A Framework for Embedded Digital Musical Instruments

Ivan Franco

Music Technology Area
Schulich School of Music
McGill University
Montreal, Canada

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

2019/04/11

Abstract

Gestural controllers allow musicians to use computers as digital musical instruments (DMI). The body gestures of the performer are captured by sensors on the controller and sent as digital control data to a audio synthesis software.

Until now DMIs have been largely dependent on the computing power of desktop and laptop computers but the most recent generations of single-board computers have enough processing power to satisfy the requirements of many DMIs. The advantage of those single-board computers over traditional computers is that they are much smaller in size. They can be easily embedded inside the body of the controller and used to create fully integrated and self-contained DMIs.

This dissertation examines various applications of embedded computing technologies in DMIs. First we describe the history of DMIs and then expose some of the limitations associated with the use of general-purpose computers. Next we present a review on different technologies applicable to embedded DMIs and a state of the art of instruments and frameworks. Finally, we propose new technical and conceptual avenues, materialized through the Prynth framework, developed by the author and a team of collaborators during the course of this research.

The Prynth framework allows instrument makers to have a solid starting point for the development of their own embedded DMIs. It is composed of several interoperating hardware and software components, publicly available as open-source technologies and continuously updated to include new features. Prynth has been lauded in both academic and artistic circles, resulting in an active and engaged user community.

This community uses our framework to build many different DMIs that serve idiosyncratic goals and a panoply of musical contexts. The author himself is responsible for the design and construction of “The Mitt”, an instrument that was selected for the finals of the 2017 edition of the Guthman Musical Instrument Competition, the most prestigious event of its kind.

We believe that embedded computing will play a fundamental role in the future of DMIs. By surpassing the reliance on traditional computers, embedded DMIs have the possibility of becoming integrated and robust interactive devices, with increased usability and performance. All of these represent key factors to the evolution and growth of these musical instruments.

Résumé

Les contrôleurs gestuels permettent aux musiciens d'utiliser les ordinateurs en tant qu'instruments numériques de musique (INM). Les mouvements corporels de l'artiste sont enregistrés par des capteurs sur le contrôleur, puis envoyés sous forme de données de contrôle numériques à un logiciel de synthèse audio.

Jusqu'ici, les INM étaient largement dépendants de la puissance de calcul des ordinateurs portables et des unités centrales, mais les générations les plus récentes d'ordinateurs à carte unique disposent d'une puissance de calcul suffisante pour satisfaire aux exigences de nombreux INM. L'avantage de ces ordinateurs à carte unique sur les ordinateurs traditionnels réside dans leur taille qui s'avère bien moindre. Ils peuvent facilement être embarqués dans le corps du contrôleur et être utilisés pour créer des INM indépendants et totalement intégrés.

Cette dissertation examine les diverses applications des systèmes d'informatique embarquée aux INM. Dans un premier lieu, nous ferons un rappel de l'histoire des INM, puis nous exposerons certaines des limites associées à l'utilisation d'ordinateurs génériques. Par la suite nous présenterons une analyse des différentes technologies applicables aux INM embarqués et l'état de l'art des systèmes et instruments. Enfin, nous proposerons de nouvelles voies conceptuelles et techniques, matérialisées par le système Prynth, développé par l'auteur ainsi qu'une équipe de collaborateurs dans le cadre de cette recherche.

Le système Prynth permet aux fabricants d'instruments de disposer d'une solide base de développement pour la création de leurs INM embarqués. Il est composé de nombreux éléments matériels et logiciels en interopération, de technologies disponibles en open-source et continuellement mises à jour pour intégrer de nouvelles fonctionnalités. Prynth a été acclamé à la fois dans les cercles académiques et artistiques, avec pour conséquence une communauté très active et motivée.

Cette communauté utilise notre système pour construire de nombreux INM qui ont un objectif idiosyncratique et une large panoplie de contextes musicaux. L'auteur lui-même est responsable de la conception et de la construction de l'instrument nommé « The Mitt » qui a été sélectionné pour la finale de l'édition 2017 de la Guthman Musical Instrument Competition, l'événement le plus prestigieux de sa catégorie au monde.

Nous croyons que l'informatique embarquée jouera un rôle fondamental dans le futur des INM. En surpassant la dépendance aux ordinateurs classiques, les INM embarqués ont la possibilité de devenir des appareils interactifs solides et intégrés, disposant d'une utilisabilité et de performances améliorées. Tous ces critères représentant des éléments-clés de l'évolution et de la croissance de ces instruments de musique.

Acknowledgments

I would like to express my gratitude to the people and institutions that supported me during the course of this work.

Thanks to my supervisor Marcelo Wanderley, for his precious knowledge, dedicated tutorship and wonderful enthusiasm.

Thanks to all of my colleagues at the Input Devices and Music Interaction Laboratory (IDMIL) and the Centre for Interdisciplinary Research in Music Media and Technology (CIRMMT), especially those who have in some way contributed to this research: Ian Hattwick, John Sullivan, Johnty Wang, Eduardo Meneses, Jerônimo Barbosa, Harish Venkatesan, Ajin Tom, Antoine Maiorca and Darryl Cameron.

Thanks to all Prynth users who believed in this project and offered their valuable feedback and support.

Finally I would like to thank my family for their love: my partner Lígia Teixeira, my newborn daughter Margot, my mother Isabel and my brother Rafael.

This research was sponsored by the PhD program of the Fundação para a Ciência e Tecnologia (FCT), a public agency under responsibility of the Portuguese Ministry for Science, Technology and Higher Education.

Contents

List of Figures

List of Tables

List of Acronyms

ADC	analog-to-digital converter
AJAX	asynchronous JavaScript and XML
ALSA	advanced Linux sound architecture
API	application program interface
ARM	Advanced RISC Machine
ASIC	application-specific integrated circuits
BBB	BeagleBone Black
C64	Commodore 64
CAD	computer-aided design
CAN	controller area network
CCRMA	Center for Computer Research in Music and Acoustics
CEMAMu	Centre d'Etudes de Mathématique et Automatique Musicales
CIRMMT	Centre for Interdisciplinary Research in Music Media and Technology
CMMR	International Symposium on Computer Music Multidisciplinary Research
CPLD	complex programmable logic device
CPU	central processing unit
CRT	cathode-ray tube
DAC	digital-to-analog converter
DAW	digital audio workstation
DC	direct-current
DIY	do-it-yourself
DMA	direct memory access
DMI	digital musical instrument
DRAM	dynamic random-access memory
DSL	domain-specific language
DSP	digital signal processor; digital signal processing
FFT	fast-Fourier transform

FIFO	first-in-first-out
FM	frequency modulation
FPGA	field-programmable gate arrays
GPIO	general purpose input/output
GPL	general public license
GPS	global position system
GPU	graphics processing unit
GUI	graphical user interface
HCI	human-computer interaction
HDL	hardware description language
HDMI	high-definition multimedia interface
HTML	hypertext markup language
HTTP	hypertext transfer protocol
I/O	input/output
I^2C	inter-integrated circuit
I^2S	inter-ic sound
IBM	International Business Machines Corporation
IDE	integrated development environment
IDMIL	Input Devices and Musical Interaction Laboratory
IP	internet protocol; intellectual property
IRCAM	Institut de Recherche et Coordination Acoustique/Musique
ISPW	IRCAM signal processing workstation
JIT	just-in-time
JS	JavaScript
JSON	JavaScript object notation
LADSPA	Linux audio developer's simple plug-in API
LCD	liquid crystal display
LV2	Linux Audio Developer's Simple Plug-in API version 2
MAC	multiply-accumulate
MIDI	musical instrument digital interface
MIPI	mobile industry processor interface
MTG	Music Technology Group
NIME	New Interfaces for Musical Expression
NWDR	Nordwestdeutscher Rundfunk
OnCE	on-chip circuit emulation
OS	operating system

OSC	open sound control
PC	personal computer
PCB	printed-circuit board
PCI	peripheral component interconnect
PCM	pulse-code modulation
PD	Pure Data
PLD	programmable logic device
POSIX	portable operating system interface
PRU	programmable real-time units
PSoC	programmable system-on-chip
PWM	pulse-width modulation
RAM	random-access memory
RDF	Radiodiffusion Française
RISC	reduced instruction set computer
RPi	Raspberry Pi
RSS	RDF site summary
RX	reception
SBC	single-board computer
SD	secure digital
SDK	software development kit
SDRAM	synchronous dynamic random-access memory
SID	sound interface device
SIMD	single instruction multiple data
SoC	system-on-chip
SPI	serial peripheral interface
SSE	streaming SIMD extensions
STEIM	Studio for Electro-instrumental Music
TCP	transmission control protocol
TRRS	tip-ring-ring-sleeve
TX	transmission
UART	universal asynchronous receiver-transmitter
UDP	user datagram protocol
UPIC	Unité Polyagogique Informatique du CEMAMu
URL	uniform resource locator
USB	universal serial bus
VHDL	very high speed integrated circuit hardware description language

VPU	video processing unit
VST	virtual studio technology
WIMP	windows, icons, menus and pointers
XML	extensible markup language

Chapter 1

Introduction

A digital musical instrument is usually defined as a musical instrument composed by two distinct elements: a computer running real-time audio synthesis software and an external gestural controller to play it. Gestural controllers allow computers to be played using a wider palette of physical gestures than those permitted by the standard mouse and keyboard interfaces. In turn, the computer allows for the use of different types of synthesis software, ranging from virtual emulations of hardware devices to custom programs developed in specialized computer music programming languages. The decoupling between the controller and the sound producing device allows for numerous combinations of hardware and software, through which musicians can create custom musical setups, better tuned to their creative and functional needs.

The DMI model of the computer/controller combo has been used almost since the inception of computer music, but recent advances in embedded computing technologies suggest new architectures that could radically change DMIs. In the last decade, the mobile industry has pushed for the development of a new generation of processors that are smaller, cheaper and consume much less power than the ones used in desktop and laptop computers. Today, those new types of embedded processors have enough computing power for real-time audio processing and many of them are available in *form factors* that allow for their incorporation inside the body of the controller itself. DMIs with embedded computing surpass the need for peripheral devices and have the potential to become significantly different interactive devices.

1.1 Motivation

Like almost any other kid in my neighborhood of the late 1980s, my first incursions into music-making began with garage bands. Born from blue-collar suburban life, the manifest of punk culture influenced my generation far beyond music, dictating fashion, political ideals and social behavior. All of the suburban tribes of the time had a very strong relationship with music. It defined our ideals and was one of the main forms of artistic expression at our disposal. These underground movements were strongly connected to live performance and concerts were the main social event. They were also strongly connected to a do-it-yourself (DIY) culture. We created our own independent music markets by trading tapes, distributing homemade fanzines and organizing small concerts at social centers, many times sharing musical equipment or switching between the roles of musician, sound engineer and roadie.

At the time, avant-garde music was not so easily accessible as it is today. We had to search for it. In those quests, my interests soon diverged to more eclectic and experimental types of music, such as the works of John Zorn or Bill Laswell. Some of the sounds in their compositions were synthetically generated, so I began to try to understand how I could emulate such styles. I bought a computer and started playing with tone generators and sequencers, soon to realize that it would be difficult to be seriously considered by my peers, simply because this music could not be played live. Pressing the play button on stage was not an option. This newfound problem haunted me for a couple of years, until around 1994, when I crossed paths with the excellent artwork of the Catalan artist Marcellí Antúnez Roca (Roca 2018). *Epizoo* was an interactive mechatronic performance where the audience was given control over a robotic exosqueleton attached to Antúnez himself. I was fascinated by this art piece because it showed both conceptually and technically how a human could be much intimately connected to the machine than I had ever realized. Later I discovered that *Epizoo*'s technical system had been developed by a researcher named Sergi Jordà. He was a lecturer at the recently launched master in digital arts program of the Pompeu Fabra University in Barcelona and that was where I was heading. By 1999 I had been exposed to the work of artists like Michel Waisvisz and Laetitia Sonami, who in my perspective had taken live computer music performance to a new level by creating and performing their own custom DMIs.

In the following years I developed and performed with many of my own DMIs. I was also invited to go back to my home country Portugal and fill the position of R&D Director of YDreams, a newly founded company in interactive digital media that later raised significant capital investment. YDreams became my dream job, where I was able to conduct research, art and business under the same umbrella. I also had the opportunity to work with several Top 100 companies and collaborate with some of the best applied research labs in Europe and Silicon Valley, including the Fraunhofer Institute, Phillips Research, Hewlett-Packard and International Business Machines Corporation (IBM).

In parallel I continued to develop my artwork and around 2007 I finally had the opportunity to do my first artistic residence at the Studio for Electro-instrumental Music (STEIM), under the tutorship of my longtime hero: Michel Waisvisz. Waisviz had an accumulated knowledge about DMIs like few others. In my opinion he was one of the most important DMI performers because he truly mastered his instrument, "The Hands" (Torre et al. 2016). He spent years learning and refining it, until he felt truly connected to the instrument, like an expert with the tool of the trade: effortlessly but proficiently operated. Michel was also a true thinker and someone who didn't shy away from critics and discussion. Realizing he was willing to coach individuals, I grabbed tightly to that opportunity in the hope for a candid and fruitful discussion. I decided to confront Michel with the role of the computer in our instruments. Even with significant experience in DMI use, from a user perspective, I had always felt that the computer kept me from being able to deeply engage with my instruments. If there was a laptop display anywhere within range, I would inevitably focus my attention on the numeric representations of the controller's sensor data, instead of focusing on the instrumental qualities of the tangible interface. Waisviz corroborated this problem and said he simply kept the computer far away from the stage. "Put it under the table! Only then you'll forget about it", he told me. I tried to follow his advise, but somehow it felt difficult.

During 2011 I was on sabbatical and living abroad, but that summer I went to visit YDreams. I told them I had a few free days and that I wished to engage hands-on with some of the newer research projects at the laboratory. After much time in a management position, I wanted to develop something practical during that break. My colleagues presented me with a fresh project they were working on: an aquatic drone. The goal was to have a toy submarine that could be controlled in real-time using a mobile application, with virtual controls and a live video feed from the drone's point-of-view. At the time the BeagleBone (*BeagleBoard* 2018) had just been released. It was one of the first easily accessible Advanced RISC Machine (ARM) single-board computers (SBC) in the market and it seemed perfect for this project: it was very small and worked pretty much like a computer. We ordered some BeagleBones and I ended up developing the video server for the drone. From that moment on, I immediately foresaw the potential of this new class of minicomputers in DMI applications.

A couple of years later I finally decided to pursue my PhD. I contacted Professor Marcelo Wanderley and told him about my ideas about using embedded computing in DMIs, only to find out that, since the launch of the BeagleBone, there had been significant research work in this area, part of it happening right there at the Input Devices and Music Interaction Laboratory (IDMIL). The remainder of this story is presented throughout this dissertation.

1.2 The Promises of Embedded DMIs

The initial motivation for applying embedded computing to DMIs is simply that it seems like a logical progression for this type of instruments. After all, once the DMI has been programmed, the computer serves only as a processing unit and doesn't play any fundamental role in the playing of the instrument itself. If the controller could incorporate the computer, it would no longer be a controller but a full-fledged instrument. The terminology "digital musical instrument" would immediately make more sense. But the more we think about the daily usage of DMIs, the more it becomes evident that embedded computing could solve other problems. DMIs would become much simpler to setup and operate. Their reliability and longevity could be increased due to their solid-state designs. They could embody the concept of computers made specifically for playing music, with specialized designs and increased usability. From now on we will call them embedded DMIs.

1.3 Contributions

Research on DMIs is still somewhat at its infancy. It's only been about 20 years since we have had access to the computing power required for real-time audio applications. When it comes to embedded computing, we are looking at an even narrower time interval. Just during the course of this research, we have witnessed the introduction of many new technologies and applications of embedded DMIs. It reinforces our perception that it represents an area of considerable interest, but it also means that much of the current research is somewhat speculative. We are just giving our first steps in what can be done with these new technologies and how to apply them in DMIs. Only time and a continued research effort could prove that we are heading in the right direction. For now, we are still planting the first seeds.

This dissertation's main contributions to the field are:

1. State of the Art—an in-depth state-of-the-art review on applications of embedded technologies to DMIs. Because this is a considerably new area of research, many of the presented cases were yet to be compiled into a compact and comprehensive format.

2. The Prynth framework—a group of tools for the creation of embedded DMIs, developed by myself and a team of collaborators at IDMIL. The Prynth framework offers many advantages for researchers and artists that want to build their own embedded DMIs. It provides higher-level components that work as a good starting point for development, diminishing the complexity and duration of implementation. It also suggests novel interaction models for embedded DMIs and respective workflows.

3. Academic Publications and Conferences—Our work on embedded DMIs resulted in three academic papers. The first was "Practical evaluation of synthesis performance on the Bea-

gleBone Black (BBB) (Franco and Wanderley 2015), presented in the 2015 edition of the international conference on New Interfaces for Musical Expression (NIME), which took place at the Louisiana State University in Baton Rouge, Louisiana. The research presented was part of our initial findings with ARM-based SBCs, before Prynth was officially released to the public. At the time, we were using the BeagleBone Black to prove that this class of SBCs could handle the computing required in a DMI. Since then we abandoned the BeagleBone Black and decided to migrate Prynth to the Raspberry Pi (RPi).

The second paper is titled "The Mitt: case-study in the design of a self-contained instrument" (Franco and Wanderley 2016), presented at the 12th International Symposium on Computer Music Multidisciplinary Research (CMMR 2016), which took place at the University of São Paulo in Brazil. In this paper we described *The Mitt* (see section 6.2.1), the first instrument to use the Prynth framework. The initial goal of this paper was solely to present this particular DMI, but the vivid discussions we had with our peers during the conference led to an early showcase of Prynth as the underlying technology that made the Mitt possible. The reception was so enthusiastic that we were invited by the committee to further develop our paper for the extended notes of CMMR 2016, published by Springer. This extended paper was then retitled "Prynth: a framework for self-contained digital music instruments" (Franco and Wanderley 2017).

4. Dissemination activities—the dissemination of the Prynth framework resulted in many other parallel activities, including public presentations, press articles and participations in concerts and art festivals.

5. Instruments built with Prynth—a collection of instruments built using the Prynth framework, allowing us to examine the individual goals of users and how our tools respond to their needs.

6. Discussion—a critical reflection on the advantages of Prynth's embedded DMIs and potential avenues for future research.

1.4 Collaborators

During the Winter of 2018, three graduate students contributed to the development of the Prynth framework. The first two were Harish Venkatesan and Ajin Tom, two students of McGill's music technology master in arts. They created an instrument with Prynth for Marcelo Wanderley's class and later we discussed possible improvements and new features of the framework. Venkatesan and Tom offered their help and this boost in manpower allowed us to invest in a full reimplementation of the acquisition system, adding support for digital sensors, more advanced data processing and the revision of communication protocols. We also improved Prynth's code editor and added important interaction features, such as parenthesis matching actions and syntax highlighting. Their contribution is reflected in sections 5.6 and 5.9.8. The third collaborator was Antoine Maiorca, an exchange student from Université of Mons in Belgium. Maiorca was in charge of researching technologies for Prynth's graphical user interfaces (GUI) and implementing a functional prototype. After some iterations, we reached a satisfactory model, later polished and included in the official Prynth distribution. Maiorca's work is reflected in section 5.9.1.

1.5 Structure of the Thesis

This thesis is divided into the following chapters:

- Chapter 2—"Background": introduces computer music research and its path towards the development of DMIs.
- Chapter 3—"Limitations of DMIs": discusses some of the limitations of the current generation of DMIs, with special emphasis on those that derive directly from the use of traditional computers.
- Chapter 4—"Embedded Digital Musical Instruments": starts by defining what is an embedded DMI, after which it presents the technologies that could be used in the construction of such instruments and the state of the art, constituted by a selection of cases that include

both instruments and development frameworks.

- Chapter 5—“Prynth: a Framework for Embedded DMIs”: describes the features and technical implementation of the Prynth framework.
- Chapter 6—“Results”: gathers all the results from the Prynth project, including dissemination activities and several instruments built by users.
- Chapter 7—“Discussion”: presents a reflection about the reasoning behind Prynth’s architecture and how the framework addresses some of the problems identified in chapter 3. We also discuss Prynth’s strengths and weaknesses and how it compares to competing frameworks. Finally, we suggest future directions of research on embedded DMIs.

Chapter 2

Background

In this chapter we offer a brief overview of the evolution of computer music and discuss some of the unique qualities that make the computer an important tool for electronic music. Armed with this knowledge, we can better understand the context that led to the development of DMIs. We draw special attention to how these instruments fulfill an important role of bridging human and machine, by focusing on human gesture, language and intention. DMIs allow computers to be used in ways that are closer to more traditional instruments, resonating deeply with musicians whose learned skills are deeply related to an embodied interaction.

2.1 The Experimental Nature of Electronic Music

Even with almost a century of history, electronic music continues to evolve. It is in constant motion, following the frantic rhythm of its technological foundations. Researchers and companies continue to invest in the fields of digital signal processing, human-computer interaction (HCI) and artificial intelligence with the goal of creating better musical tools. In turn, artists continue to find new creative avenues in composition, performance and co-creation with machines.

Electronic music is an art form that is intrinsically connected to experimentation. Its appeal resides exactly in the discovery of the new: a search for intellectual and emotional challenge. To meet this challenge, the musician must chart new territories of artistic expression and be willing to try, fail and reiterate. Any electronic musician could comfortably sit between the artist and the scientist.

This thesis is the reflection of a research imbued with this spirit. It proposes the new, crossing into the field of speculative design. It tries to imagine a possible future for digital musical instruments, by offering the tools to shape what that future could look like. But to speculate about the future, we must first look into the past and understand the path that led us here and now.

2.2 From Mainframes to Laptops

In 1928, Lev Termen invented the Theremin, considered one of the first electronic music instruments and a symbol of the promises of technological progress. An instrument played in the air, without physical contact, surely supported Arthur C. Clarke's law that "any sufficiently advanced technology is indistinguishable from magic" (Clarke 1982). The Theremin won some notoriety with the acclaimed virtuosic performances of Clara Rockmore and was also used to create some of the signature sounds of Hollywood's science fiction and horror movies of the early 1050s. Nevertheless, the Theremin and other early instruments, such as the Telharmonium, the Trautonium or the Ondes Martenot, remained largely unknown to the larger public. The more serious inception of electronic music happened by the early 1050s, with the emergence of three academic schools:

- The French, led by Henry and Schaeffer at the studios of Radiodiffusion Française (RDF), exploring the manipulation of closed loop records and magnetic tape in the creation of Musique Concrète.
- The German, associated with the development of analog synthesis at the radio studios of Westdeutscher Rundfunk (WDR) in Cologne, with composers such as Karlheinz Stockhausen and Gottfried Michael Koenig.
- The American, with Max Mathews at Bell labs creating the first computer programs to generate sound.

Naturally there were other important contributors to electronic music, like the British at BBC or the Japanese, with the inception of companies like Yamaha and Korg. The strict categorization of these early schools is slightly reductionist, and although there was an inevitable cross-pollination between establishments, it shows the relations to particular technologies and its consequence to musical aesthetics. The French became the precursors of what can now be considered sampling, the German of analog synthesis and the American of digital synthesis.

In this historical panorama, computer music is often considered to be a later effort in the genesis of electronic music, probably due to the expensive and heavy monolithic mainframes of the time, where the computation of the simplest algorithm could take hours or even days. But no matter how cumbersome or inconvenient, computers were quickly embraced by those who had access to them. The computer became a strong bridge for the formalization of music as mathematics, reflected in the work of Iannis Xenakis (Xenakis 1992) and Gottfried Michael Koenig (Koenig 1983), who were among the first composers to pursue algorithmic composition. The computer was also capable of rendering entirely synthetic sounds, detached from the impositions of nature's mechanic and acoustic phenomena. Jean-Claude Risset pioneered the extensive use of synthetic sound generated with the computer.

It is beyond our scope to extensively present all the details of early computer music. For those interested, we point to the comprehensive texts of Joel Chadabe (Chadabe 1997), Peter Manning (Manning 2013) and Nick Collins (N. Collins et al. 2013). We will instead jump forward a few years and discuss the repercussions of the subsequent democratization of computers.

This next important period for computer music happened by the early 1980s, with the advance of microprocessor technology and the commercialization of affordable personal computers. In the market from 1983 to 1986, the Commodore 64 (C64) was one of the pinnacles of this era. It had much better multimedia capabilities than the competition, outselling IBM, Apple, Atari and Sinclair in their 8-bit home market. With a dedicated audio chip, the sound interface device (SID) (Dillon 2015), it was capable of running three audio channels in parallel, each with an oscillator (sawtooth, triangle, pulse or noise waveforms), amplitude envelope and ring modulation. The resulting mix was then passed through a multimode filtering stage. The C64 also ran the first software sequencers, called music trackers, characterized by their vertical timeline arrangement. Their interfaces were similar to a modern spreadsheet application, with arrays of cells with numeric data that progressed over time, similarly to a music box. These trackers could also be used to send control data to external gear via musical instrument digital interface (MIDI) protocol (*MIDI Manufacturers Association* 2018). With these revolutionary multimedia capabilities, the Commodore 64 and its later rival, the Atari ST, became so popular that they gave birth to a unique musical style called Chiptune (Leonard et al. 2014). Chiptune, characterized by primitive digital synthesis, is still performed to this day, continuously attracting new musicians and audiences.

In the beginning of the 1990s, computers based on the IBM x86 architecture and coupled with the popular Sound Blaster sound cards became the first viable systems for running full-fledged audio sequencers. The Sound Blaster cards had a polyphony of up to 11 voices of frequency modulation (FM) synthesis. By this time Chowning's technique (Chowning 1973) became one of the most efficient solutions for timbre variety, leading to its extensive licensing in the multimedia market. Later Sound Blaster versions went from 8-bit to 16-bit architectures, with full-duplex audio at 44.1 kHz (compact disk quality) and incorporating Ensoniq's wavetable solutions, commonly used as a cheap alternative to hardware samplers.

Desktop computers, sound cards and sequencers kept evolving to become *de facto* tools in professional sound studios. Together with the widespread Internet access, they also opened the doors for the concept of an affordable home studio, allowing almost anyone to compose, perform, produce and distribute music.

Software virtualization is another technology that became fundamental to the digital audio workstation (DAW). All modern sequencers can incorporate processing from third-party software modules, commonly called *plug-ins*. These emulate audio processors, like a reverb or a compressor, or instruments, like a violin or a hardware synthesizer. They typically have a front end GUI that mimics their hardware counterpart through a skeuomorphic design. Using the DAW, a user can instantiate plug-ins at will and arrange them into serial or parallel audio processing chains, in the same fashion as audio engineers did for decades with hardware devices and patching of audio signals.

While the revolution of the recording studio was taking place at home, academic computer musicians kept investing in advancing algorithmic composition and synthesis languages. Since his early efforts in the late '50s, Max Mathews kept developing new versions of Music-N languages (Manning 2013), which in turn became models for other domain-specific languages (DSL) of the 1980s and the 1990s. These include CMusic by Richard Moore (Moore 1982), CMix by Paul Lansky (Lansky 1990), and Csound by Barry Vercoe (Vercoe 1993). Another notable derivative of Music-N languages was Common Lisp Music (Schottstaedt 1994). Lisp was frequently used in algorithmic compositions of the Serialist tradition because it facilitated the representation and transformation of arrays with musical data like pitch or duration.

Although these languages had their differences in syntax and architecture, they all shared the same underlying model where the user would write musical programs split into two components: an arrangement of DSP blocks defining a proto-orchestra of synthesizers and a score to play those same virtual instruments. A compiler would then process the source code and output the corresponding sound file. This procedure could take from minutes to hours, and only after its completion could the composer hear the resulting sound.

All of the Music-N derivatives established themselves through a significant following by academic composers. Some are still seeing improvements to this day, especially Csound, which has since integrated real-time processing and more recently reinvented itself as a library to be used in conjunction with other programming languages (Lazzarini, Yi, Timoney, et al. 2012; Lazzarini, Yi, Fitch, et al. 2016).

The next significant step in computer music was the advent of real-time DSP. Although previously demonstrated with mainframe computers, it was only by the mid-1990s that real-time DSP became a reality for most users. With it, new interaction paradigms and programming languages emerged. The computer could now become not only a composition tool but also an interactive instrument.

One important software of this era was Max (Cycling '74 2018), developed by Miller Puckette in 1988, at the Institut de Recherche et Coordination Acoustique/Musique (IRCAM). Originally called Patcher, Max was first developed to control IRCAM's Sogitec 4X synthesis system (Puckette 2002) over MIDI. The big novelty in Max was visual programming (Puckette 1988). Musicians could now write algorithms using a patching system, creating complex programs from smaller functional black boxes that were connected using virtual cables. Visual programming opened the doors of computer music to less experienced programmers, quickly becoming one of its *lingua francas*. In 1989 IRCAM developed the IRCAM signal processing workstation (ISPW), an expansion DSP hardware that, when combined with the NeXT computer, allowed for real-time audio processing in Max (Puckette 1991). By 1990, Opcode Systems licensed Max and developer David Zicarelli took over the refinement of Max for commercial purposes. By 1996, as real-time processing advanced, Puckette released the first versions of Pure Data (PD) (Puckette 1997), an open-source alternative to Max that brought to general-purpose computers the same DSP capabilities that were before only possible with the ISPW platform. By 1997 Max had also added audio processing in the form of a library called "Max Signal Processing" (MSP). With Max/MSP and Pure Data musicians could now develop all kinds of audio synthesis and processing using lower-level objects such as signal generators, filters and delay lines. Furthermore, these audio processes could now be strongly tied to interactive behaviors, which would pave the way for Digital Musical Instruments. After this period Max continued to evolve, adding new functionalities, including video processing through Jitter, a package for matrix calculation and vector graphics, and a software development kit (SDK), allowing third-parties to create their own sets of objects called "externals".

In parallel with Max and PD, another notable software that had much influence in the development of interactive music systems was SuperCollider (McCartney 1996; McCartney 2002). Created by James McCartney in 1996, SuperCollider is a scripting language with similarities to Smalltalk and C, making it more akin to Music-N languages than dataflow languages. Unlike Max or PD, SuperCollider contemplates the dynamic management of real-time audio processes. It uses an architecture where a virtual machine interprets and executes new programs on-the-fly, contrarily to the compilation stage required in more traditional software. Dynamic programming adds extreme flexibility to interactive music systems because it allows the creation and destruction of DSP processes during performance and without interruption. Since its inception, SuperCollider has seen three major revisions, with the latest becoming freeware and open-source. Like Max or PD, SuperCollider is highly extensible, and its developer community continues to enhance its features.

Innovation in software was accompanied by its hardware counterpart. The increase in computational power led to a consumer market with affordable computers that had enough power for real-time DSP processing. These machines also became much smaller and thus portable, easily finding their way onto musical setups and stage performances in the form of laptop computers. One of the landmarks in this new generation of computers was the Powerbook G3, an Apple Macintosh laptop commercialized between 1997 and 2000, which became almost omnipresent in most computer music performances of the time.

This is a very short overview of the history of electronic music, omitting many important details and people for the sake of brevity. It does however offer a glimpse of the path that led to the more widespread adoption of computers in music and the creation of the tools and base technologies that would form the foundations of computer music.

2.3 Idiosyncrasies of Computer Music

With the widespread access to computers and mature software, computers became an essential tool in music production. Today any recording studio uses the computer as a central control unit. Producers use virtual plug-ins not only to mix and master sessions, but also creatively, with heavy use of particular software instruments or audio processing techniques that become blueprints for specific music styles. A good example is Dubstep, a genre that evolved in the end of the 1990s from early UK styles like 2-Step and Drum 'n' Bass. This type of dance music uses a specific set of techniques, like basses with heavy sawtooth waveforms that are filter-modulated to create a wobbling effect. Many different synthesizers can accomplish these effects, but producers often point to a specific virtual instrument—Massive by Native Instruments (Native Instruments 2018). The connection between Massive and the Dubstep musical style is evident in the amount of YouTube videos explaining how to accomplish the wobble effect on that particular software instrument. Another example is the Auto-Tune plug-in, commercialized by Antares Audio Technologies (Antares 2018). This effect was originally developed to correct slight pitch deviations in vocal material, but modern producers soon discovered that extreme parameters created a robotic vocal effect similar to Moog's vocoder (Bode 1984). The subversion of the original intention of this plug-in saw extensive creative use by artists such as Cher and late 2000s hip-hop act T-Pain.

The computer not only left a mark in popular music production but it has also profoundly impacted experimental music. Those more inclined to the exploration of unconventional sonic territories see the computer as a door that opens to a new world of possibilities. The more significant of these new possibilities is probably the creation of synthetic timbres. The main synthesis method employed in early analog devices is commonly called subtractive synthesis, a source-filter model where a combination of unit generators form a sound source that passes through an equally arbitrary combination of filters to shape its spectrum. Subtractive synthesis is probably the most disseminated synthesis method. It has been in use for 50 years and musicians have mastered its intricacies and techniques. Similarly to the analog synthesizer, the computer can also perform subtractive synthesis, with the difference that it can easily surpass the limited number of generators and filters that are viable through analog circuitry. But the real strength of the computer lies in its ability to perform mathematical calculations and run complex algorithms used in more advanced sound synthesis, such as granular synthesis (Roads 2004), physical modeling (Karplus et al. 1983; Smith 1992) or the phase vocoder (Flanagan et al. 1966; Portnoff 1976).

Granular synthesis relies on the playback of very small sound fragments (the grains) at extremely high frequencies, producing an auditory illusion of continuity, a technique that could find its analogy in Pointillism from visual arts.

Physical modeling simulates sound sources and resonant bodies to replicate natural acoustic phenomena. Although the original goal of physical modeling was to achieve compelling simulations of acoustic instruments, musicians have also used its principles to create fictional instruments that are only virtually possible, like a wind instrument with an extremely long pipe or a string that can vibrate for much longer than its real-world counterpart (Eckel et al. 1995).

The phase vocoder allows for the direct modification of sound spectra. Using the fast-Fourier transform (FFT) to convert a sound between its temporal and spectral domains, it is possible to modify the phase and amplitude content of particular frequencies and finally resynthesize the sound for playback. The most common uses of the phase vocoder are independent time stretching or pitch shifting, but as with physical modeling, musicians have creatively explored its possibilities to sculpt unusual sounds.

Algorithmic composition (Essl et al. 2007; Edwards 2011) is another practice that is deeply related to the possibilities of the computer. This type of musical composition uses mathematical algorithms to create musical scores. Instead of a linear score, the composer develops a computer algorithm that will produce music according to specific rules. The origins of music composition as mathematics date back to the serialist tradition of composers like Schoenberg, Webern and Boulez, using permutations of an arbitrary fixed number of values to create all the notes of a musical piece.

The use of mathematical modeling in computer music also provides a handle to the inclusion of other scientific fields, like computer science, biology or physics, and it is common in today's musical pieces to use fractals, stochastic models, genetic algorithms or artificial intelligence.

Artificial intelligence is another field of particular interest to music because it represents the possibility of machines participating more actively not only in music composition but also performance. Examples of computers that play music with humans include the commercial Band In A Box (PG Music Inc. 2018), offering a full accompanying virtual band that can play numerous popular styles like Jazz, Rock or Bossanova, to more complex systems developed in academia, like Rob Rowe's Cypher (Rowe 1992) that explores the computer as an active player that can listen, comprehend and react to musical input from human performers.

Another important feature of the computer is connectivity. Virtually any modern computer includes networking technologies such as Ethernet, Wi-Fi or Bluetooth, through which it can connect to other computers or the Internet. Mobile devices can access cellular networks anywhere and at anytime, inferring their precise location via satellite connection to the global position system (GPS). Computer music has incorporated all of these technologies, from ensembles that share musical data through local or remote networks (Barbosa 2003), to interactive audio guides that deliver audio content according to the users geographic location (Gaye et al. 2006).

These examples represent just a small fragment of the many idiosyncrasies of computer music. They offer opportunities for artistic creation that are not possible with other musical instruments. They also demonstrate how the computer can become a valuable tool for musicians interested in experimental music and the crossing to other knowledge domains that could contribute to new forms of composition and performance.

2.4 Performing with Computers

The consolidation of computers as musical machines and their increasing presence on stage also gave origin to new types of performance practices. The concept of *laptop performance* implies that the computer surpasses its function as an accessory, to instead become the main (and sometimes sole) instrument of a musical performance. A typical laptop performance consists of the real-time manipulation of synthesis parameters on a computer program, such as a sequencer or a Max patch. These interactions are generally conducted via traditional computer interfaces, like the mouse, keyboard and display.

The interaction with traditional computers involves the use of windows, icons, menus and pointers (WIMP), which present serious limitations regarding the performative gestures and musical styles they permit. A virtual instrument or audio effect will often have a GUI, with spatially distributed buttons, sliders or dials that are mapped to single parameters. The user moves the pointer via mouse to a specific control point on the GUI, clicking and dragging it to modify values. In this scenario, the user can only control one parameter at a time and multidimensional control is often left to predefined automation. We will further discuss the topic of mapping in digital musical instruments, but for now, we will simply point to the work of Andy Hunt and collaborators, who have conducted in-depth studies about the importance of multi-parametric and cross-coupled mapping (Hunt, Wanderley, and Paradis 2003). Input accuracy is also limited by the resolution of the mouse and display, which could also impact the ability for rigorous continuous control.

The losses in multi-parametric and sensible input are greatly compensated by the flexibility of the medium, where composers and performers are free to choose software with radically different interfaces and interaction models. Those capable of programming their own software using languages like Max or SuperCollider can achieve even greater sonic variety and explore the freedom of creating their own *modus operandi*. Since there are so many distinct approaches to computer music performance, specific techniques often tend to become trademarks for particular artists or subgenres. One good example is the work of Autechre, the English duo of Rob Brown and Sean Booth, who by the mid-1990s created some of the music that was later described as essential to all of the subsequent post-digital and glitch scene. Autechre incorporated generative and algorithmic processes in their music, pushing to a greater audience new perspectives on musical structure. Many people had never heard such music and were naturally curious about the processes behind such fresh sonic results. During the following years, Autechre's techniques became a topic of discussion among electronic musicians, until it finally came to light that Brown and Booth were developing their own software, using an obscure program called Max (until then somewhat confined to academic circles) and that it would be difficult (if not impossible) for others to reach the same exact results.

Given all this diversity, it is useful to understand if there is a possible categorization of approaches to interactivity in laptop performance. Joseph Malloch and colleagues have taken Jens Rasmussen's models for human information processing (Rasmussen 1986) and extrapolated them to a possible categorization of interaction models for digital musical instruments, divided into skill-, rule- and model-based models.

> "Briefly, skill-based behavior is defined as a real-time, continuous response to a continuous signal, whereas rule-based behavior consists of the selection and execution of stored procedures in response to cues extracted from the system. Model-based

> behavior refers to a level yet more abstract, in which performance is directed towards a conceptual goal, and active reasoning must be used before an appropriate action (rule- or skill-based) is taken" (Malloch, Birnbaum, et al. 2006).

This categorization is useful because it highlights one of the main aspects of laptop performance. They often tend to fall more into a category of rule- and model-based interactions, since the musician fulfills the role of a conductor that operates a machine through complex semiotics. In contrast, the direct manipulation of a tactile sound-producing object, such as any traditional musical instrument, relies heavily on skill-based interactions dependent on the dexterity and embodied knowledge of the player.

Another important aspect of laptop performance is that it doesn't strictly require energy input from the performer. Ultimately, the computer has the ability to keep playing without any human intervention. Even when it requires motor action, the spatial ranges of the mouse and keyboard are quite narrow. Additionally, the computer is often placed on a table, in a stationary position. This leads to a considerable reduction of mobility on stage, since the computer requires almost constant visual focus. Due to this low amplitude of gestures, it becomes difficult to establish a clear causality between gesture and respective sonic output or to observe any ancillary gestures or emotional expressions of the performer.

2.5 Causality, Ritualism and Expression

Kim Cascone, one of the early proponents of laptop music, recognizes the problem of establishing causality in electronic music:

> "Unfamiliar codes used in electronic music performance have prevented audiences from attributing 'presence' and 'authenticity' to the performer ... During laptop performances, the standard visual codes disappear into the micro-movements of the performer's hand and wrist motions, leaving the mainstream audience's expectations unfulfilled" (Cascone 2003).

