plate model. In *Proceedings of the New Interfaces for Musical Expression Conference (NIME-17)*, Copenhagen, Denmark, 2017.

[114] Thibaut Meurisse, Adrien Mamou-Mani, Simon Benacchio, Baptiste Chomette, Victor Finel, David Sharp, and René Caussé. Experimental demonstration of the modification of the resonances of a simplified self-sustained wind instrument through modal active control. *Acta Acustica United with Acustica*, 101(3):581–593, January 2015.

[115] Thibaut Meurisse, Adrien Mamou-Mani, René Caussé, Baptiste Chomette, and David B. Sharp. Simulations of modal active control applied to the self-sustained oscillations of the clarinet. *Acta Acustica United with Acustica*, 100:1149–1161, 2014.

[116] Geist Meyer and Ben Kleinerman. Simultaneous transmission of a video and an audio signal through an ordinary telephone transmission line, 1975.

[117] Romain Michon. A hybrid sound installation: the blackbox. On-line, December 2012. `https://ccrma.stanford.edu/~rmichon/blackbox/`.

[118] Romain Michon. faust2android: a Faust architecture for Android. In *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, September 2013.

[119] Romain Michon. Faust tutorials. On-line, 2017. `https://ccrma.stanford.edu/~rmichon/faustTutorials`.

[120] Romain Michon and John Granzow. Hybrid clarinet project. Technical report, CCRMA, Stanford Univeristy, December 2013.

[121] Romain Michon, Sara R. Martin, and Julius O. Smith. Mesh2Faust: a modal physical model generator for the Faust programming language – application to bell modeling. In *Proceedings of the International Computer Music Conference (ICMC-17)*, Shanghai, China, October 2017.

[122] Romain Michon and Yann Orlarey. The faust online compiler: a web-based ide for the faust programming language. In *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, California, 2012.

[123] Romain Michon, Julius Smith, and Yann Orlarey. New signal processing libraries for Faust. In *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, May 2017. Paper accepted to the conference but not published yet.

[124] Romain Michon, Julius Smith, Matthew Wright, Chris Chafe, John Granzow, and Ge Wang. Passively augmenting mobile devices towards hybrid musical instrument design. In *Proceedings on the New Interfaces for Musical Expression Conference (NIME-17)*, Copenhagen, Denmark, May 2017.

[125] Romain Michon and Julius O. Smith. 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 2011.

[126] Romain Michon, Julius O. Smith, Chris Chafe, Matthew Wright, and Ge Wang. Nuance: Adding multi-touch force detection to the iPad. In *Proceedings of the Sound and Music Computing Conference (SMC-16)*, Hamburg, Germany, 2016.

[127] Romain Michon, Julius O. Smith, Matthew Wright, and Chris Chafe. Augmenting the iPad: the BladeAxe. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Brisbane, Australia, July 2016.

[128] Romain Michon and Julius Orion Smith. A hybrid guitar physical model controller: The BladeAxe. In *Proceedings of ICMC|SMC 2014*, Athens, Greece, September 2014.

[129] Romain Michon, Julius Orion Smith, and Yann Orlarey. 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 2015.

[130] Kurt Miller. Print me a stradivarius - how a new manufacturing technology will change the world. On-line – The Economist, February 2011. `http://www.economist.com/node/18114327?Story_ID=18114327`.

[131] Eduardo Miranda, Ross Kirk, and Marcelo Wanderley. *New Digital Musical Instruments*. A-R Editions, Middleton, WI, 2006.

[132] Ananya Misra, Georg Essl, and Michael Rohs. Microphone as sensor in mobile phone performance. In *Proceedings of the New Interfaces for Musical Expression conference (NIME08)*, Genova, Italy, 2008.

[133] Ali Momeni. Caress: An enactive electro-acoustic percussive instrument for caressing sound. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME-15)*, Baton Rouge, USA, June 2015.

[134] Joseph Derek Morrison and Jean-Marie Adrien. Mosaic: A framework for modal synthesis. *Computer Music Journal*, 17(1):45–56, Spring 1993.

[135] Robert Moses, Norman Durkee, and Charles Hustig. System for carrying transparent digital data within an audio signal, 1997.

[136] Teemu Mäki-Patola. User interface comparison for virtual drums. In *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression (NIME05)*, Vancouver, Canada, 2005.

[137] Alexander Müller, Fabian Hemmert, Götz Wintergerst, and Ron Jagodzinski. Reflective haptics: Resistive force feedback for musical performances with stylus-controlled instruments. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*, Sidney, Australia, June 2010.

[138] Charles Nichols. The vbow: Development of a virtual violin bow haptic human-computer interface. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME'02)*, Dublin, Ireland, May 2002.

[139] Charles Nichols. *The vBow: an Expressive Musical Controller Haptic Human-Computer Interface*. PhD thesis, Stanford University, USA, 2003.

