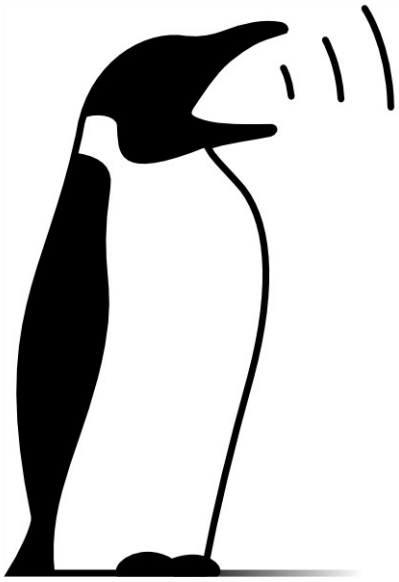
Proceedings of the

Linux Audio Conference 2019

March 23rd – 26th, 2019

Center for Computer Research in
Music and Acoustics (CCRMA)

Stanford University, USA



"In Ping(uins) we trust"

Published by
CCRMA, Stanford University, California, US
March 2019

http://lac.linuxaudio.org/2019
ISBN 978-0-359-46387-9

Credits
Layout: Frank Neumann and Romain Michon
Typesetting: LaTeX and pdfLaTeX
Logo Design: The Linuxaudio.org logo and its variations copyright Thorsten Wilms ©2006, imported into "LAC 2014" logo by Robin Gareus

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

Partners and Sponsors

Linuxaudio.org

Center for Computer Research in Music and Acoustics - Stanford University

CENTRE NATIONAL DE CRÉATION MUSICALE

Stanford | Arts Institute

Foreword

Welcome everyone to LAC 2019 at CCRMA!

For the second time in its seventeen year history, the Linux Audio Conference (LAC) is hosted in the United Stated of America by the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University. With its informal workshop-like atmosphere, LAC is a blend of scientific and technical papers, tutorials, sound installations, and concerts centered on the free GNU/Linux operating system and open-source free software for audio, multimedia, and musical applications. LAC is a unique platform during which members of this community gather to exchange ideas, draft new projects, and see old friends.

In these times of increasing political tensions and of rising extremism throughout the world, we believe that emphasizing and promoting the universality of this type of event is of the utmost importance. The Linux audio community exists worldwide; we believe it should remain a priority to diversify LAC's geographical location from year to year for the benefit of those who can't afford to travel to the other side of the world.

This year, a large portion of presenters and performers is coming from the Americas and Asia. LAC-19 features six paper sessions, five concerts, four workshops, one keynote, as well as various posters, demos, and side events happening in various locations on Stanford University campus.

We wish you a pleasant stay at Stanford and we hope that you will enjoy the conference!

Romain Michon (LAC-19 Co-Chair)
CCRMA, Stanford University (USA) & GRAME-CNCM, Lyon (France)

Conference Organization Core Team

- Romain Michon – Co-Chair & Organizer
- Fernando Lopez-Lezcano – Co-Chair & Organizer
- Constantin Basica – Music Chair & Organizer
- Elliot Canfield-Dafilou – Organizer
- Carlos Sanchez – CCRMA System Administrator & Organizer
- Matthew Wright – CCRMA Technical Director & Organizer
- Nette Worthey – CCRMA Administrator
- Chris Chafe – CCRMA Director
- Bruno Ruviaro – Organizer

Volunteers

- Alex Chechile
- David Goedicke
- Benjamin J. Josie
- Kyle O. Laviana
- Vidya Rangasayee
- Travis Skare
- Xingxing Yang

Paper Chair/Administration and Proceedings

- Frank Neumann

Stream Team

- David Kerr
- Carlos Sanchez

Keynote Speaker

- Fernando Lopez-Lezcano

Also Thanks to

- **Session Chairs:**
  - Chris Chafe
  - Albert Gräf
  - Elliot Kermit Canfield-Dafilou
  - Fernando Lopez-Lezcano
  - Yann Orlarey
  - Julius O. Smith
- Santa Clara University Laptop Orchestra (SCLOrk)

Review Committee

Joachim Heintz	University for Music Drama and Media Hanover, Germany
IOhannes Zmölnig	IEM, University of Music and Performing Arts (KUG), Graz, Austria
Elliot Canfield-Dafilou	CCRMA, Stanford University, United States
Robin Gareus	Germany
Martin Rumori	Institute of Electronic Music and Acoustics, Graz, Austria
Orchisama Das	CCRMA, Stanford University, United States
Henrik von Coler	Technische Universität Berlin, Germany
Mark Rau	CCRMA, Stanford University, United States
Stéphane Letz	GRAME-CNCM, France
Romain Michon	GRAME-CNCM, France & CCRMA - Stanford University, USA
Jörn Nettingsmeier	Netherlands
David Runge	Native Instruments, Germany
Fons Adriaensen	Huawei Research, Germany
Harry van Haaren	OpenAV, Ireland
Steven Yi	United States

Table of Contents

Papers

Posters

CREATING A SONIFIED SPACECRAFT GAME USING HAPPYBRACKETS AND STELLARIUM

Angelo Fraietta

UNSW Art and Design
University of New South Wales, Australia
a.fraietta@unsw.edu.au

Ollie Bown

UNSW Art and Design
University of New South Wales, Australia
o.bown@unsw.edu.au

ABSTRACT

This paper presents the development of a virtual spacecraft simulator game, where the goal for the player is to navigate their way to various planetary or stellar objects in the sky with a sonified poi. The project utilises various open source hardware and software platforms including Stellarium, Raspberry Pi, HappyBrackets and the Azul Zulu Java Virtual Machine. The resulting research could be used as a springboard for developing an interactive science game to facilitate the understanding of the cosmos for children. We will describe the challenges related to hardware, software and network integration and the strategies we employed to overcome them.

1. INTRODUCTION

HappyBrackets is an open source Java based programming environment for creative coding of multimedia systems using Internet of Things (IoT) technologies [1]. Although HappyBrackets has focused primarily on audio digital signal processing—including synthesis, sampling, granular sample playback, and a suite of basic effects–we created a virtual spacecraft game that added the functionality of controlling a planetarium display through the use of WiFi enabled Raspberry Pis. The player manoeuvres the spacecraft by manipulating a sonic poi[1], which is usually played in the manner shown in Figure 1. The poi contains an inertial measurement unit (IMU), consisting of an accelerometer and gyroscope; and a single button. The goal of the

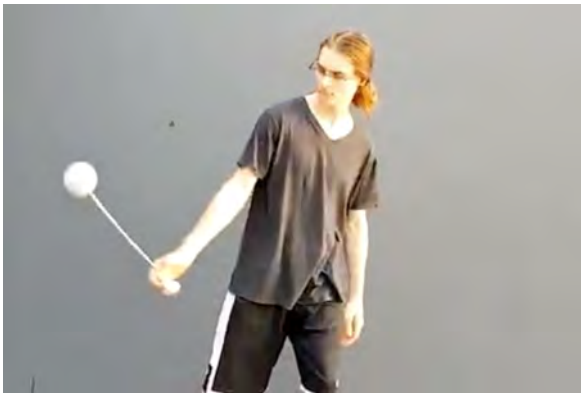


Figure 1: *The conventional way of playing a sonic poi.*

game is for a player to choose an astronomical object, for example a planet or star, and to fly towards that object. This enables the player to view other objects, including planets, moons, stars and galaxies in the field of view. For example, Figure 2 shows how the player might view Saturn from Earth, while Figure 3 shows how the player may view Saturn from their spacecraft. The sonic poi generates sound that is indicative of the player's field of view. Additionally, the poi provides audible feedback when the player zooms in or out.

Figure 2: *Saturn viewed from the ground from Stellarium.*

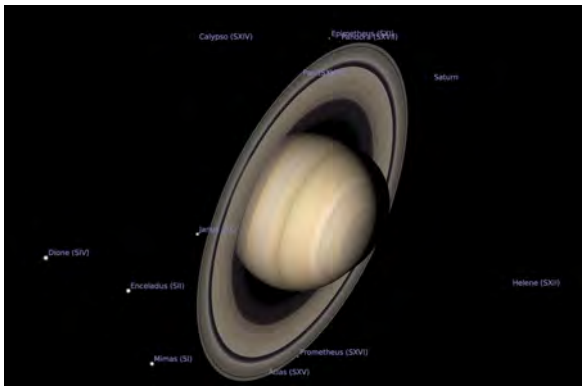


Figure 3: *A closer view of Saturn from Stellarium.*

The University of New South Wales required a display for their open day to showcase some of the work conducted in the Interactive Media Lab. The opportunity to develop an environment whereby visitors could engage with the technology we were developing would not only facilitate attracting possible future students, it was also a way to develop and test the integration of various research components we were conducting. Many managers and business seek to engage new customers through gamification [3]—in this case, prospective customers were potential students. Furthermore, research indicates that visualisation and interpretation of software behaviour de-

[1] "Poi spinning is a performance art, related to juggling, where weights on the ends of short chains are swung to make interesting patterns." [2, p. 173]

veloped as part of a game is more memorable, which facilitates locating errors or developing methods for improvement [4]. Developing a game, therefore, would not only engage the visitors, it would provide us with a more memorable way of seeing how our system was behaving.

The technology to develop the game required two different versions of Raspberry Pi, installation of planetarium software onto one of the Pis, and the creation of a Java API to join the different systems. This paper details the strategies and techniques to integrate the different technologies and describes some of the workarounds for unresolved issues. We also discuss the goals, rules and rewards used to define the game and the methods we used to entice prospective players. Finally, we lists areas where the research can be extended.

2. BACKGROUND TO RESEARCH

The research was inspired by a previous project developed by one of the authors that correlated what a viewer saw in the night sky through binoculars with data obtained from on-line astronomical data catalogues [5]. One installation, which was conducted in conjunction with the Newcastle Astronomical Society on one of their field viewing nights, was particularly successful [6]. More than twenty members of the public were enticed into viewing the night sky through high powered binoculars while sound that was based on data from the stars they were viewing was playing through loudspeakers on the field.

Another set of performances was conducted with an improvising ensemble that featured various astronomical photos displayed as a slide show [7]. The stellar data was mapped as MIDI and successfully functioned as inspirational impetus for the performers, but was unsuccessful from an astronomical point of view. First, the ability for viewers to look through the equipment was directly dependant upon the weather. One performance, for example, had a night sky complete with thick black cloud, heavy rain and lightning. Moreover, when the weather was favourable for viewing, the audience were often content to just watch the performers rather than venture out of their chairs to view through the binoculars [5]. The audience feedback from the was that although they really liked the slide show, many were unaware that the binoculars were even there for viewing. Instead of providing a slide show at the next performance, an improvisation using Stellarium from a laptop computer was used on the screen. The audience's response was extremely favourable, inspiring the idea of using Stellarium as a visual stimulus instead of binoculars.

2.1. Raspberry Pi

The Raspberry Pi was originally developed in 2011 [8] for education by the Raspberry Pi Foundation, a UK based educational charity [9][10]. The Raspberry PI has a very large user base and a significant number of plug in sensors available for it [11], and supports a 128GB SD card, which can be used to store more than 200 hours of high-quality audio. The Raspberry Pi foundation officially supports a derivative of the Linux distribution Debian known as Raspbian [12]. Raspbian's inclusion of compilers, support for multiple coding languages, and the ability to run multiple programs provides the flexibility that enables a system to expand as an interactive platform as newer technologies become available. The game project used two different versions of Raspberry Pi and Raspbian. The sonic poi required a small form factor, low power consumption but did not require a GUI, and consequently, Pi Zero running Raspbian Stretch Lite was selected. The device used to display the graphics required significantly more power but did not have size restrictions, so a Raspberry Pi B+ running the desktop version of Stretch was selected for this.

2.2. HappyBrackets

HappyBrackets commenced as "A Java-Based remote live coding system for controlling multiple Raspberry Pi units" [13] where a master controller computer sent pre-compiled Java classes to selected Raspberry Pi devices on a network. Unlike the Arduino sketch, which is effectively a single program [14], the HappyBrackets composition is not a standalone executable program. The HappyBrackets core has a thread that listens for incoming bytecode classes, and after receiving the class, executes the new class's functionality through a Java interface. This allows for multiple concurrent compositions that can be easily created or updated during composition or the creative coding performance [1]. This research was extended with the development of the Distributed Interactive Audio Device (DIAD) [15], which contained an IMU consisting of an accelerometer, gyroscope and compass. The devices were handled by the audience and incorporated into the environment. The DIADS not only responded to user manipulation, they also responded to one another. Furthermore, DIADS were configured to automatically connect to the wireless network, and once a DIAD came into range of the network, became a part of the DIAD multiplicity. The main focus of this development was the implementation of a reusable platform that allowed creators to easily develop interactive audio and easily deploy it to other devices. Although HappyBrackets runs on many embedded platforms, the main research has been with the Raspberry Pi, primarily due to the availability and low cost of the devices. HappyBrackets is licensed under the Apache License 2.0[2] and is available through Git Hub[3].

A prebuilt disk image—which contains the Java Virtual Machine (JVM), the I2C drivers to enable access to the IMU, and libraries to access the GPIO—enables users to flash an SD card and start using HappyBrackets without ever having to connect their device to the Internet. The licence for the Oracle JVM, however, appeared to prohibit embedding the Oracle JVM into a prebuilt image and was therefore legally problematic. We found that the AZUL Zulu JVM was available under the GNU GPLv2 licence[4], enabling an embedded distribution within an image. Medromi et al. conducted a study that compared the two JVMs [16]. Their tests revealed that Zulu created more threads and classes than Oracle, indicating that Zulu probably used more memory, making it more susceptible to garbage collection issues. Furthermore, their tests showed that Zulu also used a greater percentage of CPU, indicating greater power consumption. The report, however, did not detail the difference in performance speed between the two JVMs. Our own initial tests did not show any difference between the two JVMs and there was no noticeable performance degradation, however, this is an area we still need to research. It is possible to change the default JVM used in the Raspberry Pi from the terminal, which would make switching between JVMs when performing comparative tests relatively easy.

[2] www.apache.org/licenses/ [accessed November 2018]

[3] github.com/orsjb/HappyBrackets [accessed November 2018]

[4] www.gnu.org/licenses/old-licenses/gpl-2.0.txt [accessed November 2018]

2.3. Stellarium

The advancement of computing power over the last two decades has made the availability of planetarium software available on both desktop computers and mobile devices commonplace. Moreover, many of these software packages—including *RedShift*[5], *SkySafari*[6], *StarMap*[7], *The SkyX*[8], and *Stellarium*[9]—have become valuable tools for astronomers. They facilitate the identification of objects and in the planning of viewing and astro-photography sessions by enabling sky simulation for any particular location, date and time [17].

Stellarium is an open source software project distributed under the GNU General Public Licence with the source code available through Git Hub[10] . Stellarium functions as a virtual planetarium; calculating positions of the Sun, moon, stars and planets based on the time and location defined by the user. Moreover, the viewing location does not even need to be on Earth. For example, Figure 4 displays Stellarium rendering Jupiter viewed from its moon Io.

Figure 4: *A simulation of Jupiter viewed from Io.*

Stellarium is used by both amateur and professional astronomers, and is used by the European Organisation for Astronomical Research in the Southern Hemisphere to facilitate distribution and sharing of visual data among scientists [18]. Stellarium has a very high quality graphical display, supporting spherical mirror projection that can be used with a dome [19]. Stellarium is used in many schools and museums because it is both scientifically accurate and visually engaging [18]. Moreover, it is suitable for demonstrating basic through to advanced astronomy concepts [18]. Stellarium has a built in library of 600 000 stars, with the ability to add an additional 210 million [19]. Moreover, Stellarium can display constellations from several different cultures and has labels translated to more than 40 languages, making Stellarium both culturally aware and inclusive [18].

Although it is quite straightforward to control Stellarium using a keyboard and mouse, there are many plugins that allow third party integration with the software. The plugin we were particularly interested in to control Stellarium was the Remote Control, which enabled control of Stellarium through HTTP [21]. Stellarium also contains a powerful scripting engine that enables one to program and run complete astronomy shows. The scripts, written in JavaScript, control Stellarium through a series of objects that represent the Stellarium application components [20].

3. RELATED WORK

Video games rose from obscurity in the 1970s, into a video arcade industry grossing $8 billion dollars in 1982 [22, p. 88]. The video game moved from the arcade into the home with Nintendo and Atari game consoles [22, 23]. Iconography games like *Space Invaders*, *Defender*, *Spaceward HO!* and *Star Wars* were often replaced with interactive games that became more realistic [23]. Wolf suggests that there are more than forty different genres of video games [23], however, we were only particularly interested in the "Training Simulation" genre.

One study showed that video game expertise developed over long-term playing had a beneficial effect on the spatial skills in the player, supporting the hypothesis that "video expertise could function as informal education for the development of skill in manipulating two-dimensional representations of three dimensional space" [22, p. 93]. The aerospace industry has employed training simulators for many years, with the advancement in virtual reality environments leading to the availability of a new technology known as "serious gaming" [24, p. 655]. This technology exploits popular high-quality computer games, making it available via Software Development Kits (SDKs) to developers of "serious"[sic] applications such as defence, surgery, education and aerospace [24, p. 686].

One particularly interesting training simulation project was a prototype environment for training astronauts in a simulated zero gravity environment for the purpose of controlling and handling objects [25]. Rönkkö et al. noted that astronauts discovered using a laptop in a zero gravity environment was completely different to using it on Earth, and that the whole concept of a laptop computer in a zero gravity environment was questionable [25, p. 183].

There have been various implementations of third party integration with Stellarium. Although it is possible to remotely control a telescope using Stellarium as the master controller [26], some researchers have developed projects whereby Stellarium becomes the slave. Tuveri et al. developed two planetarium control systems for driving Stellarium on a Laptop computer [27]. They extended the Stellarium code in order to send it application messages before the Remote Control plugin was available in the standard Stellarium distribution. One interaction implementation was through a touch screen, while the other was through a Kinect gesture controller [27].

The Remote Control Stellarium plugin was developed by "Florian Schaukowitsch in the 2015 campaign of the ESA Summer of Code in Space programme" [20, p. 110], and was used for a visual art installation in the MAMUZ museum for pre-history [21]. The installation, *STONEHENGE. A Hidden Landscape*, consisted of a single computer driving five projections onto a 15x4m curved screen.The presentation was automated with a Raspberry Pi that triggered a script via an HTTP request every twenty-five minutes via a cron job. This Remote Control plugin is now a standard part of the Stellarium installation. This use of both scripting and HTTP control was the mechanism we employed in our game.

4. DEFINING THE GAMIFIED EXPERIENCE

One of the intentions of creating the gamified environment was to engage visitors. In the gamified experience, four parties are involved: players, designers, spectators, and observers [28]. The key to a de-

[5] www.redshift-live.com [accessed November 2018]
[6] www.southernstars.com [accessed November 2018]
[7] www.star-map.fr [accessed November 2018]
[8] www.bisque.com [accessed November 2018]
[9] stellarium.org [accessed November 2018]
[10] github.com/Stellarium/stellarium [accessed November 2018]

veloping successful gamified experience is to identify who the parties are and how to engage them for the purpose of creating a positive and memorable experience [3], each with different levels of involvement or immersion [28]. Players were the visitors who physically controlled the virtual spacecraft, and in a sense, were the competitors and highly immersed in the experience. Spectators were people who do not directly compete in the game, but instead, influenced the game indirectly by encouraging the player and were also highly immersed in the experience. Observers were other visitors in the space that were passively involved and had no direct impact on the game. They were, however, mildly involved and often moved to become players or spectators [28].

Research indicates that the three main factors in developing an enjoyable game were challenge, fantasy and curiosity [29]. We provided challenge in that we set a goal that had increasing levels of difficulty. As the user was closer to the planet, the spacecraft became more difficult to control.