Without meaningful gestures or an understanding of causality, a laptop performance (like acousmatic music before it) tends to shift the focus to the musical content and to be less concerned about the means of creation. This concentration on audition is conducive to activities like *deep listening*, a term coined by Pauline Oliveros (Oliveros 2005) to refer to a mindful act of listening that requires attention and a meditative state of mind. People engaged in deep listening often close their eyes to shutoff any visual stimuli. Cascone calls it an “active reception”, as opposed to a “distracted reception”, characterized by the immediate gratification of popular entertainment (Cascone 2003).

While this reasoning certainly constitutes a valid artistic manifesto, it is impossible to deny that laptops defy the conventions of stage performance. For most people, a concert is the opportunity to witness the exquisite instrumental skill of individuals that embody that particular music. It is also not an individualistic experience. Instead, it fulfills a social function and the reward is the shared celebration of artistic talent.

Garth Paine argues that live music “is far more than purely entertainment; it is a ritualized form of collective conscience, a rare opportunity within modern Western society for communal catharsis” (Paine 2009).

Marc Leman also sees live music as a social and ritualistic activity that generates “human attraction and an urge for interaction”. People are “energized” and empowered by the rewarding of their expectations, fundamental to Leman’s models for expressive interaction.

> “Expressive interaction with music thereby relies on the remarkable capacity of humans that they are able to align movements with sounds ... in music performance we are dealing with corporeal articulations that are time-critical and fine-motoric, often in combination with physical effort and in relation to mediating roles of motivation and reward systems” (Leman 2016).

These rewards relate to our capacity to anticipate the outcomes of a musical gesture executed by the performer, something that is almost impossible in laptop performance.

Laptop performances can also present limitations in terms of musical style. Most rule- or model-based performances depend significantly on prepared materials, so it becomes difficult to improvise or to play music that depends on fast and radical shifts. Furthermore, most computer interfaces cannot capture the finesse of micro-gestures, like the ones used for producing a vibrato on the violin or ghost notes in percussion.

> "It is critical that new instruments be developed that facilitate and nurture this expression, musical instruments that facilitate subtlety and nuanced expressivity of the same granularity as traditional acoustic musical instruments" (Paine 2009).

Computer music performance seems stuck in a conflict of principles and aesthetics related to who should occupy the central role: the human or the computer. While some defend that physicality is irrelevant because the machine could perform much faster and accurately, others consider that traditional computer interfaces are not only insufficient to some forms of musical expression but also less rewarding to an audience. The latter group embarked on a quest for physicality in computer music, eventually leading to considerable research on haptic interfaces that could bring human skill back into the equation.

2.6 Haptic Interaction

Human-computer interaction is a discipline that studies the methods through which humans interact with computers, combining knowledge from computer science with fields like cognitive science and design. Its primary objective is to provide novel interaction models that would contribute to increased usability in computer applications.

Music seems to be a natural candidate for experimentation in the HCI field, particularly in regards to tangible interaction. This can be observed in the work of the early proponents of tangible interaction: Bill Buxton (Buxton 1977; Buxton et al. 1978), George W. Fitzmaurice (Fitzmaurice 1996) or Joe Paradiso (Paradiso 1997).

In the Tangible Bits project, Hiroshi Ishii introduced the notion of physical objects as potential handles to digital information, defending the cultural importance of instruments and the need to reflect on the value of their tangibility.

> "Long before the invention of personal computers, our ancestors developed a variety of specialized physical artifacts to measure the passage of time, to predict the movement of planets, to draw geometric shapes, and to compute ... We were inspired by the aesthetics and rich affordances of these historical scientific instruments, most of which have disappeared from schools, laboratories, and design studios and have been replaced with the most general of appliances: personal computers. Through grasping and manipulating these instruments, users of the past must have developed rich languages and cultures which valued haptic interaction with real physical objects" (Ishii et al. 1997).

Buxton constantly refers to the importance of artists and musicians in HCI research, alluding to how performers might be responsible for the most demanding scenarios in interaction design.

> "... there are three levels of design: standard spec., military spec., and artist spec. Most significantly, I learned that the third was the hardest (and most important), but if you could nail it, then everything else was easy" (Buxton 1997).

The justification for Buxton's remarks lies in the fact that traditional computer interfaces are not up to par with the skill of trained performers. They are not able to capture human gesture with enough finesse to fully conduct the performer's artistry. Therefore we should invest in the development of computer interfaces that empower those skills, so musicians might take better advantage of a competence earned through years of practice.

2.7 Musical Controllers

Tangible interfaces are a significant part of how we interact with a computer. They are the physical input devices through which the user operates the machine. Though the keyboard and the mouse are still the most common input devices, there has always been a considerable investment in the creation of alternative tangible interfaces. They can increase usability by being specifically designed or adapted to particular tasks. A computer flight simulation is more realistic when controlled by a joystick. With a graphics tablet a visual artist can draw more naturally on the computer. Professional video editors use jog wheels to more efficiently shuffle video footage in editing software.

One of the first input devices for computer music was developed by Max Mathews. The Radio Baton (Mathews 1989; Mathews 1991) was composed of a horizontal electromagnetic plate that was used to track two handheld batons in tridimensional space. Mathews used the Radio Baton as an expressive controller for GROOVE (Mathews and Moore 1970), the first computer program to control a synthesizer in real-time.

Another early example of alternative input devices in digital synthesizers was the light pen, a digitizing device with the format of a regular pen and that was used to draw directly on cathode-ray tube (CRT) screens. Peter Vogel and Kim Ryrie used the light pen to control the Fairlight (Levine et al. 1980), one of the earliest fully digital sampler systems. Xenakis also adopted the light pen to draw interactive musical scores for the Unité Polyagogique Informatique du CEMAMu (UPIC) system (Lohner 1986).

The video game industry was perhaps one of the most important proponents of the commercialization of alternative input devices. The light gun, using a principle similar to that of the light pen, was commonly used in shooting games. Joysticks were often used in flight simulators and steering wheels combined with foot pedals in automotive games. The Power Glove (Zeltzer et al. 1994), developed by Mattel for the Nintendo Entertainment System, was a low-cost version of the types of gloves used in virtual reality. It was capable of detecting the inclination, finger flexure and spatial position of the hand. Power Gloves are still often sought-after in the secondary market of vintage controllers. Guitar Hero (K. Collins 2008) was a series of rhythm music games developed by Harmonix and Red Octane in the mid-2000s. They were played using toy controllers that resembled musical instruments, like the guitar or the drum kit. A few years later, the gaming industry introduced motion sensing. The Wii controllers (K. Collins 2008), developed by Nintendo, were a pair of wands resembling Mathews' Batons, capable of detecting position and orientation of the player's hands. Microsoft developed the Kinect (Zhang 2012), a full-body gestural tracker for the Xbox video game console. The Kinect used the time-of-flight of a cloud of projected infrared points to virtually recreate the skeletal positions of up to 4 players that controlled video games via free body gestures, similarly to playing the Theremin. Over the years, many of these video game controllers were repurposed so they could be used in musical applications.

In parallel with these industries, musicians also developed their own input devices for the computer. The greatest catalyst for the development of digital musical controllers happened in the beginning of 1983, with the formulation of the musical instrument digital interface standard, commonly abbreviated as MIDI (Moog 1986). This type of interface would enable electronic instruments to interconnect and share control data. The most immediate use case was that a single controller could now be used for the manipulation of several external sound synthesis and effects units.

The musical keyboard quickly became the most popular MIDI controller, since it represented a known interface that many trained musicians could already play. The bias towards the keyboard is obvious in the MIDI specification itself, which is designed around the abstraction of the piano and the pitches of Western music. MIDI notation is often represented in DAWs using a piano roll metaphor. But these limitations didn't keep MIDI from growing into the *lingua franca* of interactive devices and to become widely adopted even in non-musical applications, such as stage lighting, robotics, video playback or laser shows. As MIDI grew, manufacturers also created many other musical controllers: mixing consoles, breath controllers, drum pads and many others.

Computers quickly became central to the MIDI workflow. Software could be used to more efficiently control, record and playback performances with synthesizers. MIDI tracks can be easily edited or overdubbed with new parameters, offering great control and precision in the studio. Swapping instruments or adding further instrumental layers became a trivial matter. MIDI data could also be easily transformed by mathematical operations, facilitating musical processes like transposition, filtering or harmonization.

With external digital controllers, control became independent from sound source. A percussionist or a saxophonist could now use their skills to play synthetic sounds and even control sound parameters other than musical notes. All this flexibility led to the creation of custom setups and idiosyncratic musical practices that were heavily based on the expert use of musical controllers.

2.8 Digital Musical Instruments

As new musical interfaces quickly became an active discipline of academic research, one of the first significant surveys in this area was *Trends in Gestural Performance of Music*, where Wanderley and Battier (Wanderley et al. 2000) gathered publications on the most relevant research tendencies and organized a round table discussion with several of the pioneers of electronic music, including Bill Buxton, Bob Moog, Max Mathews and Michel Waisvisz.

In 2001 a workshop dedicated to HCI and music was held at the Association for Computing Machinery Conference on Human Factors in Computing Systems. Having attracted a considerable audience, the organizers decided to establish it as an annual academic conference titled "New Interfaces for Musical Expression". During the years following the inception of NIME, we would witness an immense growth of new instruments using alternative controllers combined with the computer, from here onward referred to as digital musical instruments, as defined by Miranda and Wanderley:

> "An instrument that uses computer-generated sound can be called a digital musical instrument (DMI) and consists of a control surface or a gestural controller, which drives the musical parameters of a sound synthesizer in realtime" (Miranda et al. 2006).

The separation between sound source and control mechanism allows musicians to create their own instrumental setups from the combinations of different controllers and software. Moreover, the user decides how the controller will influence the synthesis parameters. This flexibility and creative freedom is even further extended by the capability of developing custom controllers or software. It is undeniable that DMIs provide a fertile ground for experimentation through the ability to customize or even create whole new instruments.

DMIs also add fresh perspectives to computer music performance because musicians cannot only create new sounds but also new ways of playing them. From an artistic standpoint, the effort and expertise required to play the instrument becomes as relevant as the sound it creates. The instrument itself may also acquire artistic significance. For example, it might have a form factor that favors theatrics in detriment of efficiency or ergonomics. In essence, DMI performances recontextualize and attribute new meanings to the relationship between object and actor in live computer music.

2.9 The Architecture of a DMI

Most DMIs share the same fundamental model: the user manipulates an external controller that sends control messages in real-time to an audio software running on a computer.

All musical controllers have some form of embedded sensing to infer how the instrument is being played. Sensors monitor changes in mechanical, thermal, magnetic, electric or chemical properties and generate output signals to quantify those properties. Inside the controller, an electronic circuit reads the sensor signals and sends them to the computer as control values.

Sensors can be either analog or digital. Analog sensors will output a signal in the form of a variable voltage, so using them with a digital system requires a conversion stage. This operation is done using an analog-to-digital converter (ADC), a component that digitizes the analog values and transforms them into digital data. Digital sensors autonomously perform this same operation using integrated components. They readily output digital signals, which can be transported through digital data buses, such as the standards serial peripheral interface (SPI) or inter-integrated circuit (I^2C).

The musical controller incorporates electronic circuits dedicated to sensor signal acquisition, data processing and communication with the computer. The central component of these circuits is a microcontroller. These small and cheap integrated circuits have a dedicated processor, memory, digital-to-analog converters (DAC) and addressable input/output (I/O) pins. They can handle mixed-signals (analog and digital) and exchange data with other devices using standard serial communication buses.

Microcontrollers are scaled for embedded applications and are much simpler and cheaper than computers. They also operate very differently. While computers can run several concurrent processes, managed automatically by the operating system (OS), the microcontroller can only run one program at a time. It does not have an operating system and is usually dedicated to smaller tasks. Since there is no concurrency in microcontrollers, they are optimal for applications that require deterministic processing. Determinism could be defined as the ability to control the flow of the program and assure the completion of tasks with precise timing and no jitter. Assuring determinism is crucial to musical controllers because they need to have a consistent behavior, even if the response is not the fastest or the most precise.

The microcontroller can also be used to apply mathematical transformations to raw sensor data, in order to reduce noise, rescale values or transform the data into more meaningful forms, like for example converting a vector between cartesian and polar coordinate systems.

After data conditioning, the microcontroller must encapsulate said data into a digital protocol understood by the computer, the most common being MIDI and open sound control (OSC) (Wright 2005). There are also different ways to connect a controller to a computer. MIDI's serial data can be sent via 5-pin Deutsches Institut für Normung connector (DIN) and universal serial bus (USB) cables, or using radio frequency transmission via wireless protocols, such as Wi-Fi or Bluetooth. OSC is typically transported using computer network interfaces, via Ethernet or Wi-Fi protocols.

The computer runs a real-time sound synthesis software that listens to these communication ports and internally routes the incoming data to specific musical parameters. Virtually any MIDI or OSC compliant software can be used in a DMI, ranging from virtual instrument plug-ins, to custom software developed in Max, PD or SuperCollider. The latter offer more possibilities in the creation of customized systems. The user is free to define any aspects of the instrument, including sound synthesis and mapping schemes, but also any other interactive behaviors of the device. A

DMI could offer additional forms of visual or kinetic feedback, for both aesthetic and functional purposes, or automate certain musical processes, such as following a score, generating new musical material or even co-playing with the performer.

2.10 Instrument Makers

The development of a custom DMI requires considerable knowledge in technical areas like programming and electronics. At first glance this could seem like a daunting endeavor, especially to those without a formal engineering training. Yet, the development of custom electronic music instruments is a stronger practice today than it ever was. The reason for this growth is a consequence of two major factors: knowledge sharing and widespread access to technological markets.

The Internet has revolutionized communication and learning. It has fueled the expansion of niche communities from all corners of the world, that can now communicate free from geographical barriers. Interest groups altruistically share knowledge on topics they are passionate about, often without any monetary compensation. Content channels such as forums, websites or video providers abound with educational content that challenges the norms of more conventional pedagogy. From new cooking recipes, to electronics or programming, it is almost certain to find individuals on the Internet that are willing to explain, teach and coach. This cultural background became fundamental for the *Maker* community, a DIY subculture that is engaged in the independent development of new artifacts, merging disciplines like arts, crafts, electronics and programming. Websites like Make (Maker Media, Inc. 2018), Instructables (Autodesk, Inc. 2018) or Hackaday (Hackaday.com 2018) are excellent archives of this communal knowledge.

Global markets and the strengthening of worldwide distribution channels were also fundamental to this technological appropriation. Today any consumer can directly access many electronic components that were once reserved to large scale manufacturing. Online distributors offer extensive information about their products through digital catalogs and ship low volumes of components directly to the consumer. Custom printed-circuit boards (PCB) can now be manufactured in low quantities and delivered to the door in less than two weeks.

The easy and cheap access to all of these resources became crucial to DMI makers, but the levels of complexity involved in electronics development or computer programming could still be discouraging for many, no matter their enthusiasm or autodidact skills. Higher-level development frameworks serve these less experienced developers, so they can concentrate more efficiently on reaching their goals. They can circumvent the hurdles of low-level development and possibly the reinvention of a worst wheel. The Maker community is actively engaged in the production of said frameworks, most of them freely shared with the public under open-source licenses.

2.11 Crafting a DMI

There are different motivations for the creation of a new musical instrument. The musician could be interested in exploring a particular set of performative gestures, finding new forms to control a specific type of sound synthesis or increasing engagement with an audience. Interesting instruments can also emerge from the sonification of mechanical contraptions or the exploration of accidental discoveries in electronics or programming.

Prototyping usually starts by understanding the relationship between the instrument's properties and how they are going to be measured and applied as control signals. Knowing what to measure conditions the selection of an appropriate set of sensors, which involves an evaluation of technologies, complexity, cost and form.

The next step is to create electronic circuitry prototypes, usually done using breadboards or veroboards, where components and cables can be easily connected or swapped. These prototype circuits are sometimes sufficiently functional to test signal acquisition programs and sensing technologies. The choice of which microcontroller to use depends on many factors, but usually the bottleneck in sensing applications is the number of analog inputs, since each analog sensor requires a dedicated ADC channel.

The next stage is to develop the microcontroller program that will not only acquire the sensor data but also condition and encapsulate it in a communication protocol. Here the developer is interested in creating an efficient program that presents the lowest latency possible and minimizes data loss. When using a non-standard protocol, the data stream received by the computer has to be parsed and reformatted into a protocol understood by the destination software.

If the instrument uses a commercial virtual synthesizer, the only remaining task is to map the signals to the available control parameters. Custom applications developed in Max or Super-Collider require additional steps, namely the elaboration of a synthesis scheme and possibly the creation of more elaborate mappings or system interactions.

Once the system reaches its first functional versions, it is finally possible to play it and evaluate the results. Typically this leads to an iterative phase of refinement of both controller and software, which lasts until satisfactory results are met.

Theoretically, the development stage would now conclude, but in practice many of the aforementioned tasks will still be required throughout the instrument's life span. Many custom DMIs demand continuous maintenance, due to their fragile construction or plain deterioration. Moving parts or contact materials tend to degrade and need replacement over time. Electronic components or cabling can also fail, in which case many instruments need to be reopened to trace errors and substitute components, often a laborious and time-consuming process.

Many DMIs are designed for idiosyncratic artistic purposes and not necessarily for commercialization. They serve individual goals, where factors like usability, reliability or production cost become less relevant. But whatever the purpose of the instrument, a design stage could still benefit from the principles and methodologies of user-centered design. In participatory design activities there is an involvement of users and field experts in design activities. They are responsible for testing prototypes and offering feedback based on their personal experiences. Participatory activities could include the creation of non-functional prototypes or enacting possible interactions via *wizard-of-oz* experiments. These activities are often sufficient to demonstrate design problems that can be rectified before further investment. Texts on interaction design by authors Don Norman (Norman 2013) or Bill Moggridge (Moggridge 2007) certainly offer knowledge and guidelines that are applicable to DMI development.

2.12 Conclusions

Computers have become essential tools for music production and performance. The fast-paced progress of music technologies has had two fronts: utilitarian and artistic.

From a utilitarian perspective, computers made their way into music production studios and became essential tools for sound recording, editing and mastering, rendering these tasks more productive and precise. Expensive and monolithic hardware audio devices were substituted by their cheaper and convenient virtual counterparts.

From an artistic standpoint, computers also propelled the creation of new forms of music. They allowed us to hear purely synthetic sounds that were generated using novel synthesis techniques. They were also used as tools for algorithmic composition, pushing the concept of music as a set of rules and mathematical processes. Some of these compositions were true products of computation, oftentimes surpassing the limits of human cognition or dexterity. Early computer-aided compositions were presented as acousmatic or electroacoustic music, with synthetic sound reproduced from tape, compact disk or file.

With the advent of real-time DSP, musicians turned to the possibility of using the computer like an instrument. DMIs offer innovative musical experiences to both audience and performers, drawing closer to centennial traditions in stage performance and allowing for a more symbiotic relationship between human and machine. Since then there has been a significant investment in both applied and academic research with the objective of finding new strategies for musical control. The industry responded with the standardization and commercialization of MIDI controllers, which made DMIs widely accessible to a larger public.

One of the big advantages of DMIs is that they offer many levels of flexibility and customization. Permutable combinations of controllers, software and mappings allow the musician to create personal instruments that better serve individual workflows or artistic goals. With the increased access to technological components and peer knowledge sharing, many musicians learned not only to play but also develop their own DMIs.

Chapter 3

Limitations of DMIs

DMIs represent new ways of playing electronic music and a path to repositioning the human gesture as an important aspect of computer music performance. They offer great flexibility, considering they can be tailored to meet individual needs and be continuously upgraded to incorporate new functionalities. From casual to professional users, DMIs serve many different target groups and use cases.

Given all these advantages and possibilities, it is also pertinent to question the limitations and weaknesses of DMIs. Why aren't they more popular? Shouldn't most musicians aspire to explore this much creative potential? Shouldn't companies be more interested in offering such innovative products? We will discuss some of the weaknesses of computer-based instruments, to later elaborate on how embedded computing could provide solutions to overcome them.

3.1 A Culture of Musical Objects

Before the computer, almost any activity related to music (except perhaps for singing or writing a score) was intrinsically related to the manipulation of physical objects. Musicians have a long tradition of interaction with instruments, including not only musical instruments but also many other electric devices: microphones, sound processors, guitar pedals, amplifiers, cables, adapters, signal converters, mixers, tape recorders, speakers or headphones. All these instruments (in their broader sense) are self-contained, produce consistent results, and have a single and well-defined function. In many instances, they can also be easily combined in a myriad of ways.

The cultural pull created by this heritage of self-contained and single-purposed devices is paradoxically present in the design of most audio software. A large part of virtual audio plug-ins emulate real-world devices. From vintage reverbs to pedal boards, most plug-ins have skeuomorphic GUIs that mimic their hardware counterparts. Software can also emulate physical activities that are common in the studio, like routing signals with virtual cables or stacking processing units on a virtual rack.

Although there are undeniable advantages to virtualization, like lower financial investment, increased mobility or the shear possibility of having a large number of synthesizers on a single track, we think they still work as mere substitutes for the real thing. If we were to give an unrestricted choice between a virtual plug-in and a hardware device, many musicians would probably still prefer the latter. This predilection might be related to particular properties of hardware, such as sound quality or stability, but also from the fact that these are devices designed specifically for music. They have a single purpose and a set of recognizable tangible *affordances* (Norman 2013), therefore representing a more straightforward and clear *embodiment* of a musical activity.

We believe that this strong relationship between musicians and their tangible devices is never going to fade away, so the question is: can computers claim their place as such objects within the musician's workflow? Can they be easily identified by their properties and function? We don't think so, which is even more evident in the case of DMIs. If we were to present a full DMI setup (controller and computer) to any layperson and ask where is the musical instrument, they would probably point to the controller. This means that the computer has no apparent relation with the musical object. It is an external peripheral that only diminishes the autonomy and embodiment of the instrument. This fragmented nature is a potential barrier to considering the DMI as a whole and full-fledged instrument.

3.2 Cognitive Activities

While external controllers and real-time processing brought important instrumental qualities to computers, during the regular operation of a DMI, the user is still forced to use the computer's native interaction models. The development stage of a DMI involves tasks like programming or configuring a virtual synthesizer, but even daily use requires at least starting the computer, loading applications, routing data signals and finally testing that everything works. This is, of course, optimistically assuming that there are no errors or unsolicited requests from the computer itself.

Cognitive psychology is not our field of expertise, but we believe it is safe to say that interacting with a computer involves mental models and cognitive activities that are significantly different from the ones used when playing a skill-based musical instrument. Taking Rasmussen's model (see section 2.4), the computer's operating system will always be closer to a rule-based system.

Modern computers are operated through sequential actions on objects. For example, correcting the amplitude of a sound file implies the following sequence of operations:

- Locate and start the audio editing program.
- Locate and activate the *open file* command on the application menu.
- Navigate through a nested folder structure to find the desired file on the hard disk.
- Select and open the file.
- Navigate to a menu to find the amplitude correction function.
- Apply the amplitude correction with the desired parametrization.
- Save the file.

This *modus operandi* of the computer implies that, before any significant action, the user must first build a functional mental model towards a particular goal, taking into account the rules of operation. Most operating systems with GUIs also require the localization and interpretation of visual information in the form of icons, directories and menus of WIMP interfaces.

Computers are also very intolerant to error. Any deviation from the rules of operation or even a malfunction of the software, and the computer will simply refuse to work. The feedback for these errors is usually in the form of technical (and sometimes indecipherable) jargon and the user is prompted to engage in the construction of further mental models to solve them.

On the other hand, playing a skill-based musical instrument is a considerably different activity. It involves the direct manipulation of mechanisms that provide real-time sonic feedback. There is a parallel control of multiple variables and the player is engaged in a cycle of constant monitoring, evaluation and control of the instrument's state.

Furthermore, there shouldn't be a wrong way to play instruments. They allow us to build our own mental models, sometimes breaking the established rules of how to play them. In the case of new instruments like DMIs, the user must actually go through an exploratory phase and learn the characteristics of the instrument, so that appropriate playing techniques may emerge. To do so, a musical instrument shouldn't malfunction as a consequence of an unexpected operation. It should continue to work, even if the result is not the intended or the most pleasant to the ear.

For these reasons, we argue that interaction with a computer and playing most musical instruments are significantly different activities. The constant shift between these activities, imposed by any DMI, is detrimental to the musical experience. This is probably why many musicians dismiss the use of the computers in their work. Any mention of the computer immediately conveys an idea of complexity and a barrier to music making. The notion of musical expression with an instrument is tightly related to an embodied and playful activity, not dealing with the higher-level cognitive load of a complex machine like the computer.

3.3 Immediacy

An acoustic instrument is always available to be played. An analog synthesizer isn't much different; the only additional step is powering the device. On the other hand, the DMI requires significant preparation until it reaches a ready-to-play state. It involves establishing several connections, booting the computer, launching at least one computer application and finally verifying that it all works as intended. This preparation phase depends on many factors, ranging from the complexity and stability of the system to the proficiency of the user. In the best-case scenario, this setup phase will inevitably take at least a couple of minutes, if not longer.

Most other instruments can just be “picked-and-played”, but with a DMI we can easily imagine some of the following hypothetical scenarios:

- A friend asks you to demonstrate your fantastic new instrument. It doesn’t matter if you still don’t know how to properly play it. The only request is for a simple demonstration in these spare couple of minutes...
- You wake up in the middle of the night having finally been visited by the nocturnal muse. You drowsily tumble through the corridors of the house, until you finally find your instrument. Now to start it, so we can finally capture that... What was it?
- You are happily playing your DMI in this small gig, when there is a sudden power outage. In a couple of minutes, everyone is up-and-ready and staring at you, waiting for the computer to do its thing...

These simple anecdotes easily illustrate the problem. Most DMIs do not facilitate musical impetus. A request to switch musical styles could be problematic. Sometimes the (apparently) simple adjustment of a single parameter could involve considerable time and effort. These could even represent optimistic scenarios, since they don’t consider sudden and unpredictable behaviors from the computer. This lack of immediacy leads to situations where the DMI might seem more of a barrier than a vehicle for artistic expression. Its workflow is not fluid and deterministic, like with most other instruments.

3.4 Reliability

Computers are not very reliable machines. For musicians this is strongly discouraging, particularly to those who play live music, where a critical failure during a stage performance can have catastrophic consequences. This problem has always plagued live computer music and anyone working in the field will corroborate the unease that this uncertainty brings to stage performance. For this reason, in big productions like arena concerts, computer systems are often duplicated for redundancy and include methods for graceful recovery. This is rarely the case with DMIs. The reliability of the computer is undermined by several different factors:

- Multiple Duties

  Because computers are expensive devices, it is difficult for users to allocate a machine just to music production. The computer that serves the DMI is often used for writing documents, navigate the Internet, doing tax reports or even playing video games. This pattern of use often compromises the stability of the system, due to the overload of persistent processes and third-party libraries of multiple software. Operating systems should be resilient to these problems but unfortunately they are not. A freshly installed operating system will almost always display better performance than another that has been in use for some time and for multiple purposes.

- Poor Software

  Some software can be poorly implemented, which results in reduced performance, unexpected behaviors or simply total failure (commonly called a *crash*). In the case of commercial software, the user has little power over these issues. Sometimes it is possible to trace the actions that trigger the problem and even circumvent them, but most of the times the user will not be able to truly fix anything. If instead the user decides to write custom software, there is an added responsibility of assuring its quality and stability. Even then, most software development requires third-party libraries that the developer might not control. Good-quality development tools should help the user in avoiding the pitfalls of programming, but naturally there is only so much any intelligent integrated development environment (IDE), good domain-specific language or optimized compiler can help with.

- Low-quality Devices

Most DMIs run on consumer-grade computers, often built to be competitive in a market inundated with choice. Some manufacturers cut production costs by producing machines with lower-quality components or materials with relative short life spans. Poorly engineered computers could exhibit problems like overheating, sensitivity to supply currents or the failure of integrated-circuit controllers, all of which compromise the performance and stability of a DMI.

- Interference

  Computers might also be subjected to interference from surrounding electrical systems. A good example is radio frequency communication, used in wireless technologies such as Bluetooth or Wi-Fi. The performance of radio frequency communication is greatly compromised by the amount of local transceivers operating on the same spectrum. Another good example is electrical ground loops and their interference with the electronic components and mechanical parts of the computer, often resulting in poor audio quality or unwanted audible artifacts.

- Malware and Bloatware

  The computer can also malfunction as a result of malware. Commonly called *viruses*, malware is computer software designed by third-parties for malicious purposes, like accessing private data or covertly using computing resources. *Bloatware* is software that is pre-installed by manufacturers and can severely compromise the overall performance and security of a computer.

Over the last few years some of these problems have become less severe, but they still reduce the level of confidence on the computer. No live musician will feel comfortable dealing with such a high-risk tool, and one that in the end might create more problems than the ones it solves.

3.5 Longevity

DMIs also tend to exhibit poor longevity. Any machine is bound to eventually fail overtime but computers are particularly problematic. They are relatively delicate machines that nevertheless must resist stresses like prolonged use or harsh environments. The accumulation of dirt or the failure of a mechanical part, such as a fan of the cooling system, could be enough to render a computer useless.

But even if hardware components could withstand physical degradation and show better resilience, the computer is still bound to quickly become obsolete, due to the accelerated pace of its industry. Most computer software is written and compiled to work on specific versions of an operating system (OS). When there are major revisions of an OS, which in today's industry standards happens almost yearly, much of the software also needs revisions to work properly. These short cycles of innovation leave smaller developers in a tight spot because sometimes they lack the resources to keep up with such quick turnarounds. Great software is sometimes abandoned in just a few of these iterations. Those developers that can keep up usually charge additional fees for upgrades, which is also prejudicial for the consumer, who is forced to invest more money to be able to continue to use the product.

Computer hardware faces similar problems. About every five years we see the introduction of new processing architectures, physical connectors and data transport protocols. While they provide better performance, bandwidth or connectivity, they also introduce incompatibilities with previous-generation software and peripherals, which will then need to be ported, adapted or fully substituted by their more modern counterparts. We hope that modern I/O standards like USB C deliver in the promise of one type of connector for all digital devices and longer life cycles.

In this context of accelerated innovation and consumption, the commercial industry expects the user to purchase new technology instead of maintaining old one. There is no motivation for companies to offer continued support or to create products that exhibit significantly longer life spans. Most modern computers have low serviceability and it is almost impossible to independently source and substitute components. For all of these reasons, even an extremely reliable computer will eventually become obsolete.

The reduced longevity of computers is a cause of concern not only for DMIs but to all media art. Many of the early pieces from the pioneers of computer music are today totally irreproducible. They cannot be played live or studied in detail. Fortunately, since the late 1990s, some museums, galleries and artistic institutions have come to understand the importance of devising strategies and concerted efforts for the conservation of digital media art (Digital Art Conservation 2018; Variable Media Network 2018; DOCAM 2018; Rhizome 2018).

3.6 Conclusions

In this chapter we examined some of the problems, limitations and disadvantages of DMIs. While there are certainly many more important aspects that could be pondered, we have concentrated on the issues that stem directly from the reliance on computers.

In a traditional DMI architecture, the controller is used for the main activity of playing the instrument, while the desktop or laptop computer is used as a peripheral device for data processing and setup of the instrument. The fragmented nature of this architecture results in a lack of embodiment of the musical object, which might be a considerable barrier to many potential users.

The operation of a DMI requires long and complicated setups. It also imposes a significantly high level of interaction with the computer and the use of its interfaces and mental models. In the daily operation of a DMI, these are secondary activities and many times completely unrelated to music. Computers are also fragile and unreliable, a deficiency that compromises their use in live performance and that severely impacts the longevity of an instrument.

Many people think of playing music as an activity that suggests playfulness and enjoyment. The computer does not suggest those feelings. It is a machine originally made for work and almost imposed in today's most mundane tasks. It is also a general-purpose tool, with several usability problems when adapted to work as a musical instrument. It lacks important properties like determinism, immediacy, reliability and longevity, all of which are crucial to successful musical instruments.

There is a considerable effort in the research and development of new gestural controllers but there seems to be less preoccupation about many other aspects that influence the usability of DMIs, many of which stem directly from the dependency on the traditional computers. In the next chapter we will introduce embedded DMIs, a new category of instruments that could help overcome at least some of the limitations of the current generation.

Chapter 4

Embedded Digital Musical Instruments

In this chapter we begin by introducing the concept of embedded DMI. We then present a survey of different supporting technologies that could be used to build such instruments and discuss their advantages and disadvantages. Finally, we present the state of the art on embedded DMIs, with examples of different instruments and development frameworks.

4.1 A Definition of Embedded DMI

An embedded system can be defined as an electronic component that operates as an internal subsystem of a larger device and that is responsible for handling one or more functions of that same device. Embedded systems are part of virtually any modern electronic device. Home appliances, industrial machinery, modern cars, toys and many other objects have embedded circuitry.

Embedded computing can be classified as a subclass of embedded systems, where the principal hardware component has a computer-like architecture. Like in other types of embedded systems, the computer interacts with various hardware components, with the difference that it can run arbitrary user programs. This broader definition allows for the inclusion of microcontrollers or field-programmable gate arrays (FPGA), processors that don't necessarily run an operating system but that can be arbitrarily programmed to run complex algorithms.

In some embedded computing applications it can be difficult to say if the device is or not in itself a computer. The best example are smartphones. Are they phones with embedded computing or just computers with a different form factor?

We've already presented a definition of a DMI. It is a musical instrument composed of a controller and a computer, and that can be programmed to have different sonic and interactive behaviors. An embedded DMI is a possible nomenclature for a DMI that uses embedded computing and where the computer is incorporated inside the body of the controller. This subsystem is responsible for carrying out the same exact tasks that were once performed by the desktop or laptop computer. This merge between the controller and the computer makes the instrument fully self-contained and free from the need of any other external peripherals. In other words, the external desktop or laptop computer would disappear from the equation. We could then ask a similar question: is this a musical instrument with computing capabilities or a computer designed specifically for music?

User-level programming is one of the fundamental characteristics that distinguishes a DMI from other electronic music instruments. Therefore it is relevant to differentiate instrument programming from operations of configuration, since they can be easily confused. Some users refer to programming synthesizers as any operation other than playing music. Programming a DMI implies implementing a synthesis scheme, mapping control parameters and developing any other interactive behavior of the instrument. With this definition of instrument programming it is easier to distinguish between embedded DMIs and other standalone electronic music instruments.

As discussed in section 2.3, DMIs are capable of complex audio synthesis and interactive behaviors that many standalone synthesizers cannot reproduce and an embedded DMI should continue to offer those same distinctive features and possibilities.

4.2 Base Technologies

There are many different technologies that can be used in the construction of embedded DMIs, including microcontrollers, digital signal processors, field-programmable gate arrays or systems-on-chip (SoC). We will start by describing these base technologies and then highlight their benefits, limitations and applicability to embedded DMIs.

4.2.1 Microcontroller

Microcontrollers are integrated circuits with fairly small processors and memories. They are usually found in applications that don't require vast amounts of processing power, like home appliances or simpler industrial control. They have many different applications because they can run different programs, stored on their volatile onboard memory and automatically loaded at start-up.

Microcontrollers interact with other electronic components via their addressable input and output pins. Some operate only in the digital domain but most can manage mixed-signals, handling both digital and analog signals. They usually include convenience inputs and outputs for audio, pulse-width modulation (PWM) and serial communication over digital buses (such as SPI or I^2C). More powerful microcontrollers also have dedicated floating-point engines and more advanced connectivity through standards like USB.

One popular family of microcontroller processors is Atmel's AVR (Microchip Technology 2018c), acquired by Microchip in 2016. They are the heart of the Arduino platform (Arduino 2014), which became very popular with the academic and hobby communities for its ease-of-use and support. An 8-bit AVR typically operates at a speed of 8 to 16 MHz and uses a flash memory with sizes from 16 to 256 KB. It exposes several pins for digital and analog I/O, a subset of which generate PWM signals. PWM pins can be used for generating low-quality audio signals (8-bit/16 kHz) by applying a low-pass filter to the PWM signal and correcting the direct-current (DC) offset. Higher-quality depends on the addition of a dedicated DAC that receives serial or parallel digital audio data directly from the microcontroller.

Microchip recently introduced more powerful microcontrollers with 16- or 32-bit buses and central processing units (CPU) with higher clock rates, like the PIC32 or the AVR32 (Microchip Technology 2018b). Some of these newer microcontrollers, like the dsPIC (Microchip Technology 2018a), can do DSP directly on-chip.

There are some examples of electronic music instruments that use AVR microcontrollers, such as the Nebulophone by Bleep Labs (Bleep Labs 2018), the Shruthi by Mutable Instruments (Mutable Instruments 2018) or the Anode by MeeBlip (Anode 2018), a fully open-source synthesizer.

4.2.2 Digital Signal Processor

DSPs are microprocessors that are more efficient in the mathematical operations commonly used in signal processing. They have hardware architectures that promote parallel computing in one single clock, where one instruction can be applied to a data vector: a Single Instruction Multiple Data (SIMD) architecture, according to Flynn's taxonomy (Flynn 1972). With direct memory access (DMA), the processor can access memory blocks without the need to suspend the main program instructions. Circular buffers are also characteristic of DSPs. They save computing cycles by using a single pointer that loops through adjacent memory blocks, thus minimizing the computation to address samples. These techniques can drastically increase performance in applications such as finite-impulse response filters, where multiple coefficients are processed in parallel in a single clock cycle. DSPs are also optimized to perform specific mathematical operations, like multiply-accumulate (MAC), which are the basis for many algorithms in computer music, such as the fast-Fourier transform, heavily used in the conversion of audio data between the time and spectral domains.

DSPs are extensively used in applications such as data compression/decompression, multimedia processing and data analysis. In musical applications, many of today's digital effects racks or guitar pedals use DSPs, although in those applications their functionality is usually predefined and limited to parameter editing.

Early examples of custom DSP audio systems include the work by Barrière et al. (Barrière et al. 1989) with the Motorola DSP56001 chip. Later Putnam describes Frankenstein (Putnam et al. 1996), a system with an arrangement of eight simultaneous Motorola DSP56002 chips, also proposing methodologies for DSP data sharing and interconnection between the processors.

Like microcontrollers, DSP chips include memory and can be freely programmed. Some of this programming can be done in C, but it is more common to use the assembly language for DSP-specific functions. This lower-level operation makes DSPs more complex to program than computers or microcontrollers. For this reason, DSP programming is usually done using specialized IDEs.

Systems like Sonic Core's SCOPE (Sonic Core 2018) or Symbolic Sound's Kyma (Symbolic Sound 2018) offer turnkey DSP solutions for musicians. Their external processing units or DSP cards work in tandem with a host computer that runs simple software to configure the systems. On SCOPE, the user can load virtual synthesizers, including a simulator of a modular analog synthesizer with virtual patching, while Kyma uses a dataflow language similar to Max.

An example of a self-contained instrument built using DSPs is the TouchBox (Bottoni et al. 2007), a touch screen device developed at the Computer Science Department of the Sapienza University of Rome. It uses the Atmel DIOPSIS740, a dual core chip that incorporates both an ARM7 and a mAgic VLIW DSP. The ARM7 core controls execution, GUIs and I/O, while the DSP performs the complex vector operations in a single cycle. To program and upload the new sound synthesis, mappings or GUIs, the TouchBox must be connected to a computer running its host software.

More recently, the use of standalone DSPs in musical applications has decreased substantially. The enormous advance of general-purposed microprocessors reached a point where they can outperform DSPs by shear brute force, making it difficult to justify their cost and complexity. The new trend in DSP chip technology is to use them as processing accelerators that work in tandem with other types of microprocessors.

4.2.3 Field-Programmable Gate Array

In the last decade FPGAs have grown substantially and are currently starting to provide competitive advantages in mass-market deployment.

FPGAs are integrated circuits composed of logic blocks that can be reconfigured after deployment (hence the name), as opposed to application-specific integrated circuits (ASIC) that once manufactured cannot be modified to update their functionality. Since FPGAs can completely rearrange their gate array logic, they represent one of the most flexible solutions for embedded computing. They are also one of the most efficient, since they virtually mimic an integrated circuit.

FPGAs use programs loaded from adjacent nonvolatile memory, setting up the fabric for a specific application. Predesigned modules, called intellectual property (IP) cores, can be loaded to emulate common circuitry functionality and can even implement complete soft microprocessors, occupying only a section of the fabric, while keeping the remainder free for other purposes. Input and output pins are freely addressable and many models are capable of mixed-signal operations, handling both analog and digital signals.