[140] Sile O'Modhrain. *Playing by feel: incorporating haptic feedback into computer-based musical instruments*. PhD thesis, Stanford University, 2001.

[141] Yann Orlarey, Stéphane Letz, and Dominique Fober. *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming". Delatour, Paris, France, 2009.

[142] Francesco Orrù. Francesco orrù's portfolio. On-line. `https://www.myminifactory.com/users/4theswarm`.

[143] Dan Overholt. The overtone violin: A new computer music instrument. In *Proceedings of the 2005 International Computer Music Conference (ICMC-05)*, Barcelona, Spain, September 2005.

[144] Sile O'Modhrain and Chris Chafe. The performer-instrument interaction: A sensory motor perspective. In *Proceedings of the International Computer Music Conference (ICMC-00)*, Menlo Park, USA, 2000.

[145] Joseph A. Paradiso. Electronic music: new ways to play. *IEEE Spectrum*, 34(12):18–30, December 1997.

[146] Tae Hong Park and Oriol Nieto. Fortissimo: Force-feedback for mobile devices. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, KAIST, Daejon, Korea, May 2013.

[147] Olivier Perrotin and Christophe d'Alessandro. Adaptive mapping for improved pitch accuracy on touch user interfaces. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Daejeon, South Korea, May 2013.

[148] Trevor J Pinch, Frank Trocco, and TJ Pinch. *Analog days: The invention and impact of the Moog synthesizer*. Harvard University Press, 2009.

[149] Cornelius Poepel. Synthesized strings for string players. In *Proceedings of the 2004 Conference on New Interfaces for Musical Expression (NIME04)*, Hamamatsu, Japan, 2004.

[150] Nick Porcaro, David Jaffe, Pat Scandalis, Julius Smith, Tim Stilson, and Scott Van Duyne. Synthbuilder: a graphical rapid-prototyping tool for the development of music synthesis and effects patches on multiple platforms. *Computer Music Journal*, 22(2):35–43, Summer 1998.

[151] Miller Puckette. Infuriating nonlinear reverberator. In *Proceedings of the International Computer Music Conference (ICMC-11)*, Huddersfield, UK, 2011.

[152] Miller Puckette. Playing a virtual drum from a real one. *The Journal of the Acoustical Society of America*, 130(4):2432, 2011.

[153] Zhimin Ren, Ravish Mehra, Jason Coposky, and Ming C. Lin. Tabletop ensemble: Touch-enabled virtual percussion instruments. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D '12)*, Costa Mesa, California, March 2012.

[154] Stuart Rimell, David M Howard, Andy M Tyrell, Ross Kirk, and Andy Hunt. Cymatic: Restoring the physical manifestation of digital sound using haptic interfaces to control a new computer based musical instrument. In *Proceedings of the Iternational Computer Music Conference (ICMC-02)*, Gothenburg, Sweden, 2002.

[155] Xavier Rodet. One and two mass model oscillations for voice and instruments. In *Proceedings of the International Computer Music Conference (ICMC-95)*, pages 207–214, Banff, Canada, 1995.

[156] Roland Corporation. *Aerophone AE-10 – Owner's Manual*, 2016.

[157] ROLI Ltd., London, UK. *ROLI Seaboard GRAND – User Manual*, 2015.

[158] Thomas D. Rossing and Robert Perrin. Vibrations of bells. *Applied Acoustics*, 20(1):41–70, December 1987.

[159] Martin Russ. Yamaha VL1 - virtual acoustic synthesizer. Sound on Sound, July 1994.

[160] Edward Kingsley Rutter, Tom Mitchell, and Chris Nash. Turnector: Tangible control widgets for capacitive touchscreen devices. In *Proceedings of ICMC|SMC*, Athens, Greece, September 2014.

[161] Spencer Salazar and Ge Wang. Auraglyph – handwriting input for computer-based music composition and design. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Goldsmiths, University of London, UK, 2014.

[162] Gary Scavone and Perry Cook. RtMidi, RtAudio, and a synthesis toolkit (STK) update. In *Proceedings of the 2005 International Computer Music Conference*, Barcelona, Spain, 2005.

[163] Gary P. Scavone. THE PIPE: Explorations with breath control. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, Montreal, Canada, 2003.

[164] Greg Schiemer and Mark Havryliv. Pocket Gamelan: tuneable trajectories for flying sources in Mandala 3 and Mandala 4. In *Proceedings of the 6th International Conference on New Interfaces for Musical Expression (NIME06)*, Paris, France, 2006.

[165] Daniel Schlessinger and Julius Orion Smith. The Kalichord: A physically modeled electro-acoustic plucked string instrument. In *Proceedings of the 9th International Conference on New Interfaces for Musical Expression (NIME-09)*, Carnegie Mellon Univeristy, USA, June 2009.