We utilised fantasy in that we implement two modes of play: terrestrial and spaceship. Terrestrial mode allowed the player to use gravity in a familiar way, provided wide fields of view that showed large amounts of sky and provided course control. Spaceship mode showed less fields of view, displaying significantly less sky and provided finer control; however, the player was not allowed to use gravity in their control. We enabled the player to zoom in and out by performing a quick twist action of the ball around the string. If the gyroscope pitch value exceeded the set threshold, the field of view would change, simulating a zoom in or out. When the user changed their field of view to less than 30 degrees, the play mode went from terrestrial to spaceship. We provided an audible feedback that sounded like a zipper when the level of zoom was changed.

The only controls available at the time on the poi were accelerometer and gyroscope [11], while the only feedback was audio generated by the poi and the Stellarium display. In the same way that a laptop could not be used conventionally in a zero gravity environment [25], a player would be unlikely to control the game successfully using the poi by spinning it around their body [2]. Figure 5 shows the poi with three axes of accelerometer and gyroscope on the left and right respectively.

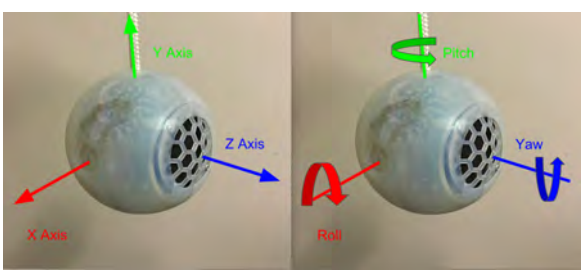


Figure 5: *Sonic Poi accelerometer and gyroscope input.*

In terrestrial mode, we wanted to simulate a viewer on the ground lifting and turning their head to view the sky as one would on Earth, which is essentially increasing the altitude and rotating the azimuth. The player "lifts their head" by raising the ball of the poi in an arc, using the point where the player holds the rope as the centre, and measuring Y axis acceleration through the IMU in the poi. Rotating the viewer's head was simulated by detecting the pitch value of the gyroscope, as shown on the right side of Figure 5. Gyroscope values only change while the object is rotating, whereas gravitational accelerometer values are maintained when the object is stationary.

In the spaceship mode, we wanted to simulate the player navigating through space in a zero-gravity environment. The yaw and the pitch were used as input, whereby the user had to roll the ball in their hands to move the display. This was completely foreign to users at first because there was no haptic feedback, nor any sense of grounding for the user or the control. In a sense, it was similar to balancing on a ball in space because you could not fall off—you would just float in an unintentional direction. Furthermore, it was not easy to detect which axis was which because the poi was a ball shape. Furthermore, rotating one axis would affect the cognition of the other axis. Consider a player in Figure 5 rotating the ball forward around the X axis with the poi producing a positive yaw. If the player then turned the poi 180 degrees around the string, rotating the ball forward again would now produce a negative yaw, which would mean the screen would start moving in the opposite direction to what they experience a moment earlier. The result was that controlling the display required constant mental adjustment, which we suggested might simulate to some degree the sense of strangeness an astronaut may feel controlling objects in outer space [25, p. 183].

In order to run an attractive and engaging display that would trigger the visitors' curiosity when they entered the room, we ran Stellarium scripts that functioned as standalone astronomy shows. We invited visitors to manipulate the poi and watch the display move while the script was running. When we saw they were interested and enjoyed the novelty of interacting with the display through the poi, we offered them the opportunity to start from Earth and navigate to one of the planets in our solar system. As they zoomed in closer to Saturn, they became quite excited when they saw the rings and realised that they could also see Saturn's moons. For those who were particularly enthusiastic, we suggested finding Jupiter next, informing them that they would also be able to see the four Galilean moons that night at home with a standard pair of binoculars. We also asked them to imagine that rolling the ball to control their movement might be as strange as moving about in a zero gravity environment. Although a few of the players gave up after a few minutes, the majority of players continued for more than ten minutes, had a lot of fun, and exhibited a sense of achievement in being able to navigate into outer space.

5. DEVELOPMENT

The system was originally developed as a tool for evaluating the performance, behaviour and suitability of networked control of Stellarium as part of a potential interactive audio visual artwork. We intended to calculate the azimuth and altitude position in space calculated from the rotation and manipulation of poi. These values would be used as input to Stellarium on another device, sent via the network, which would then display the sky based on those values. Additionally, we sent commands to change the field of view on Stellarium, which effectively acted as a zoom function. The poi also played audio as a series of ten uniformly distributed pseudorandom sine waves between 500 and 550Hz, giving a sense of cosmic background microwave noise.

```
float freq = hb.rng.nextFloat() * 50 + 500;
Envelope envelope = new Envelope(1);
WaveModule soundGenerator = new WaveModule();
```

[11] The button control was added to the poi later.

```
soundGenerator.setFequency(freq);
soundGenerator.setGain(envelope);
soundGenerator.connectTo(masterGain);
return  envelope;
```

A metronome iterates though each of the envelopes, adding segments that cause each frequency to momentarily pop out of the background as a beep.

```
hb.createClock(5000).addClockTickListener((
    offset, this_clock) -> {
Envelope e = envelopes.get(envelopIndex++ %
    TOTAL_OSCILLATORS);
final float LOUD_VOL = 20;
final float LOUD_SLOPE = 20;
final float LOUD_DURATION = 200;

e.addSegment(LOUD_VOL, LOUD_SLOPE);
e.addSegment(LOUD_VOL, LOUD_DURATION);
e.addSegment(1, LOUD_SLOPE);
});
```

As the user zooms in, the metronome becomes faster, increasing the beep rate, generating a sense of sonic tension.

5.0.1. *Starting Stellarium*

The first challenge was starting Stellarium on the Pi from within HappyBrackets. HappyBrackets has a simple facility to execute shell commands or create processes through both the Java Runtime exec and the ProcessBuilder [30]. We attempted a script to run Stellarium from a process command, which ran successfully when executed from a terminal; however, we could not get HappyBrackets to run the script after each fresh reboot of the device—the program was unable to access the display. Interestingly, If we killed the JVM and the started HappyBrackets again from a terminal, then Stellarium started from within HappyBrackets with no problem. The problem was that the HappyBrackets installation script had configured the Raspberry Pi to automatically start the Java application when the device first boots by executing a script in /etc/local.rc as defined in the Raspberry Pi documentation[12]. In order to run GUI programs from Java, the Java program needs to be started when the desktop starts, which was effected by moving the script command to /.config/lxsession/LXDE-pi/autostart[13]. The HappyBrackets installation scripts were consequently modified to detect whether a desktop version was used, and added the HappyBrackets start-up script command accordingly.

5.0.2. *Controlling Stellarium*

Examples of controlling Stellarium through the Remote Control API were provided on the plugin developer page[14], which made use of the *cURL [sic]*[15] command line utility[16] and executed via an SSH terminal connection to the Pi. Although we did not intend to use curl in our actual program because Java has its own networking interface, curl was extremely useful for examining and diagnosing through the terminal. Querying the state of Stellarium was performed by issuing a curl *GET* command. For example, executing the following command in the SSH terminal retrieves the current view direction of Stellarium as a JSON encoded string.

```
curl -G http://localhost:8090/api/main/view
{"altAz":"[0.954175, 9.54175e-06,
    0.299249]","j2000":"[0.240925, 0.147495,
    -0.959271]","jNow":"[0.241334, 0.148053,
    -0.959082]"}
```

Setting the position of Stellarium is executed with the curl *POST* command, with the parameters added as JSON parameters. Executing the following command would set the display to horizontal by setting the altitude to zero.

```
    curl -d 'alt=0' http://localhost:8090/api
        /main/view
```

Having tested the functionality using curl through the terminal, we implemented calls using the standard Java URL connections [31]. We sent control message from the poi via UDP to the slave using HappyBrackets and then immediately sent the HTTP message on the slave to Stellarium. We found that although the message arrived from the poi to the slave in less than a few milliseconds, the time to execute the post message on localhost, be actioned by Stellarium, and then return typically took between 80 and 120 milliseconds. This produced accumulative latency when the player continually moved the poi. The accelerometer and gyroscope typically update every 10ms, so constantly rotating the device for two seconds would generate approximately 200 messages. These values would become queued inside the slave and sequentially executed, which would result in an accumulating latency over a twenty second period. A method was required that would immediately send the last received position change when the last message was complete, but would discard previous values that were not yet actioned. We accomplished this through an independent thread for executing the post command. This thread would be effectively dormant while waiting for an event. When a message arrives on a different thread, the event is triggered, at which point the thread wakes and sends the message. We effected this through the use of Java synchronisation objects. The functionality that sends the post messages to Stellarium executes in an indefinite loop, laying dormant through the `altAzSynchroniser.wait()` call.

```
new Thread(() -> {
    while (!exitThread) {
        synchronized (altAzSynchroniser){
            try {
                altAzSynchroniser.wait();
            } catch (InterruptedException e)
                {
                e.printStackTrace();
            }
        }
        sendAltAz(currentAz, currentAlt);
    }
}).start();
```

The thread will wait indefinitely until it receives a signal from variable `altAzSynchroniser`. When a message to change the altitude arrives from the poi, the class variable `currentAlt` is set and the `altAzSynchroniser` object is notified, which in turn

[12] www.raspberrypi.org/documentation/linux/usage/rc-local.md [accessed November 2018]

[13] www.raspberrypi.org/forums/viewtopic.php?t=139224 [accessed November 2018]

[14] stellarium.org/doc/head/remoteControlApi.html

[15] curl.haxx.se

[16] cURL should not be confused with the curl programming language. ec.haxx.se/curl-name.html [accessed November 2018].

causes the thread shown above to wake and then call `sendAltAz` with the new azimuth and altitude to the localhost.

```
public void changeAltitude(double
    control_val) {
  synchronized (altAzSynchroniser){
   currentAlt = control_val  / 2 * Math.PI;
   altAzSynchroniser.notify();    }
}
```

We found that modifying the azimuth and altitude directly often produced a jittery display due to the 100ms latency coupled with discarding of values that were not actioned while waiting for the `sendPostMessage` call to return. We reduced this problem significantly by sending arrow key messages and moved the display left and right instead of sending an azimuth. This produced a smooth display rotation when rotating the ball. It was not possible to use this for the altitude in the terrestrial mode because we were using the accelerometer value to determine the height. In the spaceship mode, however, this proved very effective as we were able to just send up, down, left and right messages based on gyroscope action.

6. FUTURE WORK

There were several issues that we discovered through running the game. The first problem was that the Raspberry Pi would often crash when running the display after a certain period of intense manipulation, however, we were able to run it for several days if we did not demand too many rapid changes from Stellarium. We substituted the Pi with a Mac Mini in order to determine where the problems were. We found that we were able to reproduce an error in Stellarium on the Raspberry Pi when running the script *double_stars.ssc* that comes with Stellarium, however, the Mac ran with no errors. Running the kernel journal showed errors indicating an inability to allocate memory within the GPU[17]. The VC4 OpenGL driver required to run Stellarium is still experimental, and it is probably that this is where the error lies. Research and development in this area is still required to make a stable Raspberry Pi installation of Stellarium.

We found that when the player started rotating the ball fast, the zoom would activate, requiring them to stay within certain rotation rates. We modified the game so changing zoom required the player to hold the button down when performing a zoom action.

Messages are sometimes lost over UDP, which became evident when a zoom message was sometimes not delivered to the slave. We have performed some tests comparing different routers and different Raspberry Pis for packet loss. Additionally, we tested code in both Java and C++. We discovered that as packet intervals exceeded 10ms, the percentage of packet loss increased. Interestingly, we found that there was less packet loss using Java than C++ using the standard compilers distributed with Raspbian. Furthermore, the quality of router had a significant impact. Some routers, although supporting multicasting, stopped sending multicast messages to devices after about ten minutes. We intend to perform more tests regarding the packet loss, however, the real concern is that broadcasting and multicastling of OSC over UDP is not satisfactory [33].

We found that the Just In Time (JIT) compiler took time to convert the downloaded Java byte code into machine code [34], producing a brief stuttering effect when executed for the first time. The problem became exacerbated when using the Pi Zero with ten oscillators running simultaneously due to the limited power of the Pi Zero. Once the JIT compiler had converted the code, subsequent code changes were not affected. Although only an issue when the program starts, we need to examine strategies to overcome this.

[17] github.com/Stellarium/stellarium/issues/550 [accessed November 2018]

7. CONCLUSIONS

During our research we were able to integrate various open source programs to create a system where we could develop and evaluate Stellarium as a controllable display element, create inter process and device communication using the HappyBrackets Java environment, and to experiment with the use of the sonic poi as a performance tool. We used this system to create a gamified environment where visitors were engaged with our technology, providing them with a positive and memorable experience. We capitalised on this opportunity to observe and evaluate how our system was behaving, which was more memorable to us by virtue of it being part of a game that was played repeatedly. We leveraged the quality the Stellarium display coupled with a wireless control device to create a game that was challenging, fun, engaging and educational. Moreover, the technical goal was to be able to control Stellarium during a performance with HappyBrackets, with an example available at `https://youtu.be/NhXRdd-MNoo`

The research obtained from developing this game can be used as a starting point for the development of an interactive educational installation. Furthermore, we found a way to expose issues with OpenGL driver on the Raspberry Pi, Java JIT, and UDP packet loss and performance using both Java and C++.

8. ACKNOWLEDGEMENTS

A special thanks to Ben Cooper who designed and built the Sonic Poi. Many thanks to the members of the Newcastle Astronomical Society in their support and encouragement for this project. I would like to acknowledge the support of Georg Zotti and Alexander Wolf from the Stellarium development team for their advice and guidance in using Stellarium.

9. REFERENCES

[1] Sam Ferguson and Oliver Bown, "Creative coding for the Raspberry Pi using the HappyBrackets platform," in *Proceedings of the 2017 ACM SIGCHI Conference on Creativity and Cognition*. ACM, 2017, pp. 551–553.

[2] Eleanor Farrington, "Parametric equations at the circus: Trochoids and poi flowers," *The College Mathematics Journal*, vol. 46, no. 3, pp. 173–177, 2015.

[3] Karen Robson, Kirk Plangger, Jan H Kietzmann, Ian McCarthy, and Leyland Pitt, "Game on: Engaging customers and employees through gamification," *Business horizons*, vol. 59, no. 1, pp. 29–36, 2016.

[4] Nergiz Ercil Cagiltay, "Teaching software engineering by means of computer-game development: Challenges and opportunities," *British Journal of Educational Technology*, vol. 38, no. 3, pp. 405–415, 2007.

[5] Angelo Fraietta, "Musical composition with naked eye and binocular astronomy," in *Australasian Computer Music Conference 2014*. Victorian College of the Arts, 2014, p. 47.

[6] Angelo Fraietta, "Echoes from the fourth day - a segue through the southern night sky for FM synthesiser and binoculars," 2014, Performed in Brickworks Park in collaboration with the Newcastle Astronomical Society.

[7] Colin Bright, "spa-c–e," 2013, Performed live at Colbourne Ave Glebe, Sydney, Australia May 23rd 2013 by The Colin Bright Syzygy Band and Angelo Fraietta.

[8] Simon Monk, *Raspberry Pi cookbook: Software and hardware problems and solutions*, " O'Reilly Media, Inc.", 2016.

[9] Samuel Aaron, Alan F Blackwell, and Pamela Burnard, "The development of Sonic Pi and its use in educational partnerships: Co-creating pedagogies for learning computer programming," *Journal of Music, Technology & Education*, vol. 9, no. 1, pp. 75–94, 2016.

[10] Matt Richardson and Shawn Wallace, *Getting started with Raspberry Pi*, " O'Reilly Media, Inc.", 2012.

[11] Ivica Ico Bukvic, "Pd-l2ork Raspberry Pi toolkit as a comprehensive Arduino alternative in k-12 and production scenarios.," in *NIME*, 2014, pp. 163–166.

[12] Maik Schmidt, *Raspberry Pi: A Quick-Start Guide*, Pragmatic Bookshelf, 2014.

[13] Oliver Bown, Miriama Young, and Samuel Johnson, "A Java-based remote live coding system for controlling multiple Raspberry Pi units," in *ICMC*, 2013.

[14] Yusuf Abdullahi Badamasi, "The working principle of an Arduino," September 2014, pp. 1–4, IEEE.

[15] Oliver Bown, Lian Loke, Sam Ferguson, and Dagmar Reinhardt, "Distributed interactive audio devices: Creative strategies and audience responses to novel musical interaction scenarios," in *International Symposium on Electronic Art*. ISEA, 2015.

[16] Hicham Medromi, Laila Moussaid, and FAL Laila, "Analysis of the allocation of classes, threads and cpu used in embedded systems for Java applications," *Procedia computer science*, vol. 134, pp. 334–339, 2018.

[17] Joseph Ashley, "Computers and computer programs," in *Astrophotography on the Go*, pp. 151–161. Springer, 2015.

[18] K Berglund, "Using free, open source Stellarium software for iya2009," in *Preparing for the 2009 International Year of Astronomy: A Hands-On Symposium*, 2008, vol. 400, p. 483.

[19] Matthew Mc Cool, "Touring the cosmos through your computer: a guide to free desktop planetarium software," *CAPjournal,(7)*, pp. 21–23, 2009.

[20] Georg Zotti and Alexander Wolf, "Stellarium 0.18.0 user guide," 2018.

[21] Georg Zotti, Florian Schaukowitsch, and Michael Wimmer, "The skyscape planetarium," 2017.

[22] Patricia M Greenfield, Craig Brannon, and David Lohr, "Two-dimensional representation of movement through three-dimensional space: The role of video game expertise," *Journal of applied developmental psychology*, vol. 15, no. 1, pp. 87–103, 1994.

[23] Mark JP Wolf, "Genre and the video game," *The medium of the video game*, pp. 113–134, 2001.

[24] Robert J Stone, Peter B Panfilov, and Valentin E Shukshunov, "Evolution of aerospace simulation: From immersive virtual reality to serious games," in *Recent Advances in Space Technologies (RAST), 2011 5th International Conference on*. IEEE, 2011, pp. 655–662.

[25] Jukka Rönkkö, Jussi Markkanen, Raimo Launonen, Marinella Ferrino, Enrico Gaia, Valter Basso, Harshada Patel, Mirabelle D'Cruz, and Seppo Laukkanen, "Multimodal astronaut virtual training prototype," *International Journal of Human-Computer Studies*, vol. 64, no. 3, pp. 182–191, 2006.

[26] Jitong Chen, Lingquan Meng, Xiaonan Wang, and Chenhui Wang, "An integrated system for astronomical telescope based on Stellarium," in *Advanced Computer Control (ICACC), 2011 3rd International Conference on*. IEEE, 2011, pp. 431–434.

[27] Elena Tuveri, Samuel A Iacolina, Fabio Sorrentino, L Davide Spano, and Riccardo Scateni, "Controlling a planetarium software with a kinect or in a multi-touch table: a comparison," in *Proceedings of the Biannual Conference of the Italian Chapter of SIGCHI*. ACM, 2013, p. 6.

[28] Karen Robson, Kirk Plangger, Jan H Kietzmann, Ian McCarthy, and Leyland Pitt, "Is it all a game? Understanding the principles of gamification," *Business Horizons*, vol. 58, no. 4, pp. 411–420, 2015.

[29] Thomas W Malone, "Heuristics for designing enjoyable user interfaces: Lessons from computer games," in *Proceedings of the 1982 conference on Human factors in computing systems*. ACM, 1982, pp. 63–68.

[30] JW van der Veen, R de Beer, and D van Ormondt, "Utilizing Java concurrent programming, multi-processing and the Java native interface," *Running Native Code in Separate Parallel Processes," Report on behalf of the Marie-Curie Research Training Network FAST*, 2012.