FPGAs are very useful in parallel computing because they are capable of true concurrent processing, as opposed to microprocessors with serial processing. They also have a true real-time behavior, with latency kept within a few clock cycles, while a single data bit is passed between subsequent digital signal processes. FPGAs can also generate different clocks, which allows for a more precise management of processing tasks that happen at different rates.

Another advantage of FPGAs is their scalability and portability. A core can be reused in series or parallel, or easily coupled with others to form more complex signal processing chains. Existing programs are easy to adapt for bigger fabric sizes and different brands. The scalability of FPGAs also adds extreme flexibility. For example, increasing the number of voices in a synthesizer might be as simple as moving to a bigger fabric and cloning the same process as many times as necessary.

The main drawback of FPGAs is their intricacy and complexity because they are programmed using low-level hardware description languages (HDL) such as Verilog or very high speed integrated circuit hardware description language (VHDL). There are several efforts to create code translators from other text-based languages (C and Python) and also common scientific computing environments (MATLAB, LabVIEW), but they usually fail to provide the efficiency of native HDL implementations. Additionally, most of the times FPGAs must still exchange data with other components and devices, so it becomes necessary to deal with the lack of interoperability layers that are usually native to other platforms like microcontrollers or computers. This lack of interoperability could easily boost the complexity of implementations on FPGAs.

To address this problem there is an increasing number of hybrid systems that couple discrete microprocessors with FGPAs. The microprocessor facilitates operations like system communication with other components and shares digital data with the FPGA fabric via memory blocks made available to both chips. These hybrid systems provide much more flexibility and a wider range of deployment scenarios by using FPGAs as parallel processing devices that boost efficiency on particular processes. An example is the Terasic's Cyclone family of boards (Technologies 2018), which couple the FPGA fabric with ARM processors to provide audio, video and networking capabilities.

In the past, FPGA development boards were relatively expensive items, but today there are several companies making them more accessible. These include Digilent (Digilent Inc. 2018) and Terasic (*Terasic* 2018), which have a number of new low-cost products in the form of compact boards, accessible to smaller-scale developers, hobbyists and educators. There are also inexpensive FPGA add-on boards that can be used together with single-board computers (see section 4.2.4), such as the LOGI family of boards by ValentF(x) (ValentF(x) 2014).

There have been several efforts to use FPGAs in synthesizers. Synthup (Raczinski, Marino, et al. 1999; Raczinski, Sladek, et al. 1999) is a peripheral component interconnect (PCI) board developed by Raczinski and team, who report the ability to simultaneously run 800 oscillators with interpolation. Saito also describes the implementation of MAC units for audio synthesis on FPGA, showing it can outperform DSPs (Saito et al. 2001). In this application a host computer runs Pure Data, which works in tandem with the FPGA. While the more intensive DSP calculation (filter, oscillators or amplifiers) is done by the FPGA, the Pure Data patch conveniently takes care of signal routing and timing.

FPGAs have also been extensively used in physical modeling based on finite-difference schemes. Motuk (Motuk, Woods, and Bilbao 2005; Motuk, Woods, Bilbao, and McAllister 2007) proposed plate and membrane models and Gibbons created a model for 1D wave propagation (Gibbons et al. 2005). Pfeifle and Bader also conducted extensive research, modeling the banjo and the violin (Pfeifle et al. 2009; Pfeifle et al. 2011) and proposing new methodologies for real-time control of physical modeling parameters (Pfeifle et al. 2013). Other musical applications with FPGAs include wave-field synthesis (Theodoropoulos et al. 2009) or high-frequency sensor signal acquisition for the construction of more responsive physical controllers (Avizienis et al. 2000; Kartadinata 2006).

4.2.4 System-on-Chip

The conventional desktop personal computer (PC) is composed of several integrated circuits, such as the CPU, random-access memory (RAM) or graphics processing unit (GPU), which interoperate through communication buses on a large PCB, commonly called motherboard. SoCs integrate all of those components into a single chip. They work very similarly to a computer, by running an operating system that controls several concurrent processes that share resources like the CPU or memory. Today, many SoCs use the Linux operating system, which allows them to run much of the software that might have been originally written for the x86 architectures of desktop and laptop computers, without the need for a significant refactoring.

The rapid advance of the mobile industry has been an important factor for the improvement of SoCs, since compactness and low-power operation are crucial in products like smartphones and tablets. There are several types of SoCs on the market but the most popular architecture is ARM. Contrarily to Intel, AMD or NVidia, ARM does not produce or distribute chips but instead licenses intellectual property to about 300 companies (Texas Instruments, Apple, Qualcomm, ...), who are then responsible for manufacturing their own chips. This decentralized business approach circumvents lock down by chip manufacturers, which led many companies to consider the ARM architecture a sound investment. It is expected that ARM architectures will continue to grow in the upcoming years, even threatening the x86 architecture in the desktop and laptop markets.

There are several families of ARM processors, from the more modest Cortex-M series with about 0.84 DMIPS/MHz to the Cortex-A with up to 3.5 DMIPS/MHz. Mid-tier processors, such as the Cortex-A8 have 2.0 DMIPS/MHz, with clock speeds between 300 and 1000 MHz. ARM processors are also common in hybrid architectures that integrate DSPs or FPGAs.

Single-board computers that incorporate SoCs are commonly used in embedded applications. The terminology originates from the fact that these computers are composed of a single PCB, where all the components are pre-soldered, in contrast to regular motherboard-based computers, where components are modular and connected to communication buses via interfaces like PCI.

In the case of SBCs that use SoCs, we normally find additional components to expand connectivity, such as Ethernet controllers, memory expansions or serial buses. In most SBCs, the SoC's chip pins are also exposed via general purpose input/output (GPIO) interfaces. The result is extremely small-sized and low-powered computers, with relatively high processing power and a myriad of I/O options. In essence they are a sort of hybrid between microcontrollers and regular computers.

The Raspberry Pi (Raspberry Pi Foundation 2018) and the BeagleBone (*BeagleBoard* 2018) are two examples of ARM-based SBCs that became very popular in education and hobby communities in the same fashion as the Arduino.

There are reports of several implementations of DMIs with SBCs. The El-Lamellophone (Trail et al. 2014) is a *hyperinstrument* that uses piezo-electric pickups mounted on the underside of the instrument's body for direct audio acquisition and processing using Pure Data. RANGE (MacConnell et al. 2013) is an autonomous processing unit for guitar effects, with pitch tracking and several potentiometers to control audio synthesis/processing parameters. The authors refer to the high level of autonomy of their systems and the versatility that DSLs like Pure Data provide, with a "wide range of possibilities and for both digital audio effects and their control" (MacConnell et al. 2013).

4.2.5 Analysis

Processing Power

When selecting a technological platform to support the development of an embedded DMI, the first preoccupation will inevitably fall on processing power and the ability to perform relatively complex audio synthesis with sufficient sound quality and acceptable latency. In terms of pure processing power in DSP applications, FPGAs will stand out due to their capacity for parallel processing and low-level control over DSP processes.

Microcontrollers can also achieve low latency but have much less processing power than FPGAs, so their applicability in scenarios of more demanding audio synthesis is somewhat limited.

Single-board computers with ARM processors use a more traditional form of serial processing. They are capable of running complex audio synthesis but at the cost of increased latency. There are several efforts to categorize the performance of these systems in real-time audio processing. MacConnell (MacConnell et al. 2013) measured latency values that ranged from 10–15 ms. Topliss performed more rigorous tests that consisted in "measuring the time delay between the sound produced by pressing a button on the keyboard and a sinusoidal audio output triggered on the computer by pressing the button itself" (Topliss et al. 2014), for several different setups of buffer and period sizes at 16-bit and a sampling rate of 44.1 kHz. The results show latencies that span from 2.5–12 ms in the various setups, with no more than 1 ms jitter, which is well within the generally accepted latency magnitudes for real-time performance (Wessel et al. 2002).

One of the most common methods to evaluate synthesis performance of a system is to count the maximum number of simultaneous oscillators it permits. Although this constitutes a valid measurement it is not very informative, considering that computer musicians use many different types of synthesis techniques. We proposed a new methodology consisting of a categorization of synthesis techniques and the evaluation of the maximum number of simultaneous voices possible with each one (Franco and Wanderley 2015). The test was performed on the BeagleBone Black SBC, with a 1 GHz ARM R Cortex-A8 processor and running SuperCollider. The BeagleBone Black was able to reproduce 184 voices of wavetable synthesis, 26 of granular synthesis, 12–20 voices of two types of cross-synthesis and 16 voices of a pitch shifter based on the phase-vocoder. With these results, we became confident that an SBC has enough computing power for the DSP of many DMIs.

Integration and Standards

Many DMIs have to communicate with other peripherals or integrate third-party components and software. This type of interoperability is possibly easier to implement in microcontrollers and SBCs, which have built-in hardware interfaces and controllers. SBCs also have the increased advantage of the middleware and software stacks of the operating system, which considerably facilitates addressing devices from user programs.

There are also strong advantages in the adoption of software and communication standards in component integration. These include communication protocols (MIDI, OSC), plug-in formats (virtual studio technology, Audio Unit, LADSPA), IC communication buses protocols (I^2C, SPI), programming languages (PD, Max, SuperCollider) and many others. All of these standards are generally more difficult to implement in DSPs or FPGAs.

Although FPGAs can fully emulate soft processors, they still rely extensively on third-party proprietary solutions. Many IP cores have restrictive licenses that impede their modification and distribution.

Complexity

Programming an advanced DMI on an FPGA is significantly more complex than with SBCs or microcontrollers. The trade-off between the complexity and performance of FPGAs might prove unbalanced for musicians, who could be tech savvy enough to engage in some level of programming but will unlikely choose to develop signal processing algorithms from the ground up. While IP cores might somewhat mitigate this problem, they cannot fully overcome the need for significant knowledge in FPGA architectures. On the other hand, SBCs and even microcontrollers have complete computer music DSLs that are more in tune with the needs of musicians and that offer a quicker path to creativity and productivity with DMIs.

Scalability and Portability

FPGAs are extremely scalable and portable because their programs can be easily adapted to different fabrics without the need for significant refactoring. Solutions based on microcontrollers or SBCs might be harder to port to new systems because they depend more heavily on specific hardware implementations. Nevertheless, such problems can be alleviated by good programming practices and the adoption of well-supported libraries and middleware.

Interactivity

A DMI can provide a lower or a higher level of interactive behavior, from simple audio processors to instruments using artificial intelligence to play together with the performer. The possibility of complex interactive behavior is one of the main differences between DMIs and other standalone synthesizers or sound processing units. In this sense, SBCs are easier to program, can resort to many third-party libraries and are connected in networks. They are more appropriate for complex interactive behavior, while FPGAs or microcontrollers might be more limited for such approaches. FPGAs are complex to program and have less third-party libraries, while microcontrollers are more limited in processing power.

Availability and Cost

Two other important considerations in the choice of a base technology for embedded DMIs are availability and cost. Powerful FPGAs are still relatively expensive, while ARM SBCs and microcontrollers typically cost less than USD 50 and are available in many electronics stores. The previously mentioned Cyclone and LOGI boards (see section 4.2.3) constitute notable exceptions and prove that there is definitely a growing interest in the use of FPGAs. Nevertheless, they still represent a market at its infancy, and for now SBCs and microcontrollers are still a more sound choice for makers and educators.

Growth and Communities

The mobile and embedded computing industries are bound to continue to stimulate a substantial growth of ARM architectures. New ARM-based SBCs and microcontrollers are constantly introduced in the market and many of them made available to the general public. The Arduino platform was introduced in the mid-2000s and was largely responsible for the dissemination of microcontrollers for hobbyists. Its major influence in electronics education and knowledge sharing is undeniable. SBCs like the Raspberry Pi or the BeagleBone also found their way to the hands of the Maker community and are now going through the same exact growth process.

It is therefore safe to say that ARM processors have the potential to dominate the embedded computing market and are possibly a good bet for a technology that will continue to offer better specifications and continued support in the upcoming years. Embedded DMIs could benefit from adopting these industry trends.

Conclusions

Microcontrollers are cheap and simple to use but have less processing power. They are optimal for the construction of simpler low-fidelity instruments.

FPGAs have optimal architectures for high performance and low-latency digital signal processing, but are complex to program, expensive and often use proprietary technologies. They are also less widely available and less used by the Maker community, although we are seeing the introduction of more affordable and user-friendly products. FPGAs might be optimal for the construction of high-performance DMIs but less appropriate for the development of advanced interactive behaviors or frequent user-level programming.

SBCs are widely available, cheap and represent a fast growing market. They offer extreme flexibility because they can run different programs and use proven and mature Linux libraries. SBCs also have architectures that are much closer to traditional computers and thus are possibly better to create a programmable DMI that offers the same advanced features and interactivity that characterizes previous-generation DMIs.

4.3 State of the Art

In this survey we present past and present examples that constitute the state of the art on embedded DMIs. The objective is to provide an overview of how DMI developers have explored the possibilities of embedded computing. There is a large number of synthesizers that use these technologies, but we are most interested in highlighting those that maintain the most important characteristics of computer-based DMIs. Programming (or at least a high freedom of configuration) is probably the most important of those characteristics, so our selection criteria concentrates in cases where programming is featured by design and a clear part of the specification.

Some examples date back to almost two decades and range from commercial products to academic research. This level of heterogeneity becomes more manageable by aggregating our cases into three main categories:

- Early examples: some of the first concepts and technologies on embedded DMIs.
- Products: commercial products available directly off-the-shelf.
- Frameworks: hardware and software development tools for instrument makers.

It is important to note that we did not have direct access to many of these systems. The information hereby presented is mostly collected from specification sheets, manuals, marketing materials, academic publications and online educational videos. Therefore, it is difficult to guarantee a consistent categorization of the source data, although we believe this information holds enough value to inform our analysis.

At the time of writing, Prynth has been available to the public for two years. It could now be considered part of the state of the art, but we abstained from including it in this survey since it will be described and discussed in subsequent chapters.

4.3.1 Early Examples

Soundlab

The SoundLab (Curtin 1994) was developed in 1994 by Steven Curtin, during a research residency at STEIM. It was an incremental development of a previous device, the SensorLab, a portable signal acquisition system for wearable haptic interfaces.

The SoundLab had sensor signal acquisition but also onboard real-time audio synthesis with the addition of a DSP. The system was composed of the Siemens 80C535 microcontroller, for up to 8 channels of analog-to-digital conversion, communicating with a Motorola 56004 DSP chip, to compute and output audio via an inter-ic sound (I^2S) serial bus. This family of Motorola DSP chips was equipped with a debugging port called on-chip circuit emulation (OnCE), that could communicate with a host computer to upload, download and debug DSP programs written in OnCE-Forth.

The SoundLab also included a library with several audio processing algorithms, implementing functions such as multimode oscillators, delays, waveshaping and pitch shifting. Unlike most systems at the time, the SoundLab was a completely self-contained unit that could be easily reprogrammed.

Gluiph

The Gluiph (Kartadinata 2003) was a system designed by Sukandar Kartadinata in 2003 and directly inspired by the SoundLab. It was probably one of the first DMI development systems to run on a single-board computer.

The base system was originally developed by Mtronix for multimedia set-top boxes and had a Philips Trimedia CPU, synchronous dynamic random-access memory (SDRAM) and Flash memory, a complex programmable logic device (CPLD) and onboard audio with 2 inputs and 10 outputs. An added microcontroller subsystem was responsible for analog sensor acquisition and digital sensors handled directly through the CPLD. For audio synthesis and mapping, the system used a modified version of PD, adapted to interface with the sensor subsystem, audio drivers and file system of the Mtronix SBC. There is no specific reference to the programming workflow or to its level of dependency on a host computer.

Gluiph became the technological foundation for at least two instruments: Skrub, a keyboard sampler with additional trackpads for sample manipulation, and Rajesh Mehta's meta trumpet, a hybrid between a trumpet and a trombone that includes a gyroscope sensor, preset buttons and a display screen.

Fungible Interfaces

In 2010, Hollinger, Thibodeau and Wanderley presented an embedded hardware platform used in the construction of their Fungible Interfaces (Hollinger et al. 2010). The system ran on a programmable system-on-chip (PSoC) by Cypress Semiconductor. The PSoC is a peculiar architecture, composed of a CPU (an ARM7 processor) and a mixed-signal array, similar to the programmable logic device (PLD) used in Gluiph. The PSoC signal array is programmed using digital and analog blocks that can be freely routed between themselves or to a GPIO interface.

In Fungible Interfaces, the system uses an I^2C bus for communication to peripheral components and USB to connect to a host PC. The onboard 10-bit converters are used for both high-speed analog sensor acquisition and audio output with 1 μs settling time. The ARM processor is responsible for mapping and synthesis, with examples of two physical modeling implementations, the FitzHugh-Nagumo excitable cell and the Ishizaka-Flanagan vocal fold, both of which were used in Thibodeau's augmented trumpet.

The authors of this platform refer to the modularity of the PSoC and how the use of high-level DSP nodes could greatly accelerate DMI prototyping.

4.3.2 Products

Nord Modular

The Nord Modular (Nord Keyboards 2018) was a synthesizer produced by Clavia and introduced to the market in 1997, with a second version (the G2) produced from 2004 to 2009. It had different formats (keyboard, rack module and tabletop) and digitally emulated an analog modular synthesizer.

Its strongest feature was that it was specifically designed so that the user could reconfigure the DSP engine. This was done by connecting the Nord Modular to a host computer and creating new patches using a GUI application resembling an analog modular synth, in which the user could virtually add processing modules and interconnect them using cables. All the processing was carried out by the Nord's onboard DSP chips, allowing the instrument to be programmed and later used in standalone mode.

The OWL

The OWL (*The OWL* 2018) is a programmable audio platform developed by Rebel Technology and available since 2013 in two different formats: an audio effects pedal and an eurorack module. The pedal version has a foot switch and 4 knobs, while the eurorack version has 8 knobs, 5 control voltage inputs, and I/O triggers. The main processing unit is an STM32F4 microcontroller, with a 32-bit ARM Cortex M4 processor, 192 kB RAM, 1 MB Flash, 8 Mb SDRAM and a USB connector. Audio is handled by a Wolfson WM8731 stereo codec, with two inputs and two outputs at 24-bit depth and sampling rates of up to 96 kHz.

The system is programmed by connecting the OWL to a computer and using an online compiler with a full C++ application program interface (API), apparently based on the JUCE framework (ROLI Ltd. 2018). The online compiler can also be used as a source-to-source compiler (a process also called "transpilation"), converting PD and Faust (Orlarey et al. 2009) programs to C/C++, and optionally create an interactive JavaScript preview for testing a DSP program. A recently announced feature is that the OWL can now use Max patches with the gen∼ object, since they can be easily exported to C++ code.

The OWL developers encourage the user community to share their programs, so there is a contribution of about 180 freely accessible patches available online. The hardware schematics and software for The OWL are open-source and released under a GNU general public license (GPL).

MOD Duo

The MOD Duo (MOD Devices GmbH 2018) is a commercial audio effects unit, created by MOD Devices in 2016. It has the typical format of a pedal board, with 2 audio inputs and outputs, headphone output, 2 liquid crystal displays (LCD), 2 foot switches, 2 knobs, MIDI input and output, 2 USB and an Ethernet connector. Although the company does not provide any particular details about the processing unit, there are several references to a single-board computer with added custom electronics. All audio effects of the MOD Duo are based on the Linux audio developer's simple plug-in API version 2 (LV2) plug-in format.

The MOD duo is not exactly programmable but it is highly configurable. It mimics the work-flow of a guitar pedal board, with complex audio processing chains created from an arrangement of virtual pedals and sound processors. It is configured using a web application that presents a visual canvas that the user can freely populate with instances of the virtual processors. The physical knobs and switches of the device can then be freely mapped to any of the exposed parameters.

New modules are available on a virtual online store and the creators of the MOD Duo have also opened the development to third-party developers, so they can create and sell their own plug-ins. The company has also recently announced support for Max with gen∼, similarly to the OWL.

Trinity

Bastl's Trinity (Bastl Instruments 2018) is a collection of palm-sized devices, available as kits or preassembled units. They all have similar interfaces, with 3 knobs, 5 to 7 switches and a volume control. At the heart of these small devices is an ATmega328, the same chip found on the most common versions of the Arduino boards. The audio output is a TRS 3.5 mm jack, with a relatively low-fi audio signal (16 kHz, 14-bit). On the other hand, due to their low power requirements, Trinity instruments can be powered by a 9 V battery. A remarkable feature is the inclusion of a small side connector to daisy chain units to share power, audio and MIDI.

These little devices are intentionally made to be reprogrammed using the Arduino programming language via the Mozzi library (examined in the frameworks section).

Patchblocks

Patchblocks (Patchblocks 2018) are miniature synthesizers with a similar functionality to Bastl's Trinity. The microcontroller used is an LPC1343, with an ARM Cortex M3 processor at 72 MHz. They have two knobs, two buttons and stereo I/O, with a sampling rate of 20 kHz and bit depth of 10 bits. They also have a data bus, transporting MIDI and digital audio for lossless processing, and a battery autonomy of 10 hours. A micro-USB connector is used to charge and program the modules. Patchblocks are programmed using their own dataflow language and dedicated host computer application.

The company behind the Patchblocks has recently announced the discontinuation of these products.

4.3.3 Frameworks

Mozzi

Mozzi (Barrass 2018) is an audio toolkit for Arduino microcontrollers, composed of several high-level DSP building blocks. It includes an implementation of separate interrupt routines for audio and control, solving one of the main difficulties of using microcontrollers in audio processing. The audio sampling rate is of 16.384 kHz, while sensor acquisition and control signals operate at 64 Hz. The output buffer has a size of 256 samples, resulting in a maximum latency of 15 ms. Mozzi works with the AVR 8-bit microcontrollers, found on the more common Arduino boards, and also some of their variants, like the Teensy, which uses an ARM microcontroller with an onboard 12-bit

DAC. The Mozzi library is freely distributed under a ShareAlike Creative Commons license. The supporting website contains examples, an API documentation and a dedicated user forum.

Axoloti

Axoloti (Taelman 2018) is a preassembled electronics board with an STM32F427 microcontroller and assorted connections, like stereo I/O at 24-bit/48 kHz, headphone output, MIDI, USB and a secure digital (SD) card slot. The circuit board also contains several soldering pads for custom I/O, including 16 channels of analog to digital conversion, SPI and I^2C interfaces.

The system is programmed by connecting the Axoloti to a host computer via USB and using a custom editor similar to the Nord Modular. The editor is written in Java but generates C++ code, which is then compiled and uploaded to the board.

There is no specific mention to an intellectual property license, but both the circuit schematics and software are freely available online.

Satellite CCRMA

Planet CCRMA is a Linux distribution for personal computers, maintained by the Center for Computer Research in Music and Acoustics (CCRMA) of Stanford University. Planet CCRMA contains system-level enhancements and precompiled applications, making Linux PC audio systems more accessible to novice users. Satellite CCRMA (Berdahl et al. 2011) follows the same principles but it has been adapted for single-board computers, like the BeagleBoard-xM and the Raspberry Pi.

By adding an Arduino microcontroller and a USB audio card, Satellite CCRMA provides a good starting point for embedded DMIs, but still requires relatively high proficiency with Linux command line tools. Without any update in several years, it appears Satellite CCRMA is currently not being maintained.

Edgar Berdahl, author of Satellite CCRMA, was one of the first researchers to identify important properties of embedded DMIs, like quicker setups and higher reliability.

Bela

Bela (McPherson 2017) is a platform for the creation of DMIs, developed by the Augmented Instruments Laboratory at the Queen Mary University of London. This system uses the BeagleBone Black SBC, coupled with custom electronics that provide a stereo output, 8 channels of analog signal I/O and 16 of digital I/O, with latencies as low as 100 μs. This highly efficient system is possible due to the BBB's programmable real-time units (PRU), two co-processors that can access the SoC's internal memory, and Xenomai, a real-time Linux system that implements a co-kernel for high-priority scheduling. There is an audio expansion board to transform the audio output to 6 channels at 44.1 kHz or 10 channels at 22.05 kHz, and also a multiplexer board to expand the 8 analog inputs to 64 at a lower frequency of 2.75 kHz.

Bela is programmed using a browser-based IDE and can compile C/C++ programs directly on the BeagleBone. It can also run PD patches and SuperCollider programs, albeit with some restrictions. PD patches cannot be edited directly on the IDE and SuperCollider cannot be programmed interactively (see section 5.7.3). Bela's IDE also includes features like a console window, a terminal prompt, file management, a virtual oscilloscope, code examples and an API documentation. The hardware and software components are open-source and preassembled circuit boards are sold by Augmented Instruments Ltd.

4.3.4 Analysis

Goals

The clearest difference between these various cases is that some are commercial instruments and others frameworks for instrument development. They also require different levels of technical knowledge. In a guitar effects pedal, the user will mostly control parameters, while operating a synthesizer requires more knowledge in sound processing. The creation of a new DMI is an even more demanding proposition, involving multiple disciplines (electronics, mechanics, computer programming, interaction design) to fully conceptualize, develop and assemble a new instrument.

These target groups also have very different sizes. There are certainly many more guitar players willing to adopt a device like a configurable pedal than experimental electronic musicians willing to develop idiosyncratic devices. Still we can find offerings for all levels of complexity and approaches to music making.

Programming

All DMIs are meant to be programmed in some capacity. From our pool, both the Nord Modular and the MOD Duo are instruments that are more configurable than programmable. They virtually mimic the analog modular synthesizer and the pedal board, where audio processing modules are chained and their parameters mapped to controllers. Maybe an instrument like the MOD Duo cannot be truly called an embedded DMI, but the Nord Modular is somewhat programmable because it can have state machines or perform nonlinear transformations.

Nord Modular, Patchblocks and Axoloti have their own dataflow languages, while the OWL, MOD Duo or Bela can use transpiled Max or PD patches created on a desktop computer. Dataflow languages are an easier entry point to programming, but not without their share of disadvantages and limitations. Some types of programs can be difficult to construct and represent visually, requiring excessive patching for operations that can be achieved succinctly with procedural object-oriented languages. Other operations like control centralization, recursion or refactoring can also prove challenging. As a unidirectional and automated process, transpilation of dataflow languages to lower-level code can also become problematic. Implementation or performance issues are difficult to trace and correct, and reverse transpilation is usually impossible, especially if the user edits the generated code. Finally, a program that requires a transpilation post-process might be limited by specific procedures or subsets of the original libraries.

Text-based languages generally offer better performance and the ability to create advanced algorithms, with the disadvantage of a considerable increase in complexity. This is the case with the OWL, Trinity or Bela, which are focused on C/C++ programming.

Programming Tools

Most of the referred systems are programmed on a local computer and the result uploaded to the instrument via USB connection. The OWL's compilation goes a step further, offering compilation of programs via Internet services. The MOD Duo and Bela have a different approach. Their IDEs also require a web browser, but all code editing and compilation operations happen directly on the device, without the need for any special software or compilation toolchains on host computers.

Tangible Interfaces and Electronics Development

In this review, we refer to both instruments and frameworks to build them. Products like the OWL, MOD Duo, Patchblocks or Trinity already have physical interfaces, with various buttons and knobs, that can be freely assigned to synthesis or functional parameters.

Axoloti, Mozzi or Bela don't have any predefined interface. The user must develop a custom interface by connecting analog or digital sensors to the system. The Axoloti is possibly the easiest to use because it does not require any development other than connecting the sensors. Bela focuses on high-performance analog signal acquisition, through dedicated ADCs that communicate with BeagleBone Black.

Audio Quality

Most DMIs have high-quality audio signals, accomplished via dedicated codec chips that run at a minimum bit depth of 16 bits and a sampling rate of 44.1 kHz (equivalent to CD audio quality), but that can go up to 24 bits and 96 kHz respectively. Trinity, Patchblocks or Mozzi are the exceptions, with the lower audio quality of on-chip DACs or PWM. Most applications use stereo inputs and outputs, but single-board computers can also be used for multichannel output.

Latency

Latency is another important specification for DMIs. The time delay between actuation input and sound reproduction can result from the accumulation of processing delays at many different stages, like AD/DA conversion, communication protocols and DSP. Latency can be determinant to the usability of a DMI, especially in applications where tight musical timing is required (eg. percussive instruments).

Computers have higher audio latency than microcontrollers or dedicated signal processing units, with values that can go up to 20 ms. Nevertheless, it is possible to tune single-board computers for better real-time performance (Topliss et al. 2014), through optimization of system kernels, binary compilation and vector buffering. When it comes to latency, Bela is a notable exception because achieves sub-millisecond values with its PRU and Xenomai solution.

Musical Control Protocols

MIDI and OSC are the two most widely adopted musical protocols. MIDI connectivity is available in most of our cases, either through the older 5-pin DIN connector or the more modern USB. MIDI is a simple serial protocol, while OSC uses the user datagram protocol (UDP) messaging over internet protocol (IP) networks, requiring a more complex networking middleware. SoC-based instruments, like the MOD Duo or Bela, already have this networking subsystem, which might be more difficult to implement in microcontrollers.

MIDI is still the most used musical protocol, a result of its legacy and commercial omnipresence, but OSC is a much more flexible and modern protocol. OSC allows the user to define a custom addressing system and transport any type of arbitrary data, while MIDI is bound by its strict protocol and low resolution data. Furthermore, networking can also be used in other creative or functional scenarios, from the multicast control of a large number of instruments to pushing and pulling data from remote servers.

Sound Synthesis

Early analog synthesizers use subtractive synthesis, a method by which source oscillators with basic waveforms are combined and processed by a series of filters. Digital systems spawned several new synthesis techniques. Sampling and its derivatives, like wavetable or granular synthesis, are only possible in the digital realm, due to the ability to hold and play arbitrary sound samples stored in volatile memory. Physical modeling uses computation to simulate the mechanical and acoustical properties of excitation sources, like strings or membranes, and the resonance of physical bodies according to their materials and form. The phase vocoder makes extensive use of FFT by decomposing sound and directly modifying its spectral content.

Instruments like the Nord Modular, Patchblocks or MOD Duo are emulators of source-filter models and mostly limited to subtractive synthesis, while the Axoloti adds the ability to manipulate sound samples. Devices such as the MOD Duo and The OWL are capable of performing more complex digital synthesis because they can use dataflow languages like PD or Max/MSP. Finally, formal languages like C++ or SuperCollider are more adequate for the development of highly demanding algorithms or DSP chains.

A method that is often used to measure synthesis performance is to count the maximum amount of sine wave oscillators that can run simultaneously, but the author suggests a more comprehensive method (Franco and Wanderley 2015). It consists of selecting commonly used synthesis techniques and counting the maximum number of simultaneous voices. Synthesis performance is an important metric but the inability to access all the described systems prevents us from presenting concrete performance results. Still, we can theoretically speculate that audio engines that rely on virtual machines, such as SuperCollider or PD, are probably going to have lower performance than binaries created from a lower-level C++ code, good programming practices and tuned compilers. Nevertheless, the opposite situation can also occur, where a less experienced developer has difficulties in properly managing the technical details of a lower-level implementation and negatively impact performance.

Mapping

Mapping is the process by which gestural control signals are transformed and applied to specific parameters of the synthesis engine. It is the bridge that establishes the connection between a tangible interface and the types of interactions it permits. There is significant research in the field of mapping and some of its basic concepts might help in the evaluation of our cases. The first notion is that most control signals need some sort of conditioning operation, like scaling or noise reduction. The second is that a control signal can be mapped to a single parameter (one-to-one mapping) or to many parameters (one-to-many mapping). It is also possible to control a parameter from the convolution of several signals (many-to-one mapping) (Hunt and Wanderley 2002).

We can extrapolate that advanced mapping depends directly on the ability to program the system. The Nord Modular or MOD Duo are mostly restricted to direct assignment, while dataflow programming allows further transformation of signals (table lookup, filtering or nonlinear transfer functions). Advanced mapping techniques, like machine learning, might be better to implement in procedural languages like SuperCollider or C++.

Availability and Support

Last but not least, we should evaluate availability and longevity. Most commercial products have a limited shelf life, before they are substituted by better versions or simply discontinued due to market saturation or poor sales. Such is the case of the Nord Modular, which many still consider to be one of the most compelling and flexible synthesizers ever built. Unfortunately, it cannot be purchased anymore.

On the other hand, open-source solutions have the potential for continued support by developer communities, but even those face the risk of sudden unavailability of third-party components, which could significantly impact projects like Bela that are highly dependent on the features of a particular hardware.

4.4 Conclusions

In this chapter we introduced the concept of embedded DMIs. They use small processors that can be incorporated inside the body of the controller and that have enough processing power for audio synthesis.

We have also reviewed different technologies that could be used in the construction of an embedded DMI, including microcontrollers, FPGAs, DSPs and SoCs. Each technology presents its own advantages and disadvantages. FPGAs and DSPs are able to run demanding signal processing tasks with the best possible performance but they are complex and less accessible to musicians. Microcontrollers are much easier to use but they tend to have lower audio quality and cannot run complex audio synthesis algorithms. The SoCs found in single-board computers have several advantages. They have architectures that are much closer to traditional computers and have enough processing power for many of the synthesis techniques that are essential to computer music. They also have the advantage of being able to use mature software libraries and applications for the Linux OS. SoCs are a growing market and very accessible to independent developers and small-scale businesses. Therefore, we believe they represent the best balance of features for the construction of embedded DMIs.

Embedded DMIs are not exactly new. There are several past and present examples of efforts to develop and commercialize them. Based on the proximity to our own definition of an embedded DMI, we have selected and analyzed some of those cases, including both products and development frameworks. They all converge to the same goal of discarding the computer but they also have different workflows, limitations and base technologies. But the most important conclusion we can draw from this state-of-the-art is the validation that embedded DMIs constitute a relevant topic of research and development.

Chapter 5

Prynth: a Framework for Embedded DMIs

In this chapter we introduce Prynth, a hardware and software framework for the development of embedded DMIs. We discuss the motivations for the creation of Prynth, present an overview of its features and then offer a more detailed description of each of its technological components and their implementation.

5.1 Motivation

One of the first goals of our research on embedded DMIs was to build a small set of prototypes that would allow us to explore new designs and test the viability of single-board computers. As our work advanced, we began to implement many of the features we would like to see on our ideal embedded DMIs. In this process we gathered a significant set of tools, methods and designs that were systematically used in our prototyping activities. As our technical implementations matured, we came to realize that they could be polished and packed into a more comprehensive framework. If properly tested and documented, it could be shared with others and serve a wider audience. On September 20, 2016, we released the first public version of Prynth.

Prynth immediately attracted the attention of specialized media that focuses on topics like electronic music, synthesizers and the DIY culture. This quick diffusion resulted in an unexpected and quick growth of a user community, which led us to believe in the value of continuing the development and support of our framework.

Prynth is a set of software and hardware tools that are free and open source. Our target audiences are artists, educators and academic researchers that wish to create custom electronic musical instruments. Therefore, we believe in the importance of offering development frameworks that account for the resources commonly accessible to these user groups. We insist in the use of off-the-shelf components that can be easily sourced and assembled at home using basic tools and procedures.

Like any other framework, Prynth's features and supporting technologies impose particular methodologies and workflows that may not fit every user profile. Still, many of its choices take into consideration the author's experience of almost 20 years of study and creation of DMIs. We believe Prynth represents a good balance between ease of use, performance, flexibility and usability. Since its inception and release to the public, Prynth has also incorporated numerous features requested by its users and will hopefully grow to include many more.

Another important aspect of Prynth is modularity. It is based on several decoupled components that operate independently and communicate using standard interfaces and protocols. This level of modularity facilitates the modification and substitution of these components in the creation of custom applications. It also means that porting Prynth to different single-board computers should be a relatively easy task, considering the possibility of obsolescence or discontinuation of third-party technologies (see section 3.5).

Finally we believe that Prynth offers a relevant number of features that advance the research on embedded DMIs and their interaction models. We will now describe those features and their implementation.

5.2 Requirements

In our model for embedded DMIs we have taken into account more than just fitting a headless DSP system inside the body of a gestural controller. An embedded DMI should be easy to use, conveniently programmable and capable of the complex sound synthesis, mapping schemes and other interactive behaviors only possible with computers. In essence, it should strive to maintain the many features that differentiate DMIs from traditional hardware synthesizers.

With this mindset we elaborated the following set of requirements to guide Prynth's development:

1. Real-time audio processing—enough computing power to process relatively demanding synthesis algorithms in real-time.
2. Sensor signal acquisition—easy connection of sensors for the construction of tangible interfaces.
3. Programmability—ability to program new synthesis, mapping and interactive behaviors of the instrument.
4. Dimensions—small and lightweight, so it may fit the body of a handheld instrument.
5. Integrity—fully operational without any significant dependence on the physical connection to external peripherals like displays, keyboards or mice.
6. Driver and software independent—configurable or programmable without installing specific drivers or software on host computers.
7. Plug-and-play—once powered, the instrument should reach a ready-to-play state autonomously and quickly.
8. Networking—capable of connecting to shared networks to control or exchange data with other digital instruments, devices or remote services.

9. Transparency—the implementation should be as open as possible and avoid the obfuscation of any processes, computer code or hardware designs.

10. Modularity—a modular design, to facilitate the upgrade, substitution or modification of components.

11. Maturity and Richness—built on top of technologies and standards that have shown some degree of maturity through their longevity, features, stability and large user communities.

12. Accessibility—using only common components that can be sourced directly from electronic shops, assuring there is no lockdown by small providers or obscure business models.

13. Ease of use—the framework should be relatively easy for users with a minimal expertise in DMI development (programming, soldering).

5.3 General Overview

5.3.1 Hardware Overview

The main hardware elements that form Prynth's base system are the Raspberry Pi 3 single-board computer, a custom circuit board with an embedded Teensy microcontroller, a set of 10 multiplexing circuit boards and an audio codec.

The single-board computer is responsible for running and supervising all of the main computation tasks, including audio synthesis, mapping and various other interactive functions. The RPi lacks any type of analog-to-digital conversion, so it requires additional circuitry for sensor input. In Prynth, this is done using an additional microcontroller, the Teensy 3.2. The connection between the RPi, the Teensy and the sensors is done through our custom circuit board that routes their respective inputs and outputs. The Teensy acquires sensor data and sends it to the RPi through a serial connection using Prynth's custom communication protocol.

The signal acquisition circuit board accepts the direct connection of 10 analog sensors but this number can be expanded using up to 10 smaller add-on boards with 8-channel multiplexers that are driven by a bit clock generated by the Teensy. With the multiplexing add-on boards, up to 80 analog sensors can be directly connected to a Prynth system. The circuit board also accepts the connection of up to 48 digital sensors that communicate via standard I^{2}C and SPI buses. All the physical connections are done using female jumper cables with the standard 2.54 mm pin spacing. Because the multiplexing daughterboards are separate from the main board, they can be located anywhere inside the body of the instrument.

Analog sound input and output is done through an external codec that connects to the single-board computer through USB or I^{2}S. The single-board computer can also communicate with other devices on the same network, using standard protocols like transmission control protocol (TCP) and UDP, on top of interfaces such as Ethernet or Wi-Fi.

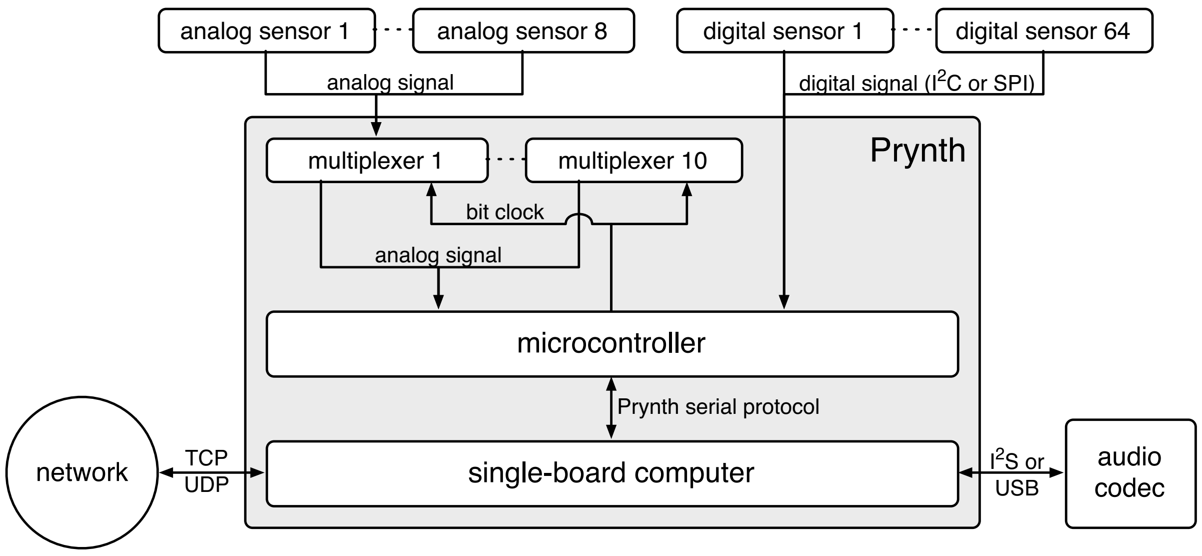


Fig. 5.1 Hardware architecture overview.