[166] Norbert Schnell and Marc Battier. Introducing composed instruments, technical and musicological implications. In *Proceedings of the 2002 Conference on New Instruments for Musical Expression (NIME-02)*, Dublin, Ireland, May 2002.

[167] Stefania Serafin, Matthew Burtner, Charles Nichols, and Sile O'Modhrain. Expressive controllers for bowed string physical models. In *Proceedings of the 1st International Conference on Digital Audio Effects*, Limerick, Ireland, December 2001.

[168] Stefania Serafin, Richard Dudas, Marcello M. Wanderley, and Xavier Rodet. Gestural control of a real-time model of a bowed string instrument. In *Proceedings of the International Computer Music Conference*, Beijing, October 1999.

[169] Stefania Serafin and Diana Young. Bowed string physical model validation through use of a bow controller and examination of bow strokes. In *Proceedings of the Stockholm Music Acoustics Conference (SMAC03)*, Stockholm, Sweden, August 2003.

[170] Hang Si. Tetgen, a delaunay-based quality tetrahedral mesh generator. *ACM Trans. Math. Softw.*, 41(2):11:1–11:36, February 2015.

[171] Stephen Sinclair. Implementing the synthbuilder piano in stk. Technical report, McGill University, 2006.

[172] Stephen Sinclair, Gary Scavone, and Marcelo M. Wanderley. Audio-haptic interaction with the digital waveguide bowed string. In *Proceedings of the International Computer Music Conference (ICMC-09)*, Montreal, Canada, 2009.

[173] Julius O. Smith. Physical modeling using digital waveguides. *Computer Music Journal*, 16(4):74–91, Winter 1992.

[174] Julius O. Smith and Perry Cook. The second-order digital waveguide oscillator. In *Proceedings of the International Computer Music Conference (ICMC-92)*, pages 150–153, San Jose, USA, 1992.

[175] Julius O. Smith and Scott A. Van Duyne. Commuted piano synthesis. In *Proceedings of the International Computer Music Conference (ICMC-95)*, pages 319–326, Banff, Canada, 1995.

[176] Julius Orion Smith. Virtual electric guitars and effects using Faust and Octave. In *Proceedings of the Linux Audio Conference (LAC-08)*, pages 123–127, KHM, Cologne, Germany, 2008.

[177] Julius Orion Smith. *Physical Audio Signal Processing for Virtual Musical Instruments and Digital Audio Effects*. W3K Publishing, `https://ccrma.stanford.edu/~jos/pasp/`, 2010.

[178] Julius Orion Smith and Romain Michon. Nonlinear allpass ladder filters in Faust. In *Proceedings of the 14th International Conference on Digital Audio Effects*, Paris, France, September 2011. IRCAM.

[179] Richard R. Smith. *The history of rickenbacker guitars.* Centerstream Publications, 1987.

[180] Starr Labs, San Diego, California. *Ztar MIDI – User's Manual*, 2008.

[181] Scott Summit. System and method for designing and fabricating string instruments. US Patent, 2014.

[182] SwitchScience. Audio jack modem for iphone and android. On-line, 2016. `https://www.switch-science.com/catalog/364/`martin.

[183] Atau Tanaka. Mobile music making. In *Proceedings of the 2004 conference on New interfaces for musical expression (NIME04)*, National University of Singapore, 2004.

[184] Atau Tanaka. Mapping out instruments, affordances, and mobiles. In *Proceedings of the 2010 Conference on New Interfaces for Musical Expression (NIME 2010)*, Sydney, Australia, June 2010.

[185] Adam R. Tindale. A hybrid method for extended percussive gesture. In *Proceedings of the 2007 Conference on New Interfaces for Musical Expression (NIME07)*, New York, USA, 2007.

[186] Caroline Traube, Philippe Depalle, and Marcelo Wanderley. Indirect acquisition of instrumental gesture based on signal, physical and perceptual information. In *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, Montreal, Canada, 2003.

[187] Dan Trueman and Perry Cook. Bossa: The deconstructed violin reconstructed. *Journal of New Music Research*, 29(2):121–130, 2000.

[188] Friedrich Türckheim, Thorsten Smit, and Robert Mores. The semi-virtual violin – a perception tool. In *Proceedings of 20th International Congress on Acoustics (ICA10)*, Sydney, Australia, August 2010.

[189] Vesa Valimaki and Timo I Laakso. Principles of fractional delay filters. In *Acoustics, Speech, and Signal Processing, 2000. ICASSP'00. Proceedings. 2000 IEEE International Conference on*, volume 6, pages 3870–3873. IEEE, 2000.

[190] Maarten van Walstijn and Pedro Rebelo. The prosthetic conga: Towards an actively controlled hybrid musical instrument. In *Proceedings of the International Computer Music Conference (ICMC05)*, page 786–789, Barcelona, Spain, 2005.