[31] Cay S Horstmann and Gary Cornell, *Core Java 2: Volume I, Fundamentals*, Pearson Education, 2002.

[32] Matthew Wright, Adrian Freed, et al., "Open SoundControl: A new protocol for communicating with sound synthesizers.," in *ICMC*, 1997.

[33] Angelo Fraietta, "Open sound control: Constraints and limitations.," in *NIME*, 2008, pp. 19–23.

[34] Anderson Faustino Da Silva and Vitor Santos Costa, "An experimental evaluation of Java JIT technology.," *J. UCS*, vol. 11, no. 7, pp. 1291–1309, 2005.

BROWSER-BASED SONIFICATION

Chris Chafe

Center for Computer Research in Music and Acoustics (CCRMA)
Stanford University, USA
cc@ccrma.stanford.edu

ABSTRACT

TimeWorkers is a programming framework for coding sonification projects in JavaScript using the Web Audio API. It is being used for sonification workshops with scientists, doctors, and others to facilitate ease of use and cross-platform deployment. Only a browser and text editor are needed. Using Free and Open-source Software (FOSS) the system can run standalone since No Internet is Required for Development (NIRD). Workshop participants rapidly master principles of sonification through examples and are encouraged to bring their own datasets. All mapping code is contained in a project's .html landing page. A single generator function iterates over the project's data series and provides a fine-grained interface to time-varying sound parameters. This loop and its internals are patterned after similar constructions in the Chuck language used by the author in earlier sonification tutorials.

1. INTRODUCTION

Sonification shares much with other kinds of computer music making including the wide range of programming tools which can be used. Sonification also shares in the kinds of decisions found in photography and soundscape recording. Gathering, selecting, framing and contrast enhancement are a part of working with material from the (outside of music) outside world. On the other hand, another key part of creating a sonfication, mapping, has affinities with algorithmic composition. TimeWorkers is a browser-based software framework described in this paper which, while not limited to sonification, provides in it's initial rollout functional support for decisions specific to such work.

Specialized programming languages have evolved and continue to evolve which are custom-designed to express musical relationships, especially timing and concurrency. I've used several over the course of composing computer music with succeeding generations of hardware platforms, for example, Pla[1], MIDILisp[2], Common Music[3] and Chuck[4], all of which are examples of computer music languages with ways of programmatically expressing organization of sound in time.

TimeWorkers is written in JavaScript and provides a readily available computation environment for my sonification workshops. To give a glimpse of what will be explained later in detail, the name comes from its use of the Web Worker API[5] for composing musical layers or voices which unfold in time. The software uses browsers' existing means for sound generation, in this case the built-in computer music capabilities of the Web Audio API[6]. The added functionality provided by TimeWorkers provides ways to compose higher-level aspects of musical timing and texture.

Stepping back for a moment, it's worth reflecting on how computers and music have been mingling their intimate secrets for over 50 years. These two worlds evolve in tandem and where they intersect they spawn practices that are entirely novel. One of these is sonification, the practice of turning raw data into sounds and sonic streams to discover new relationships within the dataset by listening with a musical ear. This is similar to exploring data visualization with strategies made for the eye to reveal new insights from data using graphs or animations. A key advantage with sonification is sound's ability to present trends and details simultaneously at multiple time scales, allowing us to absorb and integrate this information the same way we listen to music.

Kramer, et al.'s prescient *Sonification Report* [7] (2010) merits quoting here at length and will be revisited in the conclusion section. The paper identified "three major issues in the tool development area that must be tackled to create appropriate synthesis tools developed for use by interdisciplinary sonification researchers." The TimeWorkers framework addresses some (but not all) of the following points.

"Portability: Sonification scale places demands on audio hardware, on signal processing and sound synthesis software, and on computer operating systems. These demands may be more stringent than the requirements for consumer multimedia. Researchers dealing with problems that go beyond the limits of one system should be able to easily move their sonification data and tools onto a more powerful system. Thus, tools must be consistent, reliable, and portable across various computer platforms. Similarly, tools should be capable of moving flexibly between real- time and nonreal-time sound production."

"Flexibility: We need to develop synthesis controls that are specific and sophisticated enough to shape sounds in ways that take advantage of new findings from perceptual research on complex sounds and multimodal displays and that suit the data being sonified. In addition to flexibility of synthesis techniques, simple controls for altering the data-to-sound mappings or other aspects of the sonification design are also necessary. However, there should be simple 'default' methods of sonification that allow novices to sonify their data quick and easily."

"Integrability: Tools are needed that afford easy connections to visualization programs, spreadsheets, laboratory equipment, and so forth. Combined with the need for portability, this requirement suggests that we need a standardized software layer that is integrated with data input, sound synthesis, and mapping software and that facilitates the evaluation of displays from perceptual and human factors standpoints."

2. USING THE FRAMEWORK

Meant to be very hands-on, my 2-hour workshops ask the participants to bring their own laptop and headphones. I first take them through a simple example which has a been an early "etude" assignment in my course, "*Computer Music Fundamentals* "[8], taught at Stanford's CCRMA. The goal is to get students to start working with their own datasets as soon as possible and get them exploring a range

of sonifications through experimentation.

A dataset to play with can be scouted out by searching the web and copied or exported from a spreadsheet or other format. For starters, it's simply a single column of numbers in plain text. The range of values doesn't matter because it will be automatically rescaled when read by the framework's file input layer. In my own development work, examples and code repository are all linux-based and other operating systems work equally well.

2.1. Basic Sonfication How-to

2.1.1. What you'll need

The browser can be a recent version of Firefox, Chromium, Chrome, or Edge. A simple text editor like Gedit is all that's required for developing the code and preparing an ASCII data file.

2.1.2. Testing the demo

Open the demo URL `https://ccrma.stanford.edu/~cc/sonify` to see a page that looks like Figure 1. There's a default time series "*tides.dat*" that can be played by clicking on the demo icon (the small globe is a button).

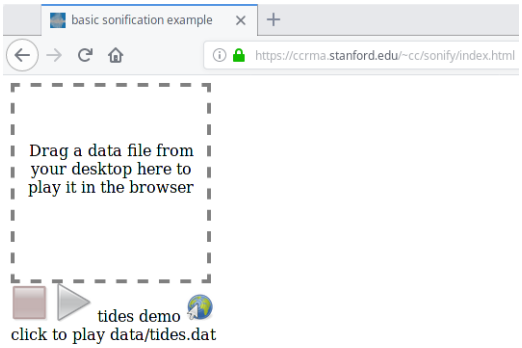


Figure 1: *An example page with options for playing a default time series or dragging in a data file.*

Alternatively, a data file can be dragged from the desktop onto the page to sound it with the same preset sonification parameters.

The demo was created by Chris Hartley, a biologist who participated in the first workshop (in 2016) at the University of British Columbia. In it, "*You can hear the rising and then falling chirp-chirp-chirp of the major high tides, which get highest at the new and full moons, and then the slightly lower trill of two roughly equal high tides per day, which occurs during the quarter moons.*" Hartley's sonfication plays a year's worth of tidal data at a fast rate using a sine tone.

After starting the demo or after loading a data file the stop and play buttons on the web page become activated, Figure 2.

2.1.3. Modifying the demo

To practice modifying the demo, a good first goal is to make the rate of running through the data much slower. To accomplish this, we'll make a local copy of the demo, test it and then edit it.

Go to its repository `https://cm-gitlab.stanford.edu/cc/sonify` and download a snapshot. The downloaded .zip file

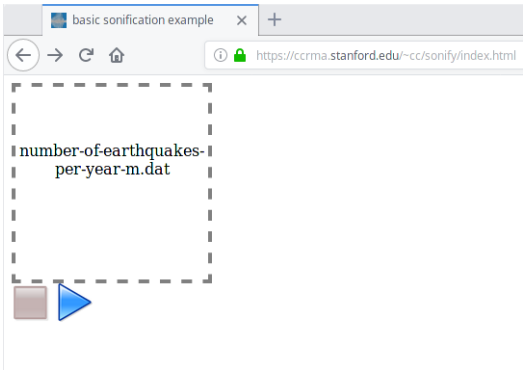


Figure 2: *Stop and play buttons become activated after starting the demo or dragging in a data file.*

will have a long name that depends on the version. Extract the contents of the .zip file and open its index.html file in a browser (use Firefox because it will allow the demo to run as a local file without manual intervention).

This will allow you to test the local copy of the landing page in a browser and make sure it's working identically to the version on the workshop's web server. If it's all good, then the local copy of the landing page can be opened in a text editor. Search for the line

`let dur = 0.005`

and assign a new value, for example:

```
function* sonify(data) {
  let dur = 0.05
// duration between data points in seconds
```

Save the modification in the text editor and then refresh the browser page to load the changed file. The example can then be played as before but the rate will now be $10x$ slower.

Further modifications are quickly explored with the same work flow of edit-save-refresh-play. For example, in the mapping function

`map(v)`

where, for a given value of v, sound parameters are determined for pitch and loudness (respectively, kn in MIDI key number units and db in a decibel range from -100 to 0). These in turn are used to calculate values which will be applied to the sine tone's frequency (Hz) and amplitude (range 0.0 to 1.0):

```
function map(v) {
  let kn = 60 + v * 40
  let f = mtof(kn)
  let db = -30 + v * 10
  let a = dbtolin(db)
  return {pit: f, amp: a}
}
```

`map(v)` returns pitch frequency and loudness amplitude in an object created by an object initializer. Its argument, v, is expected to lie in the range 0.0 to 1.0. In a hidden step which happens when the data is loaded, the data series has been automatically normalized to this range. `map(v)` is set so that the lowest data value will be sounded at Middle-C (MIDI key number 60) and the highest will be 3 Octaves and a Major Third above. Intermediate values will be linearly interpolated across key number values (using fractional quantities, in other words, not quantized to integer key numbers). Code for

the utility functions `mtof` and `dbtolin`, respectively for conversion from MIDI key number to frequency in Hz and dB loudness to amplitude, have been borrowed from Hongchan Choi's Web Audio API Extension (WAAX) project [9].

The sonify generator function sets a new target pitch when processing each new data value and starts a glissando (a smooth frequency ramp) to reach the target pitch in the length of time specified by the data update period, `dur`. The ramp is a linear function which updates the sine tone's frequency each audio sample. Amplitude is smoothly modulated in the same way.

The complete sonify generator function for this example is listed below and includes a definition of the sound source along with a mechanism for applying updates to its parameters. The new function `Sin(timeWorker)` instantiates a SinOsc and several methods which start the oscillator, apply parameter updates to it and stop it. After instantiation as a local object `s`, it is initiated with the first values from the mapping function and a gain of 0. Ramps are set in motion and the process pauses until they reach their targets with `yield dur` after which the loop continues and cyclically churns through each data point until all have been "performed." The last few lines ramp the oscillator to 0 and then stop and finish.

```
function* sonify(data) {
  let dur = 0.005
  let datum = data.next()
  function map(v) {
    let kn = 60 + v * 40
    let f = mtof(kn)
    let db = -30 + v * 10
    let a = dbtolin(db)
    return {pit: f, amp: a}
  }
  function Sin(timeWorker) {
    let s = new SinOsc(timeWorker)
    s.start()
    this.setPit = function(freq) { s.freq(
        freq ) }
    this.setAmp  = function(gain) { s.gain(
        gain ) }
    this.rampPit = function(freq,dur) { s.
        freqTarget( freq,dur ) }
    this.rampAmp = function(gain,dur) { s.
        gainTarget( gain,dur ) }
    this.stop = function() { s.stop() }
    this.ramps = function (f,a,d) {
      this.rampPit(f,d)
      this.rampAmp(a,d)
    }
  }
  let sin = new Sin(this)
  if (withFFT) postMessage("makeFFT()")
  let params = map(datum.value)
  sin.setPit(params.pit)
  sin.setAmp(0)
  while (!datum.done) {
    sin.ramps(params.pit, params.amp, dur)
    yield dur
    if (withSliderDisplay) postMessage("
        move1D("+datum.value+")")
    if (withChart) postMessage("move2D()")
    datum = data.next()
    params = map(datum.value)
  }
  sin.rampAmp(0,0.1)
  yield 0.1
  sin.stop()
  postMessage("finish()")
}
```

Workshop discussions are mostly focused on customizing the above code and demonstrating extensions described later in this report. What follows in the next section is a discussion of the TimeWorkers framework "under the hood." This can be skipped if one's main interest is in customizing sonifications rather than digging into the underlying system.

3. PROGRAMMING STRUCTURE AND SUPPORTING FUNCTIONS

The framework has no dependencies. It is a lightweight project which is Free Open-source Software (FOSS) and has the additional feature of No Internet Required for Development (NIRD). Workshops and individual work are equally possible online and offline, for example, during field work with no connectivity. A project's .html landing page loads a single associated script file, engine.js, which contains all supporting functions. Files and modules are shown schematically in Figure 3.

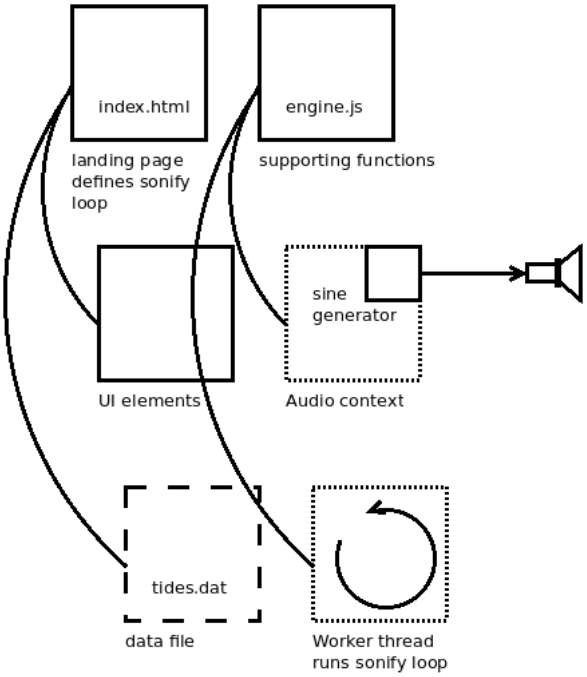


Figure 3: *Structure and modules.*

The project landing page sets up web-related configurations, specifies the user interface (UI), loads the script file, engine.js, and is where the sonification is "composed." Various "hardwired" globals need to be declared which will be communicated to the script file, including a default value for `dataFileName`. Likwise, the script file expects a "hardwired" generator function with the name `sonify` (which should be defined using JavaScript's function* syntax [10]).

Table 1: project files

web landing page	supporting script
index.html	**engine.js**

Table 2: **index.html** elements

<head>	<body>	<script>
<meta> specifies metadata	configures UI elements	sets global and local variables
(optional) <script> loads any auxiliary script files	(options to hide or expose)	loads **engine.js**
e.g., graphing library	e.g., drag and drop	must define function* sonify(data)

Table 3: **engine.js** tasks, **classes** (and optional functionality)

set locals	polish UI	specify web worker(s)	set up spork mechanism	define DSP ugens
audio context	check browser capabilities	**WorkerThread**	**TimeIterator**	e.g., **SinOsc**
data source	get UI elements	uses inline definitions	play / stop	e.g., **FM**
timing cushion	set UI element states	(add graphing capability)	nextEventAt	uses setValueAtTime,
worker arrays	(add drag and drop)	(connect real-time UI elements)	uses async / await	linearRampToValueAtTime

This function instantiates any unit generators (ugens) it will be using, for example with

`new SinOsc(timeWorker)`

as shown above, and specifies data-to-sound parameter mappings which unfold through time.

For brevity's sake the script file, engine.js, is not reproduced here but can be found in `js/` subdirectory of the project repository[11]. This script provides the TimeWorkers structure through its class definitions, functions and own variable settings. Any special tokens which are referenced by the sonify generator function, e.g. `SinOsc` will be resolved against what is defined or declared in the global scope after engine.js has been loaded.

The script file contains several parts. Setting local variables, polishing the UI and a system for "performing" sonifications composed with the sonify generator function.

A *WorkerThread* interface sets up and runs this time-sensitive apparatus in separate threads. The *TimeIterator* class provides a mechanism which waits between events in the sonify generator's loop and compensates for timing jitter. It uses the `performance.now()` clock to compare real time with expected logical time. Finally, the ugen part of the script file defines any synthesis or DSP patches which are used.

```
var context
```

is declared to hold the *window.AudioContext* which gets instantiated at sound start and closed at sound stop,

```
var workerThreads = []
```

is the array containing the pool of `WorkerThread` instances and

```
var uwta = []
```

is a multi-dimensional array (whose name is shorthand for "*ugenWorkerThreadsArrays*") that contains the set of all ugens in all WorkerThreads.

A programming pattern often used in sonification in the Chuck language [4] has two aspects. The first is the `spork` function which calls a given function in a parallel, separate thread with its own logical timebase. (A child process spawned by a sporked function can also spork its own child processes.) The second construct is a means for looping over data, in Chuck this is usually a `while` loop where event time advances each iteration. The loop executes in its own thread. The present framework supports both features using its WorkerThread and TimeIterator constructs.

When `makeWorkerThread` (Table 1) creates a new instance, the spawned JavaScript Worker [12] is of a special inline type (as opposed to the more common type which is usually created by loading a dedicated script file).

```
var blob = new Blob([script])
var worker = new Worker(URL.createObjectURL(
    blob))
```

The script passed into the new Blob sets up a mechanism for dynamic object definition. It calls `addEventListener` on the new worker and sets how the worker will handle incoming messages. By telling it to handle them with an `eval` (in the global scope), the worker's set of variables and functions is literally "grown" by posting message strings to be evaluated which contain the desired definitions and settings. One of these, for example, is the sonify function defined back in the landing page. Dynamically defining timeWorkers in this way allows the sonify function to also spork processes which will become its own new child workers each of which runs in a separate thread.

The `spork` function itself instantiates a time-sensitive data iterator with `makeTimeIterator`. A TimeIterator will pause a generator for a given duration with its method `nextEventAt()` which is an async function utilizing JavaScript's async / await ([13]) pausing functionality. When sporked, a sonify generator's loop is started with `nextEventAt("start")` that executes its first cycle. A subsequent `yield` in the sonification loop will set the amount of time to pause on the next call to `nextEventAt` (which calls itself recursively) and the loop continues.

In the definition below, `fstar` is the sonify generator defined in the landing page and `args` contains a data iterator with the provided data series (which has had its range normalized).

```
function spork(fstar, ...args) {
  let ti = makeTimeIterator()
  ti.sporkScript = fstar.apply( ti, args )
  ti.nextEventAt("start")
}
```

To reiterate, calling `spork` with both a sonify generator and a TimeIterator containing the data as shown

```
spork(sonify,data)
```

will create a pattern comparable to a Chuck-based sonification which consists of essentially the same parts: spork a new thread which sets up a sound source and mapping strategy, and then loops through a conditioned data series, pausing after each data point.

In Chuck, pausing is written using the syntax

```
dur => now;
```

whereas the TimeWorkers equivalent uses

```
yield dur
```

A `yield` in the sonify generator loop invokes a JavaScript `Promise` in the TimeIterator object whose `setTimeout` is set to the duration to `await`.

3.1. SinOsc ugen example

Custom ugens comprise patch definitions made with the Web Audio API's audio nodes. The `makeSinOsc` example shown here instantiates an oscillator with gain control using the API's createOscillator() and createGain() methods[6].

```
function makeSinOsc()
{
  let o = context.createOscillator()
  let g = context.createGain()
  o.type = "sine"
  o.frequency.value = 440
  g.gain.value = 0.1
  o.connect(g)
  g.connect(context.destination)
  g.connect(dac)
  return { osc:o, gain:g }
}
```

The object gets instantiated in a wrapper called `SinOsc` which when instantiated itself with `new` also includes methods to alter its parameters, for example, by changing its frequency with the following custom `freq()` method:

```
    freq: function (hz) {
        let n = this.dsp
        postMessage(ugens+"["+n+"].osc.
            frequency.setValueAtTime("+hz+",
            "+(myThread.now+cushion)+")")
      }
```

`this.dsp` refers to the ugen itself which is held in the main thread's array `ugens[]`. The message posted to the main thread looks up the osc field of the ugen and changes its frequency using the Web Audio API's `setValueAtTime` (which corresponds to the worker thread's "now" plus a constant offset). A full ugen definition comprises instantaneous setters for all parameters, as well as custom time-varying envelopes, for example made with the Web Audio API's `linearRampToValueAtTime`. Note that the patch code also includes a connection from the patch's summing point to a global summing point called `dac`.