The base system, comprised of the Raspberry Pi, coupled with the main circuit board and a USB DAC, measures approximately 12 x 6 x 3 cm. Each multiplexer board measures 5 x 2.5 x 1.2 cm and connects to the main board with a jumper cable.

Figure 5.2 shows the inside of a Prynth instrument called "The Mitt" (described in section 6.2.1).

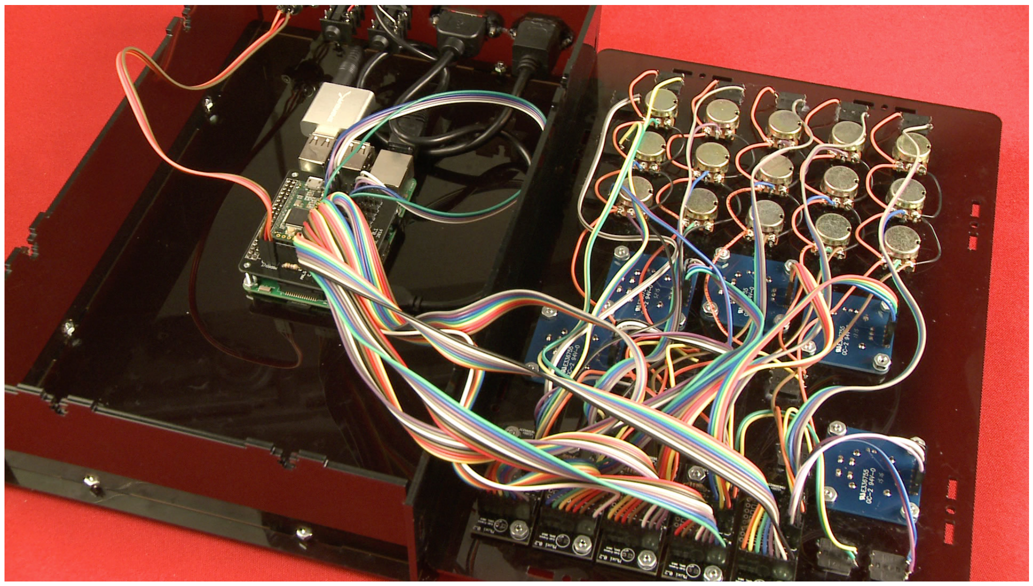


Fig. 5.2 The Mitt with the Prynth hardware inside.

5.3.2 Software Overview

Prynth is tightly integrated with SuperCollider, one of the most modern and powerful computer music languages. SuperCollider itself is divided into two main applications: the SuperCollider server (scsynth), the audio engine responsible for running DSP, and the SuperCollider Interpreter (sclang), which runs user programs and sends control messages to the server. The communication between the two applications is done via internal OSC messaging.

The official SuperCollider release works on a personal computer and uses its own IDE program, in which the user writes new programs and controls the states of the SuperCollider server and language applications. Prynth substitutes this native IDE by a web front end application served by the instrument itself and that runs on any standard web browser, allowing access from any thin client (computer, tablet or smartphone) connected to the same network. The functionality is similar to that of the original IDE, with an embedded text editor and various other controls to manage system settings. The web application is only necessary for programming and configuring the instrument. Once powered, the instrument can autonomously reach a ready-to-play state by running a predetermined user program.

Other relevant software components include the firmware that runs on the microcontroller and a bridge application that runs on the Raspberry Pi. The microcontroller firmware is responsible for acquisition and conditioning of sensor data. It then encapsulates the data into a custom protocol and sends it to the bridge application via serial interface. The bridge application is a middleware between the microcontroller and the RPi. It receives the sensor data from the microcontroller, runs an algorithm that decodes and checks the integrity of the serial messages, and then repacks and sends the data to SuperCollider via local OSC messages.

The Prynth server not only serves the front end web application but also supervises and controls all other software components. It can dynamically inject new SuperCollider programs, receive real-time log messages via a Unix standard I/O stream, change parameters on the microcontroller via bridge application and interact directly with Linux via portable operating system interface (POSIX) commands.

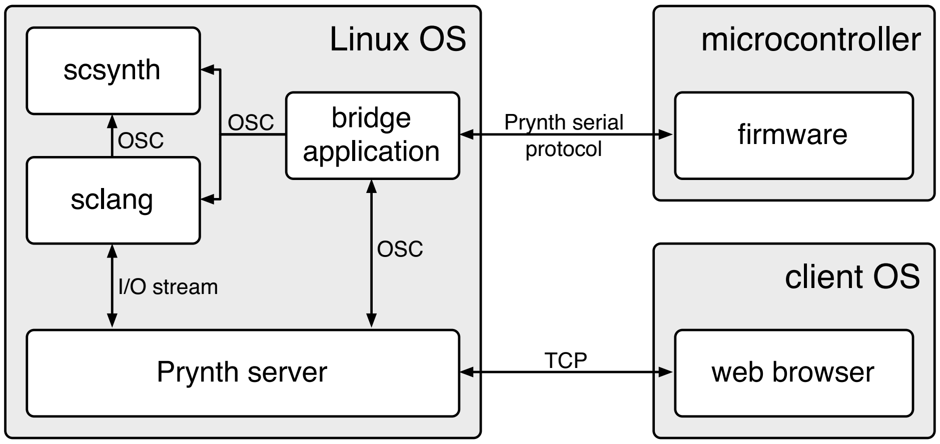


Fig. 5.3 Software architecture overview.

Prynth's software is distributed as a Linux image file that can be downloaded from our website (Franco 2018a) and loaded into the SBC's memory card using a desktop computer. The microcontroller firmware and PCB design files are included in this image, but also independently available for download on the project's website.

5.4 Single-board Computer

In Prynth we have chosen to work with the Raspberry Pi, since it is unarguably the most popular and better supported SBC in the market. It was specifically created for education and it is widely available for a price of approximately USD 35. There are several versions of the RPi in the market. Prynth was developed to work on top of the Raspberry Pi 3 model B, although several users have reported correct operation with the previous Raspberry Pi 2 model B, which has a very similar architecture. At the time of writing, a revised Raspberry Pi 3 model B+ has just been released to the market, but it features only minor incremental improvements and preliminary tests also show no issues with this Raspberry Pi model. From here onward we will be referring only to the Raspberry Pi 3 model B, used in the development and testing of Prynth.

The RPi is a single-board computer with 85.6 x 56.5 x 17 mm. Its top surface area is approximately equivalent to that of a credit card. Like a regular computer, the RPi offers a wide variety of input and output connections, including 4 USB 2.0 ports, Ethernet, high-definition multimedia interface (HDMI), analog audio and video through a 3.5 mm tip-ring-ring-sleeve (TRRS) jack and two mobile industry processor interfaces (MIPI) for cameras and screens.

A 2 x 20 pin header serves as the RPi's GPIO interface with the SoC. These pins provide addressable digital input and output pins and standard communication buses, such as user datagram protocol (UART), SPI, I^{2}C and I^{2}S. It is through the GPIO interface that the RPi connects to any external electronic circuitry, sometimes in the form of stackable PCBs that are mounted directly on the header. A full map of the GPIO pins can be found on the technical documentation available at the official Raspberry Pi website (Raspberry Pi Foundation 2018).

The RPi is designed around the Broadcom BCM2837 SoC, which integrates an ARM Cortex-A53 processor, a GPU, Wi-Fi and Bluetooth. The ARM Cortex-A53 processor has a 64-bit quad-core CPU, based on the ARMv8-A architecture and is capable of clock speeds of up to 1.2 GHz. Roy Longbottom is a UK engineer that has offered reputable and comprehensive computer synthetic benchmarks since 1997 (Longbottom 2018). In his tests, the RPi's Cortex-A53 running at a clock speed of 1200 MHz scored 711.6 MWIPS on the Whetstone benchmark and 2469 VAX MIPS on the Dhrystone 2 benchmark. For comparison, a modern Intel Core i7 x86 processor running at a clock speed of 3900 MHz achieves 3959 MWIPS on Whetstone and 16356 VAX MIPS on Dhrystone. In rough terms, a Raspberry Pi has about one sixth of the processing power of a recent desktop/laptop computer.

The integrated GPU of the RPi is a Dual Core VideoCore IV multimedia co-processor, with Open GL ES 2.0, hardware-accelerated OpenVG and 1080p30 H.264 high-profile decoding capabilities. This GPU subsystem is clocked at 400 MHz and its 3D core at 300 MHz, which results in a throughput of 1 Gpixel/s, 1.5 Gtexel/s or 24 GFLOPS with texture filtering and DMA infrastructure.

The BCM2837 SoC also includes integrated radio frequency transceivers for wireless communication using 802.11n Wi-Fi and Bluetooth 4.1.

The two other integrated chips found on the RPi are the system's RAM and a USB controller. The RAM is a DRAM module with 1 Gb of LPDDR2 memory, manufactured by Elpida (recently acquired by Micron). The USB controller is a LAN9514, manufactured by SMSC (recently acquired by Microchip). It provides an integrated solution to a 4-port USB 2.0 hub and a 10/100 Ethernet controller using a single 25 MHz crystal.

Like most single-board computers, the RPi lacks integrated nonvolatile memory storage, such as hard disks or solid-state drives. Instead it makes use of a microSD card, manually inserted into the onboard memory card slot, and containing all the necessary operating system and user files. MicroSD cards have lower read/write performance than standard hard drives and are also less durable but in the case of embedded SBCs they offer the advantage of easy substitution and backup at low-cost. The highest performance is achievable by opting for microSD cards of speed class 10, which offer a read/write speed of 10 MB/s but that can go up to 80 MB/s in higher-end offerings. For comparison, a modern solid-state disk drive should have average read/write speeds of about 500 MB/s.

The RPi is powered through a 5 V microUSB DC power adapter but all of its remaining circuitry operates at 3.3 V. Although the processor only consumes up to 1.34 A in turbo mode, the recommended supply current is of 2.4 A, in order to assure proper operation of the radio frequency transceivers and any other external peripherals or add-on circuitry.

Table 5.1 summarizes the technical specifications of the Raspberry Pi.

Table 5.1 Raspberry Pi 3 Model B technical specifications.

CPU	1.2 GHz 64-bit quad-core ARM Cortex-A53.
Architecture	ARMv8-A (64/32-bit).
SoC	Broadcom BCM2837 SoC.
Clock speed	600 MHz base and 1.2 GHz in turbo mode.
RAM	1 GB of RAM.
Power connector	microUSB.
Audio output	3.5 mm TRRS jack.
Video input	CSI (MIPI Serial Camera)
GPU	Broadcom VideoCore IV @ 250 MHz
Video output	HDMI (rev 1.3 with up to 1080p resolution), composite video (3.5 mm TRRS jack), DSI (MIPI Serial Display)
GPIO	2x20 female pin header, with UART, I^{2}C bus, SPI bus with two chip selects, I^{2}S audio, +3.3 V, +5 V and ground.
Network	10/100 Mbps Ethernet and 802.11n Wi-Fi.
USB	4 x USB 2.0 port.
RF	Bluetooth 4.1.
Size	85.60 x 56.5 x 17 mm
Weight	45 g
Power Ratings	from 300 mA (1.5 W) while idle to 1.34 A (6.7 W) in turbo mode.

5.5 Audio

The RPi has an onboard 3.5 mm TRRS jack that transports analog video and audio. The audio signal is generated directly via the BCM2837's PWM and passed through a low-pass filter. The BCM2837's phase-locked loop (PLL) is capable of frequencies up to 100 MHz, so theoretically it should be able to work at audio sampling rates as high as 48 kHz. Unfortunately, the corresponding bit resolution is limited to 11 bits, which severely compromises audio quality due to its low signal-to-noise ratio.

Given these limitations, most RPi-based audio applications resort to the use of external audio cards with discrete DAC codecs, usually relying on sigma-delta modulation and subsequent conversion to a more conventional pulse-code modulation (PCM) signal. These types of audio cards are capable of higher sample rates (44.1, 48, 96, 192 kHz) and bit-depths (16, 24 bits), resulting in much better audio quality than the RPi's PWM output. There are two types of external audio cards directly usable with the RPi: USB and I^{2}S sound cards.

There is a considerable number of different USB sound cards in the market. The cheapest USB 1.1 cards have prices as low as USD 15 but are usually limited to stereo output and mono input at 48 kHz. More expensive USB 2.0 cards can cost hundreds of dollars, but they have enough bandwidth for multiple input/output channels, higher sampling rates and high-quality signal converters.

I^{2}S audio cards connect directly to the RPi's GPIO header and are typically available in the form of stackable PCBs. They have onboard codecs that directly accept the digital audio signal of the BCM2837, in the form of an I^{2}S bitstream. I^{2}S cards can work at very high sampling rates (up to 192 kHz) and generally have better signal/noise ratios than their USB counterparts. They also tend to have lower latency, less audio artifacts and better overall performance in real-time audio processing. The price of these audio cards is highly dependent on the onboard codec of choice, and can range from the USD 12 of a Fe-Pi (Fe Pi 2018) up to USD 100 for a DACBerry PRO (OSA Electronics 2017).

The RPi's I^{2}S audio stream is mapped to 4 specific pins on the RPi's GPIO:

- LR Clock - GPIO pin 19
- Data In - GPIO pin 20
- Data Out - GPIO pin 21
- Bit Clock - GPIO pin 18

The Prynth software works with any audio device available to the advanced Linux sound architecture (ALSA) (ALSA Project 2018), the default OS audio layer in Linux (see section 5.7.2). This includes any USB or I^{2}S supported by the Linux kernel and/or produced specifically for the

RPi. In Prynth, instead of imposing a particular audio solution, the user can pick the one that best suits a specific application and budget.

Nevertheless, I^2S devices are always preferable because the Raspberry Pi does not have the best USB implementation. In our tests with three different USB 1.1 audio cards, we found that there was a random degradation of audio quality over time. The RPi community traced the root of this problem and attributed it to the loss of split transactions[1], required to manage USB 1.1 full speed devices (12 Mbps) on the faster USB 2.0 data stream (480 Mbps). In audio applications, packet loss and clock drift can lead to audio artifacts such as clicks, pops and background noise. To circumvent this problem, the Raspberry Pi development team included a temporary fix to avoid packet corruption, in the form of a boot flag (dwc_otg.speed) that limits the kernel to USB 1.1 speeds[2]. We have verified that this option indeed solves audio problems with USB 1.1 devices, with the downside that USB 2.0 devices may show inconsistent behavior or simply not work at all.

5.6 Sensor Signal Acquisition

Some ARM single-board computers like the Raspberry Pi do not have onboard ADCs, so it is impossible to connect analog sensors directly to these boards. But even if SBCs did have ADCs, real-time signal acquisition could still present significant challenges. Like regular computers, SBCs are designed for concurrent processing, running several interleaved processes that are automatically managed by the operating system. The CPU might be occupied by many other computing tasks that, together with signal acquisition, compete for processing power. Since processing load is variable, it is impossible to assure constant and high-rate signal acquisition, which can be problematic for real-time musical applications in which timing is paramount.

For these reasons, although the RPi could be programmed to control and receive data from

[1] A high speed USB 2.0 host uses split transactions to exchange data with a low speed USB 1.1 device by breaking the data of the slower device into smaller packets that are sent in intervals. With this mechanism, the bus can continue to be used for high speed data in the interim without being blocked by the completion of the low speed messages.

[2] A boot flag is a start-up parameter that is passed to the kernel when the computer starts.

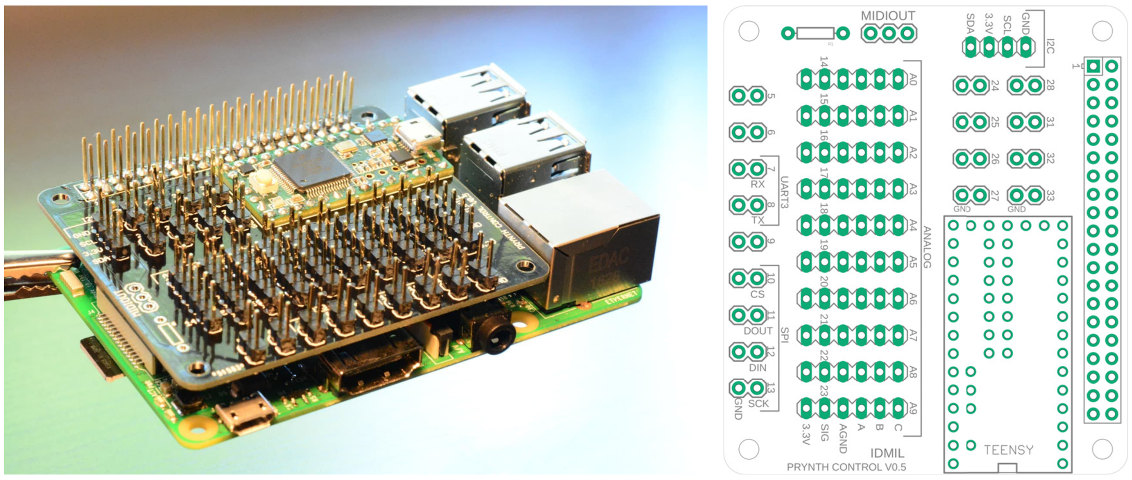
external ADC chips, we opted for the development of an independent signal acquisition subsystem that works in tandem with the RPi, offloading it from any signal acquisition tasks. This subsystem is composed of a main circuit board and up to 10 optional daughterboards that can be used to increase the number of analog input channels on the system.

The design files for these PCBs are distributed with Prynth in the form of Gerber files (Ucamco 2018), the standard computer-aided design (CAD) format used by the PCB manufacturing industry. In the last 5 years, low-quantity PCB manufacturing became extremely affordable and accessible to the Maker community. To manufacture custom PCBs, the customer logs to the manufacturer's site, uploads the PCB design files and pays via credit card, to have the PCBs delivered to the door with typical turnaround times of 3 to 5 days plus shipping. The estimated price of manufacturing Prynth's boards is of about USD 20.

5.6.1 Control Board

The main circuit board of the acquisition system is called *Control* and incorporates a Teensy 3.2 microcontroller. The Teensy family of microcontrollers is developed by Paul Stoffregen (PJRC: Electronic Projects 2018a). They quickly became a popular alternative to Arduino because they share the same programming language but are equipped with ARM-Mx processors. They offer much better performance at significantly lower prices, when compared to the Arduino's older 8-bit Atmel processors, still widely used by many DMI developers. The Teensy 3.2 has a Cortex-M4 processor with a clock speed of 72 MHz, but that can be overclocked to 96 MHz. It accepts a power supply range from 3.6–6.0 V, which is then internally regulated to the operating voltage of 3.3 V. It has 34 pins for digital I/O, 2 x 13-bit ADCs multiplexed over 21 pins and one analog output DAC with 12-bit resolution. The Teensy 3.2 also includes several communication buses: 3 serials UART with speeds of up to 115 kbaud, 2 x I^2C, SPI, I^2S and controller area network (CAN). A full map of the Teensy's pins can be found on the microcontroller's official documentation (PJRC: Electronic Projects 2018b).

With the Control board, the Teensy's pins are remapped to a series of 2.54 mm pin headers, facilitating the connection of any external sensors and peripherals to the microcontroller's digital buses or ADC pins. The Control board has the same footprint as the RPi, so it can be stacked directly on top of the GPIO to form a compact unit. The two boards exchange data bidirectionally through serial communication over their respective UARTs, at a speed of 3 Mbps. Figure 5.4b shows an overview of the Control board design and its connectors. A detailed description is available in the appendix A.1.



(a) Control board mounted on the RPi. (b) CAD screenshot.

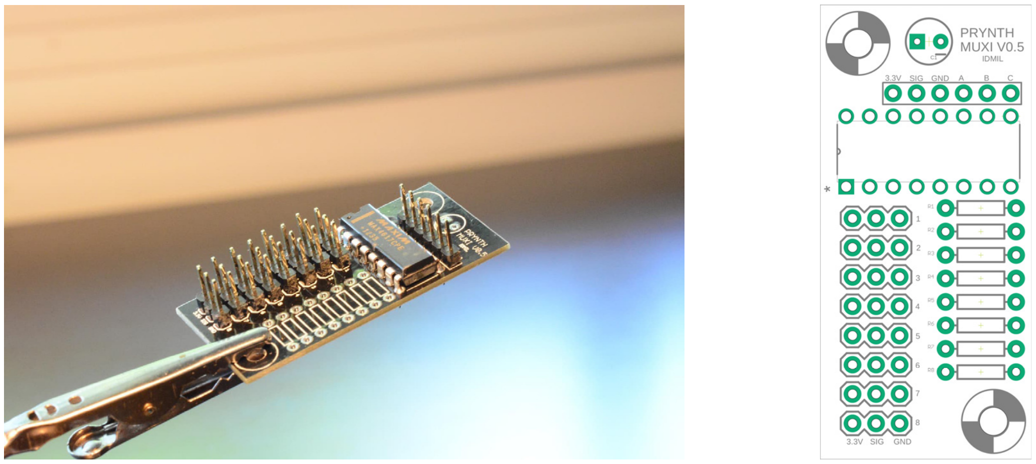
Fig. 5.4 Control board.

Soldering the Teensy microcontroller directly to the Control board is possibly one of the hardest steps for electronics beginners. The problem lies in the fact that some of the Teensy's pins are located on the bottom side of the microcontroller, including the I^2C bus pins used by Prynth. Therefore, the microcontroller must be assembled by flowing solder through the pin holes of both boards to create bridges. This is a difficult procedure, to which we still haven't found a better solution. On the other hand, if the user has no need for I^2C communication with digital sensors, we offer an easier assembly solution that uses male pin headers to stack the Teensy on top of the Control board.

5.6.2 Muxi Daughterboards

The number of ADC pins on a microcontroller is sometimes insufficient in applications that require a large number of analog sensors. In those cases, a common practice is to use multiplexing. An analog multiplexer chip contains several inputs that can be arbitrarily routed to a single output. This operation is done by setting a specific combination of the multiplexer's control pins to high or low. By cycling through the input channels, it becomes possible to use a single ADC channel to read multiple sensors.

In Prynth, we have created the *Muxi* daughterboards, that use a standard 4051 8-channel analog multiplexer to expand the 10 original analog inputs of the Control board. With the Muxi daughterboards, a Prynth system can use a total of 80 analog sensors. The Muxi daughterboards are connected directly to the Control board via a 6-pin cable, transporting power, analog input and ground, and also the digital signal to control the gates of the multiplexer. The multiplexer control signal is sent over three wires labeled A, B and C, constituting a 3-bit counter that constantly cycles at the specified rate.



(a) Muxi daughterboard after assembly. (b) CAD screenshot.

Fig. 5.5 Muxi daughterboard.

The sensor input connectors of the Muxi daughterboards have three pins: 3.3 V power, ground and analog signal input. Each of the sensor inputs has a corresponding pull-down resistor, required in the use of variable resistors or switches. Figure 5.5b shows an overview of the Muxi board design and its connectors.

The Muxi daughterboards also include the footprint for a capacitor, which could be used to prevent possible current drops. This capacitor is included mostly as a preventive measure and its use entirely optional.

5.6.3 Microcontroller Firmware

Just like Arduino, the Teensy microcontroller is very easy to program. In fact, it uses the same programming language and IDE, which is probably part of the reason for its success.

To program the Teensy, the user must first install Teensyduino, a software add-on for the Arduino IDE. Another support application included in this package is the Teensy Loader, responsible for managing the Cortex-M4 compilation and upload stages. After the installation of this software, the user can connect the microcontroller to a computer via USB cable and use the Arduino IDE to write, compile and upload new programs.

Although it is common to call Arduino a programming language, it is actually a simplified form of C/C++, imbued with a set of functions to facilitate microcontroller programming. These functions include the ability to easily address analog or digital pins or the automatic prototyping of functions in the preprocessing stage. The Teensyduino library follows the same exact conventions of Arduino but with additional functions specific to the Cortex-M4.

Prynth's distribution ships with a ready-to-use Teensy program that the user must compile and upload. This program readily handles sensor signal acquisition, data conditioning and transmission to the Raspberry Pi. It can also receive commands from the RPi, used to set states on the microcontroller program. The tasks carried out by the Teensy could be summarized as:

- Analog signal acquisition.
- Digital signal acquisition.
- Driving multiplexer boards.

- Management of data memory banks.
- Data conditioning.
- Send sensor data to the Raspberry Pi.
- Receive commands from Raspberry Pi.
- Encoding and decoding of serial messages.
- Message integrity checks.

Classes and Main Loop

There are two principal classes in the Teensy's firmware program: the *SensorBuffer* class and the *Filter* class. The SensorBuffer class creates a single first-in-first-out (FIFO) buffer, in which sensor data is queued before processing and transmission. The Filter class contains several different processing units that apply mathematical transformations to the raw sensor data.

Sensor data and processing objects are stored in bidimensional arrays, allowing for the indexing of 16 channels with 8 sensors each. The first 10 channels correspond to the 10 multiplexers and the remainder 6 are reserved as placeholders for digital sensor data. All sensor data is represented by 32-bit normalized float numbers.

The main loop of the program scans the SensorBuffer object's queue for data and retrieves new entries, each containing a sensor identification number and the raw data value. The data value is then processed using the Filter class and sent to the Raspberry Pi. Lastly, it checks for any input messages on the serial input buffer and processes them before continuing the loop.

Signal Acquisition

Parallel to the main loop, there is a timed callback function that matches the sampling frequency of each sensor. At each tick, this callback function increases a sensor index, writes the corresponding values to the digital pins of the multiplexer bit clock and triggers the reading of the corresponding ADC pin. The data value is then added to the Sensor Buffer queue together with a sensor index.

This function is attached to an interrupt handler, which stops the main loop during execution, thus guaranteeing constant acquisition frequency and avoiding any data corruption when pooling incomplete data from SensorBuffer. All analog data is sampled at 12 bits, corresponding to 4085 possible values.

Digital sensors have many different specifications. Sometimes they use dedicated software libraries to access the IC's memory registers. To use digital sensors with Prynth, the user must add sensor specific code to the base Teensy program. The latter readily includes blank digital sensor initialization and reading functions. Once the raw data is held, it is equally added to the sensor buffer for processing, using an arbitrarily chosen channel and sensor number from those available for digital sensor signals (channels 11–16).

Data Conditioning

The Filter class implements three different types of processing units: a simple first-order low-pass filter, the One Euro filter (Casiez et al. 2012) and a Boolean processor.

A finite impulse response low-pass filter is commonly used to smooth sensor data, eliminating jitter at the cost of added lag. It is a simple and cheap process but still useful for basic noise filtering.

The One Euro filter aims at being an efficient algorithm for signals that require high precision and responsiveness. The One Euro filter is essentially a low-pass filter but with an adaptive cutoff frequency that changes according to the signal differential. At low speeds a low cutoff is used to reduce jitter at the cost of the introduced lag, but at higher speeds the cutoff value is set to a higher value, reducing lag and increasing responsiveness.

The last processing unit is what we loosely call a Boolean processor. When the raw value of a sensor crosses 0.5 upwards, the value is set to 1 and when it crosses it downwards it is set to 0. The Boolean processor is particularly useful for digitizing values from switches connected directly to Muxi's analog inputs.

Serial Communication

Once data is acquired and queued in SensorBuffer, it can be retrieved, normalized and sent to the Raspberry Pi via serial connection. Prynth's custom serial protocol uses a packet composed of a sensor identification number, the sensor data value and a checksum used for the verification of the message's integrity. The sensor data value can be arbitrarily expressed in 1 byte (low resolution) or 4 bytes (high resolution). The full packet then passes through an encoding function, which adds two delimiting characters to mark the beginning and end of the message, namely the $ and * characters. The message is then ready for dispatch and written to the serial output object. The UARTs at both ends (RPi and Teensy) are set to a baud rate of 3 MHz.

As previously mentioned, the Teensy can also receive control messages from the Raspberry Pi. These messages are used to set sampling frequency, data resolution, data processor parameters and to enable or disable sensors. The main loop of the program starts by checking the serial input buffer of the Teensy. If there is data available, the timer and interrupt functions are stopped and the serial data parsed and processed. The delimiter characters are striped from the message and a checksum algorithm verifies its integrity. If the message is valid, an acknowledgment packet is sent back to the Raspberry Pi. If this acknowledgment is not received, the RPi will try to send the message again, with up to ten attempts, after which an error is thrown. Once all incoming serial messages have been processed, timer and interrupts are once again enabled and the main loop restarts.

5.7 Support Software

5.7.1 Linux Operating System

The official Linux operating system for the Raspberry Pi is called Raspbian. It is a derivative of Debian (Debian 2018), modified to work with the RPi and available on the Raspberry Pi Foundation's website (Raspberry Pi Foundation 2018).

Raspbian is distributed in the form of a disk image, a file format that can be used to preserve all the contents and structure of an operating system volume. The disk image is loaded into a blank microSD card using any computer with a memory card reader and software to adequately copy the files. This operation can be done either through a command line interface, using the Unix *dd* command line utility, or using GUI software, like the multi-platform Etcher (Balena 2018).

There are two official Raspbian versions: the regular desktop version that includes a window manager and several bundled applications, and a lite version with the bare-bones of a Linux system and no window manager. Prynth is built on top of the latter and thus stripped from any dispensable processes or software that might compete for resources. Although there are some modifications to the original Raspbian Lite distribution, we have made an effort to keep it as close as possible to the original. This lean approach assures that Prynth can be easily ported to future versions of the Raspberry Pi and that the large knowledge base created by RPi users remains mostly accessible.

Next we describe the modifications made to the original Raspbian Lite.

UART Reassignment

The Raspberry Pi 3 has two UARTs, the PL011 and the so called mini UART, respectively mapped in the filesystem as */dev/ttyAMA0* and */dev/ttyS0*.

The mini UART is a less capable device than the PL011. It has a smaller buffer, with only 8 symbols deep FIFO for both receive and transmit, while the PL011 has separate 16 x 8 symbols for transmit and 12 x 12 for receive. The mini UART also has a variable baud rate, due to the fact that it shares its clock with the GPU's video processing unit, managed by its own frequency governor.

Prynth uses the PL011 to achieve the maximum performance in serial data transmission. It benefits from its stable 400 MHz clock and its bigger buffer, helping to minimize any potential data packet loss or corruption. By using the PL011, the serial communication between the Teensy and the Raspberry Pi can be pushed to a baud rate of 3 Mbaud, which is a considerably higher than usually found in standard control data applications (usually under 115200 baud).

By default, the PL011 is linked to the Bluetooth transceiver, while the mini UART is set to provide a debugging serial console. These default mappings are loaded at boot time but they can be modified to free the UARTs for other purposes. The steps to remap the Raspberry Pi's UARTs are described in appendix A.2.

USB Data Speed

As previously mentioned in section 5.5, the Raspberry Pi sometimes malfunctions with USB 1.1 full speed devices that require a constant data transmission. To avoid this problem the LAN9514 controller can be forced to work at 1.x speeds at boot time by setting the *dwc_ otg.speed* kernel module option to 1 in */boot/cmdline.txt*.

```
dwc_otg.speed=1 dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4
elevator=deadline fsck.repair=yes rootwait
```

CPU Governor

Like most modern processors the BCM2837 can work at different frequencies, scaling the clock speed to meet processing load. Frequency scaling is useful in applications with low processing requirements or long idle times, where power consumption and heat can be reduced by working at lower speeds. In Linux, this process is controlled by a software called CPU governor. By default, the CPU governor is configured with an *on-demand* state, where the application monitors system load and sets the CPU frequency accordingly.

In Raspbian we have verified that the stability of the audio processing carried out by SuperCollider is greatly compromised by the frequency switching of the CPU governor. The CPU behaves so that it is constantly shifting between 600 and 1200 MHz, creating peeks and drops that negatively affect audio quality. This problem can be avoided by forcing the governor to always work at full speed, which can be done by setting the CPU governor to *performance* on start-up.

```
sudo echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

Another method to set the Raspberry Pi to full speed is to completely bypass the CPU governor, which can be done by adding the following option in */boot/config.txt*.

```
enable_turbo = 1
```

Job Scheduler

Cron is a software used for job scheduling in Unix systems. Processes can be set to trigger at desired time intervals or at boot time. The latter is useful to launch the Prynth services at start-up. Setting new cron jobs can be done by executing the following shell command:

```
crontab -e
```

This command opens a text editor that handles a cron job file with all the scheduled jobs. The Prynth server starts at boot time with the following command added to the default user crontab file:

```
@reboot sudo node /home/pi/prynth/server/bin/www
```

5.7.2 ALSA and JACK

ALSA is a Linux software framework that provides an application programming interface for audio devices. ALSA is included in the Linux kernel and it is through ALSA that applications access sound devices such as audio cards.

Like many other Linux audio applications, SuperCollider does not access ALSA directly. Instead it uses a sound server that acts as a bridge between an audio application and the audio device driver exposed by ALSA. JACK (JACK Audio Connection Kit 2018) is the *de facto* sound server for Linux audio applications and provides several benefits over the lower-level ALSA. It works as an audio patchbay, handling audio streams as packets that can be shared between applications and audio devices, allowing for the audio to be freely routed between different applications. JACK also provides a low latency infrastructure and fine control over system performance parameters like sample rate, buffer size and number of periods.

JACK facilitates the development of audio applications because it can be used as an abstraction layer that bypasses the need to handle the specificities of particular audio or computer hardware. Another interesting feature of JACK is that it is not the client program that manages audio threading. The audio program must provide a callback function that is then synchronously triggered by JACK itself.

There are two different implementations of JACK: JACK 1 and JACK 2. Both use the same API and protocols, but the difference between both is that version 2 introduces multi-processor support and a C++ re-implementation. JACK 2 was supposed to substitute JACK 1, but developers of the original version kept maintaining it, so the two options are still available.

Prynth uses JACK 2, which is launched by invoking the *jackd* binary and passing the appropriate parameters for audio device, sample rate, buffer size and number of periods.