[191] Ashlee Vance. The world's first 3d-printed acoustic guitar. On-line – Bloomberg Business, October 12 2012. `http://www.bloomberg.com/bw/articles/2012-10-11/the-worlds-first-3d-printed-guitar`.

[192] Marc-Pierre Verge. *Aeroacoustics of Confined Jets with Applications to the Physical Modeling of Recorder-Like Instruments*. PhD thesis, Eindhoven University, 1995.

[193] Sonal Verma, Andrew Robinson, and Prabal Dutta. Audiodaq: Turning the mobile phone's ubiquitous headset port into a universal data acquisition interface. In *Proceedings of the Conference on Embedded Networked Sensor Systems (SenSys)*, Toronto, Ontario, November 2012.

[194] Vesa Välimäki and Tapio Takala. Virtual musical instruments - natural sound using physical models. *Organised Sound*, 1(2):75–86, August 1996.

[195] Marcello M. Wanderley and Philippe Depalle. Gestural control of sound synthesis. In *Proceedings of the IEEE*, volume 92, pages 632–644, 2004.

[196] Marcelo Wanderley and Nicola Orio. Evaluation of input devices for musical expression: Borrowing tools from hci. *Computer Music Journal*, 26(3):62–76, Fall 2002.

[197] Ge Wang. Ocarina: Designing the iPhone's Magic Flute. *Computer Music Journal*, 38(2):8–21, Summer 2014.

[198] Ge Wang. *Artful Design – Technology in Search of the Sublime*. Stanford University Press, 2018. To be published in 2018.

[199] Ge Wang, Georg Essl, and Henri Penttinen. Do mobile phones dream of electric orchestra? In *Proceedings of the International Computer Music Conference (ICMC-08)*, Belfast, Northern Ireland, 2008.

[200] Ge Wang, Jieun Oh, and Tom Lieber. Designing for the iPad: Magic fiddle. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, May 2011.

[201] Paul Warner. System for transmitting data simultaneously with audio, 1983.

[202] Paul White, Rachel Fletcher, and Paul Farber. Yamaha wx5 wind controller. On-line – Sound on Sound, July 1998. `http://www.soundonsound.com/sos/jul98/articles/yamwx5.html`.

[203] Gerhard Widmer, Davide Rocchesso, Vesa Välimäki, Cumhur Erkut, Fabien Gouyon, Daniel Pressnitzer, Henri Penttinen, Pietro Polotti, and Gualtiero Volpe. Sound and music computing: Research trends and some key issues. *Journal of New Music Research*, 36(3):169–184, 2007.

[204] Matthew Wright and Adrian Freed. Open Sound Control: A new protocol for communicating with sound synthesizers. In *Proceedings of the International Computer Music Conference*, Thessaloniki, Greece, 1997.

[205] Yamaha Corporation, P.O. Box 1, Hamamatsu, Japan. *VL1 Virtual Acoustic Synthesizer - Owner's Manual*, 1993.

[206] Yamaha Corporation of America, Buena Park, California. *Yamaha Digital Programmable Algorithm Synthesizer – Operation Manual*, 1983.

[207] Diana Young. The hyperbow controller: Real-time dynamics measurement of violin performance. In *Proceedings of the 2002 Conference on New Instruments for Musical Expression (NIME-02)*, Dublin, Ireland, May 2002.

[208] Diana Young and Ichiro Fujinaga. Aobachi: A new interface for japanese drumming. In *Proceedings of the 2004 Conference on New Interfaces for Musical Expression (NIME04)*, Hamamatsu, Japan, 2004.

[209] Neng-Hao Yu, Li-Wei Chan, Seng-Yong Lau, Sung-Sheng Tsai, I-Chun Hsiao, Dian-Je Tsai, Lung-Pan Cheng, Fang-I Hsiao, Mike Y. Chen, Polly Huang, and Yi-Ping Hung. Tuic: Enabling tangible interaction on capacitive multi-touch display. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, Vancouver, Canada, May 2011.

[210] Michael Zbyszynski, Matthew Wright, Ali Momeni, and Daniel Cullen. Ten years of tablet musical interfaces at cnmat. In *Proceedings of the 2007 Conference on New Interfaces for Musical Expression (NIME07)*, New York, USA, June 2007.

[211] Amit Zoran. The 3d printed flute: Digital fabrication and design of musical instruments. *Journal of New Music Research*, 40(4):379–387, December 2011.

[212] Amit Zoran and Pattie Maes. Considering virtual & physical aspects in acoustic guitar design. In *Proceedings of the 2008 Conference on New Interfaces for Musical Expression (NIME08)*, Genova, Italy, 2008.

[213] Amit Zoran and Joseph Paradiso. The chameleon guitar - guitar with a replaceable resonator. *Journal of New Music Research*, 40(1):59–74, March 2011.