Different sound sources can be made available by expanding the library of ugens defined in engine.js. Each would comprise a "make the patch" portion and a wrapper (with the ugen name) which includes the set of parameter altering methods.

3.2. FM patch

For example, a simple two-oscillator FM patch could look like the following:

```
function makeFM()
{
  let mod = context.createOscillator()
  let modGain = context.createGain()
  mod.type = "sine"
  mod.connect(modGain)
  let car = context.createOscillator()
  let g = context.createGain()
  car.type = "sine"
  modGain.connect(car.frequency)
  car.connect(g)
  g.connect(context.destination)
  g.connect(dac)
  let cFreq = 2200
  let index = 33
  let mRatio = .1
  modGain.gain.value = cFreq * index
  mod.frequency.value = cFreq * mRatio
  car.frequency.value = cFreq
  g.gain.value = 0.1
  return { osc:car, gain:g, mod:mod, modGain:
      modGain }
}
```

All ugens need to be accessible in the timeWorker thread in which the sonify loop is running. A last step, then, in ugen creation is to add the ugen wrapper, for example `FM`, to the list of functions which gets dynamically installed inline when a new WorkerThread is instantiated.

4. EXTENSIONS

Changing the sound source, sounding multiple time series and adding graphing capabilities are extensions which complement the basic example described above 2.

4.1. Voicing

Changing to a more interesting sound source is possible in the sonify generator itself. This approach relies on combinations of ugens defined in the engine.js script. Where the basic example uses a single `SinOsc` ugen as its instrument, the example here demonstrates additive synthesis built by summing multiple sines which are harmonically tuned. The new instrument `Harmonics` is defined directy within the sonify generator.

```
function Harmonics(nSins,timeWorker) {
  this.sins = new Array
  for (let i = 0; i < nSins; i++) this.sins.
      push( new SinOsc(timeWorker) )
  this.sins.forEach(function(x) { x.start()
      })
  function fi(f,i) { return f*(i+1) }
  function ai(a,i) { let h = (i+1); let odd =
       (h%2) ? a : a*0.1; return odd/h }
  this.setPitch = function(freq) { this.sins.
      forEach(function(x,i) {x.freq( fi(freq,
      i) )}) }
```

```
  this.setGains = function(gain) { this.sins.
      forEach(function(x,i) {x.gain( ai(gain,
      i) )}) }
  this.freqTarget = function(freq,dur) { this
      .sins.forEach(function(x,i) {x.
      freqTarget( fi(freq,i),dur) }) }
  this.gainTarget = function(gain,dur) { this
      .sins.forEach(function(x,i) {x.
      gainTarget( ai(gain,i),dur) }) }
  this.stop = function() { this.sins.forEach(
      function(x) {x.stop()}) }
  this.ramps = function (f,a,d) {
    this.freqTarget(f,d)
    this.gainTarget(a,d)
  }
}
```

One of these instruments is then instantiated in the sonfication loop, for example, with

```
let vox = new Harmonics(8,this)
```

to create an harmonic series of 8 SinOscs. Given a pitch frequency f `function fi(f,i)` sets their tunings. Amplitude relationships in `function ai(a,i)` create a clarinet-like structure favoring odd harmonics. A convenience function `ramps` is provided which applies frequency and amplitude updates to the entire additive synthesis patch.

The following set of extensions are turned on or off with flags in the index.html file. By default, the withDemo flag is set. Only one option is allowed at a time, so remember to set

```
    withDemo = 0
```

before exploring these others.

4.2. Polyphony from multiple data series

Multiple time series are interesting to sonify at the same time, for example, to hear correlations by ear. Data can be input from two or more separate data files as in this example which combines monthly USA gross domestic product (GDP) from 1969 to 2016 and global CO_2 level for the same period. The curves shown in Figure 4 have been normalized to the same range.

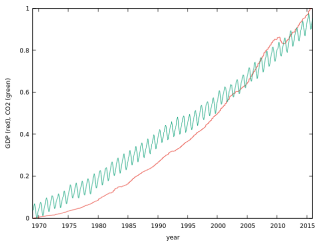


Figure 4: *GDP and CO_2.*

The example landing page, index.html, has a provision for hearing these two playing together, as two independent musical voices. Change the state of `withDemo` and this flag for this to take effect:

```
    withTwoFiles = 1
```

Two data files will now be specified and will spawn two TimeWorker threads both using the single sonify generator as defined. In this example, one can hear details like the 2008 financial downturn and the seasonal flux in global CO_2. Overall, the two quantities follow a coincident rising trend.

4.3. Animated Chart

Similar to the interest in multi-modal data presentation described in [14], sonification in the present framework can be combined with graphing. Chart.js is a FOSS project for interactive plotting in the browser and is integrated into the project by loading a single script file (which can be locally sourced for creating a NIRD environment).

Again, the example landing page, index.html, has a provision for demonstrating this extension by changing `withDemo` and this flag:

```
    withChart = 1
```

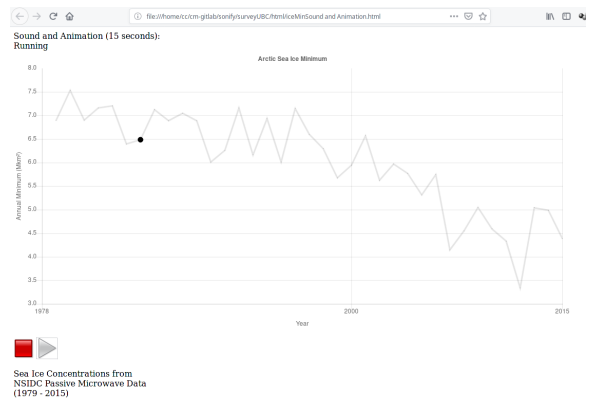


Figure 5: *Simultaneous sound and graph of Arctic Sea Ice Minimum per year.*

Playing the sonification in Figure 5 animates the black dot on the curve. Syncronized sound and animation is accomplished with `postMessage("moveGraph()")` inside the loop in the sonify generator. Each successive call advances the black dot to the next data point in an array of 2D data points that was input from a multi-column data file (columns are year and value).

4.4. Real-time FFT display

Likewise, change `withDemo` and the following flag in the example landing page, index.html, and the sonification's audio output will be displayed as a time-varying spectrum.

```
    withFFT = 1
```

An FFT analyzer computes the spectrum of the global summing point `dac` in real time.

5. CONCLUSIONS

A 40+ year tradition has evolved a well-known pattern for sequencing scores and real-time synthesis in languages like Pla[1], Common Music[3], Chuck[4] and others. The sonify generator's loop is a descendant written in JavaScript. Running in the browser, it allows flexible programming using the full power of the language and can be rapidly experimented with on any browser-equipped system.

Table 4: TimeWorkers framework in terms of goals suggested by *Sonification Report* [7]

attribute	goal	now	soon	never
Portability	consistent		x	
Portability	reliable	x		
Portability	portable across various computer platforms	x		
Portability	moving between real-time and non-real-time sound production		x	
Flexibility	simple controls for altering the data-to-sound mappings	x		
Flexibility	simple "default" methods of sonification that allow novices to sonify their data quick and easily	x		
Integrability	easy connections to visualization	x		
Integrability	easy connections to visualization programs, spreadsheets, laboratory equipment			?
Integrability	standardized software layer			?

Sonifications created using the framwork run equally well on mobile and other smaller systems.

Pla's *voices* are analogous to sonify generator loops because they constitute groups of time-ordered events which can themselves be voices (recall that spork-ed child threads can spork their own children). Other pertinent features of Pla also have bearing on the present framework (these are distilled a 1983 description): "Higher levels of musical control are implemented as *voices* and *sections* ..." "...notes that somehow belong together are grouped under the rubric of a voice." "Arbitrarily large groups of voices can be organized into a *section,* which then becomes nearly equivalent to a voice." "Another kind of grouping is based on voices... voices can create other voices to any level of nesting."

Common Music's similar features involve multiple types: "*Thread – A collection that represents sequential aggregation.* A single timeline of events is produced by processing substructure in sequential, depth-first order." "*Merge – A collection that represents parallel aggregation, or multiple timelines.* A single timeline of events is producted by processing substructure in a scheduling queue." "*Algorithm – A collection that represents programmatic description.* Instead of maintaining explicit substructure, a single timeline of events is produced by calling a user-specified program to create new events."

The TimeWorkers framework described here offers a way to construct the above relationships in browser-based platforms and offers solutions for some, but not all of the goals cited in *Sonification Report* [7]. Table 4 lists the boxes it checks off.

In the future, faster-than-sound soundfile writing will be directly supported though for now, file output is only by browser sound capture plug-ins (which run in real time). Faster-than-sound is a highly-desirable feature and is something that's been supported in both Common Music and Chuck. Regarding the former, "Realization in Common Music can occur in one of two possible modes: run time and real time. In run-time mode, realized events receive their proper 'performance time stamp,' but the performance clock runs as fast as possible. In real-time mode, realized events are stamped at their appropriate real-world clock time." For the latter, Chuck's "silent mode" is the equivalent.

The recently standardized AudioWorklet [15][1] will be integrated into the framework in the coming months. Of particular interest is another recently proposed enhancement to Web Audio to support multi-channel output.

Also for the future, direct real-time sonification from live sensor data can be contemplated. This important feature opens up applications such as bio-feedback [16] or other kinds of feedback such as providing real-time "cracking" sounds to operators of fracking pumps (where presently feedback is provided after the fact and one can imagine the problems resulting from the over-stimulation of shale gas wells). It has become vital in medical applications, even making inroads on traditional treatment practices in cases where listening to data provides equal or better sensitivity and specificity compared to visual means. The brain stethoscope, for example, allows rapid detection of non-convulsive seizures by non-specialists. [17]

Interest in sonification is burgeoning as sensors and data collections become an increasingly ubiquitous part of daily life. Employing well-known sound generation techniques from computer music, sonification can play a role in the work of domain experts and students in sciences and arts, as well as for general communication.

[1] As of this writing, only the Chromium browser family supports AudioWorklet. It is expected soon in Firefox at which point the integration work will commence.

6. REFERENCES

[1] Bill Schottstaedt, "Pla: A composer's idea of a language," *Computer Music Journal*, vol. 7, no. 1, pp. 11–20, 1983.

[2] David Wessel, Pierre Lavoie, Lee Boynton, and Yann Orlarey, "Midi-lisp: A lisp-based programming environment for midi on the macintosh," in *Audio Engineering Society Conference: 5th International Conference: Music and Digital Technology*, May 1987 (accessed February 2, 2019), http://www.aes.org/e-lib/browse.cfm?elib=4659.

[3] Heinrich Taube, "An introduction to common music," *Computer Music Journal*, vol. 21, no. 1, pp. 29–34, 1997.

[4] Ge Wang, Perry R. Cook, and Spencer Salazar, "Chuck: A strongly timed computer music language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, 2015.

[5] Moz://a MDN web docs, *Using Web Workers*, 2019 (accessed February 6, 2019), `https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers`.

[6] Moz://a MDN web docs, *Web Audio API*, 2018 (accessed December 16, 2018), `https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API`.

[7] Bruce Walker Terri Bonebright Perry Cook Kramer, C. and John H. Flowers, *Sonification Report: Status of the Field and Research Agenda*, 2018 (accessed December 16, 2018), `http://digitalcommons.unl.edu/psychfacpub?utm_source=digitalcommons.unl.edu%2Fpsychfacpub%2F444&utm_medium=PDF&utm_campaign=PDFCoverPages`.

[8] Chris Chafe, *Music 220A*, 2019 (accessed February 6, 2019), https://ccrma.stanford.edu/courses/220a/.

[9] Honchan Choi, *Web Audio API eXtension*, 2019 (accessed January 28, 2019), http://hoch.github.io/WAAX/.

[10] Moz://a MDN web docs, *Iterators and generators*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Iterators_and_Generators.

[11] Chris Chafe, *project software repository*, 2018 (accessed December 16, 2018), https://cm-gitlab.stanford.edu/cc/sonify.

[12] Moz://a MDN web docs, *Worklet*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/API/Worklet.

[13] Moz://a MDN web docs, *async function*, 2018 (accessed December 16, 2018), https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function.

[14] Tianchu (Alex); Tomlinson Brianna; Walker Bruce N. Kondak, Zachary; Liang, "Web sonification sandbox - an easy-to-use web application for sonifying data and equations," *Proceedings of 3rd Web Audio Conference*, 2017.

[15] Hongchan Choi, *Audio Worklet Design Pattern*, 2018 (accessed December 16, 2018), https://developers.google.com/web/updates/2018/06/audio-worklet-design-pattern.

[16] Jan-Torsten Milde Baumann, Christian and Johanna Friederike Baarlink, *Body Movement Sonification using the Web Audio API*, 2018 (accessed December 16, 2018), https://webaudioconf.com/demos-and-posters/body-movement-sonification-using-the-web-audio-api/.

[17] Josef Parvizi, Kapil Gururangan, Babak Razavi, and Chris Chafe, "Detecting silent seizures by their sound," *Epilepsia*, vol. 59, no. 4, pp. 877–884, 2018.

SEQUOIA: A LIBRARY FOR GENERATIVE MUSICAL SEQUENCERS

Chris Chronopoulos

Independent Developer
Cambridge, MA
chronopoulos.chris@gmail.com

ABSTRACT

Sequoia is a new software library for musical sequencing, with generative capabilities and sample-accurate timing. The architecture supports a variety of techniques, including polymetric sequencing, clock division, probability, and other parameters which can be manipulated in real time – or even sequenced themselves. The core library is written in C and supports JACK MIDI; Python bindings are also available.

1. MOTIVATION

In recent years, the electronic music community has shown a growing interest in the use of standalone hardware units, both for studio production and live performance [1]. Among their many appeals, these devices have the advantage of being modular - drum machines, synthesizers, samplers, sequencers, mixers, and effects units can be connected and re-connected in myriad ways to accomodate a variety of workflows. Each component serves a unique role and interfaces with other components through well-defined interfaces: line-level audio, and control signals typically in the form of MIDI or CV (control voltage).

The Linux audio ecosystem is well-poised to emulate this paradigm in software; audio routing libraries like JACK, and control signal protocols like MIDI and Open Sound Control (OSC) provide a framework for connecting standalone applications into software "rigs" suitable for composition and performance alike. Indeed, such modularity is central to the Unix philosophy: programs should "do one thing and do it well" [2]. True to form, numerous drum machines (e.g. hydrogen, drumkv1), synthesizers (zynaddsubfx, amsynth, dexed), samplers (shuriken, qsampler, petri-foo, sooperlooper), mixers (jackmixer, non-mixer), and effects (calf-plugins, guitarix) are available from popular Linux repositories. Additional utilities exist for managing audio/MIDI connections (qjackctl, catia/claudia/carla) and saving/restoring sessions (lash/ladish/nsm/aj-snapshot).

Sequencers, however, are comparatively absent from this ecosystem. Perhaps the best-established example is seq24 [3], which, albeit stable and relatively comprehensive, has not been significantly updated since 2010, and suffers from usability issues which hinder on-the-fly composition. Various sequencers exist within larger DAW applications like Ardour [4], LMMS [5], Qtractor [6], Rosegarden [7], and Muse [8], but these don't fit into the modular paradigm described here. Furthermore, the predominant interface for these software sequencers is the *piano roll*, which is well suited for editing live data captured from a MIDI controller, but less appropriate for the quick manipulation of drum patterns and arpeggios typical of dance music. For this task, a traditional *step sequencer* is desired.

But step sequencers can be quite complex. They typically feature live sequence composition, real-time manipulation, and chaining of sequences. More advanced examples include generative properties like probability, ratcheting, and meta-sequencing, in addition to step-wise parameters like microtiming and control variable modulation. With such a wide variety of features, it can be challenging to design applications which cover all the bases – but this is primarily a problem of interface design. The essentials of modern sequencing – timing, synchronization, live manipulation, etc. – can be separated from the problem of application design, and distilled into a general-purpose library, as in the "model-view-controller" paradigm [9] This is the motivation for *Sequoia*.

Figure 1: *A Sequoia session is connected to two different client applications using JACK. Here ZynAddSubFX (zyn-fusion) and drumkv1 are being used to create a simple beat. Carla is used to manage audio and MIDI connections.*

2. DESIGN

The architecture of Sequoia is based on four object classes: *session*, *sequence*, *trigger*, and *port*.

A *sequence* is a discrete series of events which steps in time with a metronome. In this sense, Sequoia is a "step sequencer", but events are not required to be evenly spaced in time (see Section 4.1). The length of a sequence is the number of steps that the sequence contains. There is no limit (aside from memory) to the length of a sequence, but once specified (via instantiation), it is fixed. This is less of a constraint than it may seem, however, as sequences can be chained together and "meta-sequenced" dynamically 5.3. Sequences have several dynamic parameters: the mute state, transpose, clock division, playhead position, playhead direction, and loop boundaries can all be modified live during playback.

Triggers (or "trigs" for short) are the event objects which may populate the steps of a sequence. They store information depending on their type; the current trigger types are:

- **Null:** (an empty trig)
- **Note:** *note value, velocity, length*

- **CC:** *number, value*

Each trigger also carries a *channel number*, a *probability* and a *microtime*. Microtime is a floating-point value in the range $[-0.5, 0.5)$, where the units are in steps. Thus a trigger can be placed half a step before or after its nomimal timing, allowing for irregular rhythms, "humanization", and swing.

Sequences run within a Sequoia *session*, which controls the tempo and transport (start/stop/pause) state applied to all contained sequences. A session can have a number of *ports* for communicating with other applications – including other Sequoia sessions. The ports can be input ("inports") or output ("outports"), have descriptive names, and can be assigned to sequences individually, or on a many-to-one basis. For example, we may have 4 sequences (kick, snare, closed hat, open hat) feeding into a single outport called "drums", while another melodic sequence feeds into an outport called "synth" – all sequencing in time within the same session.

3. API

Sequoia is implemented as a C library in the "object-oriented" style: data structures are presented as custom types with associated methods for instantiation and mutation. All library functions and data types are prefixed with `sq_*`. The full API is documented on the associated GitHub wiki; here we present a simple example which constructs and plays a 2-note sequence:

```
#include "sequoia.h"

#define STEP_RES 256

int main(void) {

  sq_session_t sesh;
  sq_session_init(&sesh, "My Session",
      STEP_RES);

  sq_sequence_t seq;
  sq_sequence_init(&seq, 16, STEP_RES);

  jack_port_t *port;
  port = sq_session_create_outport(&sesh,
            "My Port");
  sq_sequence_set_outport(&seq, port);

  sq_trigger_t trig;
  sq_trigger_init(&trig);

  sq_trigger_set_note(&trig, 60, 100, 4);
  sq_sequence_set_trig(&seq, 0, &trig);
  sq_trigger_set_note(&trig, 67, 100, 4);
  sq_sequence_set_trig(&seq, 8, &trig);

  sq_session_add_sequence(&sesh, &seq);
  sq_session_set_bpm(&sesh, 120);
  sq_session_start(&sesh);

  return 0;

}
```

Here, `STEP_RES` is the step resolution, in ticks per step. This needs to be the same for all sequences in the session – attempting to add a sequence with incompatible step resolution will result in an error. We create an outport for the session called "My Port" and set the sequence to output events through it. We then create a placeholder trigger object `trig` and use it to populate the sequence. Finally, we add the sequence to the session, set the BPM, and start sequencing.