```
jackd -R -P95 -dalsa -dhw:1 -p256 -n3 -s -r44100
```

where:

- -R: for real-time operation
- -P: to set the scheduler priority
- -dalsa: to use the ALSA backend

- -dhw: to use the hardware device number 1
- -p: to set the buffer size (powers of 2, typically between 128 and 1024)
- -n: number of periods
- -s: set to softmode, which ignores xruns reports.

The hardware device number is relative to the device list available to ALSA. The first device (number 0) is automatically attributed to the RPi's onboard PWM output, while hardware device number 1 would be equivalent to the second audio device available to ALSA. In the case of Prynth, this would be the added audio card (either a USB or an I^2S solution).

When JACK fails to synchronize audio buffers (for example, if a determined audio process takes longer than the assigned buffer), it reports an error to the console, called a *xrun* in JACK lingo. When many successive xruns happen, JACK can automatically drop the connection to the audio client so that other audio processes may remain unaffected. The softmode option can be enabled to have JACK ignore xrun reports and maintain the connections with client applications even in stress situations.

Higher sampling rates will have higher sound fidelity but will also be more taxing to the system. The vector size (or number of frames) and number of periods determine the size of the audio buffer in use. Larger audio buffers represent higher latency but reduce the chances of xruns, resulting in a more stable audio output when performing demanding DSP.

The theoretical audio latency in seconds can be calculated using the formula:

```
Latency = (Vector Size / Sample Rate) x Number of Periods
```

JACK reports a theoretical latency of 17 ms with a buffer size of 256 samples, a sample rate of 44.1 kHz and three periods. Lowering the buffer size to 128 samples and using only two periods results in a latency of 8 ms. We have pushed Prynth to buffer sizes as low as 64 samples, where it was still possible to have a clean sinusoidal tone, but more demanding DSP processes might require higher buffer sizes to avoid audio artifacts. I^2S sound cards perform well with just 2 periods but USB sound cards usually require 3 periods.

5.7.3 SuperCollider

SuperCollider (McCartney 1996; McCartney 2002; Wilson et al. 2011) is a computer music programming language and synthesis engine, originally developed by James McCartney in 1996. SuperCollider saw its third major revision in 2002 and McCartney decided to make it free and open-source. Since then SuperCollider has grown to become a community effort with the contribution of many software developers.

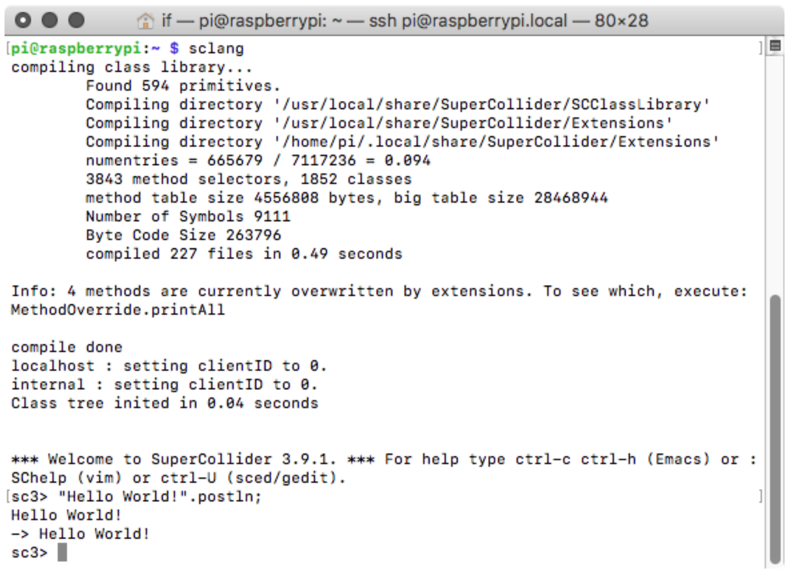
According to McCartney, SuperCollider was inspired by Music N languages but aimed to solve some of its limitations. Music N languages lacked fundamental programming principles, such as data and control structures or the ability to define user functions. Dataflow languages like Max or PD are closer to providing features like object-oriented programming and data structures, but they are still very static in consequence of the manual patching paradigm. Like Music N languages, SuperCollider uses the notion of instruments as arrangements of DSP units, but adding the flexibility of a more complete programming language to control them. The other major feature of SuperCollider is interactive programming, also called "just-in-time" (JIT) programming. Algorithms can be modified on-the-fly, updating variables, functions or routines, while the main program keeps running. This same flexibility is valid for the DSP engine, where unit generators can be inserted or removed from the signal path without any audio interruption or the need for a compilation stage. SuperCollider is also fully object-oriented and garbage-collected, avoiding many of the pitfalls of dynamically spawning and killing processes. Another important feature of SuperCollider is that it has a system of virtual buses that can be used to make any audio or control signal available to multiple processors, an extremely useful function in parallel audio processing chains or one-to-many mappings.

In essence, SuperCollider is a stable, highly efficient and full-featured language, with many functions and architectural details oriented to music and digital signal processing. All of these features make it one of the most powerful domain-specific languages in computer music and one that greatly favors real-time applications like DMIs. For these reasons, we believe SuperCollider to be an optimal choice for Prynth.

As previously mentioned, SuperCollider is divided in two different applications: scsynth and sclang. Scsynth is an independent sound server that is controlled from sclang via OSC messages. These messages can describe new DSP chains or update control parameters. A separation between DSP computation and control language has several benefits. Any error, malfunction or delay in sclang will not affect scsynth, which will continue to generate sound without interruption. Events are also time-stamped and can be scheduled to happen in the future, which benefits applications that have strict timing requirements, such as routines for sound sequencing. Scsynth can also be controlled from other programs, as long as they can generate and dispatch the correct OSC messages. A deep integration of the OSC protocol in embedded DMIs could be considered an advantage, because many of the modern embedded computing solutions already include networking interfaces. With an OSC infrastructure, it is very easy to exchange data between instruments, which can be done directly from SuperCollider user programs.

The SuperCollider binary distribution for personal computers is packaged with a third application, an IDE that also supervises and controls the scsynth and sclang applications. This desktop application has all the features expected from an IDE, such as a text editor, a debugging window and integrated documentation, complemented with several other contextual menu bar options and status monitoring panels. Alternatively, both scsynth and sclang can run as command line applications, without the need for the IDE front end. When launched using a shell, sclang prints the application's start-up information and then reaches a standby mode with its own interactive user shell. Figure 5.6 shows a "Hello World" example program written and executed from this interactive shell.

Because sclang can run as a command line application in Linux, it can also communicate with other processes or programs through the inputs and outputs commonly made available through the C Standard Library (stdio). This ability to exchange data streams between applications is


if — pi@raspberrypi: ~ — ssh pi@raspberrypi.local — 80×28

```
pi@raspberrypi:~ $ sclang
compiling class library...
        Found 594 primitives.
        Compiling directory '/usr/local/share/SuperCollider/SCClassLibrary'
        Compiling directory '/usr/local/share/SuperCollider/Extensions'
        Compiling directory '/home/pi/.local/share/SuperCollider/Extensions'
        numentries = 665679 / 7117236 = 0.094
        3843 method selectors, 1852 classes
        method table size 4556808 bytes, big table size 28468944
        Number of Symbols 9111
        Byte Code Size 263796
        compiled 227 files in 0.49 seconds

Info: 4 methods are currently overwritten by extensions. To see which, execute:
MethodOverride.printAll

compile done
localhost : setting clientID to 0.
internal : setting clientID to 0.
Class tree inited in 0.04 seconds


*** Welcome to SuperCollider 3.9.1. *** For help type ctrl-c ctrl-h (Emacs) or :
SChelp (vim) or ctrl-U (sced/gedit).
sc3> "Hello World!".postln;
Hello World!
-> Hello World!
sc3> 
```

Fig. 5.6 SuperCollider interactive shell.

considered a standard in Linux and other Unix-based applications, and arguably one of the most powerful features of these systems. For example, it is through the output function of stdio that the results of an application are printed to the shell on a command line interface. Prynth and most of its features are designed around this capability of a flexible interaction between SuperCollider and its companion Linux applications.

The Raspberry Pi and other ARM platforms are still not officially supported by SuperCollider. Fortunately, the compilation process is relatively straightforward and requires only minor adjustments in the preprocessing stage. A detailed description of the compilation of SuperCollider for Prynth can be found in appendix A.3.

5.8 Bridge Application

Pbridge is a Prynth middleware application that runs constantly on the Raspberry Pi and that acts as a communication relay between the Prynth server, SuperCollider and the Teensy microcontroller. The first two use the OSC protocol, while the microcontroller uses Prynth's custom serial protocol. The pbridge application is written in C, using the *liblo* (Liblo 2018) library for

OSC communication and the WiringPi library to control the Raspberry Pi's UART.

The most important function of pbridge is to receive sensor data from the Teensy and send it to SuperCollider for synthesis control. In pbridge, each serial message is pooled from the RPi UART, decoded and checked for integrity. If the received message is valid, the sensor data is encapsulated in an OSC message and dispatched to SuperCollider. These messages can be sent either to sclang or scsynth via their respective ports. This option can be set by launching pbridge with the appropriate flag: -s for server or -l for language. The practical difference between these two modes of sending sensor data to SuperCollider is further explained in section 5.10.

Another function of pbridge is to transcode messages that are sent from the Prynth server to the Teensy, in order to control the signal acquisition process. These include turning inputs on and off, changing acquisition sampling rate, changing parameters of the data conditioning processors or controlling any other function that happens on the microcontroller.

Finally, pbridge also sends data to the Prynth server to provide feedback to the user. For example, pbridge will throw an error after 10 failed attempts to set a state on the microcontroller, which could result from any unexpected error with the serial communication. In this case, the user is given feedback through a message sent to the Prynth server and displayed on the Prynth web client. Another useful function of pbridge is that it can be set to listen to a specific sensor value and send the data to an OSC port other than the two reserved for SuperCollider. The Prynth server can then listen to this port and relay the data in real-time to the web client for data monitoring.

5.9 Prynth Server

The Prynth server is the pivotal application that integrates all the previously described software and hardware components. The server can control child applications, manage software processes, manipulate files and interact with Linux via shell commands. User interaction with the server is done through a web client application, in the form of a webpage through which the user programs and configures the instrument.

First we will explain the functionality of Prynth's client application from a user perspective, based on the description of the application's GUI and by relating those affordances with the features of the system. After this introduction we will describe in greater detail the architecture for client/server communication and the interaction between server, applications and operating system.

5.9.1 Client Application Features and User Interface

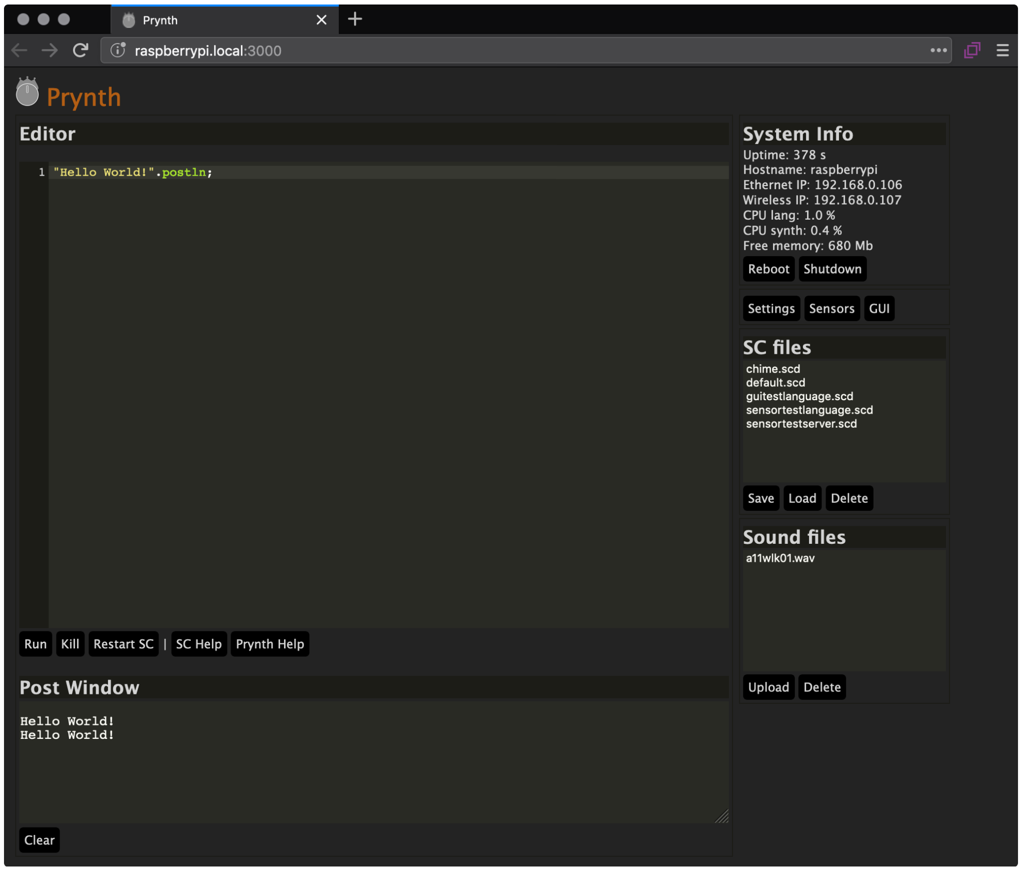
In terms of user-level interaction, Prynth's client application can be divided into the following major components: the programming text editor, file managers, settings panels and graphical user interfaces.

The client application is served by the instrument itself and runs on the web browser of any other device on the same network. It is written entirely in JavaScript, so there is no need for any extraneous software packages or plug-ins. To access the client application, the user must simply open the instrument's uniform resource locator (URL) on a browser. This URL is associated to the system's hostname on the local network. The default hostname in Prynth's distribution is "raspberrypi", so the corresponding URL would be raspberrypi.local:3000.

Main Window

One of the main purposes of the web application is to offer an interface to program the instrument. Since programming is at the core of Prynth, the main page of Prynth's web application also presents a layout similar to that of an IDE (see figure 5.7). The majority of the page is occupied by a text field, in which the user can write new programs using the full extent of SuperCollider's syntax and classes.

Prynth also inherits the flexibility of SuperCollider's interactive programming. It is possible to write and evaluate new functions of a program that is already running. In SuperCollider, running a fragment of source code can be done by selecting that portion of text on the editor and using a keyboard shortcut to trigger the execution. Interactive programming is further discussed in section 5.9.8.



```
"Hello World!".postln;
```

```
Hello World!
Hello World!
```

Uptime: 378 s
Hostname: raspberrypi
Ethernet IP: 192.168.0.106
Wireless IP: 192.168.0.107
CPU lang: 1.0 %
CPU synth: 0.4 %
Free memory: 680 Mb

chime.scd
default.scd
guitestlanguage.scd
sensortestlanguage.scd
sensortestserver.scd

a11wlk01.wav

Fig. 5.7 Main page of front end application.

Below the programming text field there is a second text field to display messages from SuperCollider. Most IDEs have some sort of console window to print feedback messages, essential to programming activities such as *debugging* a program. In the original SuperCollider IDE, the console window is also called "Post Window"[3].

Adjacent to the editor area and the post window there are several buttons with different functions. The "Run" button executes the entirety of the SuperCollider source code currently on the editor. The "Kill" button stops all scsynth DSP calculations and sclang routines, a function that is often used in SuperCollider as a panic button. "Restart SC" can be used to kill and restart

[3]Throughout this text we sometimes refer to the post window as console and vice versa.

the SuperCollider process and aid in the recovery from any locking programming error. The "SC Help" button opens SuperCollider's official documentation and "Prynth Help" opens Prynth's own documentation. Both documentations are in hypertext markup language (HTML) format and the pages open in a separate web browser tab, so they can be consulted side-by-side with the editor. Finally, the post window itself has a dedicated "Clear" button to clear all SuperCollider messages on display.

On the far right of the window we find several other panels and buttons. The first panel is a "System Info" panel, that displays real-time information of the system, including uptime, hostname, assigned IP addresses, CPU usage of both sclang and scsynth, and the amount of free RAM. Below this information panel there are two buttons to "Reboot" or "Shutdown" the system. On the bottom, we find two file managers, the first for SuperCollider programs and the second for sound files. Both panels are composed of a list of files and buttons for loading, saving or deleting files.

The three remaining buttons, labeled "System", "Sensors" and "GUI", open new browser tabs with different functions, as described next.

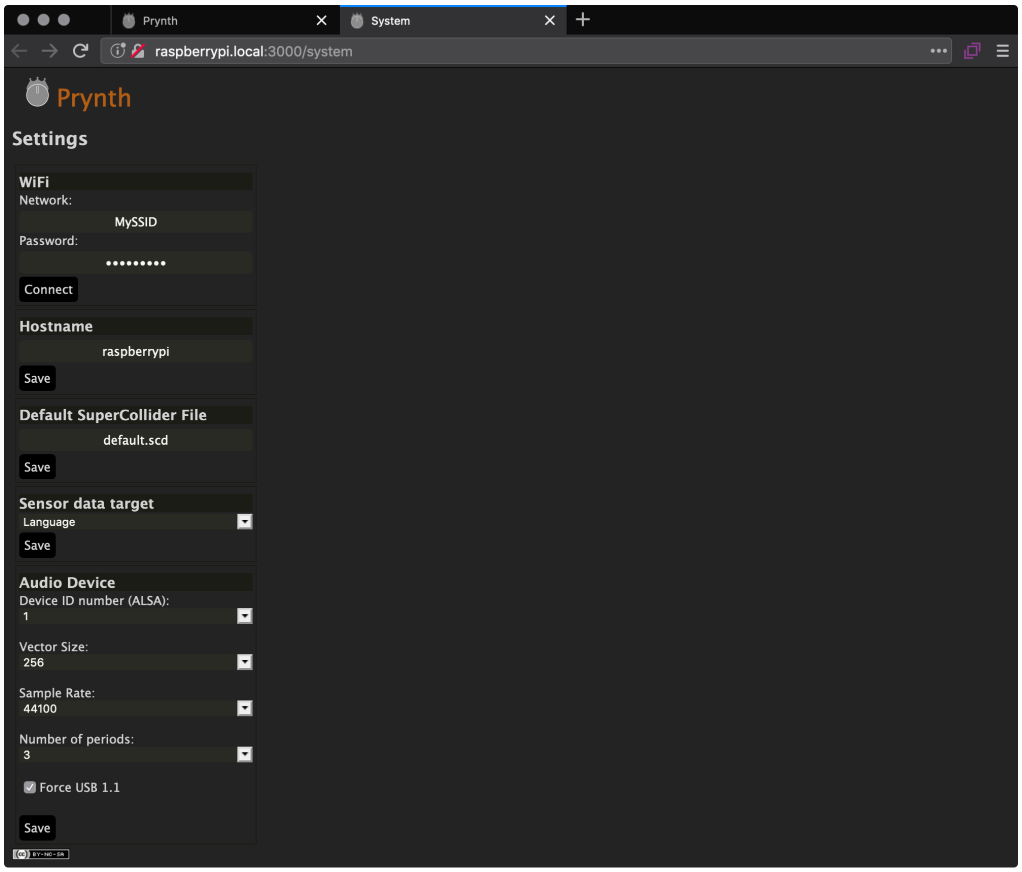
System Settings

The System page offers control over several system-wide settings (figure 5.8).

The first panel facilitates the connection of the instrument to a wireless network. Filling the text fields with the correct authentication credentials and clicking "Connect" will try to authenticate and connect to the specified wireless network, storing the credentials for future use.

The second option panel can be used to easily change the hostname. The hostname is crucial because it propagates to many important functions of the system, including the format of the OSC addresses or the server's root URL. It is convenient to change the default hostname from "raspberrypi" to any other name that would allow for an easy identification (eg. http://instrument1.local:3000).

The "Default SuperCollider File" is the name of the file that runs automatically after the instrument has reached a ready-state. The Prynth distribution ships with a default file named



Settings

WiFi
Network: MySSID
Password: •••••••••
Connect

Hostname
raspberrypi
Save

Default SuperCollider File
default.scd
Save

Sensor data target
Language
Save

Audio Device
Device ID number (ALSA): 1
Vector Size: 256
Sample Rate: 44100
Number of periods: 3
✓ Force USB 1.1
Save

Fig. 5.8 System settings.

“default.scd”, which starts the SuperCollider server and plays a simple chime to provide an auditory feedback that the booting process is complete.

The "Sensor Data Target" is another important setting because it determines which application, sclang or scsynth, will receive the OSC message streams carrying sensor data. Typically a user would choose to receive this data on the language to use it in an algorithm. On the other hand, receiving the control data directly on the SuperCollider server allows for different DSP programming approaches, where the data can be used as a generator within the DSP chain. More details about these two techniques are presented in section 5.10. In practice, changing the data target sets which option flag to be used when starting the pbridge application, as explained in section 5.8.

The remainder settings are related to JACK or to how the system interacts with the sound output device. The "Device ID Number" is used as an index pointing to the sound devices available in ALSA. In Prynth device number 0 is the Raspberry Pi's analog output (PWM) and number 1 refers to the first audio device added by the user (USB or I^2S sound card). "Vector Size", "Sampling Rate" and "Number of Periods" are the typical performance settings of a digital audio system, as explained in section 5.7.2. The final option in this panel is a checkbox that sets the speed of the Raspberry Pi USB controller, thus avoiding the audio artifacts with USB 1.1 audio cards (as explained in section 5.5).

Some of these settings require a reboot, in which cases an HTML pop-up window will prompt the user to confirm the operation.

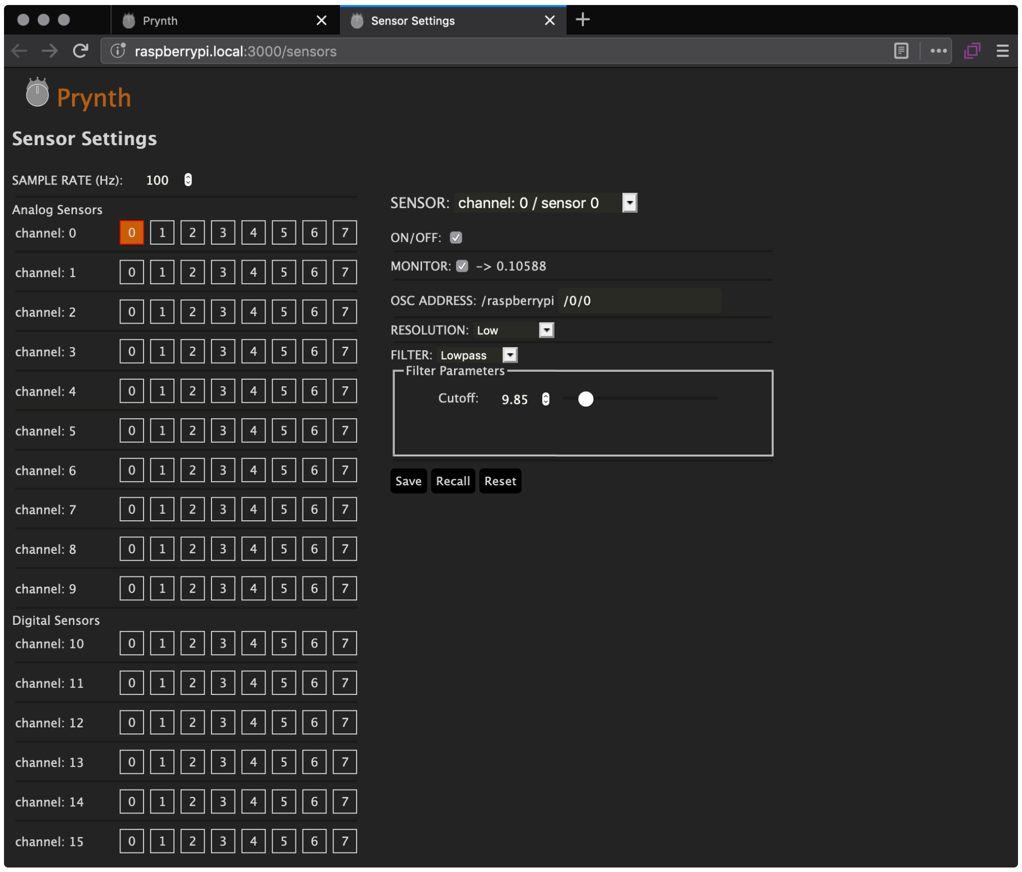
Sensor Settings

Prynth's sensor acquisition system is managed through the "Sensors" page (figure 5.9), which presents a series of controls that generate real-time messages that are sent to the Teensy via pbridge application.

The first option is the sensor acquisition sample rate, which can be set between 1 to 200 Hz. This value represents the acquisition speed for each sensor, independently of how many sensors are actively being used.

Next we find a grid that represents all possible 128 sensors in a Prynth system, divided into 16 channels with 8 sensors each. The first 10 channels correspond to each of the connected Muxi boards and the sensor number to their respective physical inputs. Each cell of the grid provides visual feedback concerning the state of a sensor. An orange background represents that the sensor is turned on and a gray background that it is off. The grid can also be used to select a specific sensor, which becomes highlighted by a red border.

The properties panel to the right is used to display and change the properties of each sensor. A dropdown menu shows which channel and sensor are currently selected (the dropdown menu itself also works as an alternative selection method). The On/Off checkbox enables or disables the


Prynth
Sensor Settings
raspberrypi.local:3000/sensors
Prynth
Sensor Settings
Analog Sensors
channel: 0 0 1 2 3 4 5 6 7
channel: 1 0 1 2 3 4 5 6 7
channel: 2 0 1 2 3 4 5 6 7
channel: 3 0 1 2 3 4 5 6 7
channel: 4 0 1 2 3 4 5 6 7
channel: 5 0 1 2 3 4 5 6 7
channel: 6 0 1 2 3 4 5 6 7
channel: 7 0 1 2 3 4 5 6 7
channel: 8 0 1 2 3 4 5 6 7
channel: 9 0 1 2 3 4 5 6 7
Digital Sensors
channel: 10 0 1 2 3 4 5 6 7
channel: 11 0 1 2 3 4 5 6 7
channel: 12 0 1 2 3 4 5 6 7
channel: 13 0 1 2 3 4 5 6 7
channel: 14 0 1 2 3 4 5 6 7
channel: 15 0 1 2 3 4 5 6 7
Filter Parameters
Save
Recall
Reset

SAMPLE RATE (Hz): 100

SENSOR: channel: 0 / sensor 0

ON/OFF: ✓

MONITOR: ✓ -> 0.10588

OSC ADDRESS: /raspberrypi /0/0

RESOLUTION: Low

FILTER: Lowpass

Cutoff: 9.85

Fig. 5.9 Sensor settings.

sensor and the "Monitor" option enables real-time monitoring of corresponding data value.

In Prynth, each sensor sends data to sclang using an OSC message with a unique address. The address starts with the hostname, followed by a configurable string. The default string for each sensor has a hierarchical format, starting with the channel number, followed by a forward slash and the sensor number. As an example, the first sensor on the first channel would send messages on the OSC address /raspberrypi/0/0. The hostname string is fixed but the rest of the address can be arbitrarily configured using the provided text field. The configuration of custom OSC messages is useful for descriptive address schemes (eg. /raspberrypi/fx1/gain). It is important to reinforce that this messaging convention takes into account the hostname of the instrument, so that changing it (as explained in the previous section) would also result in a change of the hostname prefix (eg. /instrument1/0/0). With this convention each instrument can have a unique namespace.

The data resolution of each sensor can be set to low (1 byte) or high (4 bytes). MIDI controllers typically use messages with 7 bits for data representation (equivalent to 127 possible values). Prynth's lower data resolution uses a full byte (256 values), which already represents twice the precision of the MIDI standard.

Finally, each sensor can have its own data conditioning processor, with choices of low-pass filter, One Euro filter and Boolean processor. Choosing a different processor on the dropdown menu will dynamically change the parameter controls (a low-pass filter will have one input slider, while a One Euro filter will have three).

The last set of controls has three buttons. The "Save" button saves the current configuration, so it is automatically recalled on the next start-up. This configuration can also be reloaded at any time using the "Recall" button and cleared using the "Reset" button.

Graphical User Interfaces

GUIs were not considered in Prynth's initial specifications. Our objective was to build self-contained instruments that could work without displays. GUIs seemed of little use in a system based on a textual programming language and aimed at emphasizing tangible interaction.

This preconception changed when a user described a new use case for Prynth. The goal was to build an interactive installation that would be exhibited on a museum for several months and where it would be useful to have some sort of virtual control panel that the museum staff could operate. This control panel should be accessible through a mobile device, allowing for operations like changing volume or rebooting the system.

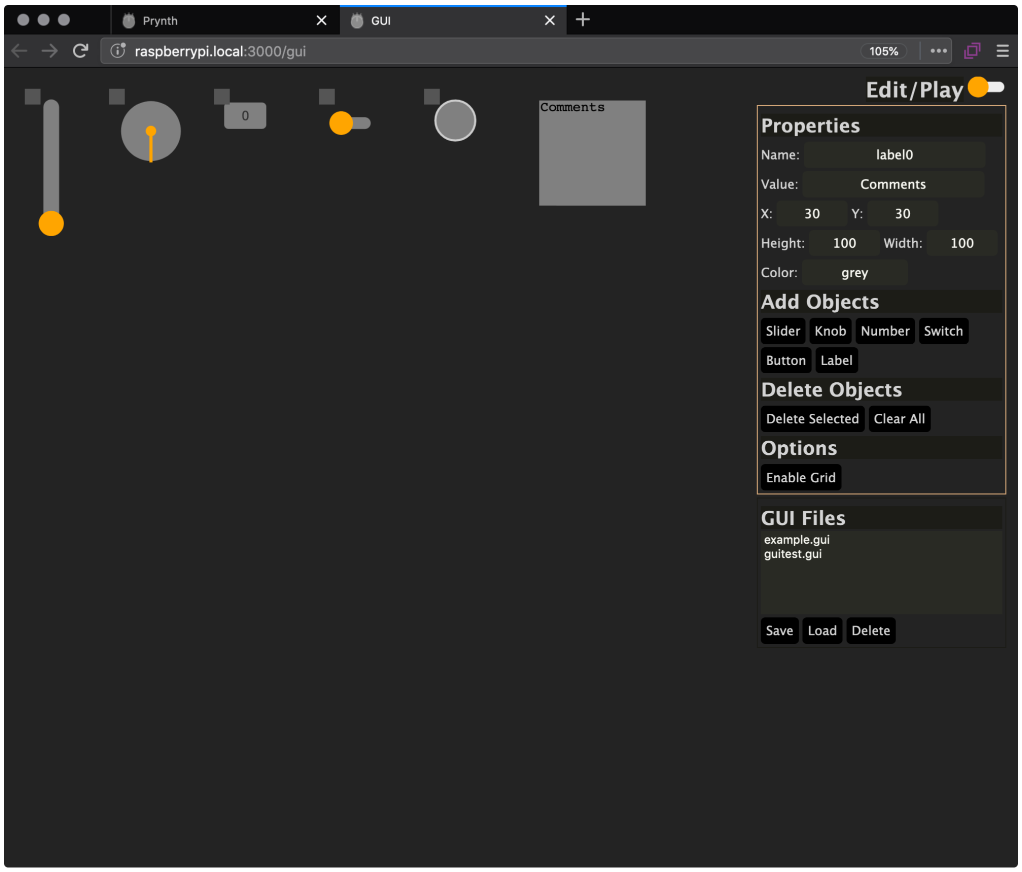
Surely this functionality could be achieved using any third-party OSC GUI software. There are many of these applications available for mobile platforms. They allow for the creation of custom GUIs by arranging virtual controls and specifying the format of their OSC output. Another solution would be to build a custom webpage to send the appropriate commands via Prynth server. However this would be more difficult to implement because it would require knowledge about Prynth's server architecture and the modification of its source code.

The described use case suggested a reevaluation of the purpose of GUIs in Prynth. We began to imagine a scenario where a musician could start the Prynth instrument and access the server via mobile device, to be presented with a webpage with GUI controls for adjusting parameters. These could include simple tuning, synthesis parameters, changing DSP chains or recalling presets. A GUI layer in Prynth could work as a complementary utility, with no obvious impact on the autonomy or workflow of the instrument. These GUIs would have to be fully configurable to fit the functionality of a particular instrument, matching the flexibility of SuperCollider and the rest of Prynth's components.

Prynth's GUI editor follows the same functional principles as many other GUI building applications. The user populates a blank canvas with different GUI objects that send OSC messages to sclang. The properties of the GUI elements can be modified, including visual properties (size, color) or other meta-properties, such as the object's name and the format of the messages it sends. Objects can be dragged and freely repositioned anywhere within the canvas.

Figure 5.10 shows a screenshot of Prynth's GUI editor, with six different types of virtual objects: a slider, a knob, a number, a switch, a button and a text label. Each object has a small square on the upper left corner, used as a handle to select or drag objects. Selected objects are highlighted so they can be easily localized in space.

To the right of the canvas we find various configuration panels. The first element is an isolated switch labeled "Edit/Play", which toggles between edit and play modes. The "Play" mode hides all the property panels and locks the position of the GUI elements on the canvas. Next we find a properties panel to display and change the values for the currently selected object. These include the name of the object, current value, x and y positioning coordinates, width and height and color. Below the properties panel there is a series of buttons to add and delete objects, and also one to clear the entirety of the canvas. There is also an option to enable a grid mode, increasing the step of the drag operation for easier alignment. To minimize pointer travel, the whole properties panel



Name: label0
Value: Comments
X: 30 Y: 30
Height: 100 Width: 100
Color: grey

example.gui
guitest.gui

Fig. 5.10 GUI editor.

can be dragged and repositioned closer to the objects that are currently being edited.

The final panel is a file manager to load, save or delete different GUIs and that operates similarly to the other file managers in Prynth.

Prynth's GUIs are built on top of two libraries: NexusUI and Draggabilly. NexusUI (Taylor et al. 2014; Nexus Web Audio Interfaces 2018) is a JavaScript framework developed by Ben Taylor and is the source of the HTML 5 GUI elements (sliders, buttons, dials). Draggabilly (DeSandro 2018) is used to add the ability of freely dragging the GUI elements on the canvas. Draggabilly facilitates this operation by using *div* HTML containers that inherit the dragging properties via HTML class. It is inside these div containers that we instantiate the NexusUI objects.

When manipulated, a GUI element sends values to the Prynth server via web socket, which are then converted to OSC messages and emitted on the SuperCollider port. The format of the OSC address is built from the object's name, preceded by the "gui" string.

```
/gui/myslider
```

A SuperCollider OSC responder function can then be set to listen to this message address and used similarly to any other sensor signal. In section 5.10 we demonstrate how to build synthesizers and map control signals in Prynth.

Prynth's GUIs can be accessed at any time, via any thin client, using the URL `http://raspberrypi.local:3000/gui` or launched in a new tab using the button on Prynth's front page. The last edited GUI is automatically loaded and displayed.

5.9.2 Node.js

The Prynth server is built using Node.js (Node.js Foundation 2018c), a technology that recently became very popular in web server development. Node.js is a runtime environment for JavaScript. JavaScript was originally developed to run exclusively on web browsers, but with Node.js it can now be used to run programs on local machines and for virtually any purpose. One of the most relevant use cases of Node.js is the creation of web servers because developers can now use the same language for both server and client applications.

Scripting languages are usually not the most efficient but in Node.js scripts are compiled to machine code at start-up time, making JavaScript programs much leaner. Additionally, Node.js uses an event-driven model with non-blocking I/O operations. Node.js serves all requests from a single thread called "event loop". When a function runs, new events are added to the event queue that proceeds with any non-blocking operations. If a blocking operation (such as reading a file or encoding/decoding data) is required, the event loop delegates this operation to a different thread pool, while the event queue continues to run in parallel. When the thread pool has processed the blocking operation, it sends data back to the event queue, which continues processing via a callback function as soon as it is free to do so. The thread pool itself is capable of multithreading through native C++ operations of the engine.

This architecture allows for non-blocking behavior, since the event loop continues to run in parallel while delegating blocking operations to a threaded pool that automatically manages concurrency. In practice, this means that a web application can continue to be responsive and interact with the server, even while other blocking processes are being computed. The downside to this model is the uncertainty as to when a function will return. To assure proper execution order, functions that depend on the output of other functions must be nested hierarchically via callback mechanism, sometimes resulting in dense and convoluted algorithms. The great advantage of using asynchronous non-blocking programming in Prynth is that the web server can be extremely efficient and lightweight, without any significant impact on the performance of high-priority audio applications.

JavaScript modules are units of independent code that can be included in a program via *require* command. They work similarly to classes and can export object literals, functions or constructors. Modules developed by third-parties are compiled into libraries and distributed through npm (Npm 2018), a web-based package manager deeply integrated with Node.js. Prynth uses many different third-party Node.js libraries, some of which depend on yet more libraries. It is out of the scope of this document to describe every library in detail, mentioning only the most relevant to Prynth's core implementation.

In JavaScript, the preferred data format is the JavaScript object notation (JSON). Prynth's configuration and GUI files are stored in this format. A JSON file consists of groups of key/value pairs and can include several data types (number, string, Boolean, array, object). JavaScript has native functions to conveniently index, query and parse JSON data.

5.9.3 HTTP Requests

The hypertext transfer protocol (HTTP) is the standard application protocol used for data communication between clients and servers of the World Wide Web. HTTP uses a request/response method. In the case of a webpage, the client (the browser) requests data from a resource on the server, which in turn responds with data that is rendered by the browser (hypertext, images, video or sounds). In the early days of the World Wide Web, the most common content was web pages, but today many web services have online APIs that can respond with any type of arbitrary data, such as geographical mapping or weather data.

HTTP requests always need some form of identifier pointing to a resource, the most common being the URL, also commonly called the "address". Besides fetching data, URLs can also be used to trigger functions on the server. The HTTP specification defines different types of requests, identified by a method (also called verb). The most common verbs are GET, used to retrieve resources from the URL, and POST, used to submit data for processing on the server. A POST request also contains a body message that can be used to carry any arbitrary data.

Modern JavaScript engines include a more powerful API for HTTP requests, called XML-HttpRequest (XHR) (World Wide Web Consortium 2018). XHR is the foundation for a series of web development techniques called asynchronous JavaScript and XML (AJAX) (Garrett 2018). With AJAX, the client can make asynchronous requests and use the response data via callback function. In practice, this means that the server no longer needs to respond with a full web page. Instead, the client can receive partial data and update the page, making it much more interactive.

As an example, the Prynth client can submit a POST request through AJAX, containing a string with a new SuperCollider program to be executed (interpreted). Here is an example using JQuery (jQuery 2018), a JavaScript library that simplifies client-side scripting:

```
$.ajax({method: "POST", url: '/interpret', data: {code: '1+1'}});
```

In the above example, $ is an alias to the JQuery object, which performs an AJAX request of type POST to the "interpret" URL, together with the code for the execution of a simple sum operation. In section 5.9.7 we explain how a client request is handled by the server.

5.9.4 Express API

Prynth also uses Express (Node.js Foundation 2018b), a web framework that facilitates the development of web applications using Node.js. Express is extremely popular due to its simplicity, strong API, good documentation and active community. Creating a server in Express is extremely straightforward. Here is a simple example of a server, adapted from Express's documentation (Node.js Foundation 2018a).

```
//use the Express framework
const express = require('express')
//start the application
const app = express() //start an application
// Set response to GET request on root URL
app.get('/', function (req, res) {
  res.send('Hello World!');
})
//start listening on Port 3000
app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

In this example we can observe that the web application includes a *get* method to respond to a GET request. The get method takes a string defining the URL relative to root path ('/'), passing the request and response objects (*req* and *res*) to a callback function. This function then invokes the send method of the res object, using the "Hello World!" string as an argument. A similar method can be used to create a response to a POST request:

```
app.post('/', function (req, res) {
 res.send('POST request to the homepage')
})
```

In Express, the definition of how an application responds to a HTTP request is called “routing”. By creating routes, the application listens to matching requests, triggering the associated callback function. This callback function can operate on the server or respond with data. There are many types of response methods, including a JSON response (for data transport) and a render function. The latter can be used to dynamically create HTML from template engines. Prynth uses the EJS template engine (Eernisse 2018), where variables can be easily passed to the render engine. Here is an example of the rendering of an index HTML page with a specific title. The router renders the “myindex.ejs” file using the variable “mytitle”.

The routing method:

```
router.get('/', function(req, res) {
  res.render('myindex', { mytitle: 'My Webpage' });
});
```

The EJS template for the webpage, called “myindex.ejs”:

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= mytitle %></title>
  </head>
  <body>
    <p>Welcome to <%= mytitle %></p>
  </body>
</html>
```

5.9.5 Sockets and Real-time Communication

Web sockets is another technology that can be used for the real-time communication between client and server. This communication protocol allows both ends to push data over a TCP connection without an explicit request. Web sockets are available to Node.js through the Socket.IO library (Socket.IO 2018).

In Prynth we use web sockets whenever we need the server to communicate a status back to the client. This includes Prynth's Post Window, which receives not only the SuperCollider *stdout* stream, but also messages concerning filesystem operations. The client's system status panel is also updated periodically by receiving a socket message that is triggered on the server using a timed routine.

OSC messages are also exchanged between client and server using web sockets as an intermediary. An example is the real-time sensor monitoring data available on the sensor settings page. The Prynth server receives an OSC message from pbridge and relays the monitored value to the client. The opposite also occurs when a user updates a parameter on the sensor settings page or manipulates a GUI element. A message is sent to the server via socket, transcoded to an OSC message and finally dispatched to pbridge or sclang.

Below is an example of how to create a web socket connection on the server and add it to the Express middleware:

```
// create sockets
var server = require('http').Server(app);
var io = require('socket.io')(server);

//add to Node.js application middleware
app.use(function(req, res, next){
  res.io = io;
  next();
});
```

It is then possible to emit socket messages to the client. Here is an example of how to send a message to the Post Window.

```
res.io.emit('toconsole', 'Process complete');
```

In turn, the client will have a function that responds to an event named "toconsole" to update the console's HTML text field:

```
<script src="javascripts/socket.io/socket.io.js"></script>
<script>
var socket = io.connect();
socket.on('toconsole', function (data) {
   $('#consoleTextarea').append(data);
});
</script>
```

5.9.6 Interacting with Linux via Child Processes

Node.js can interact with the Linux operating system by launching new subprocesses through the *child process* module. In Prynth, child processes are used to run companion applications (like JACK or pbridge) or to execute Linux shell commands.

To run local applications, we use the *spawn* method, which automatically registers event responders for the stdio streams of the child instance. It is through these event emitters and listeners that the server communicates with the application.

Shell commands are issued via the *exec* method, which also attaches a shell to the process. It then becomes possible to use standard shell functions, such as executing two commands in series or piping applications.

Below is a small example of how to write new Wi-Fi settings by concatenating a sequence of shell commands. It disconnects the networking services, after which it sets the new credentials via command line application and finally brings the networking services back up.

```
router.post('/setwifi', function (req, res) {
  var networkname = req.body.networkname;
  var networkpass = req.body.networkpass;

  var command =
  'sudo ip link set wlan0 down'+
  ' && sudo wpa_cli -i wlan0 add_network 0 && sudo wpa_cli -i wlan0 set_network 0 ssid
'+
  '\'\"'+networkname+'\"\''+
  ' && sudo wpa_cli -i wlan0 set_network 0 psk '+
  '\'\"'+networkpass+'\"\''+
  ' && sudo wpa_cli -i wlan0 enable_network 0 && sudo wpa_cli -i wlan0 save_config '+
  '&& sudo ip link set wlan0 up';

  var child = exec(command, function () {
    res.redirect('/system');
  })
})
```

5.9.7 Interacting with SuperCollider via HTTP

Written by Chris Satinger, SuperCollider JS (Satinger 2018) is a Node.js library to control Super-Collider. With SuperCollider JS it is possible to write JavaScript commands that are transformed into OSC messages understood by the sclang and scsynth applications. This library is still under development but already includes several relevant functions, such as starting the SuperCollider server, creating groups, spawning synths or setting parameters.

In Prynth we do not use SuperCollider JS to its full extent because our goal is to use SuperCollider programs written in their native language and not in JavaScript. Yet, SuperCollider JS has a very convenient *SCLang* object, that elegantly implements the stdio communication with the application. Stdout listeners are used to report sclang errors, while a *interpret* method is used to execute a string piped via stdin.

Here is a simplified program of how sclang is instantiated when the Prynth server starts:

```
var sc = require('supercolliderjs');
var sclang;

function startSclang() {
  sc.lang.boot().then(function (lang) {
    sclang = lang;
    sclang.on('stdout', function (text) {
      io.sockets.emit('toconsole', text);
    });
  });
};
```

When booting sclang, an object instance is returned and attributed to a global variable, so that it can be controlled from multiple scopes on the server. A listener function is attached to stdout, returning the messages of sclang's start-up process to the console.

In Prynth we also create a global event responder, so we can issue "interpret" commands to execute SuperCollider code from anywhere on the server. Like the start-up process, this event responder includes a callback function that returns to the client's console.

```
app.on('interpret', function (msg) {
    sclang.interpret(msg)
        .then(function(result) {
            io.sockets.emit('toconsole', result);
        })
        .catch(function (error) {
            var errorStringArray = ;
            io.sockets.emit('toconsole', JSON.stringify(error));
        });
});
```

The last step is to create the corresponding route function, to respond to client POST requests transporting the code for evaluation (see section 5.9.3). The acknowledgment of a successful request is done by returning a standard HTTP status code to the client (200 for OK).