3.1. Python Bindings

The main C library is augmented with Python bindings which obey a direct mapping between classes and methods. In Python, the example above could be written as:

```
import sequoia as sq

STEP_RES = 256

sesh = sq.session("My Session", STEP_RES)
seq = sq.sequence(16, STEP_RES)
port = sesh.create_outport("My Port")
seq.set_outport(port)

trig = sq.trig()

trig.set_note(60, 100, 4)
seq.set_trig(0, trig)
trig.set_note(67, 100, 4)
seq.set_trig(8, trig)

sesh.add_sequence(seq)
sesh.set_bpm(120)
sesh.start()
```

4. IMPLEMENTATION

A Sequoia session registers as a JACK external client whose name is the session name (specified during instantiation). Input and output ports are created as JACK MIDI ports (also named) which are served by the JACK processing callback. The API is compiled into a shared library plus header files, and can be installed e.g. in /usr/local/ for dynamic linking across multiple applications.

4.1. Timing

Timing is managed by the JACK processing thread as it executes within the context of the Sequoia session. The session keeps track of the frame count as it works to fill the JACK buffer with time-stamped MIDI events. Events are managed by the sequences which handle time as a grid of *microticks* – intervals of time much shorter than the step length which enable the microtiming functionality of the sequencer. In the code example in Section 3, the mictrotiming resolution is set to 256 ticks per step. In theory, this resolution can be set much higher, though in practice, it will be limited by CPU performance. The number of frames per tick (fpt) is:

$$\text{fpt} = 15 * \frac{\text{sr}}{\text{tps} * \text{bpm}} \tag{1}$$

where **sr** is the sample rate, **tps** is the step resolution (ticks per step), and **bpm** is the tempo in beats per minute. At 48 kHz with 256 ticks-per-step, there are 23 frames-per-tick at 120 BPM. At 4096

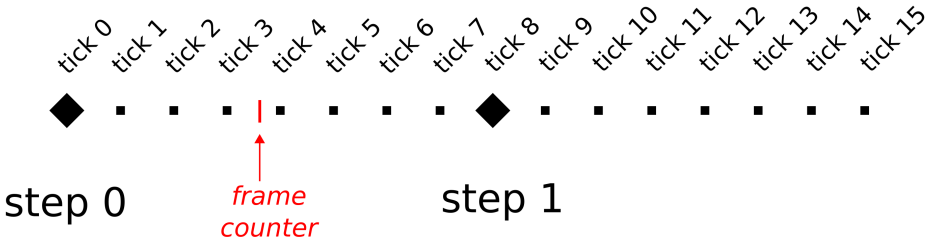


Figure 2: *Diagram visualizing the 3-tiered timing scheme used by Sequoia. At the highest level there are steps: 4 steps per beat (in the sense of "beats per minute"), and one trig per step. Going down one level, each step is composed of several "microticks" which comprise the grid for microtiming events. Here, only 8 microticks per step are shown for clarity, but a typical sequence may have 256 (or more) ticks per step. Finally, there is the frame counter, which sweeps between the microticks until it reaches a tick boundary, at which point a trigger may be fired.*

ticks-per-step, this becomes 1 frame-per-tick, which is the theoretical maximum resolution for this tempo and sample rate.

4.2. Trig-to-Microtick Translation

Although the fundamental timing grid is managed at microtick resolution, this implementation detail is hidden from the user by the trig interface. The user manages the sequence data by setting its trigs (one for each step); these trigs are then placed on the microgrid according to their microtiming. The formula is:

$$\text{tick index} = (\text{step} + \mu\text{time}) * \text{tps} \tag{2}$$

At this tick index, we place a pointer to the trig, which allows us to look up both the trig parameters (e.g. probability, length) and the sequence parameters (e.g. mute, transpose) at trig time, to ensure that we send the correct MIDI event at the correct time.

4.3. Note-Off

While note-on and control change events are recorded in the microgrid at composition time (i.e. when the user calls `sq_sequence_set_trig()`), note-off events are managed differently. To see why, consider what would happen if a C note of length 4 steps was recorded in the microgrid as a C-note-on plus a C-note-off 4 steps later. Now imagine if the sequence transpose parameter were changed in the middle of that note. The note-off would be delivered for the wrong note value, and the synthesizer downstream would be left with a hanging note. The same applies for playhead manipulation, or any number of the other sequence parameters which support live control.

The solution is to implement for each sequence a separate ring buffer, specifically for note-offs, which is always running forward. The length of this buffer is the maximum note length, which is also the length of the sequence. The buffer gets populated with a note-off (at the appropriate delay) whenever a note-on fires. When the note-off is reached by the advancing buffer pointer, it is fired, and then removed from the buffer. When a sequence (or the session) is stopped, we can optionally call a "clean" command, which sweeps through the off-buffer as quickly as possibly, delivering all remaining note-offs.

4.4. Lock-Free Parameter Control

In a running Sequoia session, the JACK thread needs immediate access to data that other threads (e.g. the UI thread) can manipulate during playback. In a non-realtime application, this would be accomplished with mutex locks [10], but in realtime audio, this is unacceptable – the audio callback must never execute code that could block for an indeterminate amount of time [11]. In lieu of mutex locks, we synchronize data between threads via lock-free message queues. For this, we use `jack_ringbuffer_t` as offered by the JACK API. We then implement a simple messaging protocol that allows for the UI thread to "set" or "get" critical data when the audio thread enters the processing callback. This allows both threads to access the data while avoiding any race conditions.

Message queuing offers a clean solution when the audio thread is running, but it can present problems when the system is in a dormant state. In this situation, for example, a queueing "getter" method would block indefinitely, waiting for the processing callback to serve the request. As another example, a user will commonly populate a sequence with trigs before adding it to a running session. If the sequence length is longer than the message queue, this would overflow the buffer and cause an error.

Ideally, the getters and setters would access data directly when operating on a dormant structure, and use message queues when the sequencer is running. In Sequoia, this branching behavior is handled automatically – the data access methods are polymorphic according to the running state of the system.

5. GENERATIVE TECHNIQUES

In addition to serving as a streamlined API for general-purpose, time-critical sequencing with real-time control, Sequoia has been designed from the ground-up with generative music techniques in mind. Here, we describe just a few of these possibilities which Sequoia enables.

5.1. Polymeter

Since there's no concept of a global step counter in Sequoia (only the per-tick frame counter managed by the session), sequences are free to run in and out of phase with each other, according to the least-common-multiple of their lengths. For example, a 16-step sequence played against a 15-step sequence will evolve through 240 steps of variation before syncing back up and repeating itself.

5.2. Probability

Trig parameters include *probability*, a floating-point value in the range $[0, 1]$ which determines what fraction of the time a trig actually fires. This applies to both note-type and CC-type triggers.

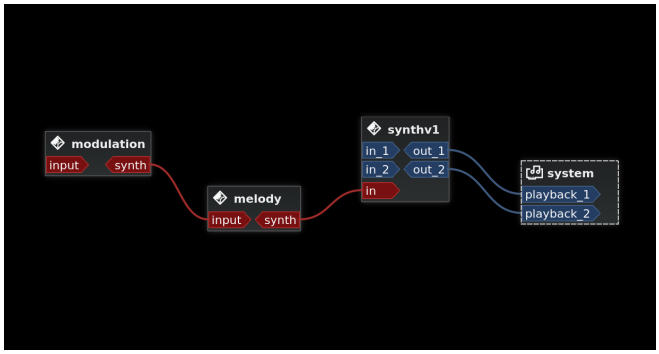


Figure 3: *Meta-sequencing. A Carla patch showing a slow modulation sequence controlling the transpose parameter of a melody sequence, which is driving the synthv1 synthesizer.*

5.3. Meta-sequencing

Meta-sequencing, simply put, is "sequences sequencing sequences". Any of the sequence parameters – playhead, loop start, loop stop, playback mode, transpose, mute state, clock divide – can be controlled live from Sequoia's MIDI-in ports. The way MIDI events map to parameter controls is determined by a mapping defined by the user upon sequence creation.

Combined with the concepts described above, this technique can be very powerful – a single, monophonic sequence can be manipulated by another (perhaps employing polymeter, probability, or clock division) to generate a much longer, stochastically evolving sequence (see Figure 3). Sequences can even be looped back into themselves to give surprising results (Figure 4) – although care must be taken in this case to avoid runaway conditions.

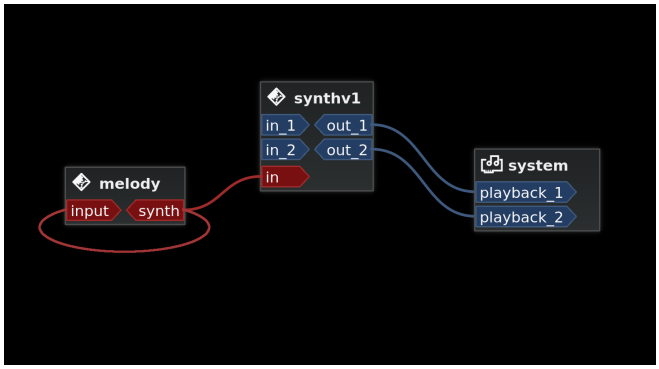


Figure 4: *Auto-sequencing. A melody sequence is fed back into itself (notice the looped-back red line from synth to input on the melody client), and the result is used to drive synthv1. Depending on the melody and the input mapping, this situation can "run away" to infinite pitch. If it doesn't, the results can be a surprising transformation of the original melody.*

5.4. Algorithmic Control

Obviously, the facility of inports and controller mappings allows for external clients (e.g. Python scripts, Pure Data patches, Geiger counters with USB connections...) to control sequence parameters in any way one might wish, thus allowing a huge variety of algorithmic methods to modulate the sequencer.

6. STATUS

Sequoia is currently in active development. The core library (libsequoia) is in a viable state, and the source code is available on GitHub under the GPL license (v3) [12]. We are also in the process of embedding the library within Ziggurat, an existing GUI sequencer application [13]. Future work will focus on developing bindings to other languages, and improving documentation.

7. REFERENCES

[1] Connor Jones, *A Live Performance Revolution is Taking Over Electronic Music*, 2016.

[2] Peter Salus, *A Quarter Century of UNIX*, Addison-Wesley, 1994.

[3] Wikipedia contributors, *Seq24*, 2019.

[4] Wikipedia contributors, *Ardour (software)*, 2019.

[5] Wikipedia contributors, *LMMS (software)*, 2019.

[6] Wikipedia contributors, *Qtractor*, 2019.

[7] Wikipedia contributors, *Rosegarden*, 2019.

[8] Wikipedia contributors, *MusE*, 2019.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Pearson Education, 1994.

[10] Michael Kerrisk, *The Linux Programming Interface*, No Starch Press, 2010.

[11] Ross Bencina, *Real-time audio programming 101: time waits for nothing*, 2011.

[12] Chris Chronopoulos, *https://github.com/chronopoulos/libsequoia*, 2018.

[13] Chris Chronopoulos, *https://github.com/chronopoulos/ziggurat*, 2018.

A JACK SOUND SERVER BACKEND TO SYNCHRONIZE TO AN IEEE 1722 AVTP MEDIA CLOCK STREAM

Christoph Kuhr

Anhalt University of Applied Sciences
Köthen, Germany
christoph.kuhr@hs-anhalt.de

Alexander Carôt

Anhalt University of Applied Sciences
Köthen, Germany
alexander.carot@hs-anhalt.de

ABSTRACT

This paper presents the evaluation of a media clocking scheme in an AVB network segment. The JACK audio connection kit on each AVB processing server is synchronized to an IEEE 1722 media clock stream, as well as each UDP Soundjack receiver on each AVB proxy server. Thus, the transmission of each packet of an audio stream is bound to the transmission interval of the media clock stream and each participant is able to recover the same media clock. In this paper we present the evaluation of this media clocking scheme and the JACK client synchronization with the AVB network segment at hand.

1. INTRODUCTION

Soundjack [1] is a real-time communication software using peer to peer connections, to connect up to five participants to each other. This software was designed as a tool for musicians and was first published in 2006 [2]. The interaction with live music over the public Internet is very sensitive to latencies, both round trip as well as one-way. Thus, this application is mainly concerned with the minimization of latencies as well as jitter.

1.1. fast-music and Soundjack

In cooperation with the two companies GENUIN [3] and Symonics [4], a rehearsal environment for conducted orchestras via the public Internet is under development as the goal of the research project fast-music. Up to 60 musicians and one conductor, who are randomly distributed throughout Germany, shall be able to play together live. The central node represents the multimedia signal processing server network under investigation, which ideally will be located in Frankfurt on the Main, since it is the largest Internet exchange node in Germany it promises the smallest round trip latencies.

1.2. Concept for a Real-time Processing Server Network

The basic signal processing functionality of the server network connects up to 60 UDP streams to each other and mixes them. A single server could easily handle mixing this amount of concurrent UDP streams with reasonably low latency, but for future research in the application of immersive audio technologies in real-time, a single server is not sufficient to handle the computational load of 60 individual audio and video streams. Thus, a scalable infrastructure is chosen to provide such signal processing capacities. The signal processing provided by the Soundjack server network involves mixing algorithms for audio and video streams. As an infrastructure for the audio signal processing stage, the JACK [5] audio server is deployed. JACK is a professional and open source audio server, that allows applications to share sample accurate audio data with each other. A large number of signal processing applications and algorithms are available for JACK. Details on the mixing application can be found in [6].

Another benefit of such a scalable approach is the minimization of service times of network packets, which is the time a packet requires to travel on the wire until it is fully held in the input buffer of the servers network interface. During the service time of a single network packet, no concurrent packets can be processed, which may introduce some hold time in the upstream buffer of each concurrent stream, adding to the overall round trip time. The reduction is not significant. The test environment considered in this paper is the Ethernet based campus network of the university.

A detailed description of the first design of the software architecture and operating system configuration can be found in [7]. Recent findings however, have revealed the first design to be flawed and not fully capable of providing the required features. A new software architecture is under development. The results presented in this paper however, are not influenced by the rework of the software architecture since the JACK server is running independently.

1.2.1. Audio Video Bridging - an Open Standard Solution

Audio Video Bridging / Time-Sensitive Networking (AVB/TSN) describes a set of IEEE 802.1 standards that operate on layer two of the OSI model [8]. These standards enable computer networks to handle audio and video streams in real-time. Operating only on OSI layer two, AVB is not routable. It is defined for local network segments only.

- IEEE 802.1AS [9]
  Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks (referred to as gPTP)
- IEEE 802.1Qat [10]
  Virtual Bridged Local Area Networks - Amendment 14: Stream Reservation Protocol (SRP)
- IEEE 802.1Qav [11]
  Virtual Bridged Local Area Networks - Amendment 12: Forwarding and Queueing Enhancements for Time-Sensitive Streams (FQTSS)
- IEEE 1722 [12]
  IEEE Standard for Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks (referred to as ATVP)
- IEEE 1722.1 [13]
  IEEE Standard for Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks (referred to as AVDECC)

The AVB standards are extensions for generic Ethernet networks providing precise synchronization, resource reservation and bandwidth shaping. Lower latencies and jitter, the avoidance of packet bursts and bandwidth shortage are addressed, providing real-time responsiveness to a computer network. These properties are used to ensure a constant streaming with low latency and jitter inside the Soundjack server network. Thus, the Soundjack client streams can be processed inside the server network, without interfering with each other.

AVB networks require special hardware for timestamping Ethernet frames with separate transmission queues for each traffic class, i.e. AVB traffic with Stream Reservation (SR) classes A/B and generic Ethernet traffic. The IEEE 802.1-2014 [14] standard defines the two stream reservation (SR) classes A and B. Both classes are used in an SRP domain to differentiate audio and video traffic from other Ethernet traffic. For SR class A, SRP reserves resources on all switch ports along the path from talker to listener to maintain a transmission interval of 125 μs (250 μs for SR class B). The implications of the transmission interval are discussed in section 2.

1.2.2. Network Synchronization with gPTP

The precise synchronization of different devices spread throughout a local area network requires a specialized protocol, i.e. PTP, which involves several steps. Each time a gPTP capable device appears on the network segment, a negotiation for the grand master role is triggered. The best master clock algorithm (BMCA) compares the clock information in announce messages, that are broadcasted by each PTP capable device on the same clock domain. A clock domain is a part of a network segment that is synchronized to the chosen grand master clock, it is separated by devices or Ethernet bridge ports that are not gPTP capable (gPTP is a special profile [9] for PTP [15]). Each gPTP capable Ethernet bridge port has a mode of its own, either master or slave. The Ethernet port of the AVB device running the grand master clock is in master mode and is the root of the hierarchical clock distribution. The bridge port of the AVB switch it is connected to, is in slave mode. It receives clocking information rather then sending it. Since the switch receives its gPTP clock from this bridge port in slave mode, all its other bridge ports are in master mode. They distribute the clock information of the grandmaster clock to the next hop or AVB device.

After this election process, the clock domain needs to be synchronized. This is achieved in two steps: Syntonization, and Offset and Delay Measurement. In the first step "SYNC" messages are send from the master to the slave port followed by a "Follow_Up" message, which includes a timestamp taken close to the media (physical layer) of the sender. Both messages are used to adjust the frequency of the slave to the master clock. The second step involves "Pdelay_Req" and "Pdelay_Resp" messages and measures the absolute time offset between the master clock and slaves local clock. The slave port adjusts its local clock to match the master clock. After this procedure each network device is synchronized to the grandmaster clock, matching its phase and frequency. For the exact mechanisms and calculations see [9] and [16].

1.2.3. Control Messages and SO_TIMESTAMPING

The CMSG macros are used by the operating system to create and access control messages, which are also called ancillary data, that are not provided by the generic payload of a raw Ethernet socket. This additional control information includes among other things the receiving interface, optional header fields, extended error description or a set of file descriptors. Ancillary data is sent with `sendmsg()`, received with `recvmsg()` and is stored as a list of `struct cmsghdr structures` with data appended to it. The use case at hand is to receive the hardware timestamp of the arrival of each AVTP packet.

The userspace interfaces to receive timestamped network packets are the following [17]:

- `SO_TIMESTAMP`:
  Generate timestamp with **microseconds** resolution for each incoming packet using the **system time**.
- `SO_TIMESTAMPNS`:
  Generate timestamp with **nanoseconds** resolution for each incoming packet using the **system time**.
- `SO_TIMESTAMPING`:
  Generate timestamp with **nanoseconds** resolution for each incoming packet using the **network hardware**.

The `SO_TIMESTAMPING` interface has to be configured on the raw Ethernet socket with `setsockopt()` and the appropriate flags have to be chosen from the following:

1. Determine how timestamps are generated with `SOF_TIMESTAMPING_TX/RX`:
   - `SOF_TIMESTAMPING_TX_HARDWARE`:
     Hardware transmission timestamp.
   - `SOF_TIMESTAMPING_TX_SOFTWARE`:
     Fallback in case of failure of `SOF_TIMESTAMPING_TX_HARDWARE`.
   - `SOF_TIMESTAMPING_RX_HARDWARE`:
     Original, unmodified reception timestamp, generated by the hardware.
   - `SOF_TIMESTAMPING_RX_SOFTWARE`:
     Fallback in case of failure of `SOF_TIMESTAMPING_RX_HARDWARE`.
2. Determine how timestamps are reported in the control messages with `SOF_TIMESTAMPING_RAW/SYS`:
   - `SOF_TIMESTAMPING_RAW_HARDWARE`:
     Return raw hardware timestamp.
   - `SOF_TIMESTAMPING_SYS_HARDWARE`:
     Return hardware timestamp converted to the system time. The correlation between the transformed hardware timestamps and the system time is as good as possible, but not perfect. Requires support by the network device and will be empty without that support.
   - `SOF_TIMESTAMPING_SOFTWARE`:
     Return software timestamp.

In addition to the `setsockopt()`, it is necessary to initialize the device driver to do hardware timestamping with an `ioctl()`-call to `SIOCSHWTSTAMP`. The `ioctl()` has to be called with the argument:

```
struct hwtstamp_config {
    int flags;
    int tx_type;
    int rx_filter;
};
```

Possible values for `hwtstamp_config->tx_type` are:

- `HWTSTAMP_TX_OFF`:
  Deactivate hardware timestamping for outgoing packets.
- `HWTSTAMP_TX_ON`:
  Activate hardware timestamping for outgoing packets is turned on. The sender decides which packets are to be time stamped.

Possible values for `hwtstamp_config->rx_filter` are:

- `HWTSTAMP_FILTER_NONE`:
  Deactivate timestamping for incoming packet.
- `HWTSTAMP_FILTER_ALL`:
  Activate timestamping for any incoming packet.
- `HWTSTAMP_FILTER_SOME`:
  Activate timestamping all requested packets plus some more.
- `HWTSTAMP_FILTER_PTP_V1_L4_EVENT`:
  PTP v1, UDP, any other event packet.

1.2.4. Hardware Configuration

Two server types with real-time capabilities are designed for the Soundjack server network, an AVB proxy server and an AVB processing server. Both server types are running on a x86_64 architecture with eight physical cores and are equipped with an Intel I210 network interface card [18]. A open source driver stack that is required to compile the kernel module (`igb_avb.ko`) with AVB support is available at Github [19]. The gPTP daemon, which is used in this setup, is also provided by this repository. All AVB servers of both types are registered for a media clock stream, which is supplied by an XMOS development board manufactured by Atterotech [20].

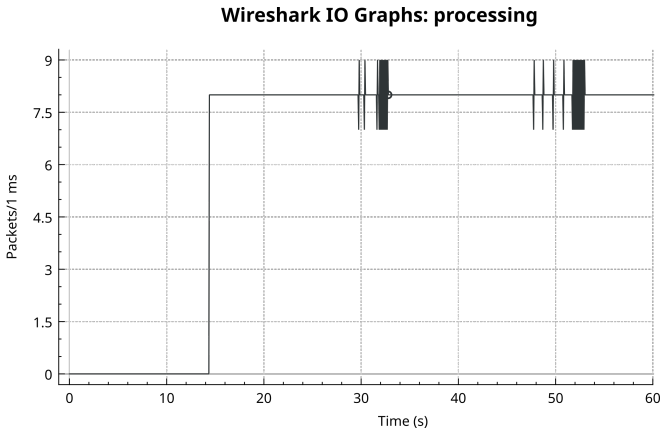


Figure 1: *Packet rate of the IEEE 1722 AVTP media clock stream originating from the XMOS talker. The MRP client of the JACK media clock backend has established the connection to the XMOS talker after 12 seconds. The figure is enhanced and clipped at 60 seconds to show the anomalies (packet rates of 7 and 9 packets per millisecond) between around 30 and 50 seconds.*

2. IEEE 1722 AVTP MEDIA CLOCK SYNCHRONIZATION CONCEPT FOR THE JACK AUDIO CONNECTION KIT

The signal processing concept is designed for a completely digital signal chain, i.e. neither analog-digital (ADC) and nor digital-analog converters (DAC) are present. Without the local clock of an ADC the processing server would have no media clock source to synchronize to. Consequently, it is not possible to adjust the local clock to match the gPTP grandmaster clock. With a media clock stream as clock source, no additional hardware besides the network interface is required. The media clock stream maintains a constant media clock originating from a gPTP derived word clock of the ADC on the XMOS development board. The ADC of the XMOS development board is running at a sampling rate of $48\ kHz$ and is configured as an AVB talker. It automatically acknowledges any connection request of a listener, without the use of IEEE 1722.1 ACMP. The different clock source concepts are explained in [16] in detail.

2.1. Packet Rate and Padded AVTP Packets

The transmission interval of $125\ \mu s$, that is defined by the SR Class A, has the same constant transmission interval for higher sampling rates as well. Instead of sending packets in a shorter interval, the amount of samples per packet is adjusted. For a sampling rate of $48\ kHz$ six samples per audio channel are written to an AVTP packet (12 and 24 samples for $96\ kHz$ and $192\ kHz$ respectively):

$$125\ \mu s = \frac{6 \text{ samples}}{48\ kHz} \Rightarrow 8 \text{ packets per millisecond} \quad (1)$$

This way the transmission interval can maintain the media clock of the talker for the listener to recover. Figure 1 shows the packet rate of 8 packets per millisecond of the media clock stream originating from the XMOS development board. Figure 2 shows the probability distributions of the transmitted AVTP packets of the media clock stream, measured on the processing server with hardware packet arrival timestamps. The calculated mean value of $124997\ ns$ and standard deviation of $309.35\ ns$ meet the defined transmission interval for a SRP class A domain of $125\ \mu s$ perfectly.

In section 3 we will evaluate the three JACK period sizes of 32, 64 and 128 samples. The remaining samples of a JACK period, that occur since six (samples per AVTP packet) is not an integer divisor of either 32, 64 nor 128, are calculated in equation (2):

$$\left\lceil \frac{N \text{ samples per JACK period}}{6 \text{ samples per AVTP packet}} \right\rceil = k \text{ packets per JACK period} \quad (2)$$

Samples	AVTP Packets
32	$\lceil \frac{32}{6} \rceil = \lceil 5 + \frac{1}{3} \rceil = 6$
64	$\lceil \frac{64}{6} \rceil = \lceil 10 + \frac{2}{3} \rceil = 11$
128	$\lceil \frac{128}{6} \rceil = \lceil 21 + \frac{1}{3} \rceil = 22$

Table 1: *Samples and packets per JACK period*

This means that for 32 samples per period every 6th AVTP packet carries a fraction of the six samples, in this case $1/3 = 2$ samples, and the remaining four samples are padded with zeros - for 64 samples every 11th packet has four samples, the rest is padded with zeros and for 128 samples every 22th packet has two samples and the rest is padded with zeros.

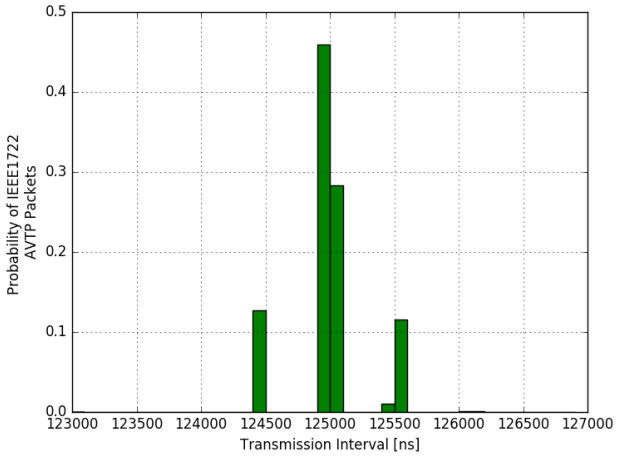


Figure 2: *Probability distribution function of the IEEE 1722 AVTP media clock stream originating from the XMOS talker. The measurement shows the network hardware receive timestamps on the server side.*

2.2. AVB Listener as JACK Media Clock Backend

The media clock listener is the same as in the AVB server implementation [7] and is integrated by a C++ wrapper that was inspired by Netjack [21], i.e. only the `Read()` (and `Write()`, which is required for proper operation) member functions are used to advance the JACK server according to the configured sample rate. As an additional configuration, the JACK AVTP backend is required to run a dummy stereo channel setup, because JACK clients could not be activated otherwise.

```
int init_1722_driver(
    IEEE 1722_avtp_driver_state_t *IEEE 1722mc,
    const char* name,
    char* stream_id,
    char* destination_mac,
    int sample_rate,
    int period_size,
    int num_periods
)
```

Called with the appropriate arguments, the initialization procedure starts a MRP thread, which takes care of the resource reservations for the media clock listener. After the Listener has established the path to the media clock talker and the JACK server has started, the backends' `Read()` member function calls the wrapped procedure:

```
uint64_t media clock_listener_wait_recv_ts(
    FILE* filepointer,
    IEEE 1722_avtp_driver_state_t **IEEE 1722mc,
    struct sockaddr_in **si_other_avb,
    struct pollfd **avtp_transport_socket_fds,
    int packet_num
)
```

This procedure is blocking until an AVTP media clock packet arrives. The `struct pollfd` was used to keep blocking and non-blocking procedure signatures consistent, since the AVB server's main process also uses a media clock listener.

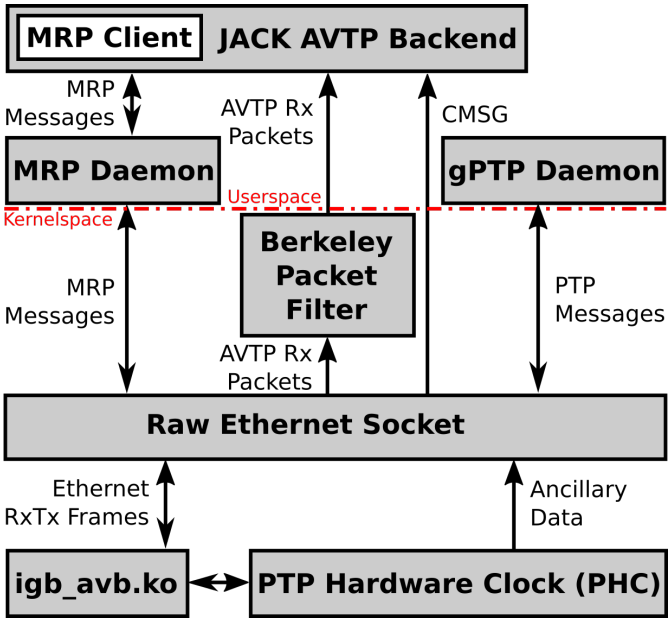


Figure 3: *Different kernel and userspace layers involved in the JACK media clock backend. The socket is filtered with a Berkeley Packet Filter (BPF) for the correct destination MAC address, Ethernet type and IEEE 1722 message type of the media clock stream packets. The stream ID is filtered after an AVTP packet is received in userspace.*

The raw Ethernet socket, that is used to receive the media clock stream, has the socket option `SO_TIMESTAMPING` set to:

```
ts_flags |= SOF_TIMESTAMPING_RX_HARDWARE;
ts_flags |= SOF_TIMESTAMPING_SYS_HARDWARE;
ts_flags |= SOF_TIMESTAMPING_RAW_HARDWARE;
```

The network device driver is configured to timestamp any incoming packet with a `struct hwtstamp_config` set to:

```
hwconfig.rx_filter = HWTSTAMP_FILTER_ALL;
hwconfig.tx_type = HWTSTAMP_TX_ON;
```

Experience has shown that `HWTSTAMP_TX_ON` has to be switched on for the reception of the media clock stream packets, even though the socket is not used for transmission, because the gPTP system service is effected otherwise and loses its synchronization to the PTP master.

Considering the following code listing, after the received packet was copied to the userspace buffer `struct msghdr msg` with the `recv_msg()` system call, the ancillary data in `struct msghdr msg` is accessed in line 8. Initially, the macro `CMSG_FIRSTHDR` returns a pointer to the first field of the ancillary data and stores it in `struct cmsghdr *cmsg`. As long as there is ancillary data available, the while-loop in line 9 cycles over the ancillary data of the received message. When a `SO_TIMESTAMPING` field is encountered, the pointers to the hardware timestamp and the hardware timestamp converted to system time are stored. The packet arrival time in nanoseconds is converted from `struct timespec`

to `unsigned int 64` and stored in the variable `pkt_arrival_ts_ns` in line 17.

The current transmission interval of the packet is calculated after the while-loop in line 25, the timestamp `last_pkt_ts_ns` of the last packet is subtracted from the timestamp `pkt_arrival_ts_ns` of the current packet. In line 26 the current timestamp is stored for the next packet as last timestamp.

The variable `pkt_num` is an argument of the procedure and supplied by the driver backend indicating the current packet number in the JACK period. When `pkt_num` matches the calculated packet numbers from table 1, a zero padded packet is sent.

If the `pkt_num` counter reaches the 6th, 11th or 22nd iteration, `adj_pkt_ts_ns` is calculated in line 30 to precisely adjust the JACK period. The remaining (modulus) samples of the JACK period divided by six samples per channel per AVTP packet, is divided by the sample rate and then scaled to nanoseconds in `unsigned int 64` representation. This calculation accounts for the padded AVTP packets calculated in table 1. The procedure returns `adj_pkt_ts_ns` to the backend driver, which can adjust the JACK period accordingly.

```
1  struct msghdr msg;
2  struct cmsghdr *cmsg;
3  uint64_t current_tx_int_ns = 0;
4  uint64_t last_pkt_ts_ns = 0;
5  ----------------------8<-----------------------
6  recv_msg(..., &msg, ...)
7  ----------------------8<-----------------------
8  cmsg = CMSG_FIRSTHDR(&msg);
9  while( cmsg != NULL ) {
10     if(cmsg->cmsg_level == SOL_SOCKET
11          && cmsg->cmsg_type == SO_TIMESTAMPING){
12         struct timespec *ts_dev, *ts_sys;
13         ts_sys = ((struct timespec *)
14                              CMSG_DATA(cmsg))+1;
15         ts_dev = ts_sys + 1;
16
17         pkt_arrival_ts_ns = ts_dev->tv_sec
18                       * 1000000000LL
19                       + ts_dev->tv_nsec);
20         break;
21     }
22     cmsg = CMSG_NXTHDR(&msg,cmsg);
23 }
24
25 current_tx_int_ns = pkt_arrival_ts_ns
26                           - last_pkt_ts_ns;
27 last_pkt_ts_ns = pkt_arrival_ts_ns;
28
29 if( pkt_num == (*IEEE 1722mc)->num_pkts -1){
30     adj_pkt_ts_ns = (uint64_t) (
31             ( ((*IEEE 1722mc)->psize % 6 ) /
32             (*IEEE 1722mc)->srate ) *
33             1000000000LL);
34 }
35
36 return current_tx_int_ns - adj_pkt_ts_ns;
```

3. EVALUATION

The quality of the synchronization to the media clock stream may be analyzed in terms of the variation between the points in time, when a JACK client is triggered and when a media clock stream packet is received. We basically observe, how many media clock stream packets are received between two successive calls of the JACK backend to the client's process callback function. The AVTP backend is based on counting the media clock stream packets, thus it is implicitly synchronized to the media clock stream source. The ALSA backend is not implicitly synchronized to the media clock stream source, which is the reason for the development of the AVTP backend. A synchronization would also be possible, since the media clock source and the servers are synchronized to the gPTP network clock. The media clock source of the XMOS development board drives its audio codec with a phase locked loop that locks onto the gPTP network clock. The local sample clock of an audio device connected to a server would also require a phase locked loop that is fed into the audio device or a continuing calculation and adjustment between the network and the audio time.

The "simple_client.c" example from the JACK source tree has been modified to make a system call to the system clock, which is synchronized to gPTP, with `CLOCK_REALTIME` every time the JACK process callback is triggered. The measured timestamps are written in the JACK shutdown callback function to file. Simultaneously, the JACK AVTP backend writes the timestamps from the ancillary data to file, as soon as JACK is shut down. In order to be able to compare the client activation times of the ALSA backend with those of the AVTP backend, a common time source is required. Instead of a local audio time that is adjusted to gPTP, we use the media clock stream as common time source. The JACK server is launched twice for this reason, one instance running with the ALSA backend and the measurement client, and a second instance running only with the AVTP backend to measure the media clock stream. The server was connected to a Focusrite Solo Gen2 USB audio interface [22], when the ALSA backend was measured.

The measurements were conducted with 32, 64 and 128 samples per JACK period with a sample rate of $48\ kHz$ over a duration of five minutes, producing between $\approx 10^5$ and $\approx 5 \cdot 10^5$ client activations, depending on the period size. Furthermore, the AVTP backend was measured with two different configurations. In the first configuration, the differences of the successive packet arrival times are accumulated, as it was explained in subsection 2.2 (AVTP Adjust). In a second configuration, a constant difference of $125,000$ (nanoseconds) is added each time, a media clock stream packet arrives (AVTP Const). No buffer over- or underrun occurred in any of the JACK backend configurations. The results of the measurements are shown in table 2.

4. DISCUSSION

Table 2 confirms the primary motivation for the development of the JACK AVTP backend, the ALSA measurements for each sample period shows a broad distribution of client activation times, which is further emphasized by its average and standard deviation. The expected value is not met in any configuration and the deviation is significantly higher than with AVTP. This results in a JACK client and a backend, which are not synchronized to the media clock. The required media clock stream packets per JACK period from table 1 are hardly met.

Media Clock Stream Packet Count	**JACK Client Activation Count**								
	32 Samples			**64 Samples**			**128 Samples**		
	AVTP Adjust	**AVTP Const**	**ALSA**	**AVTP Adjust**	**AVTP Const**	**ALSA**	**AVTP Adjust**	**AVTP Const**	**ALSA**
1	14099	0	15012	0	5936	0	0	0	0
2	0	0	19	0	0	0	0	0	0
3	1	0	32242	0	0	0	0	0	0
4	3	1	119103	0	0	0	0	0	0
5	16353	15328	7022	0	0	0	0	0	0
6	437342	406581	266913	0	0	5437	0	0	0
7	16416	15392	34865	0	0	61360	0	0	0
8	4	1	18	0	0	17693	0	0	0
9	1	0	0	2	1	282	0	0	0
10	0	0	0	8757	3275	9	0	0	0
11	0	0	0	204408	211261	2210	0	0	0
12	0	1	0	8817	3337	95166	1	0	0
13	0	0	0	2	0	70530	0	0	9
14	0	0	0	1	0	1634	0	0	2332
15	0	0	0	0	0	0	0	0	36583
16	0	0	0	0	0	0	0	0	2562
17	0	0	0	0	0	0	0	0	7
18	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0	1	0	0
20	0	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	2824	3901	0
22	0	0	0	0	0	0	104961	107485	88
23	0	0	0	0	0	0	2891	3969	10814
24	0	0	0	0	0	0	0	0	61739
25	0	0	0	0	0	0	0	0	10292
26	0	0	0	0	0	0	0	0	54
27	0	0	0	0	0	0	0	0	0
28	0	0	0	0	0	0	0	0	0
29	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0
31	0	0	0	0	0	0	0	0	0
32	0	0	0	0	0	0	0	0	0
33	0	0	0	0	0	0	0	0	0
34	0	0	0	0	0	0	0	0	0
35	0	0	0	0	0	0	0	0	0
Average	5.8	6.0	5.2	11.0	10.7	10.6	21.6	22.0	21.3
Standard Deviation	0.88	0.26	1.35	0.28	1.61	2.54	2.71	0.26	3.79

Table 2: JACK client activation count in respect to media clock stream

Comparing the two AVTP backend configurations for each sample period size shows, except for some outliers that account for less than 1% of the activation counts, that both configurations provide a equivalent solution. The averages and standard deviations of all sample period configurations imply a synchronized JACK client and backend. The required media clock stream packets per JACK period from table 1 are mostly met, with slight deviations.

5. CONCLUSIONS

Inherently, the ALSA backend for JACK adds some drift to the signal processing chain inside the Soundjack server network. Therefore, an experimental IEEE 1722 AVTP media clock backend for JACK was developed to overcome this problem. We could show that our solution for this problem is working and provides the desired synchronization and it is not necessary to adjust the duration of each JACK period with nanosecond accuracy.

Since the AVTP backend only receives AVTP packets, it is theoretically possible to run the backend on any PTP enabled device, even when no prioritized transmission queues are provided by the hardware - Intel I217 for example.