```
router.post('/interpret', function (req, res) {
  res.app.emit('interpret', req.body.code);
  res.sendStatus(200);
});
```

5.9.8 SuperCollider Editor

Prynth's code editor is developed on top of CodeMirror (CodeMirror 2018), a JavaScript text editor that runs entirely on the browser. CodeMirror's text buffer is a plain HTML *textarea*, which greatly facilitates access through basic JavaScript get/set operations. CodeMirror also implements many important features of a programming editor, including history, bracket matching, syntax colorizing and configurable key bindings.

Interactive programming is one of SuperCollider's distinctive features. The user can write a part of the program, execute it, and then continue to write other subprograms during runtime. In the desktop version of SuperCollider this behavior is implemented on the IDE itself. A code fragment can be selected and triggered to execute via a shortcut key (Command+Enter on a Macintosh and Control+Enter on Linux and Windows systems). If there is no text selection, SuperCollider will try to execute the code within a matching pair of parenthesis. If there is no matching pair, it will try to execute the line where the cursor is positioned. Although this behavior may seem confusing at first, it quickly becomes second nature after some experience with SuperCollider. Interactive programming may be used in many other contexts, but it is particularly interesting in the context of DMIs. For example, the fine-tuning of a synthesis algorithm, the incremental adjustment of mappings or the morphing between machine states can be done interactively while the instrument is being played.

In Prynth we have tried to emulate the exact same behavior. When the execution shortcut key is pressed, there is an algorithm that retrieves the string hierarchically through either parenthesis matching, string selection or string at cursor position and passes it to the interpret POST request (as previously explained in sections 5.9.3 and 5.9.7).

SuperCollider has other important default key bindings. The first is Command+. , used to stop all DSP and routine processing. In Prynth this functionality is achieved by triggering the interpret POST request with the programmatic version of the command: "CmdPeriod.run". Another important shortcut, also replicated in Prynth, is Command+d , which searches the SuperCollider documentation for the string where the cursor is resting. In Prynth, we use the HTML version of SuperCollider's documentation and inject the collected string into a search field that returns the help results.

5.9.9 Files

SuperCollider programs and other persistent data are stored in files that are read, parsed, modified and written using the *fs* filesystem module of Node.js.

In the case of SuperCollider programs, they are stored in a plain text format with the "scd" filename extension. This is the same file format used by the desktop version of SuperCollider, making files completely interchangeable between systems. To save the current program, a JavaScript function fetches the current string on the HTML textarea shared with CodeMirror and asks the user for a new file name. It then generates a POST request with address “supercolliderfiles”, containing the program string, a “save” action keyword and the name for the file. On the server side, the equivalent route is triggered, passing the message body to the *writeFile* method of the filesystem object. The respective callback function emits a message to the Post window, confirming the success of the file writing process.

```
var fs = require('fs');


router.post('/supercolliderfiles', function (req, res) {
   var path = 'home/pi/prynth/server/public/supercolliderfiles/';
   if(req.body.action === 'save'){
         fs.writeFile(path + req.body.filename, req.body.code, function (err) {
         if(err) {
            res.io.emit('toconsole', 'error saving file');
         } else {
            res.io.emit('toconsole', 'file saved');
         }
      });
   };
})
```

The loading of a file is a similar operation. The POST message sends the name of the file to be read, together with the “load” action. The corresponding route function on the server reads the file using the *readFile* method and emits the string to the client that receives it and updates the HTML textarea.

Configuration files are stored in the JSON format, which are easy to manipulate via JavaScript objects. Here is an example of Prynth’s main configuration file, which stores the system settings explained in section 5.9.1.

```
{
  "defaultSCFile": "default.scd",
  "defaultGUIFile": "example.gui",
  "wifinetworks": [
    "mySSID",
    "myPassword"
  ],
  "hostname": "raspberrypi",
  "sensorDataTarget": "l",
  "jack": {
    "device": "1",
    "vectorSize": "256",
    "sampleRate": "44100",
    "periods": "3",
    "usb1": "false"
  }
}
```

Audio files are the third type of files handled by the Prynth server. They can be uploaded to the "soundfiles" public folder and then loaded from any SuperCollider program. A binary file can be uploaded by the client application using a POST request with a multipart/form-data type, which allows for the combination of different types of data into a single body message. This multipart POST is then received and handled in the server using the Multer library (Multer 2018), which facilitates extraction and merging of the several parts of the binary file.

5.10 Sound Synthesis and Mapping in Prynth

In this section we provide some examples of how to create simple synthesizers and map control signals in Prynth. Our goal is not only to offer comprehensive examples but also to demonstrate the different programming techniques available.

5.10.1 Creating a Synthesizer

In SuperCollider, synthesizers are built using the *SynthDef* class, which holds a definition of an arrangement of unit generators and respective control arguments. Below is an example of a simple FM synthesizer:

```
SynthDef(\fm, { |freq = 220, modIndex = 0.5, modAmp = 0.5|
  var carrier, modulator;

  modulator = SinOsc.ar(freq * modIndex, 0, freq * modAmp);
  carrier = SinOsc.ar(freq + modulator, 0, 0.5);

  Out.ar([0,1], carrier);

}).add;
```

In this example we have a SynthDef with name “fm”, which defines an instrument with arguments “freq”, “modIndex” and “modAmp”. Inside the function we declare two variables that will hold the carrier and modulator signals. The signals are created using the sinusoidal oscillator object “SinOsc”, in which the first argument is frequency, the second phase and the third a value by which the output signal will be multiplied. The frequency and amplitude parameters for the modulator are derived from the multiplication of the synth’s base frequency (SynthDef’s own freq argument) by their respective modulation index or modulation amplitude. The carrier signal then uses the same base frequency, to which it adds the signal of the modulator. The final gain of the carrier signal is adjusted using a multiplier of 0.5. Finally, the "Out" object outputs the signal. It takes a first argument with the destination output and a second with the signal itself. Our destination output is the array of channels 0 and 1, equivalent to the left and right channels of a stereo output. The signal we wish to output is the carrier, which is duplicated and assigned to the left and right channels, according to SuperCollider's automatic multichannel expansion[4]. Last, we invoke the *add* method, which sends the synthesizer definition from sclang to scserver.

The synthesizer can now be instantiated and assigned to a global variable named "a".

```
a = Synth(\fm);
```

The *set* method is used to modify any of the input arguments of the synthesizer.

```
a.set(\freq, 440);
a.set(\modIndex, 0.7, \modAmp, 0.9)
```

5.10.2 Mapping Sensor Data

With an operational synth, we can now create an *OSCdef* responder function that is triggered every time sclang receives an OSC message on the defined address.

```
OSCdef(\sensor1, {|msg|
   var rawValue, cookedValue;
   rawValue = msg[1];
   cookedValue = rawValue.linlin(0,1,220,440)
   a.set(\freq, cookedValue);
```

[4]For more information on multichannel expansion please consult the official SuperCollider documentation (*SuperCollider 3.10.0 Help* 2018)

```
}, ’/raspberrypi/0/0’);
```

The OSCdef object will have at least three essential parameters: a unique name, a function and a string with the OSC address that will trigger the callback function. In our example, the trigger is set to OSC messages with address /raspberrypi/0/0, which is the default message associated to the first sensor on the first multiplexer (0/0) of the instrument with hostname “raspberrypi”.

Inside the function, the first argument is the OSC message received, assigned to the variable “msg”. The message is an array, where the first element is the trigger address, followed by any other elements of the OSC message. In the previous example, the array will have two elements:

```
[/raspberrypi/0/0, \textit{value}].
```

In the callback function we retrieve the data value by index and assign it to variable “rawValue”. Since we know that all sensor values in Prynth are normalized, we can perform a linear scaling using the *linlin* method, which accepts low and high input values (0 and 1) and scales them to the desired low and high output range. The rescaled value, now ranging from 220 to 440, can then be used to set the frequency parameter of our synthesizer.

A small drawback is that the user must create OSCdef functions for each individual address.

Another way to map parameters to synthesizers is to use control buses as intermediaries. Once we hold the sensor value in the OSCdef function, we can use it to set the value of a control bus from which the synthesizer reads continuously. Below is a complete program with a bus mapped to the frequency parameter of our FM synthesizer.

```
fork {

   s = Server.local;

```

```
  SynthDef(\fm, { |freq = 220, modIndex = 0.5, modAmp = 0.5|
    var carrier, modulator;
    modulator = SinOsc.ar(freq * modIndex, 0, freq * modAmp);
    carrier = SinOsc.ar(freq + modulator, 0, 0.5);
    Out.ar([0,1], carrier);
  }).add;

  b = Bus.control(s, 1);

  OSCdef(\sensor1, {|msg|
    b.set(msg[1].linlin(0,1,220,440));
  }, '/raspberrypi/0/0');

  s.sync;

  a = Synth(\fm);
  a.map(\freq, b);
}
```

Notice how we enclosed the whole program inside a *fork* function. Fork is a convenience method to create a routine, so that we can then control the timing of the enclosed function by suspending it for a determined time (similarly to a delay function in C). This procedure is important to SuperCollider because we must make sure that the SynthDef is received and processed by scserver before trying to instantiate a synth. In the routine, we could wait for a determined time using the *wait* method, but this would not guarantee that the SynthDef had indeed compiled. Instead, the scsynth has a convenience *sync* method (notice the server is initially assigned to global variable *s*), which tells the routine to wait for an acknowledgment from the SuperCollider server that all asynchronous commands have been completed. The routine may then continue after receiving this signal.

After adding the SynthDef, creating a control bus and creating the OSCdef function to update the bus value, we can finally instantiate our synthesizer and use the *map* method to bind the parameter *freq* to the control bus.

5.10.3 One-to-Many and Many-to-One Mappings

SuperCollider's buses are also convenient for more advanced types of mapping. Buses can have multiple listeners (synths or their mapped parameters) to create one-to-many mappings. We could use the same OSCdef to update two different control buses and then map them to different parameters of our synth (notice the different scaling operations).

```
b = Bus.control(s, 1);
c = Bus.control(s, 1);

OSCdef(\sensor1, {|msg|
   b.set(msg[1].linlin(0,1,220,440));
   c.set(msg[1].lincurve(0,1,0,1,5));
}, '/raspberrypi/0/0');

a.map(\freq, b);
a.map(\modAmp, c);
```

An example of a many-to-one mapping could involve two input values that create a third composite value. In the following example, the value of the second sensor (address /0/1) is used to modify the output range of the first sensor, in turn mapped to the frequency control of the synth. A solution would be to use the bus *get* method (asynchronous) to extract the latest value of bus c and use it in the callback function that updates bus b. Notice the use of the round method to quantize the multiplier.

```
b = Bus.control(s, 1);
c = Bus.control(s, 1);

OSCdef(\sensor2, {|msg|
  c.set(msg[1].linlin(0,1,0,1,5).round);
}, '/raspberrypi/0/1');

OSCdef(\sensor1, {|msg|
  c.get({ |value|
    b.set(msg[1].linlin(0,1,220,440*value));
  });
}, '/raspberrypi/0/0');
```

5.10.4 Using Sensor Signals as Unit Generators

The previous examples assume that pbridge is receiving the sensor values from the Teensy and relaying the data to sclang, but it is also possible to set pbridge to send the data directly to the SuperCollider server. By setting the appropriate start-up flag in pbridge, its OSC commands will instead use the SuperCollider server port, which will receive data directly on control buses. Buses can also be accessed as if they were unit generators using the *In* object. In the following example we use bus number 100 to control the frequency of a low-pass filter.

```
SynthDef(\filter, {
  var source, filter;

  source = PinkNoise.ar();
  filter = LPF.ar(source, In.kr(100).linlin(0,1,20,5000));

  Out.ar([0,1], filter);
}).add;


   a = Synth(\filter);
```

Bus 100 corresponds to the first sensor on the first multiplexer. SuperCollider can assign any number of arbitrary control buses, but in Prynth we have established a convention to facilitate addressing. The target bus number is derived from the following formula:

```
bus = (multiplexer x 10 + sensor) + 100
```

This formula concatenates the multiplexer number with the sensor number and offsets the value by 100. Note that indexes start at 0. To provide another example, sensor 5 on multiplexer 7 would be equivalent to bus 164 (6x10+4+100).

The interesting approach to this programming style is that control data can be understood as DSP signals and used in unconventional processing chains that include feedback delays, exotic chaotic noises or non-linear filters.

5.10.5 File Access from SuperCollider Programs

In Prynth we also created conventions to access program and audio files. The absolute paths to these directories are the following:

```
~supercolliderFiles = "/home/pi/prynth/server/public/supercolliderfiles/";
~soundFiles = "/home/pi/prynth/server/public/soundfiles/";
```

The variables containing the path strings are then used to access any file listed on the SuperCollider and sound file managers of the Prynth client application. Accessing other SuperCollider programs is particularly useful because it allows the division of larger programs into multiple files. A SuperCollider subprogram can be executed from within a parent program using the *load* method.

```
s.waitForBoot{
   ~supercolliderFiles = "/home/pi/prynth/server/public/supercolliderfiles/";
   (~supercolliderFiles++"chime.scd").load;
};
```

Above is the default SuperCollider program that runs when Prynth is first started. It triggers the server booting and waits until the process is complete to then execute the chime.scd file, a different SuperCollider program that plays a feedback sound.

Audio files have many uses in sound synthesis. The simplest example would be to load an audio file to memory using the *Buffer* object and then play it using *PlayBuf*, which receives three parameters: number of channels, identification number of the buffer and a looping flag.

```
fork{
   ~soundFiles = "/home/pi/prynth/server/public/soundfiles/";
   ~buffer = Buffer.read(s, ~soundFiles++'a11wlk01.wav');

   s.sync;