6. FUTURE WORK

Future work will focus on testing the Soundjack server network setup in the real world, the public Internet instead of the campus network, therefore adopting IPv6, with evaluation of the changes to the network tomography, has to be done.

Furthermore, the AVB processing server network shall in the future be migrated to function as a completely AVB capable JACK backend, not just for media clock synchronization.

It will also be of interest to achieve a synchronization between client and server via the public Internet. Mechanisms best suited for this feature are already under investigation.

Acknowledgment

fast-music is part of the fast-project cluster (fast actuators sensors & transceivers), which is funded by the BMBF (Bundesministerium für Bildung und Forschung).

7. REFERENCES

[1] (2019, Feb. 8) Soundjack - a realtime communication solution. [Online]. Available: http://http://www.soundjack.eu

[2] A. Carôt, U. Krämer, and G. Schuller, "Network music performance (nmp) in narrow band networks," in *in Proceedings of the 120th AES convention, Paris, France*. Audio Engineering Society, May 20–23, 2006.

[3] (2019, Feb. 8) Genuin classics gbr, genuin recording group gbr. 04105 Leipzig, Germany. [Online]. Available: http://genuin.de

[4] (2019, Feb. 8) Symonics gmbh. 72144 Dusslingen, Germany. [Online]. Available: http://symonics.de

[5] (2019, Feb. 8) Jack audio connection kit. [Online]. Available: https://jackaudio.org

[6] C. Kuhr, T. Hofmann, and A. Carôt, "Use case: Integration of a faust signal processing application in a livestream webservice," in *Proceedings of the 1st International Faust Conference 2018*. Mainz, Germany: Johannes Gutenberg-Universität Mainz, Jul. 17–18, 2018.

[7] C. Kuhr and A. Carôt, "Software architecture for a multiple avb listener and talker scenario," in *Proceedings of the Linux Audio Conference 2018*. Berlin, Germany: Linuxaudio.org, Jun. 7–10, 2018.

[8] H. Zimmermann, "Osi reference model -the iso model of architecture for open systems interconnection," in *IEEE Transactions on Communications, Vol. 28, No. 4*, Apr. 1980, pp. 425–432.

[9] *Timing and Synchronization for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Std. 802.1AS, Mar. 2011.

[10] *Virtual Bridged Local Area Networks - Amendment 14: Stream Reservation Protocol (SRP)*, IEEE Std. 802.1Qat-2010, Sep. 2010.

[11] *Virtual Bridged Local Area Networks - Amendment 12: Forwarding and Queuing Enhancements for Time-Sensitive Streams*, IEEE Std. 802.1Qav-2009, Jan. 2010.

[12] *Layer 2 Transport Protocol for Time-Sensitive Applications in Bridged Local Area Networks*, IEEE Std. 1722, May 2011.

[13] *Device Discovery, Connection Management, and Control Protocol for IEEE 1722 Based Devices*, IEEE Std. 17 221, Aug. 2013.

[14] *(Revision of IEEE Std 802.1Q-2011) - IEEE Standard for Local and metropolitan area networks–Bridges and Bridged Networks*, IEEE Std. Std 802.1Q-2014, Dec. 2014.

[15] *Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std. 1588-2008, Jul. 2008.

[16] H. Weibel and S. Heinzmann, "Media clock synchronization based on ptp," in *Audio Engineering Society Conference: 44th International Conference: Audio Networking*, Nov 2011. [Online]. Available: http://www.aes.org/e-lib/browse.cfm?elib=16146

[17] (2019, Feb. 8) User space api for time stamping of incoming and outgoing packets. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/timestamping.txt

[18] I. Corp. (2019, Feb. 8) Intel® ethernet controller i210-at product specifications. [Online]. Available: https://ark.intel.com/products/64400/Intel-Ethernet-Controller-I210-AT?_ga=1.64461743.1696258023.1478891344#tab-blade-1-0

[19] (2019, Feb. 8) Openavnu - an avnu sponsored repository for time sensitive network (tsn and avb) technology. [Online]. Available: https://github.com/AVnu/OpenAvnu/

[20] (2019, Feb. 8) Xmos ltd. / attero tech inc. [Online]. Available: http://www.atterodesign.com/cobranet-oem-products/xmos-avb-module/

[21] A. Carôt, T. Hohn, and C. Werner, "Netjack – remote music collaboration with electronic sequencers on the internet," in *Proceedings of the Linux Audio Conference 2009*. Parma, Italy: Institute of Telematics University, Deutsche Telekom AG Laboratories, University of Lübeck, Germany, 16–19, 2009.

[22] (2019, Feb. 8) Focusrite audio engineering ltd. United Kingdom. [Online]. Available: https://us.focusrite.com/usb-audio-interfaces/scarlett-solo

TPF-TOOLS - A MULTI-INSTANCE JACKTRIP CLONE

Roman Haefeli
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
roman.haefeli@zhdk.ch

Johannes Schütt
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
johannes.schuett@zhdk.ch

Patrick Müller
ICST (Institute For Computer Music And Sound Technology)
Zurich University of the Arts, Switzerland
patrick.mueller@zhdk.ch

ABSTRACT

Tpf-tools are used to establish bi-directional, low-latency, multichannel audio transmission between two or more geographically distant locations. The tool set consists of a server part (the tpf-server) and a client part (the tpf-client) and is heavily inspired by the JackTrip utility. It is based on the same protocol. It facilitates the handling of many concurrent audio transmissions in setups with more than two endpoints. Also, it eliminates the requirement of one endpoint having a public IP address or port forwarding configuration.

1. INTRODUCTION

The JackTrip[1] utility has proven to be a very useful and versatile tool for our research into the so-called telematic performance format (tpf), staged (musical or other kinds) events that take place simultaneously at two or more geographically distant concert venues. For these concerts, the stage is designed to blend physically present local performers with their remote counterparts, represented by means of low-latency video (UltraGrid [1]) and audio (JackTrip) transmission.

1.1. The obstacles of current IP networks

We have successfully used the JackTrip utility in many of our telematic concerts. The utility operates in two modes: client mode and server mode. For an audio transmission to take place, one end runs it in server mode listening for an inbound connection, while the other end runs it as client, thus initiating the connection. This works well so long as the client "sees" the IP address of the server. In today's Internet, most computers touched by human beings are assigned an IP address from a local area network (LAN) which is protected by a NAT router [2] . Public IP addresses are usually only assigned to headless servers and – apparently – NAT routers, but not to devices touched by humans. This topology divides the Internet in service providers and consumers and reflects the predominant capitalist ideology of today's Internet [2, Chapter 5]. At the same time, it hinders our efforts to perform telematic concerts. Running JackTrip in server mode at a concert venue requires a computer that has either a public IP assigned, or the proper port forwarding configured on the local network router. At venues where the performers are not the owners or administrators of the local network, this often bears huge administrative overheads and dealing with IT staff who may be more concerned about security than artistic achievements.

[1] Software for low-latency video transmission http://www.ultragrid.cz/

[2] NAT (network address translation) routers separate the LAN from the Internet. This increases security, because local computers are invisible from the Internet. It is also a way to deal with IPv4 address exhaustion, because all devices of a local network share one public IP address for outbound connections.

1.2. The complexity of many nodes

Another complexity we have encountered is the planning and set up of JackTrip connections when, not two, but three or (for a test situation) four venues are participating in an event. Two endpoints require one link. Three endpoints require three links, while four endpoints require six links. The number of links grows quickly with the number of endpoints. Events with more than two nodes require meticulous and careful planning.

1.3. Our motivation

We are looking for ways to streamline our processes and improve our tools in order to be able to shift our focus away from technical to more artistic aspects. JackTrip is the tool of our choice, because it is multi-platform, open source, uses JACK [3] and thus integrates well with existing professional audio software (e.g. Ardour). However, we saw an opportunity in adding a higher layer on top of the strong basis JackTrip gives us. In our efforts, we have developed a tool set that addresses the obstacles we've been experiencing:

- None of the endpoints need a public IP address.
- The client manages the audio transmissions to many endpoints and abstracts the complexity of such setups away, while presenting a simple, yet comprehensive interface to the user.

In this paper we present our tool set consisting of the tpf-client [4] (the software that is running on each participating endpoint) and the tpf-server [5] (the software that enables communication between the clients and coordinates audio transmissions).

2. VARIOUS CONNECTION MODES

2.1. Client connects to server (standard mode)

The JackTrip utility is designed so that both ends are sending similarly formatted UDP [6] packets. In server mode, it opens a listening socket that awaits for incoming connections. As soon as a packet arrives, it starts sending packets to the sender address of the incoming packets. In client mode, it immediately starts sending packets. The transmission is established as soon as both ends are up and running. This only works when the IP address of the server is visible to the client.

[3] Jack Audio Connection Kit, a sound server daemon for connecting audio applications and sound cards. http://www.jackaudio.org/

[4] The tpf-client is available at https://gitlab.zhdk.ch/TPF/tpf-client.

[5] The tpf-server is available at https://gitlab.zhdk.ch/TPF/tpf-server.

[6] User Datagram Protocol, a connectionless protocol based on the Internet Protocol that operates on the Transport Layer (Layer 4) of the OSI model. Applications with a strong focus on low latency often use it for transport.

2.2. Two clients connect to each other

A transmission can also be established when running both endpoints in client mode so long as both clients specify both bind port and peer port. The peer port of the first client matches the bind port of the second client, and vice versa.

An example of a JackTrip setup with both instances running as client:

```
$ jacktrip -c 192.168.0.12 --bindport 2000
    --peerport 3000
```

```
$ jacktrip -c 192.168.0.11 --bindport 3000
    --peerport 2000
```

This requires both ends to have an IP address visible to the other party. If one or both endpoints are hidden by a NAT-firewall, a connection cannot be established. However, this setup shows that the JackTrip design does not mandate one party to run as server.

2.3. Connection using a UDP proxy

The fact that a transmission can happen with two endpoints both running in client mode is crucial for the next step: establishing a transmission where none of the endpoints are assigned a public IP address. Since we want both endpoints to run in client mode, we need a third party that has assigned a public IP address and thus is visible for both endpoints, even when they are behind a firewall. This third party acts as proxy for both endpoints by relaying packets from client A to client B and vice versa. This technique passes most types of firewalls easily because the client initiates the connection. It works transparently for both endpoints as they do not have to know their respective peer's IP address. They simply connect to the UDP proxy. Since the JackTrip packet format is agnostic of the underlying transport protocol, all connection specific details are part of the UDP header and the payload does not contain any reference to the client address or port number. This allows the UDP proxy to relay incoming datagrams as is, without inspecting or changing the payload.

3. SUBSCRIPTION-BASED UDP PROXY

The simplest variant of a UDP proxy knows exactly two endpoint addresses and relays packets between them. However, this design mandates that each parallel transmission uses an instance of the UDP proxy, each listening on a dedicated port. The purpose of the subscription-based UDP proxy is to allow many parallel transmissions on the same port. To know which endpoints belong to a certain transmission, the endpoints send a so called token that is unique per transmission. If two clients send the same token, a transmission between those endpoints is established. This design allows an arbitrary number of transmissions to run on the same port, and each transmission is protected from intentional or unintentional interference by the token. Because of the requirement to send a token, the subscription-based UDP proxy does not work with the traditional JackTrip, at least not out-of-the-box [7] . Also, both parties intending to participate in a transmission must first agree on a common token through a separate channel.

[7] JackTrip could be wrapped into a script that first sends the token using the same bind port before it starts JackTrip

3.1. Implementation

The tpf-server presented here uses a Python [8] script as subscription-based UDP proxy. It uses two dictionaries (dicts) that are empty at start-up: a token dict and a link dict. The token dict stores the token string and sender adress when a token message is received. The token message is a UDP packet containting a string like

```
_TOKEN XXXX
```

where `XXXX` is the token string, an arbitrary string of arbitrary length. If a token message is received, its token string is looked up in the token dict. If there is no entry found, an entry is added to the token dict with the token string as key and the sender address as value. If another token message is received carrying the same token string but from a different sender address, two entries are made to the link dict. The first entry uses the address from the token dict as key and the sender address of the last token message as value. The second entry uses the same two addresses, but key and value are interchanged. After creating the entries to the link dict, the respective entry in the token dict is deleted, so that the same token may be used later by another party.

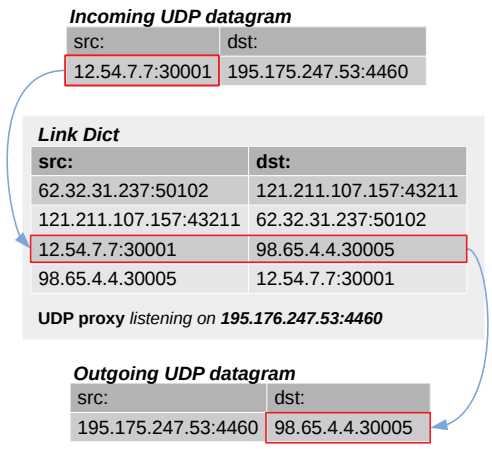


Figure 1: *Subscription-based UDP proxy.*

Since the UDP protocol does not guarantee that packets reach their destination, the client must keep sending token messages at a low rate (i.e. one message per second). When the client receives a packet for the first time, it stops sending token messages.

3.2. Considerations

Creating two entries per transmission into the link dict seems like a waste of memory, but it allows for a very quick look-up to determine the destination on an incoming packet. Keeping the latency low has the highest priority in our use case.

Although Python, as an interpreted language, is not among the fastest, it was the preferred choice for rapid prototyping and experimenting. It turned out that the UDP proxy written in Python was never the bottleneck in our performance tests and although it causes some CPU load under load, it does not seem to add a significant latency to the UDP transport. There has not yet been a pressing need to rewrite the UDP proxy in a more performant way.

[8] Python is an interpreted programming language supporting many paradigms. https://www.python.org/

4. THE TPF SERVER

The complexity of a setup increases quickly with the number of participating endpoints, as we showed before. We wanted software to manage the complex part of handling many parallel audio transmissions. The engineer should not have to deal with many terminal windows for running many JackTrip instances and know what IP address and port number each of their peer uses. Simplifying the involved processes was the main motivation for defining a protocol [3] and writing a server software implementing that protocol. It is worth noting that this part is orthogonal to the problem of audio transmission. The tpf-server is not involved in transmission of any audio data. Rather, it enables clients to know about each other and to let them initiate audio transmissions. The communication between server and clients uses TCP and runs on different ports.

4.1. Based on netpd-server and OSC

In order to reduce development efforts, the design is based on existing software – the netpd-server [9] – that was extended to implement the tpf-server presented here. The netpd-server is a relay for OSC messages and was developed for the netpd [4] project, a framework based on Pure Data (Pd) [5] that allows geographically remote clients to do electronic music together in real-time by synchronizing instrument states. The netpd framework uses OSC [6] for the communication, while OSC messages are encapsulated by SLIP [7] and transported by TCP. The OSC 1.1 specification [8] proposes SLIP to delimit OSC messages when transported by stream-oriented protocols such as TCP. While many OSC applications use UDP for transport for simplicity and speed, data integrity and correct order are crucial for the netpd framework. Also, for the tpf-server, whose purpose is to coordinate clients and allow them to share data, and which is not involved in the audio transmission directly, reliability trumps speed. TCP has a notion of connection, so for a server using TCP, there is no ambiguity in knowing when a client joins or leaves. With UDP it is much harder to clearly determine a client's state (e.g. joined or left).

4.2. netpd-server

The netpd-server defines rules about how incoming OSC messages are forwarded to the connected clients. This allows clients to send messages to specific peer clients, broadcast messages to all clients, or send messages to the server itself. The netpd-server forwards OSC messages according to the first element of the OSC path. The set of supported values for this field is listed here:

field	forwarding action
b	message is broadcast to all connected clients
s	message is intended for the server itself (not forwarded)
<*int*>	message is forwarded to the client with ID <*int*>

Table 1: List of valid receivers

4.3. The tpf-server internals

The tpf-server loads the netpd-server as an abstraction [9]. It reserves the OSC name space `/s/tpf`, which means all received messages whose OSC address starts with `/s/tpf` are handled by the tpf-server. The protocol is built on top of the protocol of the netpd-server. The exact protocol specification is part of the tpf-server package [3]. Since the protocol is based on OSC, it is agnostic of any software framework or programming language. It could be implemented in any language where libraries exist to deal with network sockets and the OSC protocol. It was implemented in Pure Data, because it uses parts already written in Pure Data. The tpf-server keeps track of the connected clients and coordinates a few common parameters that the endpoints must agree on before they are able to establish an audio transmission. It manages a few data containers and notifies clients about updates when data is changed. The tpf-server sends current data to the clients upon their request, while it is the duty of the clients to request data if they receive an update notification from the server. The data containers include:

4.3.1. Client ID And Name

When a client connects, the tpf-server assigns it a unique client ID (unique in the scope of the session). This ID, usually a small integer number, is used to identify each client. The same ID is also used to send an OSC message to a specific client by putting it into the first field of the OSC path. After establishing the connection to the server, the client registers a name (e.g. given name or location). It allows clients to display the list of connected peers in a more human-friendly way (see Client List).

4.3.2. Parameter List

The client with the smallest ID, usually the one that connects first to the server, is given a special role: it has the authority to set or change a set of parameters that all clients are mandated to use for the current session – samplerate, blocksize, bit resolution. Those parameters are distributed to all clients and the clients either adjust their settings or report an error when a mismatch occurs. The parameter list is not a hard-coded set. Instead, it is fully defined by the clients.

4.3.3. Client List

The tpf-server keeps a list of all connected clients with their ID, name, IP address and role. Whenever a client connects or disconnects, the tpf-server broadcasts an update of this list to all clients. It is therefore crucial that clients terminate their connection properly, otherwise they keep appearing in the client list until the connection is considered terminated. This period depends on the operating system.

4.3.4. Link List

In a full mesh network, each node is linked to every other node. If n is the number of nodes, the number of links (l) is:

$$l = \frac{n(n-1)}{2} \tag{1}$$

The tpf-server assigns each pair of clients a link ID, so each link ID associates two clients. The tpf-server sends each client its own list of their peer's client IDs along with the corresponding link ID. Clients use the link ID to establish the audio transmission to a specific peer. Early versions used one server port per transmission and tpf-client used the link ID as the port offset parameter for running JackTrip. In the current version, the link ID is used to generate a token string. Two clients using the same ID and thus the same token string are linked by the subscription-based UDP proxy.

[9] The netpd-server is part of the netpd framework developed by Roman Haefeli. The code is hosted at https://github.com/reduzent/netpd-server

When every transmission was using its dedicated UDP port, it seemed appropriate to let the server, as a central authority, assign link IDs to avoid collisions, but also to ensure that only IDs corresponding to an active UDP proxy would be assigned. With the subscription-based UDP proxy, this coordination task became moot, as clients could also negotiate a token by peer-to-peer communication without involving the server. Future versions of the tpf-server might remove the link list.

5. THE TPF-CLIENT

5.1. Written in Pure Data

The tpf-client is implemented in Pure Data, so it can be built on top of an already existing framework. For communication to the server, parts from the netpd client have been reused. Designed as a real-time audio programming language, Pure Data has already covered many aspects of dealing with low-latency audio. Furthermore, part of the Pd "eco system" is a vivid community that has been contributing many libraries extending the functionality of the software. Namely, there are so-called externals for parsing and formatting OSC messages (*osc*) and for accessing network sockets (*iemnet*). Pure Data has native JACK support built-in and runs on a variety of platforms.

5.2. Implementation

The purpose of the tpf-client is to manage audio transmissions to one or many peers joining the same session. It is the implementation of the client side of the tpf protocol. First drafts only implemented the management aspects in order to get the necessary information for starting the original JackTrip utility with the appropriate command-line arguments, so the audio transmission part was left completely to JackTrip. It was later decided to also re-implement the JackTrip utility as an abstraction.