   play{PlayBuf.ar(1, ~buffer.bufnum, loop:1)}
}
```

5.11 Distribution and Support Materials

5.11.1 Licensing

Prynth is entirely free and open-source. The accompanying license is a Creative Commons License Attribution-ShareAlike (CC BY-SA), allowing for free use, modification and even commercialization. The only restrictions are that credit must be given to the original developers and any modification should fall under the same license. This type of license guarantees that Prynth remains freely accessible and that users who decide to improve it maintain the same spirit of knowledge sharing.

5.11.2 Distribution

Prynth's source code is hosted at GitHub (Franco, Venkatesan, et al. 2018), an online service that stores files using the version control system Git. Git facilitates collaboration between programmers, who can push and pull different versions of the source code. New functionalities are implemented in side copies called branches and then merged with the main version once they are fully functional and tested. Github is one of the most popular online repositories for source code because it is free for open-source projects and offers valuable features, such as statistics, bug report systems and the hosting of a project website.

The Prynth end user distribution is in the form of a single compressed image file, ready for the Raspberry Pi's SD Card. Image files are large binaries, easily surpassing the file sizes permitted by GitHub (meant for the storage of small source code files). Prynth's image size is close to 1 GB, so it is instead hosted at IDMIL's servers, with the download URL referenced from the project's homepage. Downloading Prynth does not require any type of user registration.

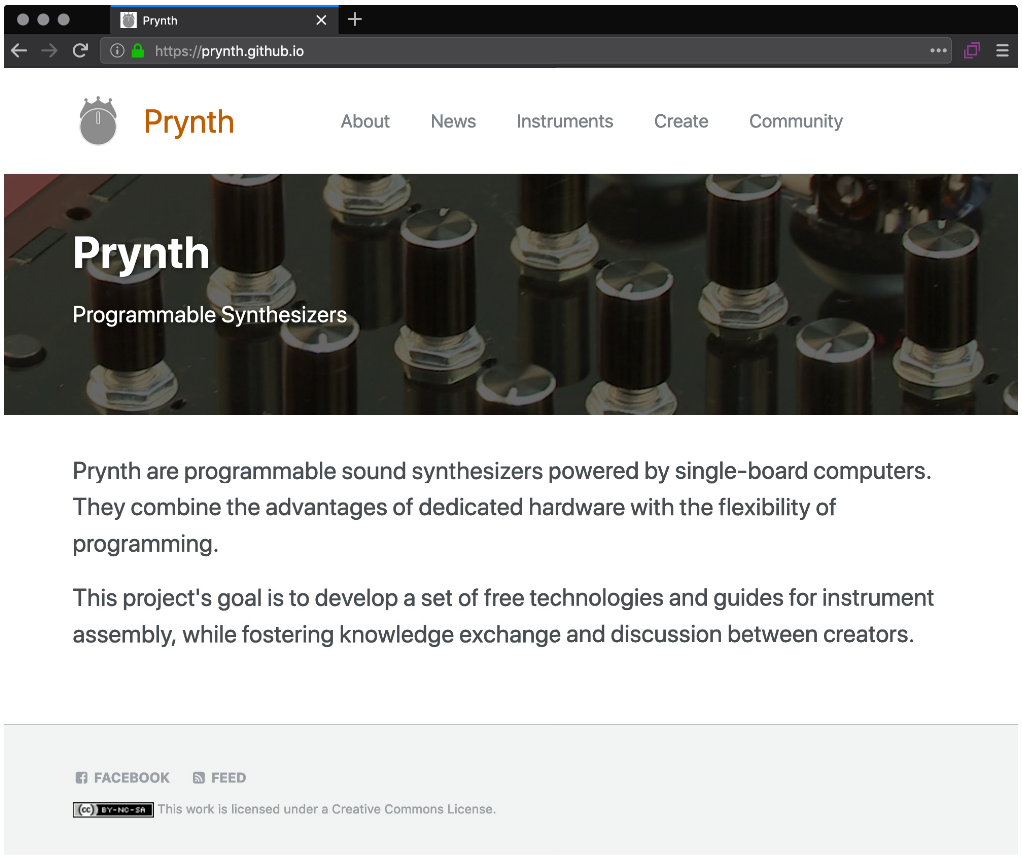
5.11.3 Website

The website is the main hub of the Prynth project. It is organized into the following sections: About, News, Instruments, Create and Community.

"About" is an introductory text, that briefly explains the project to newcomers. The "News" section is used to publish RDF site summary (RSS) feeds that advertise new releases, bug fixes and public initiatives. The "Instruments" section is a gallery of instruments built with Prynth, including images and a short description written by its makers. It illustrates the possibilities of the framework by showcasing practical examples.

The “Create” section is where users can find all the information to create their own instruments. It includes a general description of the framework, a downloads page and a documentation section. The “Downloads” page includes links not only to the Linux image file but also the PCB design files and the Teensy source code.

The “Community” section links to an online forum hosted at Google groups, requiring an authentication via a valid Google account. We have observed that most users prefer to use social media for public interaction and email for private messaging. Therefore the forum tends to be mainly dedicated to bug reports or technical inquiries that should be archived and available for future consultation.



Prynth are programmable sound synthesizers powered by single-board computers. They combine the advantages of dedicated hardware with the flexibility of programming.

This project's goal is to develop a set of free technologies and guides for instrument assembly, while fostering knowledge exchange and discussion between creators.

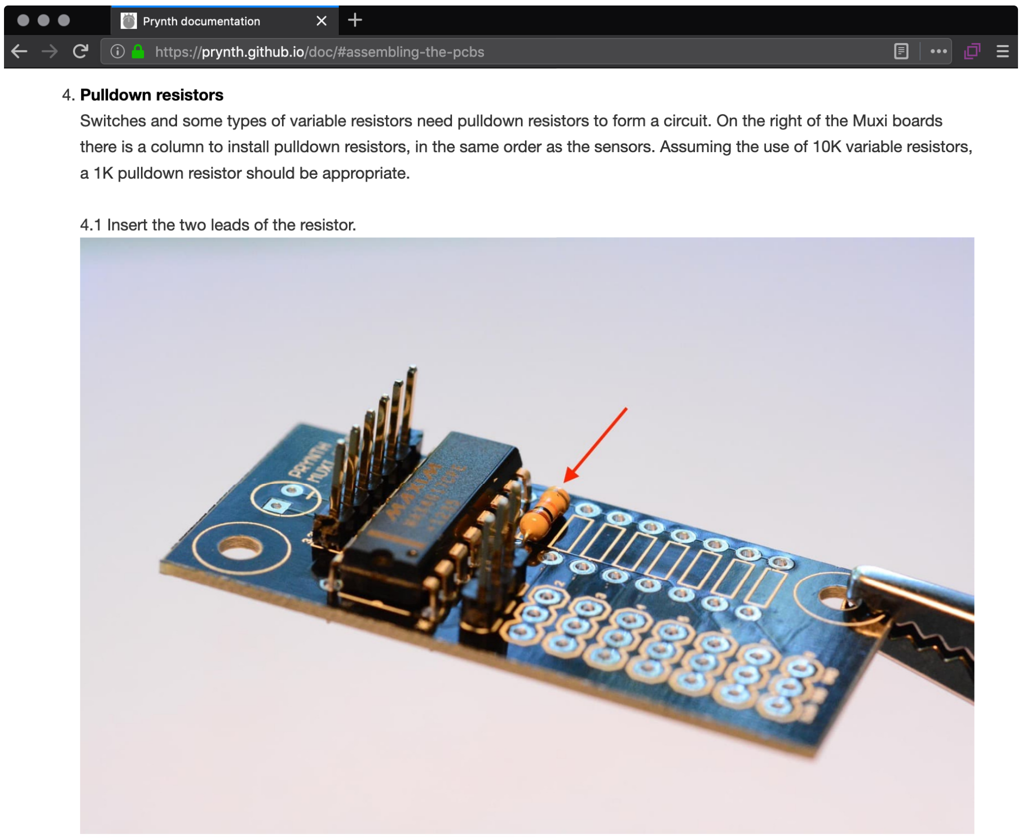
This work is licensed under a Creative Commons License.

Fig. 5.11 Prynth website.

5.11.4 Technical Documentation

Most of Prynth's documentation is in the form of *cookbooks* with step-by-step instructions. They teach new users how to assemble the base system, connect different types of sensors and program their first synthesis and mapping examples. All the instructions are accompanied by high-resolution macro photography to visually illustrate delicate operations, such as the positioning and soldering of electronic components. Many of these operations also include notes about particular obstacles or alternative methods.

Some users mentioned that they were able to easily assemble new systems by closely following these strongly descriptive and visually-oriented materials and that the cookbook format was fundamental to overcome any lack of confidence in their technical skills.



4. **Pulldown resistors**
Switches and some types of variable resistors need pulldown resistors to form a circuit. On the right of the Muxi boards there is a column to install pulldown resistors, in the same order as the sensors. Assuming the use of 10K variable resistors, a 1K pulldown resistor should be appropriate.

4.1 Insert the two leads of the resistor.

Fig. 5.12 Online documentation.

5.12 Conclusions

Prynth is a complete hardware and software solution for the development of new embedded DMIs. To create this framework we have established a series of requirements that reflect the potential advantages of embedded architectures, while maintaining the essential characteristics and features of past DMI models.

We have adopted the Raspberry Pi, the most popular single-board computer, complementing it with a dedicated signal acquisition solution, based on the integration of a supplementary microcontroller. The subsystem allows for the connection of up to 80 analog sensors and 48 digital sensors, and also performs onboard data conditioning. This solution guarantees a predictable and jitter-free signal acquisition, while freeing the single-board computer for audio DSP tasks and other interactive behaviors.

A Prynth instrument is configured and programmed using a web front end application, served by the instrument itself. It can be accessed from any thin client on the same network, without the need for a dedicated computer or any other sort of host software.

User-level programming is done through an embedded JavaScript text editor, using the SuperCollider programming language, one of the most powerful and mature solutions for computer music. The front end application also includes several control panels and side applications, including the configuration of the acquisition subsystem or the creation of custom GUIs. File managers allow for the storage of multiple programs, sound files and GUIs that can be saved and recalled at any time. The Prynth server is extremely lightweight and its idle load almost negligible due to the asynchronous processing of Node.js, which minimizes any impact on DSP processes.

Prynth aims at delivering a solution useful for both intermediate and expert users. It is especially designed to be easy to assemble at home and takes care of many of the technical aspects of building an embedded DMI. It still requires some level of electronics and programming proficiency, but we believe that it offers a good compromise between complexity and flexibility. Expert DMI developers could probably bypass Prynth and implement their own systems from the ground up, but we have received reports of significantly advanced use cases, leading us to believe that Prynth also represents a good value for this class of users.

Prynth is freely accessible and accompanied by in-depth documentation, such as cookbooks with step-by-step assembly instructions and examples of synthesis and mapping applications. Technical troubleshooting is done through a public forum, where users can exchange information first hand. Some users have explicitly reported their satisfaction with Prynth's ease-of-use and its comprehensive documentation, referring to how it helped them overcome insecurities in technical knowledge, leading to an efficient and trouble-free implementation.

Chapter 6

Results

6.1 Dissemination

In this section we present Prynth's dissemination activities, which include promotional events, press articles, and participations in conferences and art festivals.

6.1.1 Social Media

Social media has become an important and almost inevitable tool for communication on the Internet. Prynth's social media presence is done through a Facebook page (Franco 2018a), to which users can subscribe to receive direct notifications about new content. Facebook is also organized in interest groups, so it becomes easy to redirect any new content to these groups, some of which are exactly specialized in the construction of synthesizers.

At the time of writing Prynth has about 500 active Facebook followers, half of which subscribed to our page during the initial launch in November of 2016. From then on, this number has been growing slowly but steadily, with about 10 new subscriptions per month but with no signs of deceleration.

6.1.2 Public Talks

The Prynth framework was presented at the Voltage Connect Conference of 2017 (Berklee College of Music 2018), organized by the Berklee College of Music and themed “Pioneers, Players, Thinkers, Purveyors”. This conference was particularly interesting to us because is was more in touch with performers and industry than academic conferences like NIME. Speakers included people like Dave Smith (Sequential LLC 2018), Amos Gaynes (Moog Music 2018), Mark Ethier (iZotope Inc. 2018), Ernst Nathorst-Böös (Propellerhead Software 2018), Mate Galic (Native Instruments 2018), Jordan Rudess (Dream Theater 2018) and Suzanne Ciani (Suzanne Ciani 2018). Voltage Connect allowed us to showcase Prynth and receive feedback from the professional music industry. Everyone we talked to seemed to agree that SoC-based platforms, that make products like mobile phones, tablets and even the Raspberry Pi possible, need to be seriously considered because they may represent the future of computer-based musical instruments.

We also participated in the conference eNTERFACE’17 (*eNTERFACE’17* 2017), the International Summer Workshop on Multimodal Interfaces, which took place at the Centre of Digital Creativity (CCD) of Escola das Artes at the Universidade Católica Portuguesa. eNTERFACE is an informal academic gathering for researchers working on multimedia interaction design. The agenda is constituted by a series of intensive workshops and keynotes that happen over the course of one month. On the third week we did a keynote on Prynth and exchanged ideas with this group of HCI researchers. The topics of discussion centered around the design of new types of interactive musical devices and how they may fit a more conceptual outlook on the future of tangible interaction.

6.1.3 Specialized Media

One aspiration we had with this project was to be able to reach outside academia. For the public release of Prynth we sent information to online media publications related to electronic music, with the hope that they would find the project interesting and help with its dissemination. This information included a small descriptive text about Prynth and a link to a video with a demonstration of The Mitt, the first instrument built using our framework (see section 6.2.1). Prynth got a significant attention from these media outlets, which resulted in dedicated articles in online publications like FACT Magazine (Wilson 2018), Hackaday (Williams 2016) and Synthtopia (Synthtopia 2016). In the first weeks after publishing, our promotional video (Franco 2018b) got about 12,000 views and has now grown to about 20,000 views.

The interaction with these popular publications allowed us not only to reach a wider audience, but also to observe how their interests intersect Prynth.

FACT Magazine (Wilson 2018) is one of the most prestigious online music publications in the UK. It focuses on electronic music but extends to various other topics like fashion trends, youth culture and art. FACT magazine also tries to examine innovation in music and to stimulate a discussion around its cultural impact.

On the other hand, Hackaday (Williams 2016) is a website that showcases DIY projects, many of them related to electronics and computing. The discussions found in Hackaday's articles tend to be more focused on technical aspects, appealing to all those who like to play with technology. Still, some of these discussions often divert to interesting parallel topics of the Maker movement, such as sustainability or ethics. Hackaday's writer Elliot Williams had the following to say about Prynth:

> "The Most Flexible Synthesizer is DIY, Raspberry Pi ... Ivan Franco sent us this great synthesizer project that he's working on. Or maybe it's more like a synthesizer meta-project: a synthesizer construction set ... The system is brand new, but check out the Mitt synth device that Ivan already made, with a video embedded below. It's a good overview of one of the possible hardware configurations, and a sweet demo of a couple SuperCollider routines ... With the computer inside the box - the Raspberry Pi in question - you can easily make this system into a standalone musical instrument, without tethering it to your laptop. Ivan is trying to create an interesting and easy-to-use empty vessel for you to pour your hardware and software synthesizer dreams into. Everything is open source, and aside from the Pi and the price of potentiometers,

> this is a dirt-cheap device. You're not going to be limited by SuperCollider, and we're guessing that you're not going to be limited by 80 inputs either. What's the craziest sound-maker you can think of " (Williams 2016)?

Comments from Hackaday readers included:

> "Wow, Awesome! Best audio project that I have seen in a long long time and there is so much that must have gone into software development".

> "Awesome! This synthesizer really redefined the concept of 'writing' a song"!

Synthtopia is a smaller niche website, dedicated to showcasing new synthesizers to a target audience of expert electronic musicians. Discussions tend to center around new devices on the market and how they can serve these musicians. In essence, Synthtopia exposes Prynth to a group of knowledgeable and strong critics. One reader wrote the following about Prynth:

> "I've been a die-hard hardware guy since I started doing electronic music in the early 1990's, meaning I am always a bit skeptic about this sort of devices, but man, I have to say that this concept has wings to fly and I'm all in"!

These and many other opinions or interactions we had through these communication channels allowed us to create a broader understanding of how different user profiles perceive the value proposition of our framework. This marketing action was crucial for the dissemination of the project and the feedback we got gave us the confidence to believe that Prynth represents a valuable asset to a greater audience and an important tool for makers of embedded DMIs.

6.1.4 Events

There was a number of other assorted events in which Prynth marked its presence. In 2017 Marcelo Wanderley challenged his class of students to build an instrument with Prynth. Because time was short, we decided to have students build a replica of the Mitt. They were given all the necessary bare materials and about a month later we had 10 perfectly working instruments. The students reported having no difficulty whatsoever in completing the task. One student even added his own modifications, by including an inclinometer and mapping the Mitt to the Ableton Live DAW.

Prynth was also presented in two interest group meetings in Montreal. The first was Club Framboise (Club Framboise 2018), a gathering of people dedicated to showcasing all types of DIY projects that use the Raspberry Pi. This was an eclectic group, since there were many engineers of all ages, but all with little understanding about electronic music or the world of DMI creation. They all responded with great enthusiasm because for them Prynth represented a creative application well outside their usual engineering projects in fields like automation, robotics and networking. The second event was the SuperCollider Montreal user group meeting. This group of people gathers every month in Montreal for informal knowledge sharing about all kinds of music and art projects using SuperCollider. It includes a significant number of experts that represent both a major target user group and potentially the hardest critics of Prynth. It was very positive to come to the conclusion that the group shared the same interest for embedded platforms and considered Prynth to be a very complete and thoughtful implementation of SuperCollider on those platforms. Some people in the group are also contributors to SuperCollider's core development. From their own initiative, they created a contact between the development teams of SuperCollider and Prynth, which are currently engaged in discussions about better integration strategies between the two technologies.

In the summer of 2018, Prynth was also presented at the 25th edition of the Sónar Festival (Advanced Music S.L. 2018), which happens every year in Barcelona, Spain. Sónar is probably among the top ten electronic music festivals of the world, with 2018 setting the record for 126,000 visitors. We participated on a committee of artists and researchers from Canada, curated by MusicMotion (MusicMotion 2018) and sponsored by the Canada Council for the Arts. MusicMotion is a Montreal organization focused on bridging researchers and artists interested in the exploration of technology and new media art. For three consecutive days we showcased Prynth and the Mitt to hundreds of people that visited our booth. This promotional action resulted in the possibility of engaging firsthand with the public, which proved to be an extremely demanding endeavor but also a highly rewarding one.

A similar showcase happened in September of 2018, at Rythmopolis (*Rythmopolis* 2018), which took place at the Place des Festivals in the Quartier des Spectacles, Montreal. Rythmopolis is a large-scale immersive event that includes percussion, acrobatics, dance and new technology. It was curated by Fabrice Marandola and organized in collaboration between Le Vivier (Le Vivier 2018) and CIRMMT (CIRMMT 2018).

Our work was also selected to participate in many other events, such as concerts, art exhibitions and synthesizer trade shows, some of which we were unable to attend considering the scope and financial constraints of the project.

6.1.5 Conclusions

Our dissemination activities have included academic contributions in the form of publications and presentations in conferences, but also the participation in public events related to electronic music and special interest groups. Prynth was also covered in reputable press articles and social media.

We believe in the value of offering our framework to a larger community of artists and instrument developers, something that can only happen with a significant effort in good documentation, quality control and promotional activities. The result was the fast growth of a community of users that recognizes value in the technologies, concepts and technical implementation behind Prynth.

6.2 Instruments Built with Prynth

The construction of a DMI involves not only technical but also creative challenges. With an infrastructure like Prynth and the flexibility of SuperCollider, one could build many kinds of instruments, from handheld to tabletop instruments, small processing pedals or large size sound art installations. Paraphrasing Williams, the writer from Hackaday, the only limit would be one's imagination.

We asked our users to share their experiences with Prynth, by documenting and showing their creations, and reporting bugs or any other difficulties in our public discussion channels. Although we know that most users will prefer not to engage in public discussion, there is still a relevant number that do contribute. We are grateful to those people because their feedback is crucial to our research.

In this section we present a small selection of instruments built with Prynth. One of those instruments was built by the author but the remainder by other users, a few of them colleagues at McGill University. Our pool for this selection was somewhat limited because we needed direct access to the creators. We also had to make sure that there was good quality documentation for each case, leaving behind many other interesting projects that unfortunately do not. Still, we think this selection fulfills the goal of exposing the diversity of instruments made possible by Prynth, as well as the different motivations and artistic visions behind them.

6.2.1 The Mitt

Hardware Description

The Mitt is a tabletop instrument played using micro hand gestures. It uses 5 small thumb joysticks, spatially distributed to follow the morphology of the hand, so that each joystick can naturally rest under one of the five fingers. Below the joysticks there is a series of 5 buttons that follow the same spatial distribution pattern and that are accessed by slightly retracting the fingers to a more inward position. Above the joysticks, there is a matrix of controls with 4 rows and 5 columns. The first row and column are populated with buttons and the remainder with rotary potentiometers. On the lower corner of the device there are two additional buttons. The most common way to play the Mitt is to use one hand to control the joysticks, while the other is free to manipulate synthesis parameters on the control matrix.

The controls are mounted on the top panel of the instrument's quadrangular case, which has a height of 20 cm and a top surface area of 60 x 90 cm. On the backside of the device there is a series of connectors: stereo 1/4" TRS audio outputs, a USB 2.0 type B power connector, Ethernet and a 5-pin DIN MIDI output.

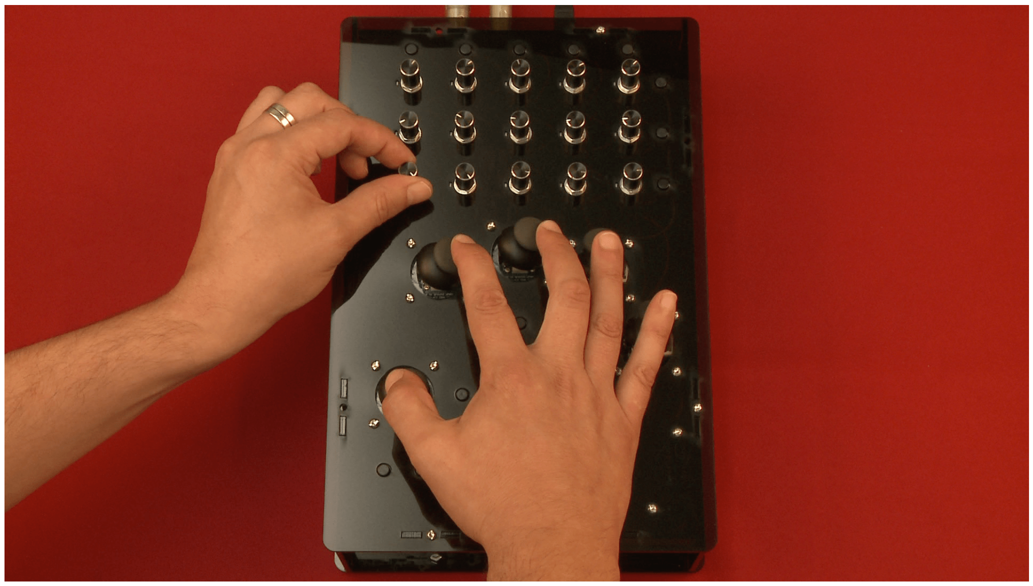


Fig. 6.1 The Mitt.

The thumb joysticks on the Mitt are appropriate for capturing micro-gestures. They are found in many game controllers, an application that requires precise tracking of small finger movements. This type of micro analog joysticks has been in the market since the '80s and they all follow the same basic design. The vertical shaft of the joystick travels inside the crossing of two internal slotted shafts. Moving the joystick up or down will move the horizontal shaft, while moving it left and right will move the vertical shaft. These shafts transform the joystick's angular motion into a combination of the rotation of these two perpendicular shafts, which in turn are connected to two analog rotary potentiometers. Because these are analog joysticks, they can be sampled at any desired speed and resolution. The vertical shaft of the joystick measures 25 mm and bends 45 degrees end-to-end. With the 12-bit ADC resolution of the Teensy, we have a theoretical precision of 0.01 degrees, which can also be expressed as an arc travel of about 0.005 mm. These joysticks also have an inner spring that once released automatically retracts the shaft to a central position. This resting position is equivalent to half the travel of each potentiometer, so there is no central dead spot, a problem commonly found in other types of joystick designs.

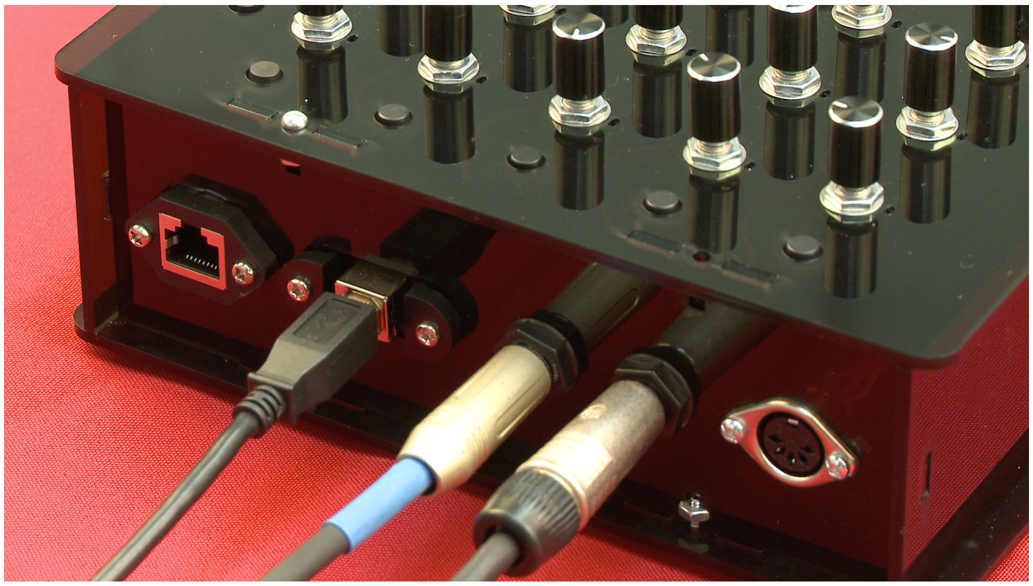


Fig. 6.2 Backside connectors of the Mitt.

Synthesis and Mapping

We tested many different synthesis and mapping schemes with the Mitt, but our preferred was a 5-channel granular synth, each voice combined with a low-pass filter and controlled by simple one-to-one mappings. The corresponding SuperCollider SynthDef is instantiated for each of the 5 channels. It incorporates both the stereo granular synthesizer and the low-pass filter and provides the following control parameters:

- Trigger gate
- Playback rate (pitch)
- Relative position of the play head
- Grain triggering rate
- Grain duration
- Sample number
- Filter cutoff frequency

The mapping is also similar for each of the 5 channels. First we transform the cartesian coordinates of each joystick into polar coordinates. We then map each joystick, so that pushing it away from the central position triggers the gate of the amplitude envelope, while the angle controls the position of the playhead. A circular motion of the joystick will result in a scrubbing effect, which can be used to modulate timbre according to the variations in the original sample. Moving the joystick back to a central position will trigger the release stage of the envelope. The threshold of the envelope trigger corresponds to a very small radius, so that the remainder of the amplitude can continue to be used to modulate the sustain level. To increase the dynamics of the resulting granular textures, we've added a small random jitter to the playback position using interpolated noise.

The buttons next to the joysticks change the sample in the buffer, cycling through an array of pre-chosen sound files for each channel. The buttons on the top of the vertical strips are used to lock the joystick position, so that a particular channel may sustain indefinitely, even if the joystick is physically released. The first potentiometer from the top is mapped to density, a meta-parameter that, when increased, reduces the grain duration but increases the grain trigger rate and the frequency of the playhead modulating noise. A high density tends to produce more static timbres, while a low density produces more variation. The second potentiometer controls pitch, with the note values converted to the corresponding playback rates. These pitches can be quantized to a particular scale and over any determined number of octaves. Finally, the third potentiometer controls the cutoff frequency of the low-pass filter, affecting the brightness of the sound by reducing its high frequency content.

Musicality

One interesting aspect of the Mitt is the exploration of micro-gestural control. The performer's gestures are amplified to the extreme, so that a very small movement of a finger can have a dramatic impact on the resulting sound. This results in a very dynamic instrument, where the performer can easily shift from an extremely controlled and subtle modulation to a major variation in timbre and amplitude.

For our live performances, we use a pair of Mitts, one for each hand. The left Mitt is the exact mirror image of the right one. The only difference is that we've removed the springs from the left unit, so that the joysticks can rest in their set positions without the need to lock the envelopes with the upper buttons.

Our performances could be broadly categorized as glitch music, with sustaining drones played with the left Mitt and more percussive sounds played with the right.

Concerts

We have performed the Mitt in several concerts, both solo and in ensembles with other musicians.

- Solstice 17—Solstice 17 was the closing event for the 2017 edition of the Montreal festival Printemps Numérique (Printemps Numérique 2018). Organized in collaboration with MusicMotion and the Société des Arts Technologiques (SAT 2018), this multidisciplinar event included interactive installations, theater and music. We participated on an ensemble, together with the Musallica (Musallica 2018) soprano quartet and percussionist Lionel Kizaba (Kizaba 2018), to create a structured musical improvisation for the performance of the interactive theater group Prism (PRISM 2018).

- Beat Dox: D+M—this event was part of the Doc Circuit Montréal (RIDM 2018), a film documentary festival that takes place yearly at the Cinémathèque Québécoise in Montreal. We were invited, together with two other artists, to perform live music for the screening of the documentary "La Pesca", by directors Pablo Alvarez Mesa and Fernando López Escrivá. The documentary was played three times in a row, with each performer offering their own live musical interpretation. For this concert we prepared an algorithm for the Mitt to play music based on the principles of serialism and inspired by the minimal music of composers like Steve Reich and Phillip Glass. The performance consisted on conducting the algorithm, which autonomously generated pitch, rhythm and articulation. This was a good exercise of creating a rule-based performance for the Mitt, rather than the skill-based performances that we tend to prefer (see section 2.4). After the concert we were asked to participate in an artist talk to explain the Mitt and how the music had been created. We were congratulated by several of the audience members that praised the project and the initiative of giving others the possibility of building similar instruments.

- Improv@CIRMMT—these concert cycles dedicated to musical improvisation take place at the Café Ressonance in Montreal and are organized by Olfer Peltz and CIRMMT. In 2017 we constituted an improvisational trio, together with Eric Lewis on trumpet and Juan Sebastian Delgado on cello. The Mitt was used to create a soundscape on top of which Lewis and Delgado could develop a tonal dialog between their instruments.

- DIY Mash Up—part of the live@CIRMMT programming of 2018, DIY Mash Up was a concert dedicated to musicians that develop their own instruments. We participated together with OrkID, IDMIL's own DMI ensemble, featuring also Ajin Tom, Harish Venkatesan, Travis West and Tiffany Matraxia. The event also counted with the special participation of Jean-François Laporte, an accomplished electronic musician from Québec, who joined in improvisations with the various ensembles of the evening.

Guthman Musical Instrument Competition

In 2017 the Mitt was selected for the final stages of the prestigious Margaret Guthman New Musical Instrument Competition (Guthman Musical Instrument Competition 2018), created by Georgia Tech alumni Richard Guthman in honor of his wife. The first editions of this competition were dedicated to piano performance, but in 2009 it changed its focus to showcasing the best new instruments created worldwide. Every year, during two days, 20 semi-finalists present their creations and perform live for audiences and judges, who have included experts such as Pat Metheny, Laurie Anderson, David Wessel, Tod Machover, David Zicarelli or Tom Oberheim. Although the Mitt was not selected for the grand finals of the competition, it was still recognized as one of the top instruments of 2017, which we consider to be a great honor and achievement.

6.2.2 Noiseboxes

Noiseboxes (Sullivan 2015) are a series of portable handheld digital musical instruments designed by John Sullivan. The first versions of these instruments were originally built using plain PD on the Raspberry Pi, but the author later moved to Prynth for the development of the following iterations. The most interesting aspect of these instruments is the inclusion of batteries and speakers, making the Noiseboxes completely autonomous and playable anywhere and at any time.

> "The stand-alone format means that there is no need for cables, connections or configuration of any kind. An internal rechargeable battery and onboard speakers makes them ready to play whenever and wherever inspiration strikes," said Sullivan.

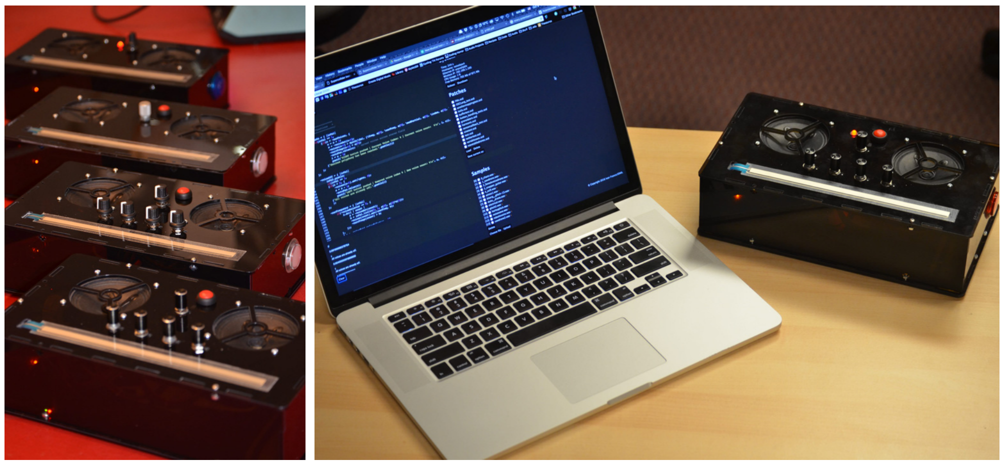


Fig. 6.3 The Noiseboxes.

The Noiseboxes use an FM synthesis engine with 8 voices to create simple harmonic tones or chaotic noises that can be sustained and layered on top of each other. The controls of the instrument include buttons, knobs and a positional sensor that can be mapped to voice selection, pitch and effect parameters. The instruments also include an accelerometer to modulate parameters by holding and tilting the whole instrument.

These instruments were used in a series of experiments that Sullivan conducted for his research on the factors that influence the adoption of DMIs. In these experiments, several units of the Noiseboxes were lent to musicians, who could take them home, learn to play them and later report their findings.

6.2.3 SONIO Soundboard

SONIO Soundboard is a musical interactive table developed by Tim Kong, Karen Cancino, Alanna Ho and Marc St. Pierre. It is composed of 52 capacitive touch sensors, distributed in a grid pattern along the surface of a table. The contact points are spaced close enough so that a young child may trigger 4 to 6 simultaneous points with the palm of the hand. Each point triggers a different FM synthesis voice and the table is large enough to accommodate up to 4 players. It also includes visual feedback, in the form of RGB led strips embedded along the sides of the table.

> "Participants described the board as having a 'futuristic' feel, where the overall theme, sound and visuals were inspired by science fiction movies like Star Wars and Close Encounters of the Third Kind", said the authors.

The SONIO Soundboard is an interesting example of how Prynth can be used in interactive devices for contexts other than live performance. It was developed for exploratory edutainment, inviting the general public to an interactive experience that lets them discover electronic music in an engaging and playful way.

Marc St. Pierre further describes the public exhibition of the SONIO Soundboard:

> "SONIO was presented at two publicly accessible events over a period of a few months. The first was titled InFlux which was hosted at the Surrey Art Gallery in Surrey, BC. There was probably a couple hundred people who attended the event. InFlux highlighted local community groups and artists by providing a venue to showcase their creative work. This was an annual event in its third year.

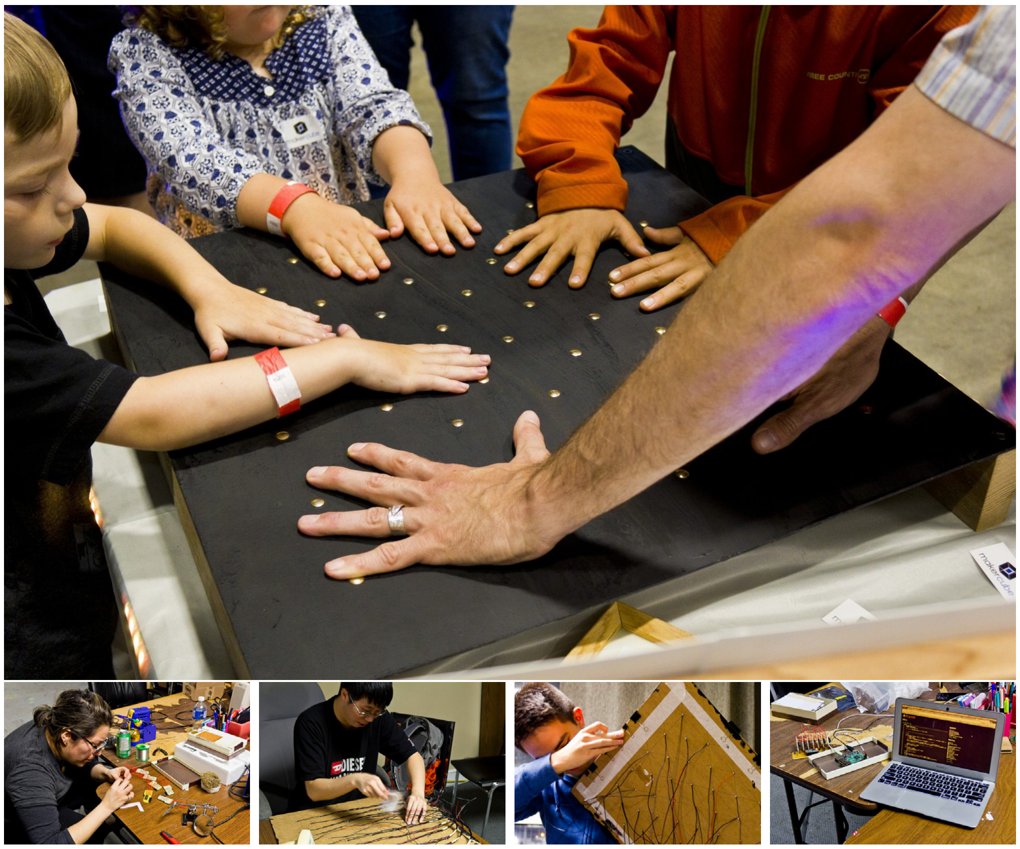


Fig. 6.4 SONIO Soundboard and its construction.

> The second event was the Vancouver Mini Maker Faire 2017, in which several thousand people were estimated to have attended. The event ran over 3 days and SONIO was a central part of our display. It was popular with both kids and adults, some of whom returned several times, bringing new people with them to try out the installation".

Interactive devices for public entertainment have different requirements than those designed for personal use. They must be extremely resilient to withstand the abuse of hundreds of people that are not aware or concerned with the sometimes fragile nature of these implementations. Most of the times they are also simpler but they still cannot malfunction or have any sort of inconsistent behavior. Otherwise they will quickly fall into discredit given the short attention span of the public in these contexts. With the SONIO Soundboard, Prynth has proven to be a reliable and stable solution for these demanding use cases.

Another interesting aspect of the development of the SONIO Soundboard is that the development team adapted the Prynth PCB designs into a format that better suited their application. The PCBs were handmade using perfboards, in order to slot the Muxi boards directly onto the Control board. We encourage this type of modification of Prynth's initial design so it can better adapt to specific use cases.

6.2.4 Modular Synths

There are two interesting applications of Prynth in modular synthesizers. The first is called RR, developed by Daniel Górny. It is a tabletop semi-modular, where Prynth's signal acquisition system was modified to allow input of analog control voltages. The instrument has 15 potentiometers, 5 control voltage inputs and a 7 x 7 patching matrix, modeled after rePatcher (Open Music Labs Wiki 2018), a similar system but for the Arduino platform. Górny told us that he plans to continue the development of RR by creating an OSCdef that responds to CV inputs, a preset system for recalling the potentiometer values and adding a potentiometer to continuously transition between said presets.

More recently, another user has sent us a report of using Prynth together with Terminal Tedium (Mxmxmx 2018), an open-source eurorack module based on the Raspberry Pi. He described his experience with Prynth and Terminal Tedium in detail:

> "I'm very impressed with the Prynth project and having a lot of fun learning Supercollider with it. I'm running Prynth on a eurorack module called Terminal Tedium.
>
> The Terminal Tedium has stereo audio codec in/out, several digital inputs and outputs, six pots and three switches. It hosts a Raspberry Pi as its compute engine—I'm using a Pi 3. The module was intended for running Pure Data but I've had good results with Csound and now Prynth. I started with the Prynth 0.5 image and ran the Terminal Tedium install script which configures the WM8731 audio codec driver and

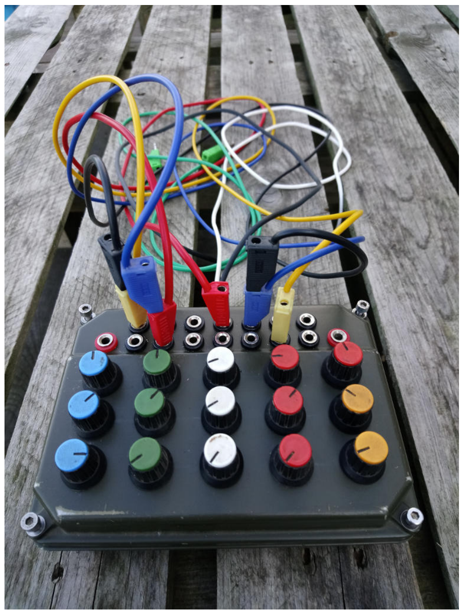


Fig. 6.5 RR semi-modular.

installs Pure Data (which I don't use). Prynth found the WM8731 sound device and required no other configuration. The Terminal Tedium install includes source code for a basic OSC client which I modified a bit to make it work better with Prynth. I added a command to /etc/rc.local to start the OSC client on boot up.

This setup works really well. I can read the digital inputs and the pots via OSCdefs. So far I've programmed a couple of simple voltage controlled oscillators, a drum machine, reverb and echo effects and a very nifty looped sample player. It's by far the most flexible module in my eurorack setup. My thanks to Ivan and the team for creating such a great project"!

6.2.5 FlexPrynth

FlexPrynth is an instrument built by Harish Venkatesan and Ajin Tom, two of Prynth's collaborators. The motivation for the construction of this DMI was that Venkatesan and Tom wanted to assess the feasibility of taking an existing instrument, the Sponge by Martin Marier (Marier

2018), and reconstruct it introducing newer technologies and features to the original design. Their interest lies in understanding how to preserve and recreate DMIs, since most of these instruments are not available in the long-term. On his website, Marier provides a set of instructions on how to recreate the original Sponge. It seems only natural that the public would take this design and reinterpret it, creating new versions that slightly deviate from the original design, especially if the original materials are no longer available. This happens oftentimes with DMIs. Instruments like the T-Stick (Malloch and Wanderley 2007) or Michel Waisvisz's the Hands (Torre et al. 2016) have had many different versions with improved technologies and new functionalities. Venkatesan and Tom also wanted to test if an embedded system and a ready-to-use framework like Prynth would effectively facilitate the development and usability of this DMI, as opposed to building it from scratch and using the traditional DMI model of a controller and computer.

The original Sponge is a controller made of a flexible foam block. It is played by bending, twisting and striking its malleable structure. The mechanical deformation is captured by a pair of accelerometers slotted inside each extremity of the foam block. On the underside, a series of button clusters are used to trigger different notes and states of the instrument. The control data is sent via RF transceivers to a computer running SuperCollider.

FlexPrynth uses the same principles but introduces pressure sensors inside the foam. Pressure sensitivity adds squeezing as a new musical gesture, one that is naturally favored by the sponge. In one design, a small wood popsicle stick works as a lever that sits on top of two pressure sensors. Pushing it upwards or downwards has an effect similar to that of a relative position slider.

Two new versions of the Sponge were built. The first was a controller, closer to Marier's architecture, and the second was a self-contained version using Prynth. After the construction of the two versions, the creators referred to the much easier implementation using a higher-level framework like Prynth:

> "The roots of the challenges [of version 1] can be traced majorly to the fact that there were many sub-systems that were independent of each other with little coherency [*sic*], forcing the developer to spend great amounts of effort in building concrete links

> between them ... The Prynth framework helped in overcoming a lot of hassle faced in building version 1".

They also referred to the ease-of-use and immediacy of a self-contained instrument versus its controller/computer combo counterpart:

> "The setup time [of version 2] is far less compared to version 1 ... version 2 is a self-contained DMI and it requires little to no set up time to get it running. One would only have to plug in the power supply and wait for the system to boot to performance state. On the other hand, the setup for version 1 involves setting up Bluetooth communication, loading mappings on libmapper, setting up a DAW with virtual instruments and routing MIDI and audio buses, which is time consuming".

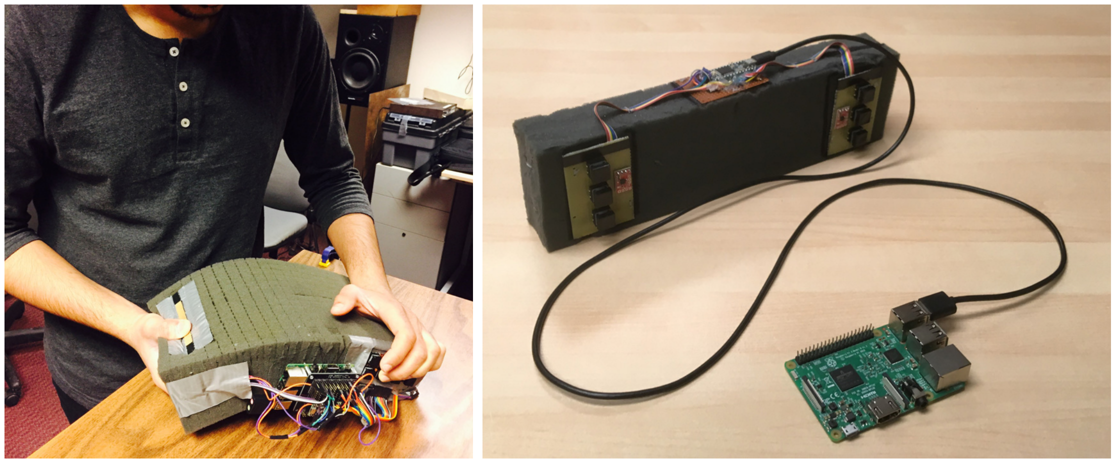


Fig. 6.6 FlexPrynth versions 2 and 3.

On the other hand, a self-contained design also brought challenges to an instrument like the Sponge. Venkatesan and Tom talk about the large size of the Prynth hardware module when compared to the smaller microcontroller of version 1.

> "This module, in comparison with that of the version 1, is bulky which means that in order to accommodate the Prynth hardware on the body of the instrument itself, the foam had to be considerably larger than version 1".

The larger Prynth hardware creates some limitations to FlexPrynth. It cannot be embedded directly inside the instrument and restricts the malleability of the foam. For these reasons, a third version was built, in which the Teensy was decoupled from the Raspberry Pi and connected with a cable. The Raspberry Pi can then be mounted around the waist, a solution similar to STEIM's SensorLab used by Waisvisz in the Hands.

6.2.6 GuitarAMI

GuitarAMI (Meneses et al. 2018), developed by Eduardo Meneses, is an example of an augmented musical instrument (AMI) using Prynth. The concept behind instrument augmentation is to take an existing instrument, such as a guitar or a violin, and use an electronic system to extend its palette of gestures and sonic possibilities. Technology-wise, an AMI works very similarly to a DMI: sensors are applied to the instrument's body and connected to a computer to control real-time DSP software. Augmented instruments use microphones (a form of sensor) to capture the acoustic sound of the instrument and apply many different effects to the source material.

The early versions of GuitarAMI were composed of a sensor module, a controller in the form of a guitar pedal, an audio interface and the computer. The sensor module is a small box with an accelerometer and a proximity sensor, which is mounted on the body of the guitar and connected to the pedal controller with a cable (later made wireless in a subsequent version). The guitar controller has several foot switches and an LCD. Inside, an Arduino performs signal acquisition of the sensors and sends the data to the computer, which runs PD with DSP like spectral freezing, FM modulation, looping and time-stretching. In some version of GuitarAMI, a conventional effects processor was added for more common effects like distortion, delay, reverb and wah-wah.

Using Prynth, the new version of the GuitarAMI discards the computer and the external audio interface, to have all of these functionalities integrated into a single pedal unit. All of the PD patches were successfully ported to SuperCollider, achieving the same functionality of previous versions in a much more integrated and robust package.

Another interesting derivative of the GuitarAMI project is that the processing unit (the pedal)

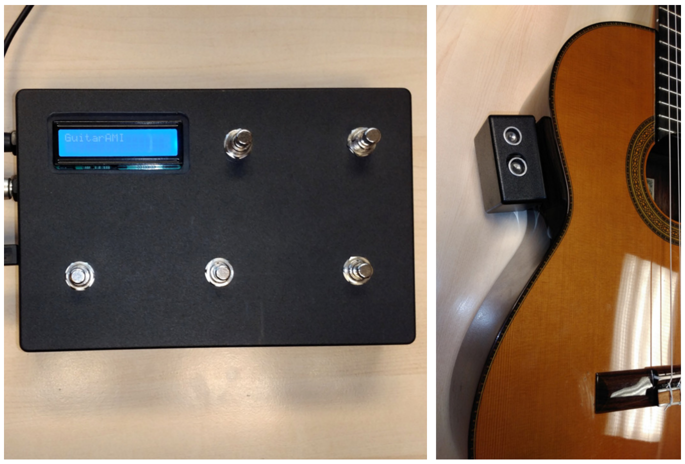


Fig. 6.7 The new GuitarAMI using Prynth.

was developed so that it can receive wireless signals from many other controllers. As an example, Meneses created a program to receive messages from the T-Stick (Malloch and Wanderley 2007), one of IDMIL's most emblematic controllers. With this general purpose processing unit, IDMIL's controllers can be demonstrated at any time, without the need for dedicated computer systems or elaborate setups.

6.2.7 Conclusions

In this chapter we have described some examples of instruments made with Prynth. We consider these to be our most important results, from which we can learn how the framework is being used and the different approaches of developers towards the design of their own embedded DMIs.

The Mitt is concentrated on exploring new forms of musical control, through its emphasis on the micro-gestures. The Noiseboxes differentiate themselves by focusing on the full autonomy of the instrument and the ability to play it anytime and anywhere. The SONIO Sounboard aims at being a multiuser social experience, manifested in the form of a hybrid between an instrument and an interactive installation, designed to be played by anyone. RR, GuitarAMI and FlexPrynth use more traditional models but augment their usability through a more integrated design.

A development framework like Prynth works as a white canvas for DMI development. It allows users to create instruments that reflect their personal interests and artistic goals. We have gathered a group of examples that reflects this diversity, concluding that Prynth responds well to a wide spectrum of use cases.

Chapter 7

Discussion

7.1 A New Generation of DMIs

The model for embedded DMIs proposed in Prynth is influenced by three major factors. The first is a cultural factor, which relates to the increasing presence of computers in our everyday lives and the sophistication of a new generation of users. The second is a conceptual factor, which draws from the understanding that future computers will assume many different forms so they can better fit specialized activities. The third is a technical factor, which relates to the new technologies that enable embedded DMIs.

7.1.1 The Cultural Push

Computers keep getting more sophisticated but so do their users. The computer training offered in schools a few years ago was limited to the use of basic software, such as word processors or spreadsheets. Today, youngsters are taught how to program computers, so they can use them more efficiently to solve complicated tasks. Programming has become an essential skill to any engineer or scientist, but even software for accounting or visual art is starting to include scripting capabilities and offer further control to power users. We are seeing the same exact tendency with electronics. Components are freely available to the public and custom circuit boards can be manufactured in low quantities. We are continuously expanding the meaning of being a technology user.

This deeper engagement with technology is also present in music applications. They offer increasing levels of control and allow for deep customization. For centuries, musicians have invented new playing techniques or created personalized setups to produce the sound palettes that became staples to their artistic entities. It is no different with computers. There is a growing interest in the creation of custom instruments for live performance of computer music.

Audio programming tools like Max have been around for a long time now, but their user base is still growing. The integration of Max with the Ableton Live DAW (Ableton 2018) is the proof that musicians also want to tap into these advanced capabilities and that they are invested in learning them. We think that the audience for a framework like Prynth is not as small as one might think. Some of our users reported that Prynth got them started in instrument building and the learning of SuperCollider. The construction of custom DMIs is on the cusp of becoming a common practice, so there is a clear need for more frameworks like Prynth.

7.1.2 The Conceptual Push

From a conceptual perspective, embedded DMIs could represent the norm for future electronic music instruments. In the same way that mobile phones gained more flexibility and intelligent behavior from the inclusion of computers, so should electronic music instruments. Embedded DMIs have the possibility of embodying the notion of smart musical objects.

The concept of smart objects can be traced back to the original ideas of the forefathers of ubiquitous and physical computing. In his 1991 article, "The Computer for the 21st Century" (Weiser 1991), Mark Weiser alludes to the future possibilities of smaller contextually-aware computers, with designs optimized for specialized tasks. Weiser's vision has now become a reality. We interact daily with many more computing devices: smartphones, tablets, watches, biological monitors and smart appliances. They all have different form factors and fulfill specialized functions. We also interact very differently with these computers. We touch them, talk to them and can even interact via emotional expressions or physiological signals. Software tracks our daily patterns and dynamically adapts to our preferences and context of use. Weiser set out to rethink the role of computers in quotidian life and soon realized they would become essential to our future society. The problem was that computers would have to change too.

> "The computer nonetheless remains largely in a world of its own. It is approachable only through complex jargon that has nothing to do with the tasks for which people use computers ... We are therefore trying to conceive a new way of thinking about computers, one that takes into account the human world and allows the computers themselves to vanish into the background. Such disappearance is a fundamental consequence not of technology but of human psychology. Whenever people learn something sufficiently well, they cease to be aware of it ... Only when things disappear in this way are we freed to use them without thinking and so to focus beyond them on new goals" (Weiser 1991).

This vision aligns with our concept for embedded DMIs. They are interactive objects geared specifically towards the activity of music making. The computer that was once required disappears from the desk and is now incorporated into the instrument itself. Embedded DMIs constitute a sort of new hybrid category between a hardware synthesizer and a computer. They are completely self-contained but can still be programmed with different DSP or interactive behaviors.

7.1.3 The Technical Push

Until recently only desktop and laptop computers with x86 processors had enough computing power for DMIs. This is no longer true. In the last years, embedded processors evolved to a point where they defy traditional computers in raw processing power. The latest iPad tablets can run applications that have the same functionality of the professional software used in musical productions of a decade ago. The mobile market is flooded with music applications that simulate acoustic instruments, pedal boards, synthesizers and sequencers. Musicians have gladly embraced mobile computing devices, which are simpler to operate and can run the most complex audio tasks.

Traditional computers are also defied by the increasing amount of applications that use cloud computing, an architecture where all of the heavy processing is done using remote servers. Some cloud computing products use specialized client applications but the tendency is to use a simple web browser. A good example of cloud computing is Google Docs, an office suite that resembles the previously indispensable Microsoft Office, but now working entirely on the cloud. Many other applications, from CAD software to videoconferencing have also moved to cloud computing. One of the advantages of this model is that there is no need for a powerful local computer or the use of specialized software. Almost any device with a web browser can be used to interact with these applications.

In this scenario traditional computers become obsolete and Prynth follows these same technological trends. It takes advantage of the new generation of embedded processors and conducts all computer-like interactions via thin client.

7.1.4 Conclusions

Prynth is the result of the alignment of groundbreaking cultural, conceptual and technological factors. We have a new generation of users that is considerably more tech-savvy and ready to accept the technological trends that will shape the way we understand computers and their quotidian use.

Artists have always been early adopters of technology because it allows them to keep pushing forward new forms of expression. Likewise, since the inception of personal computers, musicians have continuously found new ways of using them creatively. The use of DMIs is stronger than ever and so is the pursuit for more advanced solutions.

7.2 Overcoming the Limitations of Traditional DMIs

Any seasoned DMI performer would recognize the limitations of DMIs identified on chapter 3. Fortunately, embedded DMIs might hold solutions to solving many of them.

7.2.1 Embodiment

First we talked about the lack of embodiment of DMIs and how it derives from the fragmented nature of the controller/computer combo. Many people still prefer hardware synthesizers over computers because they are designed specifically for music. Hardware synthesizers behave like all the other devices of the music studio: they have a single function but can also be combined with other devices in a myriad of ways. This direct association between musical tasks and the objects to accomplish them seems to be of extreme importance to musicians. The computer offers virtual simulations of these interactions, but on top of an abstract and general-purpose machine that does not have any tangible affordances directly related to music. On the other hand, embedded DMIs have the possibility of becoming integral interactive objects, promoting dedicated functionality and a more direct correlation between task and tool, while maintaining many of the traits and advantages of computers in music.

7.2.2 Cognitive Activities

The computer is a rule-based system with a significant number of procedures and abstractions. It requires functional reasoning, which starts with a clear goal and ends in the series of complex chained operations required to accomplish it. Most DMIs cannot circumvent these cognitive activities, which are very different from the embodied interaction with a musical instrument. The computer is still at the heart of the DMI, even if only for the user.

Prynth still requires some computer-like activities but they are much more focused on the functional operation of a DMI. There is no direct interaction with an operating system and musical thinking is put in the forefront, by adopting a domain-specific language like SuperCollider and offering an easy path to the integration of physical and virtual control. In any case, these activities are only required during the development and testing of an instrument. In daily usage, it becomes predominantly a skill-based interactive device.

7.2.3 Immediacy

The computer lacks the determinism that many people expect from a dependable musical instrument: a tool that can be picked up at any time and immediately used for musical expression with consistent results. Although not completely instantaneous, Prynth instruments take only a few seconds to become operable, without the need for extraneous connections or intensive user intervention. In our many years of experience with DMIs, we have never had any instrument with an easier or faster setup time. In our concerts with the Mitt we were proud to finish our setups much earlier than our colleagues, while they dabbled in layers of technical issues and unexpected behaviors. This property of immediacy may seem like a small feature to those who have learned to live with the complexity of their DMI setups, but once experienced it considerably changes expectations.

7.2.4 Reliability and Longevity

Finally, there is little doubt that embedded computing could be key to better resilience, stability and durability of DMIs. Only a test of time can truly confirm it but it is relatively safe to speculate that a computer dedicated to a single task, with no moving parts and shielded from harsh environments will be sturdier and exhibit a more consistent and enduring performance.

We can at least confirm some level of reliability in Prynth, based on our experience with the instruments we built during the past three years. Besides two sensors that had to be substituted in a particular unit of the Mitt due to normal wear and tear, we have not observed any other noticeable issues. If anything, the weakest point of the single-board computers used in Prynth is the memory card. It has a limited number of read/write cycles and the operating system can become corrupted if the power supply is lost during critical disk operations. At the same time, and assuming frequent backups, the memory card is also the easiest and cheapest component to replace.

7.2.5 Conclusions

The use of standard computers in DMIs creates problems that stem from the purely technological to those related to design and usability. Embedded computing might help solve some of those problems. Prynth employs several strategies to make DMIs behave in ways that are closer to standalone instruments, which are still preferred by many musicians because they are more immediate, reliable and focused on musical activities.

But to say that embedded DMIs completely solve these problems would constitute a bold claim; one we cannot entirely prove at this point. We would need more in-depth psychology and usability studies to better understand the relationships between musicians and their instruments. Only then can we confidently refine our models and frameworks. Still, we recognize signs that we are in the right path and that embedded DMIs are a glimpse into a possible solution.

7.3 Strengths and Weaknesses of Prynth

Software and hardware development frameworks facilitate the implementation of technological products but they also impose their own workflows and rules of operation. They are necessarily full of compromises. Therefore, it is relevant to analyze the implications of those trade-offs and how they translate into strengths and weaknesses of Prynth.

7.3.1 Strengths

Target user level

In most of the products and frameworks we reviewed, highly flexible tools tend to be more complex. On the other hand, they have lesser constraints and are applicable to a wider variety of use cases. In Prynth we tried to find an optimal balance between flexibility and complexity, but we must not forget that building a musical instrument is in itself an ambitious proposition.

Still, we made every effort to reduce complexity. Prynth requires only basic soldering and offers an easy way to implement tangible interfaces. Its software is distributed as a ready-to-use Linux system, completely preconfigured and easy to install on a memory card. After the base system assemblage, which does not involve any other low-level implementation or direct interaction with Linux, the only remaining task is to connect sensors and program musical functions in SuperCollider.

If we were to establish the profile of a Prynth user, we would say that it is at least essential to have some basic experience with soldering and programming. That said, Prynth is free, uses relatively cheap components and a well-known programming language with many learning resources. These factors might be enticing enough for novices that want to give their first steps into these disciplines. Like previously referred, some Prynth users have reported success in building their first ever DMIs. We have also received reports of several seasoned DMI developers that still found great value in Prynth, so we trust the right balance between complexity and flexibility to be one of its strengths.

Accessibility

In Prynth we have consciously refrained from using any exotic electronic components or creating procedures that could not be easily followed at home. Sometimes even expert users have difficulties soldering surface mount components or simply do not have access to the right tools. Prynth uses only through-hole components that can be assembled using basic hand tools. These components are also easily acquired in any hobbyist electronics store and they have a very low probability of ever being discontinued by manufacturers. While the base system could have a much more advanced and compact architecture, the success of Prynth depends on this level of self-sustainability. Furthermore, we do not wish to sell components or preassembled electronics boards and transform Prynth into a business. All the contrary. By making sure that components are simple and easily sourced, we can better guarantee user adoption and a continued improvement in a logic of open-source tools and knowledge.

Modular architecture and standards

We have also made an effort to decouple many of the components that constitute Prynth. Instead of a monolithic application, Prynth is composed of many small applications that intercommunicate. Sometimes modular architectures have lower performance than highly integrated ones, but we trust that in our case there is a big advantage to this decentralization strategy. Some of the instruments built with Prynth show that by having a modular architecture, the user can decide to intersect communication protocols or substitute modules that better serve a particular implementation. This is the case with instruments like the SONIO Soundboard and FlexPrynth, in which it was important to modify functions of the signal acquisition subsystem. Instruments like the Terminal Tedium eurorack module have taken the opposite path, by discarding our electronics and using only the Prynth server and its integration with SuperCollider and Linux. The heavy reliance on standard protocols for internal messaging (OSC and HTTP) also opens up the possibility for building custom control layers. This level of modularity is beneficial for more experienced users, who can modify Prynth to better fit their goals.

Raspberry Pi

In the same way we have chosen to work with standard electronics components, we have also chosen the Raspberry Pi because it is the most widely available and popular SBC in the market. There are many other types of single-board computers, some of which with better specifications. The problem is that most of them share the risk of sudden discontinuation or lack of availability in the market. We previously referred that in the beginning of our research we used the BeagleBone Black. We decided to switch to the Raspberry Pi after the announcement that Texas Instruments had no intention of continuing to invest in the development of the SOCs used in the BeagleBone Black and that it would eventually abandon production. These news raised all the necessary red flags for us to move to the Raspberry Pi, which has since seen three processor updates, while the BeagleBone remained on the same processors it has used for several years. While we still have no control over the future of the Raspberry Pi, we at least know that it has better chances of a long-term availability. In the case it does become unavailable, we are still capable of easily porting Prynth to other SBCs, due to our low dependence on any distinctive features of the Raspberry Pi.

SuperCollider

SuperCollider is the ideal programming language for Prynth. It is a proven and mature language, designed specifically for music and with a vast DSP library. Most importantly, SuperCollider is designed for interactive programming, a feature we have preserved in Prynth and that differentiates it from all other frameworks for embedded DMIs. Prynth allows the programming of the instrument to be as interactive as playing it. Mapping and synthesis programs can be changed on-the-fly, without the need for compilation stages, which greatly facilitates modification and fine-tuning. A Prynth DMI could even be live coded, an unexpected feature in a self-contained instrument.

Web technologies

Prynth is also tightly integrated with modern web technologies, allowing the instrument to be programmed from any standard web browser. Most other systems use local applications to program them. Those applications rarely run on multiple platforms and can quickly become obsolete, rendering the DMI unusable. They also require installation on a dedicated personal computer, which the performer will need to access at all times. In essence, any proprietary software will increase the dependence on a particular computer. On the other hand, Prynth can be programmed and configured from any available computer, tablet or smartphone.

7.3.2 Weaknesses

Performance

Although the computing power of ARM processors is catching up with x86 processors, the latter still exhibit better performance. We have proven that ARM processors can run relatively complex audio synthesis (Franco and Wanderley 2015) in the context of DMIs, but it is comprehensible that a user migrating from regular computers could stumble into limitations that were not a preoccupation before. In those cases, DSP algorithms may need to be simplified, reducing the number of possible voices or post-processing effects. Latency may also increase with larger vector sizes, to compensate for the reduction in processing power. Maybe in the near future, with the fast-paced progress of ARM processors, these differences will become less pronounced, but for now they still exist.

Audio quality

Audio quality may face similar limitations. The audio cards available to systems like Prynth are generally of lower quality than the ones available for desktop and laptop computers. Many of them have lower signal-to-noise ratios, reduced spectral bandwidth, smaller bit depths and sometimes poor pre-amplification stages. Even if consumers were willing to pay higher prices for quality audio cards for the Raspberry Pi, other limitations would still come into play, such as power consumption and dimensions. Compromises in audio quality are acceptable to some users but problematic to others. In either case, the computer market still has many more options than those available to single-board computers.

User familiarity

Although we argue that standard computers are detrimental for the user experience of a DMI, we must still take into account that they have been extensively used in music for many years now. There are already many strongly established methods and tools. Like previously mentioned, Prynth imposes its own set of procedures, which might not fit the workflows of all DMI users. Someone who is highly invested in applications or programming languages that only run on regular computers might be less likely to migrate to Prynth.

Headless operation

Regular computers rely strongly on visual feedback. Dataflow programming languages like Max or PD are good examples of systems that have a great number of adepts but that strictly depend on visual information. Those types of representations might be paramount to some users. The success of dataflow programming languages among musicians might be due to the fact that they use an abstraction that is familiar to musicians: the connection of cables between functional objects. Users that are very accustomed to such types of representations might feel discouraged or lost when dealing with a system that incites textual programming and a headless operation.

7.3.3 Conclusions

Prynth aims at serving many use cases and finding a good balance between complexity and flexibility. It is designed to be easily assembled using basic tools and has a modular architecture that incites modification. It integrates modern technologies, like the Raspberry Pi and Node.js, and couples them with SuperCollider, one of the most mature and feature-rich languages for computer music. Prynth also has its share of disadvantages. ARM-based SBCs still can't match the performance and audio quality of regular computers. Prynth also proposes its own workflow for DMIs, which might represent a barrier considering the diversity of possible methodologies and technical approaches that can be used in DMIs.

7.4 Comparison with the State of the Art

In the state of the art (section 4.3), we have enumerated several DMI systems that use embedded computing in one way or another. It is difficult to compare Prynth to many of them, given such diversity in objectives, target audiences and technical implementations. That said, it is impossible to avoid a comparison between Prynth, Bela and Satellite CCRMA. Both Bela and Prynth are spiritual successors to Satellite CCMRA, which was the first system for the construction of SBC-based DMIs to become publicly available. As an early effort Satellite CCRMA was a simpler system, but it still doesn't have the level of integration and advanced features that Bela and Prynth have today.

At NIME 2015, when we first presented our results regarding the evaluation of the synthesis performance of the BeagleBone Black (Franco and Wanderley 2015), the developers of Bela gave their first workshop. At the time, Bela was not very different from Satellite CCRMA. In order to configure and program it, the user had to go through a complex sequence of command line operations and the transpilation of PD patches to C++ source code. At that time Prynth was already in development and we decided to continue it, believing we could offer a system with better usability: one where users could avoid the complexity of Linux and C++ (which we still believe to be excessive for DMI development). In the end, Bela and Prynth were developed and released within the exact same timeframes. The curiosity is that both systems wound up with very similar features. Naturally they have their share of differences, but the main concept is still the same. Both use a single-board computer with add-on circuitry and can be programmed using a web-based IDE.

Without dwelling too much into direct comparisons between the two systems, Bela has a significant advantage in the low latencies it exhibits, due to the tight integration with the programmable real-time unit of the BeagleBone Black. While this is certainly a very notable achievement, we believe that the dependence on the BeagleBone represents a threat to Bela. The BeagleBone will quickly become obsolete and it will be challenging to find an alternative SBC with a similar programmable real-time unit.

The biggest advantage of Prynth over Bela is interactive programming. Although Bela can also use SuperCollider programs, it is not capable of evaluating only portions of code and reproduce the user experience of programming in SuperCollider. Bela still requires a traditional compilation phase. After many years using all types of audio programming software, we are of the opinion that the richness of the interactive programming in SuperCollider is yet to be well reproduced by any other language. The best candidates are ChucK (Wang et al. 2003) and the more recent Pyo (Bélanger 2018), but the development of the first seems to have decelerated in the past couple years and the latter is still somewhat in its infancy. On the other hand, SuperCollider has a strong following, and being able to reproduce its stronger feature on a web-based front end of a DMI is still one of Prynth's most distinctive features.

The fact that Bela and Prynth have similar objectives and solutions, despite being developed in parallel by two different teams, cross-validates the work of both and leads us to the conclusion that there is a common understanding of the advantages and features that are important to any framework for embedded DMIs.

7.5 Future Work

Future work on embedded DMIs should fall into three major categories: Prynth improvements, users studies and trial of new interaction concepts.

7.5.1 Incremental Improvements

Prynth is an ongoing effort. There are still many aspects in which it can be incrementally improved. To use a sensor signal in the SuperCollider language, the user must create an OSC responder for each specific address. There is currently no way of specifying wildcards for hierarchical structures. If we hypothetically imagine two sensor signals represented by the addresses /coordinates/x and /coordinates/y, there is still no convenient way of writing only one OSCdef that responds to both (/coordinates/*), so that x and y could be handled together within the same function. From our understanding, this restriction is inherited directly from the OSC standard itself, but we are interested in studying ways of circumventing it, making hierarchy handling of OSC messages easier in Prynth.

There is also a chance for improvement of the Control board that hosts the Teensy microcontroller. The Teensy has breakout pins that are located on the underside of the microcontroller. Prynth uses some of those pins for I^2C and SPI communication. We have not found a definitive way of making these pins easier to solder. Currently it requires the flowing of solder through the pin holes (see section 5.6.1), which can significantly increase the difficulty of the assembly.

We have used this method successfully several times but it would be important to find a better strategy for novice users with less soldering experience.

We have chosen the Raspberry Pi 3 as the official SBC supported by Prynth, but there are other interesting offers like the Raspberry Pi Zero and the Raspberry Pi Compute Module. The Raspberry Pi Zero is a smaller SBC that discards some I/O components, such as the USB hub or the Ethernet connector. The Compute Module is an even more condensed version, with the same form factor of a DDR2 SODIMM and with just the BCM2837 SoC, RAM and an integrated embedded multi-media controller (eMMC) flash memory. The reduced footprint of these alternatives is beneficial in applications where the body of the instrument cannot accommodate the current dimensions of the Prynth base system. These other RPis even suggest that Prynth electronics could be reduced to a single integrated PCB, including onboard components like a DAC. We have knowledge of at least two Prynth users that have tried to adapt Prynth to work on the Raspberry Pi Zero, but those efforts were unsuccessful, apparently due to difficulties in the correct operation of third-party software components like JACK. In the future we would like to officially support these boards and offer the possibility of having Prynth work on even smaller and more integrated hardware solutions.

The documentation of Prynth is in the form of cookbooks to help the user assemble the base electronics system. Several users praised this format and were very satisfied with having a good starting point to their own developments. Others have reported that they would prefer instructions on how to build the entirety of an instrument like the Mitt. Initially we made a conscious decision not to offer such instructions because we wanted Prynth users to build their own instruments instead of simply replicating existing ones. Currently we are reconsidering it. If there were full instructions from start to end, novice users would have at least one example to help them assemble a fully operational instrument and eventually surpass any sense of disorientation and loss. Maybe there is still space for improvement in Prynth's documentation and training methodologies, so it can better serve novice users.

7.5.2 User Studies

We need to continue to improve our understanding of the motivations and habits of DMI users. We think there is still significant work to be done in this area. Only then can we continue to enhance frameworks like Prynth to better serve the community. At IDMIL, we have been collaborating with John Sullivan, whose work is focused on these exact research topics. Together we conducted a survey in which we inquired DMI users how they were using DMIs and in which contexts. The results of this survey are yet to be published but we can advance that we have confirmed that many DMI users have the interest and technical knowledge required to build their own instruments. We have also confirmed that there is a divide between those that prefer the computer due to its flexibility and others that prefer hardware devices due to their immediacy, tactility and robustness, confirming the core premises of Prynth.

User studies applied to DMIs are a difficult proposition. There is so much diversity that classification and analysis becomes a true challenge. At the same time, this same diversity is one of the most interesting aspects of DMIs. No matter how difficult, we are invested in continuing these avenues of research, so that in the future we can be even more assertive about the profiles of DMI users and develop research and tools that better answer their needs. DMI researchers concentrate heavily on groundbreaking technical achievements but they often tend to forget how that research must be applied outside academia to the most important actor: the musician.

7.5.3 New Conceptual Proposals

Paraphrasing Niebuhr, when we first started this research we aimed at the stars so we could reach the sky. We maintain our conviction of the benefits of completely removing traditional computer interaction models from at least some types of DMIs. At first glance this may seem like quite a discordant proposal. Even in Prynth, some kind of computer-like device is still necessary for programming the instrument, so how could we maintain the flexibility of programming with no keyboards and screens to write and display source code?

Maybe the answer lies in a more serious consideration about the possibilities of tangible programming, a concept explored in the work of researchers like McNerney (McNerney 2004), Horn (Horn et al. 2007) and Bowers (Bowers et al. 2006). The concept behind tangible programming is that physical objects can be spatially arranged to represent computing algorithms. In the work of McNerney and Horn, those objects are represented by Lego-like bricks that can be easily interconnected. Bowers has explored similar concepts in "Pin&Play&Perform", a musical application where physical widgets like buttons, sliders or potentiometers have pin-like connectors and can be freely attached to a planar surface to create ad hoc controllers. The widgets can also contain information and Bowers demonstrates how they could be used to dynamically build a Max patch. If tangible programming proved to be an effective method for creating relatively complex and rewarding DSP and mapping algorithms, then it could represent a new way of programming a DMI: one that could more closely resemble the physical interaction with hardware devices of the music studio.

There are also interesting possibilities for alternative programming methods in advanced input methods like natural language processing, recently made popular in products like Apple's Siri or Microsoft's Cortana. By coupling natural language with additional machine learning algorithms containing the principles behind DMI programming, the user could simply describe an algorithm to the machine, which in turn would be in charge of the assembly of the functional structures of the program.

These types of alternative programming methods could be the answer to more radical changes in the way we interact with DMIs. We think they would be truly revolutionary musical instruments.

Appendices

Appendix A

Prynth Technical Details

A.1 Description of Control Board

To follow this description of the Control board, please refer to figure 5.4b. The Teensy microcontroller is soldered directly on the board, in the appropriate location marked at the bottom center right of the board. To the right is the female 2x20 pin header that connects the board to the GPIO of the RPi. Above the Teensy, there is a first block of digital pins, which includes the I^2C bus on pins 29 and 30. The remainder pins 24–27 and 38–33 are available for general-purpose digital I/O. To the left of the Teensy there is a analog-to-digital conversion block, with 10 horizontally aligned ADC channels. They provide power (3.3V), ground (AGND) and accept analog signals on the SIG pin. To the right there are three more pins, labeled A, B, and C, which provide a 3-bit counter clock that controls the multiplexing daughterboards. To the far left we find another UART (pins 7 and 8) and the SPI bus (CS on pin 10, DOUT on 11, DIN on 12 and SCK on 13). Between these two buses there are 3 more general-purpose digital pins (5,6 and 9). On top of the analog-to-digital conversion section there is a MIDI output, also connected to Teensy's TX pin number 8. With the addition of a $47\,\Omega$ resistor, this output can be used to send all sensor signals to legacy MIDI devices.

A.2 Remapping of the Raspberry Pi UARTs

In this appendix we describe the steps to remap the Raspberry Pi's PL011 UART to the GPIO pins. The first step was to disable the debugging console on the mini UART, which can be done by editing the file */boot/command.txt* that holds boot parameters for the kernel. One of these parameters is *console=serial0,115200* which we deleted, resulting in the following base */boot/command.txt* file.

```
dwc_otg.lpm_enable=0 console=tty1 root=/dev/mmcblk0p2 rootfstype=ext4 elevator=deadline
fsck.repair=yes rootwait
```

Next, we mapped the Bluetooth module to work on the on top of the mini UART, instead of the PL011. This remapping is done by modifying another of the Raspberry Pi's start-up configuration files, located in */boot/config.txt*. Read at boot time by the loader, this file holds several options stored as key/value pairs. The key *dtoverlay* invokes the loading of *device tree overlays*. The *device tree* is a compiled file describing the peripherals and interfaces available to the SoC, which in turn automatically searches and loads the necessary kernel modules. A device tree overlay is a differential file that modifies the default device tree at boot time before the result is passed to the kernel. There are several preexisting device tree overlays and manufacturers of add-on boards for the Raspberry Pi can create and distribute new ones. To map the Bluetooth module to the mini UART, we have used the *pi3-miniuart-bt* device tree overlay, by adding the following to *config.txt*.

```
dtoverlay=pi3-miniuart-bt
```

After this procedure, the PL011 is free to be used in other applications and is automatically mapped to the GPIO header pins 8 for transmission (TX) and 10 reception (RX). In the case of Prynth, these pins connect to the first serial port of the Teensy (pins 0 and 1) via Control board.

A.3 Compiling SuperCollider for the Raspberry Pi

The compilation of SuperCollider for Prynth is relatively straightforward and mostly equivalent to the procedure for x86 processors, save for minor adjustments. These include disabling the CMake flags pertaining the use of the IDE, QT and any other x86-specific features, such as streaming SIMD extensions (SSE) acceleration. QT is the windowing solution for the IDE GUI and therefore irrelevant to Prynth.

```
cmake -L -DCMAKE_BUILD_TYPE="Release" -DBUILD_TESTING=OFF -DSSE=OFF -DSSE2=OFF
-DSUPERNOVA=OFF -DNATIVE=OFF -DSC_WII=OFF -DSC_IDE=OFF -DSC_QT=OFF -DSC_ED=OFF
-DSC_EL=OFF -DSC_VIM=OFF ..
```

At the time of writing, there is also a known bug in the resulting sclang binary, which will report 100% CPU usage even without any real processing load. This error is apparently related to an internal timer using the Boost library. To avoid this problem the SC_TerminalClient.cpp source file should be edited, substituting the mTimer cancel method by a conditional in function of the Boost error.

```
   //mTimer.cancel();
  if (error==boost::system::errc::success) {mTimer.cancel();} else {return;}
```

SuperCollider compiles user classes at start-up, so some of its native GUI classes will throw errors because we have have no GUI. To avoid these errors, one can simply move or delete the SuperCollider GUI classes from the class compilation path.

```
sudo mv /usr/local/share/SuperCollider/SCClassLibrary/Common/GUI
/usr/local/share/SuperCollider/SCClassLibrary/scide_scqt/GUI
sudo mv /usr/local/share/SuperCollider/SCClassLibrary/JITLib/GUI
```

```
/usr/local/share/SuperCollider/SCClassLibrary/scide_scqt/JITLibGUI
```

Bibliography

Ableton (2018). URL: `https://www.ableton.com/en/` (visited on 09/27/2018).

Advanced Music S.L. (2018). *Sónar Barcelona - Music, Creativity & Technology*. es. URL: `https://sonar.es/` (visited on 12/11/2018).

ALSA Project (2018). URL: `https://www.alsa-project.org/main/index.php/Main_Page` (visited on 04/11/2018).

Anode (2018). *MeeBlip*. URL: `https://meeblip.com/products/meeblip-anode-synth` (visited on 12/04/2018).

Antares (2018). URL: `https://www.antarestech.com/` (visited on 02/06/2018).

Arduino (2014). URL: `https://www.arduino.cc/` (visited on 11/22/2018).

Autodesk, Inc. (2018). *Instructables - How to make anything*. URL: `https://www.instructables.com/` (visited on 12/07/2018).

Avizienis, Rimas, Adrian Freed, Takahiko Suzuki, and David Wessel (2000). "Scalable connectivity processor for computer music performance systems". In: *Proceedings of International Computer Music Conference*. Berlin, pp. 523–526.

Balena (2018). *Etcher*. URL: `https://etcher.io` (visited on 05/16/2018).

Barbosa, Álvaro (2003). "Displaced soundscapes: a survey of network systems for music and sonic art creation". In: *Leonardo Music Journal* 13, pp. 53–59.

Barrass, Tim (2018). *Mozzi*. URL: `http://sensorium.github.io/Mozzi/` (visited on 11/22/2018).

Barrière, Jean-Baptiste, Pierre-François Baisneée, Adrian Freed, and Marie-Dominique Baudot (1989). "A digital signal multiprocessor and its musical application". In: *Proceedings of International Computer Music Conference*. Columbus, OH, pp. 17–20.

Bastl Instruments (2018). *Trinity*. URL: `http://www.bastl-instruments.com/instruments/trinity/` (visited on 01/30/2018).

BeagleBoard (2018). URL: `https://beagleboard.org/` (visited on 08/07/2018).

Bélanger, Olivier (2018). *Pyo*. URL: `http://ajaxsoundstudio.com/software/pyo/` (visited on 09/25/2018).

Berdahl, Edgar and Wendy Ju (2011). "Satellite CCRMA: A musical interaction and sound synthesis platform". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Oslo, pp. 173–178.

Berklee College of Music (2018). *Voltage Connect Conference*. URL: `https://www.berklee.edu/voltage-connect` (visited on 12/11/2018).

Bleep Labs (2018). *The Nebulophone*. URL: `http://bleeplabs.com/store/nebulophone/` (visited on 11/22/2018).

Bode, Harald (1984). “History of electronic sound modification”. In: *Journal of the Audio Engineering Society* 32.10, pp. 730–739.

Bottoni, Paolo et al. (2007). “Use of a dual-core DSP in a low-cost, touch-screen based musical instrument”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. New York City, pp. 394–395.

Bowers, John and Nicolas Villar (2006). “Creating ad hoc instruments with Pin&Play&Perform”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Paris, pp. 234–239.

Buxton, William (1977). “A composer’s introduction to computer music”. In: *Journal of New Music Research* 6.2, pp. 57–71.

— (1997). “Artists and the art of the luthier”. In: *ACM SIGGRAPH Computer Graphics* 31.1, pp. 10–11.

Buxton, William et al. (1978). “An overview of the structured sound synthesis project”. In: *Proceedings of International Computer Music Conference*. Illinois.

Cascone, Kim (2003). “Grain, sequence, system: three levels of reception in the performance of laptop music”. In: *Contemporary Music Review* 22.4, pp. 101–104.

Casiez, Géry, Nicolas Roussel, and Daniel Vogel (2012). “1 Euro filter: a simple speed-based low-pass filter for noisy input in interactive systems”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM Press, pp. 2527–2530.

Chadabe, Joel (1997). *Electric sound: the past and promise of electronic music*. Upper Saddle River, NJ: Prentice Hall.

Chowning, John M. (1973). “The synthesis of complex audio spectra by means of frequency modulation”. In: *Journal of the Audio Engineering Society* 21.7, pp. 526–534.

CIRMMT, Centre for Interdisciplinary Research in Music Media and Technology (2018). URL: `http://www.cirmmt.org/` (visited on 12/12/2018).

Clarke, Arthur C. (1982). *Profiles of the future : an inquiry into the limits of the possible*. 2nd rev. ed. London: Victor Gollancz.

Club Framboise (2018). URL: `https://www.facebook.com/clubframboise/` (visited on 12/11/2018).

CodeMirror (2018). URL: `https://codemirror.net/` (visited on 09/25/2018).

Collins, Karen (2008). *Game sound : an introduction to the history, theory, and practice of video game music and sound design*. Cambridge, MA: MIT Press.

Collins, Nick, Margaret Schedel, and Scott Wilson (2013). *Electronic music*. Cambridge, MA: Cambridge University Press.

Curtin, Steve (1994). “The SoundLab: a wearable computer music instrument”. In: *Proceedings of the International Computer Music Conference*. Aarhus, Denmark.

Cycling ’74 (2018). URL: `https://cycling74.com/` (visited on 11/23/2018).

Debian (2018). URL: `https://www.debian.org/` (visited on 05/23/2018).

DeSandro, David (2018). *Draggabilly*. URL: `https://draggabilly.desandro.com/` (visited on 09/05/2018).

Digilent Inc. (2018). URL: `https://store.digilentinc.com/` (visited on 11/22/2018).

Digital Art Conservation (2018). URL: `http://www.digitalartconservation.org/` (visited on 03/26/2018).

Dillon, Roberto (2015). "Ready: the commodore 64 and its architecture". In: Singapore: Springer Singapore, pp. 9–16.

DOCAM, Documentation and conservation of the media arts heritage (2018). URL: `http://www.docam.ca/` (visited on 03/26/2018).

Dream Theater (2018). *Official Website*. URL: `http://dreamtheater.net/` (visited on 12/11/2018).

Eckel, Gerhard, Francisco Iovino, and René Caussé (1995). "Sound synthesis by physical modelling with Modalys". In: *Proceedings of the International Symposium on Musical Acoustics*. Dourdan, France, pp. 479–482.

Edwards, Michael (2011). "Algorithmic composition: computational thinking in music". In: *Communications of the ACM* 54.7, pp. 58–67.

Eernisse, Matthew (2018). *EJS - Embedded JavaScript templates*. URL: `https://ejs.co/` (visited on 12/05/2018).

eNTERFACE'17 (2017). URL: `http://artes.ucp.pt/enterface17/` (visited on 12/11/2018).

Essl, Karlheinz and Julio d'Escrivan (2007). "Algorithmic composition". In: *The Cambridge Companion to Electronic Music*. Ed. by Nick Collins. Cambridge, MA: Cambridge University Press, pp. 107–125.

Fe Pi (2018). *Fe-Pi Audio Z V2*. URL: `https://fe-pi.com/products/fe-pi-audio-z-v2` (visited on 11/02/2018).

Fitzmaurice, George W. (1996). "Graspable user interfaces". PhD thesis. University of Toronto.

Flanagan, James L. and R. M. Golden (1966). "Phase vocoder". In: *Bell Labs Technical Journal* 45.9, pp. 1493–1509.

Flynn, Michael (1972). "Some computer organizations and their effectiveness". In: *IEEE Transactions on Computers* 100.9, pp. 948–960.

Franco, Ivan (2018a). *Prynth Website*. URL: `https://prynth.github.io/` (visited on 11/30/2018).

— (2018b). *The Mitt - YouTube*. URL: `https://www.youtube.com/watch?v=Oj7HfcdJhi8&t=8s` (visited on 11/30/2018).

Franco, Ivan, Harish Venkatesan, Ajin Tom, and Antoine Maiorca (2018). *Prynth GitHub*. original-date: 2016-11-07T17:58:27Z. URL: `https://github.com/prynth/prynth` (visited on 11/30/2018).

Franco, Ivan and Marcelo M. Wanderley (2015). "Practical evaluation of synthesis performance on the BeagleBone Black". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Baton Rouge, LA, pp. 223–226.

— (2016). "The Mitt: case study in the design of a self-contained digital music instrument". In: *Proceedings of the 12th International Symposium on Computer Music Multidisciplinary Research*. São Paulo.

— (2017). "Prynth: a framework for self-contained digital music instruments". In: *Bridging People and Sound*. Springer International Publishing, pp. 357–370.

Garrett, Jesse J. (2018). *Ajax: A New Approach to Web Applications*. URL: `//www.adaptivepath.org/ideas/ajax-new-approach-web-applications/` (visited on 12/05/2018).

Gaye, Lalya, Lars E. Holmquist, Frauke Behrendt, and Atau Tanaka (2006). "Mobile music technology: report on an emerging community". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Paris, pp. 22–25.

Gibbons, J. A., D. M. Howard, and A. M. Tyrrell (2005). “FPGA implementation of 1D wave equation for real-time audio synthesis”. In: *IEE Proceedings - Computers and Digital Techniques* 152.5, pp. 619–631.

Guthman Musical Instrument Competition (2018). URL: `http://www.guthman.gatech.edu/` (visited on 12/12/2018).

Hackaday.com (2018). URL: `https://hackaday.com/` (visited on 08/16/2018).

Hollinger, Avrum, Joseph Thibodeau, and Marcelo M. Wanderley (2010). “An embedded hardware platform for fungible interfaces”. In: *Proceedings of the International Computer Music Conference*. New York City, pp. 26–29.

Horn, Michael S. and Robert J. K. Jacob (2007). “Designing tangible programming languages for classroom use”. In: *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*. Baton Rouge, LA: ACM, pp. 159–162.

Hunt, Andy and Marcelo M. Wanderley (2002). “Mapping performer parameters to synthesis engines”. In: *Organised Sound* 7.02, pp. 97–108.

Hunt, Andy, Marcelo M. Wanderley, and Matthew Paradis (2003). “The importance of parameter mapping in electronic instrument design”. In: *Journal of New Music Research* 32.4, pp. 429–440.

Ishii, Hiroshi and Brygg Ullmer (1997). “Tangible bits: towards seamless interfaces between people, bits and atoms”. In: *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*. Atlanta, pp. 234–241.

iZotope Inc. (2018). en. URL: `https://www.izotope.com/` (visited on 12/11/2018).

JACK Audio Connection Kit (2018). URL: `http://jackaudio.org/` (visited on 07/05/2018).

jQuery (2018). URL: `https://jquery.com/` (visited on 07/12/2018).

Karplus, Kevin and Alex Strong (1983). “Digital synthesis of plucked-string and drum timbres”. In: *Computer Music Journal* 7.2, p. 43.

Kartadinata, Sukandar (2003). “The Gluiph: a nucleus for integrated instruments”. In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Montreal, pp. 180–183.

— (2006). “The Gluion advantages of an FPGA-based sensor interface”. In: *Proceedings for New Interfaces for Musical Expression Conference*. Paris, pp. 93–96.

Kizaba, Lionel (2018). *Kizaba - Musique électronique afro-house congolaise*. URL: `http://kizaba.ca/` (visited on 12/12/2018).

Koenig, Gottfried Michael (1983). “Aesthetic integration of computer-composed scores”. In: *Computer Music Journal* 7.4, p. 27.

Lansky, Paul (1990). “The architecture and musical logic of CMix”. In: *Proceedings of the International Computer Music Conference*. Glasgow.

Lazzarini, Victor, Steven Yi, John Fitch, et al. (2016). “Scripting Csound”. In: *Csound*. Springer International Publishing, pp. 195–205.

Lazzarini, Victor, Steven Yi, Joseph Timoney, et al. (2012). “The mobile Csound platform”. In: *Proceedings of International Computer Music Conference*. Ljubliana.

Le Vivier (2018). *Carrefour des musiques nouvelles*. URL: `https://levivier.ca/` (visited on 12/12/2018).

Leman, Marc (2016). *The expressive moment: how interaction (with music) shapes human empowerment*. Cambridge, MA: MIT Press.

Leonard, J. Paul, Bill Kapralos, Holly Tessler, and Leonard J. Paul (2014). "For the love of Chiptune". In: *The Oxford Handbook of Interactive Audio*. Ed. by Karen Collins, Bill Kapralos, and Holly Tessler. Oxford University Press.

Levine, Steve and J. William Mauchly (1980). "The fairlight computer musical instrument". In: *Proceedings of International Computer Music Conference*. New York City.

Liblo (2018). *Lightweight OSC implementation*. URL: `http://liblo.sourceforge.net/` (visited on 07/10/2018).

Lohner, Henning (1986). "The Upic system: a user's report". In: *Computer Music Journal* 10.4, pp. 42–49.

Longbottom, Roy (2018). *Roy Longbottom's PC benchmark collection*. URL: `http://www.roylongbottom.org.uk/` (visited on 11/02/2018).

MacConnell, Duncan et al. (2013). "Reconfigurable autonomous novel guitar effects (RANGE)". In: *Proceedings of the International Conference on Sound and Music Computing*. Stockholm.

Maker Media, Inc. (2018). *Make*. URL: `https://makezine.com/` (visited on 12/07/2018).

Malloch, Joseph, David Birnbaum, Elliot Sinyor, and Marcelo M. Wanderley (2006). "Towards a new conceptual framework for digital musical instruments". In: *Proceedings of the 9th International Conference on Digital Audio Effects*. Montreal, pp. 49–52.

Malloch, Joseph and Marcelo M. Wanderley (2007). "The T-Stick: from musical interface to musical instrument". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. New York, New York: ACM Press, p. 66.

Manning, Peter (2013). *Electronic and computer music*. Oxford University Press.

Marier, Martin (2018). *The Sponge*. URL: `http://www.martinmarier.com/wp/?page_id=12` (visited on 12/12/2018).

Mathews, Max (1989). "The conductor program and mechanical baton". In: Current Directions in Computer Music Research. Cambridge, MA: MIT Press.

— (1991). "The radio baton and conductor program, or: pitch, the most important and least expressive part of music". In: *Computer Music Journal* 15.4, p. 37.

Mathews, Max and F. Richard Moore (1970). "GROOVE - a program to compose, store, and edit functions of time". In: *Communications of the ACM* 13.12, pp. 715–721.

McCartney, James (1996). "SuperCollider: a new real time synthesis language". In: *Proceedings of International Computer Music Conference*. Hong Kong, pp. 257–258.

— (2002). "Rethinking the computer music language: SuperCollider". In: *Computer Music Journal* 26.4, pp. 61–68.

McNerney, Timothy S. (2004). "From turtles to tangible programming bricks: explorations in physical language design". In: *Personal and Ubiquitous Computing* 8.5, pp. 326–337.

McPherson, Andrew (2017). "Bela: An embedded platform for low-latency feedback control of sound". In: *The Journal of the Acoustical Society of America* 141.5, pp. 3618–3618.

Meneses, Eduardo A. L., Sérgio Freire, and Marcelo M. Wanderley (2018). "GuitarAMI and GuiaRT: two independent yet complementary augmented nylon guitar projects". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. Virginia.

Microchip Technology (2018a). *16-bit PIC MCUs|PIC24|dsPIC33*. URL: `https://www.microchip.com/design-centers/16-bit` (visited on 12/05/2018).

— (2018b). *32-Bit Microcontrollers (MCU)*. URL: `https://www.microchip.com/design-centers/32-bit` (visited on 12/05/2018).

— (2018c). *8-Bit MCUs*. URL: `https://www.microchip.com/design-centers/8-bit` (visited on 11/21/2018).

MIDI Manufacturers Association (2018). URL: `https://www.midi.org/` (visited on 10/05/2018).

Miranda, Eduardo Reck and Marcelo M. Wanderley (2006). *New digital musical instruments: control and interaction beyond the keyboard*. A-R Editions, Inc.

MOD Devices GmbH (2018). *MOD Duo*. URL: `https://www.moddevices.com/products/mod-duo` (visited on 11/21/2018).

Moggridge, Bill (2007). *Designing interactions*. Cambridge, MA: MIT Press.

Moog Music (2018). URL: `https://www.moogmusic.com/` (visited on 12/11/2018).

Moog, Robert A (1986). "MIDI: musical instrument digital interface". In: *Journal of the Audio Engineering Society* 34.5, pp. 394–404.

Moore, F. Richard (1982). "The computer audio research laboratory at UCSD". In: *Computer Music Journal* 6.1, pp. 18–29.

Motuk, Erdem, Roger Woods, and Stefan Bilbao (2005). "FPGA-based hardware for physical modelling sound synthesis by finite difference schemes". In: *Proceedings of the 2005 International Conference on Field-programmable Technology*. IEEE, pp. 103–110.

Motuk, Erdem, Roger Woods, Stefan Bilbao, and John McAllister (2007). "Design methodology for real-time FPGA-based sound synthesis". In: *IEEE Transactions on Signal Processing* 55.12, pp. 5833–5845.

Multer (2018). URL: `https://github.com/expressjs/multer` (visited on 09/14/2018).

Musallica (2018). URL: `https://www.facebook.com/Musallica/` (visited on 12/12/2018).

MusicMotion (2018). URL: `https://musicmotion.org/` (visited on 12/11/2018).

Mutable Instruments (2018). *Shruthi*. URL: `https://mutable-instruments.net/archive/shruthi/build/` (visited on 11/22/2018).

Mxmxmx (2018). *Terminal Tedium*. URL: `https://github.com/mxmxmx/terminal_tedium` (visited on 09/18/2018).

Native Instruments (2018). *Software And Hardware For Music Production And Djing*. URL: `https://www.native-instruments.com/en/` (visited on 12/11/2018).

Nexus Web Audio Interfaces (2018). URL: `https://nexus-js.github.io/ui/` (visited on 09/05/2018).

Node.js Foundation (2018a). *Express - Node.js web application framework*. URL: `https://expressjs.com/` (visited on 12/05/2018).

— (2018b). *Express 4.x - API Reference*. URL: `https://expressjs.com/en/api.html` (visited on 12/05/2018).

— (2018c). *Node.js*. URL: `https://nodejs.org/en/` (visited on 12/02/2018).

Nord Keyboards (2018). *Nord Modular*. URL: `http://www.nordkeyboards.com/products/nord-modular` (visited on 01/30/2018).

Norman, Don (2013). *The design of everyday things*. New York: Basic Books.

Npm (2018). URL: `https://www.npmjs.com/` (visited on 12/05/2018).

Oliveros, Pauline (2005). *Deep listening : a composer's sound practice*. New York: iUniverse, Inc.

Open Music Labs Wiki (2018). *Repatcher*. URL: `http://wiki.openmusiclabs.com/wiki/repatcher` (visited on 09/18/2018).

Orlarey, Yann, Dominique Fober, and Stephane Letz (2009). "FAUST: an efficient functional approach to DSP programming". In: *New Computational Paradigms for Computer Music*, pp. 65–96.

OSA Electronics (2017). *DACBerry PRO for Raspberry Pi*. URL: `https://www.osaelectronics.com/product/dacberry-pro/` (visited on 11/02/2018).

Paine, Garth (2009). "Gesture and morphology in laptop music performance". In: *The Oxford handbook of computer music*. Ed. by Roger Dean. Oxford University Press, pp. 214–232.

Paradiso, Joseph A. (1997). "Electronic music: new ways to play". In: *IEEE Spectrum* 34.12, pp. 18–30.

Patchblocks (2018). URL: `http://patchblocks.com/` (visited on 10/16/2018).

Pfeifle, Florian and Rolf Bader (2009). "Real-time physical modelling of a real banjo geometry using FPGA hardware technology". In: *Musical Acoustics, Neurocognition and Psychology of Music Hamburg Yearbook for Musicology* 25, pp. 71–86.

— (2011). "Real-time finite-difference string-bow interaction field-programmable gate array (FPGA) model coupled to a violin body". In: *The Journal of the Acoustical Society of America* 130.4, pp. 2507–2507.

— (2013). "Performance controller for physical modelling FPGA sound synthesis of musical instruments". In: *Sound-Perception-Performance*. Springer, pp. 331–350.

PG Music Inc. (2018). URL: `http://www.pgmusic.com/` (visited on 02/07/2018).

PJRC: Electronic Projects (2018a). URL: `https://www.pjrc.com/` (visited on 04/11/2018).

— (2018b). *Teensy and Teensy++ pinouts, for C language and arduino software*. URL: `https://www.pjrc.com/teensy/pinout.html` (visited on 11/02/2018).

Portnoff, M. (1976). "Implementation of the digital phase vocoder using the fast Fourier transform". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 24.3, pp. 243–248.

Printemps Numérique (2018). URL: `http://www.printempsnumerique.ca/en/` (visited on 12/12/2018).

PRISM (2018). URL: `http://www.prism.quebec/?fbclid=IwAR157-52yb5SQpHNVquxssaLP0k5WZP-VhBeV6-O9keDs_IwillDhgc4vvY` (visited on 12/13/2018).

Propellerhead Software (2018). URL: `https://www.propellerheads.com/` (visited on 12/11/2018).

Puckette, Miller (1988). "The patcher". In: *Proceedings of International Computer Music Conference*. San Francisco: Computer Music Association, pp. 420–425.

— (1991). "Combining event and signal processing in the Max graphical programming environment". In: *Computer Music Journal* 15.3, pp. 68–77.

— (1997). "Pure Data". In: *Proceedings of the International Computer Music Conference*. Thessaloniki.

— (2002). "Max at Seventeen". In: *Computer Music Journal* 26.4, pp. 31–43.

Putnam, William and Tim Stilson (1996). "Frankenstein: A low cost multi-DSP compute engine for the music kit". In: *Proceedings of International Computer Music Conference*. Hong Kong.

Raczinski, J. M., G. Marino, S. Sladek, and V. Fontalirant (1999). “SYNTHUP carte PCI pour la synthèse du son et le traitement de signal en temps reel”. In: *Proceedings of the JIM99*, pp. 75–82.

Raczinski, J. M., Stéphane Sladek, and Luc Chevalier (1999). “Filter Implementation on SYNTHUP”. In: *Proceedings of the 2nd COST G-6 Workshop on Digital Audio Effects*. Trondheim.

Rasmussen, Jens (1986). *Information processing and human-machine interaction: an approach to cognitive engineering*. North-Holland Series in System Science and Engineering 12. New York: North-Holland.

Raspberry Pi Foundation (2018). *Raspberry Pi*. URL: `https://www.raspberrypi.org/` (visited on 04/11/2018).

Rhizome (2018). URL: `http://rhizome.org/` (visited on 03/26/2018).

RIDM (2018). *Montreal International Documentary Festival*. URL: `http://ridm.ca/en` (visited on 12/12/2018).

Roads, Curtis (2004). *Microsound*. Cambridge, MA: MIT press.

Roca, Marcellí Antúnez (2018). *Marcellí Antúnez Roca*. URL: `http://marceliantunez.com/` (visited on 10/03/2018).

ROLI Ltd. (2018). *JUCE*. URL: `https://juce.com/` (visited on 01/30/2018).

Rowe, Robert (1992). “Machine listening and composing with Cypher”. In: *Computer Music Journal* 16.1, p. 43.

Rythmopolis (2018). URL: `https://www.rythmopolis.com/` (visited on 12/11/2018).

Saito, Takashi, Tsutomu Maruyama, Tsutomu Hoshino, and Saburo Hirano (2001). “A music synthesizer on FPGA”. In: *Field-Programmable Logic and Applications*. Springer, pp. 377–387.

SAT (2018). *Société des arts technologiques*. URL: `https://sat.qc.ca/` (visited on 12/12/2018).

Satinger, Chris (2018). *Supercollider.js*. URL: `https://crucialfelix.github.io/supercolliderjs/` (visited on 09/14/2018).

Schottstaedt, Bill (1994). “Machine tongues XVII: CLM: Music V meets Common Lisp”. In: *Computer Music Journal* 18.2, p. 30.

Sequential LLC (2018). URL: `https://www.sequential.com/` (visited on 12/11/2018).

Smith, Julius O. (1992). “Physical modeling using digital waveguides”. In: *Computer Music Journal* 16.4, p. 74.

Socket.IO (2018). URL: `https://socket.io/index.html` (visited on 12/11/2018).

Sonic Core (2018). *SCOPE*. URL: `http://scope.zone/` (visited on 11/22/2018).

Sullivan, John (2015). “Noisebox: design and prototype of a new digital musical instrument”. In: *Proceedings of the International Computer Music Conference*. Denton.

SuperCollider 3.10.0 Help (2018). URL: `http://doc.sccode.org/` (visited on 12/05/2018).

Suzanne Ciani (2018). URL: `https://www.sevwave.com` (visited on 12/11/2018).

Symbolic Sound (2018). *Kyma*. URL: `https://kyma.symbolicsound.com/` (visited on 11/22/2018).

Synthtopia (2016). *New Framework For DIY Synthesizers, Prynth*. URL: `http://www.synthtopia.com/content/2016/11/23/new-framework-for-diy-synthesizers-prynth/` (visited on 11/29/2018).

Taelman, Johannes (2018). *Axoloti*. URL: `http://www.axoloti.com/` (visited on 01/30/2018).

Taylor, Benjamin et al. (2014). "Simplified expressive mobile development with NexusUI, NexusUp, and NexusDrop". In: *Proceedings of the International Conference on New Interfaces for Musical Expression*. London, pp. 257–262.

Technologies, Terasic (2018). *Terasic - SoC Platform*. URL: `https://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=204#Category205` (visited on 12/05/2018).

Terasic (2018). URL: `http://www.terasic.com.tw/` (visited on 11/22/2018).

The OWL (2018). URL: `https://hoxtonowl.com/` (visited on 10/16/2018).

Theodoropoulos, Dimitris, Catalin Bogdan Ciobanu, and Georgi Kuzmanov (2009). "Wave field synthesis for 3D audio: architectural Prospectives". In: *Proceedings of the 6th ACM Conference on Computing Frontiers*. New York City: ACM, pp. 127–136.

Topliss, James William, Victor Zappi, and Andrew McPherson (2014). "Latency performance for real-time audio on Beaglebone Black". In: *International Linux Audio Conference*. Karlsruhe, Germany.

Torre, Giuseppe, Kristina Andersen, and Frank Baldé (2016). "The Hands: The making of a digital musical instrument". In: *Computer Music Journal* 40.2, pp. 22–34.

Trail, Shawn et al. (2014). "El-Lamellophone - a low-cost, DIY, open framework for acoustic Lemellophone based hyperinstruments". In: *Proceedings of International Conference on New Interfaces for Musical Expression*. London, pp. 537–540.

Ucamco (2018). *Gerber Format*. URL: `https://www.ucamco.com/en/file-formats/gerber` (visited on 11/30/2018).

ValentF(x) (2014). *FPGA LOGI family*. URL: `http://valentfx.com/fpga-logi-family/` (visited on 12/05/2018).

Variable Media Network (2018). URL: `http://www.variablemedia.net/e/welcome.html` (visited on 03/26/2018).

Vercoe, Barry (1993). *Csound: a manual for the audio processing system and supporting programs with tutorials*. Cambridge, MA: Massachusetts Institute of Technology.

Wanderley, Marcelo M. and Marc Battier, eds. (2000). *Trends in gestural control of music*. Published: [CD-ROM]. Paris, France: IRCAM - Centre Pompidou.

Wang, Ge and Perry R Cook (2003). "Chuck: a concurrent, on-the-fly, audio programming language". In: *Proceedings of International Computer Music Conference*. Singapore.

Weiser, Mark (1991). "The computer for the 21st century". In: *Scientific American* 265.3, pp. 94–104.

Wessel, David and Matthew Wright (2002). "Problems and prospects for intimate musical control of computers". In: *Computer Music Journal* 26.3, pp. 11–22.

Williams, Elliot (2016). *The Most Flexible Synthesizer is DIY, Raspberry Pi*. URL: `https://hackaday.com/2016/11/26/the-most-flexible-synthesizer-is-diy-raspberry-pi/` (visited on 11/29/2018).

Wilson, Scott (2018). *Prynth is a new open-source platform for making DIY synths*. URL: `https://www.factmag.com/2016/11/24/prynth-open-source-diy-synth-framework/` (visited on 11/29/2018).

Wilson, Scott, Nick Collins, and David Cottle, eds. (2011). *The supercollider book*. Cambridge, MA: MIT Press.

World Wide Web Consortium (2018). *XMLHttpRequest*. URL: `https://dvcs.w3.org/hg/xhr/raw-file/tip/Overview.html` (visited on 12/05/2018).

Wright, Matthew (2005). "Open Sound Control: an enabling technology for musical networking". In: *Organised Sound* 10.3, pp. 193–200.

Xenakis, Iannis (1992). *Formalized music: thought and mathematics in composition*. 6. Pendragon Press.

Zeltzer, D. and D. J. Sturman (1994). "A Survey of Glove-based Input". In: *IEEE Computer Graphics and Applications* 14, pp. 30–39.

Zhang, Z. (2012). "Microsoft Kinect Sensor and Its Effect". In: *IEEE MultiMedia* 19.2, pp. 4–10.