5.2.1. Rewrite of JackTrip as Pd abstraction

Implementing the audio transmission part in Pure Data has some advantages:

- The lack of a stable and feature-complete Pd external for running system commands makes it hard to consistently control many JackTrip instances from Pd. JackTrip reimplemented as a Pd abstraction is easier to control and interface with.
- An implementation of the JackTrip protocol in Pd allows to extend it, if necessary. A small addition – the subscription by sending a token message – to the JackTrip functionality was necessary to support the subscription-based UDP proxy.
- Although able to create many JackTrip connections, the tpf-client appears as one JACK client, which somewhat simplifies the process of drawing connections in the connections dialog of QjackCtl.
- Since the audio signals travel through Pd, some signal processing could be applied. The current implementation doesn't apply any processing, though.
- Since the audio signals travel through Pd, signal level monitoring can be used and graphically represented in the client user interface.
- Signal path can be used to measure round-trip time of the audio signal with built-in latency meter.

5.3. User interface

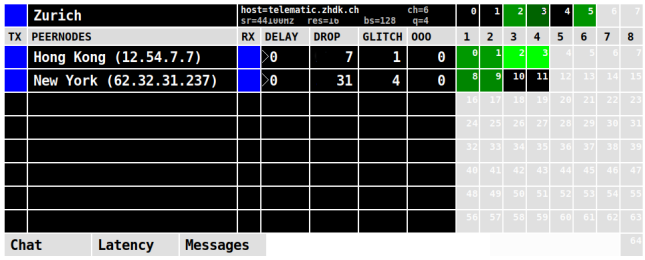


Figure 2: *The tpf-client user interface.*

The user interface displays a few configuration parameters that are settable before the connection to the server is initiated:

- name
- hostname (or IP address) of the tpf-server
- blocksize (of the JackTrip packets)
- number of channels (outgoing)
- queue buffer size

The *samplerate* and *bit resolution* cannot be changed in the client. The bit resolution is hard-coded to 16 bit. The samplerate is mandated by the JACK server and is inherited by Pd. After the connection is established, those configuration parameters become locked and cannot be changed until the session ends.

The client registers its name and either uploads the audio parameters such as *samplerate*, *blocksize*, *bit resolution* to the server or matches them against the mandated parameters, if another client already has configured those parameters. If there is mismatch between configured and mandated parameters, the client either reports an error (mismatch with *samplerate*, *bit resolution*) or silently adjusts the parameter (mismatch with *blocksize*). It is worth noting that blocksize configured in the tpf-client is decoupled from the blocksize used by the JACK server. This allows clients to run JACK with deviant blocksizes. After successfully having registered the name and matched audio parameters, the connection button (top left) turns blue to indicate that the client is ready for audio transmissions.

5.4. Managing transmissions

Peer clients are each listed in a separate row in the client interface. Audio transmissions are not started automatically, but are initiated by a user on either side by clicking the left-most button in the row. The button on the respective row on the peer's client starts flashing. Only when confirmed by the other end by clicking on the flashing button is the audio transmission started. The number of received channels is represented by the number of squares turning from grey to black in the respective row. Depending on the signal level of each channel, the square changes color from black (silence) to bright green (full amplitude). The number in each square corresponds with port number of the tpf-client in the QJackCtl connection dialog.

5.5. Transmission monitoring

During an audio transmission, three types of glitches are counted and displayed in the respective row:

DROP number of dropped packets. Late packets that miss their time frame to be played back are also considered dropped.

GLITCH number of audible audio glitches. Often, many packets are dropped in a row, resulting in one audible glitch. Thus, the number of audible artefacts is always smaller than the number of dropped packets.

OOO number of packets received out of order. If an out-of-order packet misses its time frame, it is dropped. Otherwise it is played back in correct order.

All counters are reset to zero when the audio transmission is restarted. Although those counters are not of much use during a real concert (not much can be done about bad statistics), they might help compare the quality of different network links, when testing setups or internet providers.

5.6. Message and chat window

Beside the main window, tpf-client's interface has a message window, where info, warning and error messages are displayed. There is also a built-in chat in the chat window. A channel of communication not involving audio is often desired.

5.7. Built-in latency measurement tool

To measure the overall round-trip time of the audio signal, both endpoints need to configure the audio path accordingly. The method is robust enough to allow the signal to be played back by a speaker and recorded with a microphone, even in a mildly noisy environment. The signal path of a full round-trip measurement is shown in Figure 3 .

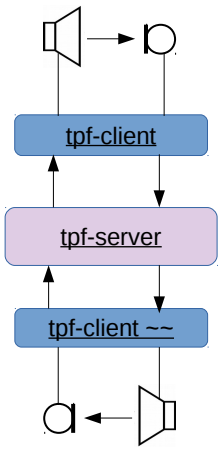


Figure 3: *Signal path of latency measurement.*

5.8. Adding artificial latency

The tpf-client allows each audio transmission to add an artificial audio delay. By adjusting the delay, it is possible to target a specific total round-trip time. Reasons for latency adjustment include:

- The performance of a certain musical piece requires the perceived latency to be aligned to the given tempo of the work.
- In a three-node setup, where one peer location is far more remote than the other, the un-adjusted latencies differ significantly, so it might be desired to "harmonize" the perceived latencies by artificially increasing the "distance" of the closer peer.

5.9. Considerations

Certain aspects of writing software in Pd are difficult. Designing a graphical user interface is relatively hard and the graphical representation is bound to pixel sizes and cannot be scaled dynamically (i.e. by resizing the window). Also, it is not possible to create dynamic interfaces that display different content depending on context. Due to those limitations, it was decided to restrict some capabilities of the client in order to provide a simple and consistent interface. The number of channels per audio transmission is limited to 8. Also, the maximum number of displayed peers and thus the number of concurrent audio transmissions is limited to 8. This limits the overall number of connected client being able to interact with each other to 9. Those limitations are not imposed by the tpf-server or the protocol, and the client could be adapted if need be. They are abitrary choices and during the past year of using the tpf-tools, those limits never have been reached in real life.

Unlike the original JackTrip implementation, each party in a setup using the tpf-tools can choose the number of channels to be sent individually. This saves bandwidth and might improve quality. Also, the configured blocksize is not dependent on the blocksize mandated by the JACK server. This can be an advantage, since the value for the most optimal JACK configuration might differ between clients.

6. EXPERIENCES AND DISCUSSIONS

We were interested to know how the usage of the tpf tools impacts audio quality and overall latency. We performed tests to compare the usage of the UDP proxy with a traditional JackTrip client-server connection. We wanted to know whether the usage of the UDP proxy has an influence on the number of dropped packets. In another test, we examined the latency differences between using a UDP proxy and a direct JackTrip connection. We also examined, whether the tpf-client imposes a penalty to the quality of the audio transmission compared to the original JackTrip.

6.1. Dropped packets imposed by UDP proxy

For counting glitches (which are a result of dropped packets), we sent a 1kHz-sine-tone through JackTrip to a remote JackTrip instance, that looped back the signal, and recorded the result for a predeterimined period of time. We used the `-z` commandline option of JackTrip, so that glitches were visually more easy to spot in the waveform. Then we counted the glitches by loading the recorded sound file into a sound editor and examining the discontinuities in the waveform. We were not able to determine a significant difference between a direct link and a link using the UDP proxy. At another instance, that was totally unrelated to the test series, we experienced many dropped packets. We later found out that the reason was a bug in the driver of the virtual network interface of the virtual machine the UDP proxy is running on. While the UDP proxy usually does not impact the number of dropped packets negatively, there is a plethora of possibilities as to why the UDP proxy might behave badly, because it

depends on hardware, on the operating system and of the software itself. These sources of error do not apply to a direct JackTrip link.

6.2. Latency imposed by UDP proxy

At the time of comparing the latency of a direct link to the UDP proxy, the tpf-client had not been written. So a simple tool in Pd was built to send a single UDP packet to a remote location, that sends back the packet immediately. The tool measures the delay betwen sending and receiving the packet. The average travel times turned out to be identical for both, a direct link a link using the UDP proxy. This behavior was consistent with different remote locations. This can be eppxlained by the fact that both, the computer taking the samples and the server running the UDP proxy, were located at the same campus. By using tools like mtr or traceroute, we found out that the number of hops between the computer taking samples and the remote computer was the same for both link types. In a scenario where both endpoints are located outside the campus hosting the UDP proxy, using the UDP proxy adds additional latency. The amount depends on how far the UDP proxy is away from the direct network path between both endpoints.

6.3. Performance of the tpf-client

We also tried to examine the impact of using tpf-client compared to the original JackTrip. It turns out that Pure Data adds one block of additional latency, because the way it communicates with JACK decouples its audio processing from the strict graph of the JACK server. Many other JACK clients like JackTrip are tightly coupled and do not add additional latency. When using a blocksize of 128 at a samplerate of 48kHz, the penalty of using tpf-client is 2.6666 ms. It increases with larger blocksizes or lower samplerates.

Because Pd interfaces the JACK server differently, it is possible that Pure Data's audio processing experiences audio drop-outs while the JACK server does not. This means that the tpf-client introduces a new source of possible buffer underruns. From our experience, this theoretical penalty has not become manifest in more glitches when using tpf-client, at least not when running tpf-client on a macOS system. On Linux, Pure Data was found, in some situations, to be the source of glitches when not running with realtime privileges. It was usually simple to remedy the situation.

6.4. Shortcomings of the JackTrip protocol

While measuring the number of glitches with different combinations of blocksize and number of channels, we found there was a sudden increase in glitch rate when the number of channels exceeded a certain value. When running two parallel transmissions with each only carrying half the channels, we experienced a low rate of glitches. By running other tests with the tool iperf, which allowed us to set the rate and size of UDP packets, we found that link capacity was only one limiting factor. Not less important was the so-called Path MTU [10] . UDP packets larger than the Path MTU are fragmented during transport. The loss of a single fragment results in the loss of the whole UDP packet. The likeliness of a UDP packet being dropped increases with the amount of fragmentation it experiences. For best performance, the UDP packet size should not exceed the Path MTU.

[10] Maximum Transmission Unit, is the maximum packet size that is a transmitted in a single network layer transaction, while Path MTU refers to the maximum packet size that is transmitted through all intermediate hops without fragmentation.

By running tests with iperf, we were not able to saturate a network link with a UDP stream, when choosing a relatively large packet size (e.g. 16000 bytes). By selecting a smaller packet size (e.g. 1400 bytes), we were able to achieve a data transfer rate close to the theorethical maximum while still keeping the number of dropped packets low. This finding shows that the JackTrip protocol is not suitable for all kinds of payloads, since the UDP packet size depends on bit resolution, number of channels and blocksize:

$$packetsize = H_{\text{UDP}} + H_{\text{jacktrip}} + N_{\text{channel}} \times \frac{b_{\text{res}}}{8} \times B_{\text{buffer}} \quad (2)$$

$$\begin{aligned} \text{where } H_{\text{UDP}} &= \text{Header size of UDP datagram,} \\ H_{\text{jacktrip}} &= \text{Header size of JackTrip frame,} \\ N_{\text{channel}} &= \text{Number of channels,} \\ b_{\text{res}} &= \text{bit resolution,} \\ B_{\text{buffer}} &= \text{buffer size} \end{aligned}$$

Larger numbers of channels or blocksize result in UDP packet sizes bigger than the optimal size. With a typical Path MTU of 1500, and a given blocksize of 128, the largerst number of channels still fitting into the Path MTU is 5 (1296 bytes). A single audio transmission with a high number of channels could be split into two or more parallel transmissions with a lower number of channels in order to reduce the resulting packet size. However, synchronization between the transmissions is not guarantueed and therefore this is not a suitable solution. The ability to detect the Path MTU and to optimize UDP packet size by splitting a transmission into many, while keeping synchronicity, are features that still need to be researched.

6.5. UDP hole punching

While there is none or only a negligible penalty for using the UDP proxy when it is located close to one participating party, it might add significantly to unacceptable latency, when the participating parties are all located geographically distant from it. In terms of network latency, using a direct link is sometimes as good, and in many cases clearly superior to using a proxy. A technique called UDP hole punching allows us to establish a direct UDP conncetion between two end-points, both acting as client, that is able to traverse many types of NAT-firewalls. NAT-firewalls usually let an incoming UDP packet pass, when its receiver address (IP and port) matches the sender address of a previously outgoing UDP packet. That is because UDP is a stateless protocol and has no notion about connection. That is how NAT-firewalls discern outbound connections (that are usually allowed) from inbound connections (that are usually blocked). Before establishing the connection, both endpoints contact a central server to learn about their peer's public IP address and port number. Then they start sending packets to the address they learned. Because this happens on both sides, the firewall on either side "thinks" the connection was initiated from a local client and it will pass incoming packets. The technique is already used in webRTC and IP telephony applications. The tpf-client supports UDP hole punching as an experimental feature. By double-clicking (instead of single-clicking) the left button in the peer row an audio transmission using a direct link is requested. There are still many scenarios where establishing a such link fails. Supporting more cases and making UDP hole punching a viable option is certainly a field worthy of further exploration.

7. ACKNOWLEDGEMENTS

Our research was funded by the *Swiss National Science Foundation* [11] and took place at the *Institute For Computer Music And Sound Technology* [12] (ICST), *Zurich University of the Arts*, Switzerland. Over the course of two years, we have been working within a team that explored many aspects of telematic performances, including scenography, audio engineering, audio and video streaming, network technology and considerations in the field of media theory. We are grateful for the collaboration with those very interesting people and feel there is still a lot waiting to be explored and further researched. We have been working with Matthias Ziegler, flutist and head of the research group, Benjamin Burger and Joel De Giovanni, video artists and scenographers, Bojan Milosevic, composer and researcher, Gina Keller and Ernesto Coba, audio engineers. We also thank all collaborating parties spread around the world for having showed the willingness to organize and perform telematic concerts with us and to use and test our tools. We appreciate being a part of this vivid community.

8. REFERENCES

[1] Juan-Pablo Caceres and Chris Chafe, "JackTrip: Under the Hood of an Engine for Network Audio," *Journal of New Music Research*, 2010.

[2] Robert W. McChesney, *Digital Disconnect: How Capitalism is turning the Internet against Democracy*, The New Press, New York, 2013.

[3] "tpf: Protocol Specification," `https://gitlab.zhdk.ch/TPF/tpf-server/blob/master/protocol_specification.txt`.

[4] Roman Haefeli, "netpd - a Collaborative Realtime Networked Music Making Environment written in Pure Data," in *Linux Audio Conference*, 2013.

[5] Miller Puckette, "Pure Data," `http://puredata.info`, 1996, Software.

[6] "Open Sound Control," `http://opensoundcontrol.org/`, Protocol.

[7] J. Romkey, "A Nonstandard For Transmission Of IP Datagrams Over Serial Lines: SLIP," Tech. Rep. RFC 1055, IETF, Network Working Group, 1988.

[8] Adrian Freed and Andy Schmeder, "Features and Future of Open Sound Control version 1.1 for NIME," in *NIME*, 2009.

[9] Miller Puckette, "Pd Documentation," `https://puredata.info/docs/manuals/pd/x2.htm#s7.1`, 2.7.1. abstractions.

[11] SNF: http://www.snf.ch/

[12] ICST: https://www.zhdk.ch/en/research/icst

A CROSS-PLATFORM DEVELOPMENT TOOLCHAIN FOR JIT-COMPILATION IN MULTIMEDIA SOFTWARE

Jean-Michaël Celerier

SCRIME
Université de Bordeaux, France
jeanmichael.celerier@gmail.com

ABSTRACT

Given the relative stagnation in single-thread performance of many processors in the recent years, made even worse by the recent security findings such as SPECTRE or L1TF which led to restrictions in existing features and decreased performance for the sake of security, it is necessary to find new ways to improve the run-time performance of dynamic multimedia systems. In this paper, we present the introduction of a just-in-time compiler in the *ossia score* interactive score authoring and playback software. We discuss in particular the creation of a toolchain and software development kit for C++ just-in-time compilation on the three major desktop platforms, the challenges and benefits caused by the use of C++ in terms of standard library requirement, but also the benefits that the system offers in terms of live-coding.

Keywords: interactive scores, just-in-time compilation, toolchains

1. INTRODUCTION

Users of multimedia software demand two features which can be hard to reconcile. On one hand, they ask for more performance, the ability to run more tracks, add more effects, etc. On the other hand, they request more dynamic behavior, and easily extensible systems – in particular, systems which do not require the user to write Makefiles and set-up a compilation toolchain. But such a dynamic behavior generally comes at a cost: for instance, Javascript, Lua or Python are often integrated with media environments, such as *Blender*, *ossia score*, and *Renoise*. These languages can have undesirable properties in low-latency audio environments: they can cause spurious dynamic memory allocations, which prevents real-time guarantees to be ensured.

Ongoing advances in just-in-time compilation can to some extent reconcile these needs. The LLVM project [7] provides simple APIs to integrate compiler and assembler in C++ software, through the MCJIT and OrcJIT sub-libraries.

The benefits of just-in-time compilation have been known for a long time [2] ; of particular interest to us is the ability of just-in-time compilers to adapt to the exact CPU type available in the user's computer. This can lead to great performance improvements: modern compilers are able to generate correctly vectorized code for vector instruction sets, such as SSE, AVX, AVX-2, AVX-512 on x86-based platforms, or Neon on ARM platforms. But in the traditional compilation model, the author of the software has to know beforehand for which instruction set the software shall provide optimized routines, and either write them manually in assembler or with intrinsincs, use compiler-specific extensions such as GCC's function multiversioning [1] or resort to manual run-time dispatch to the correct function according to detection of the user's CPU. This leads to an increase in executable size for all the users of the software, and can be quite time-consuming for the developer. Thus, we propose to leverage JIT compilation for some of the most performance-critical parts of media software so that they can be compiled in the most optimal way for the user's CPU.

The proposed system simply compiles C++ code. This is in contrast with many approaches such as Faust [11] for audio signal processing, PostgreSQL [13] for improvement of the SQL query performance or the language created by Avramoussis et al. for transformation of geometry assets in the VDB format [1]. These systems all provide custom domain-specific languages (DSL) to solve a well-defined task. This has the advantage of freeing oneself from C and C++'s complicated legacy and generally simplify the language semantics, but also means that:

- A large amount of work must be provided by the new language authors.
- The language won't necessarily be subject to new advances in compiler development unless its authors keep working on it: while some optimization phases can occur at later stage if leveraging an existing compiler framework such as LLVM, some optimizations require actual knowledge of the language's semantics and thus cannot be applied generically to any DSL.
- The language may not be able to leverage the existing corpus of libraries available in C and C++.

The system is integrated in the *ossia score* software [6, 4] for media creation. Part of the motivation is to improve run-time performance while live-coding: the software currently features a Javascript engine which can be leveraged to provide new behaviors at run-time. While it is one of the software's user-base's favorite features, it comes at a cost: no real-time safety due to the Javascript engine performing many memory allocations, and huge "context switch" costs between the native code world, and the interpreted Javascript engine world. The objective is to improve the run-time performance, while retaining some of the properties provided by live-coding: for this, Thor Magnusson gives the hard criteria that a live-coding language should not take more than five seconds between code and sound [10].

We will first give a brief overview of the *OSSIA* project, and of the way just-in-time compilation is introduced into the system. Then, we will give some pointers towards the creation of a cross-platform toolchain which allows to support JIT compilation in the three major desktop operating systems, Linux, macOS and Windows. Some performance metrics will be discussed.

2. OSSIA PROJECT

ossia [2] is an open-source software suite composed of a library (*libossia*) and a graphical user interface (*ossia score*) for managing

[1] https://lwn.net/Articles/691932/

[2] https://ossia.io/

communication, mapping and time-scripting between various software in interactive multimedia artworks. This toolset is cross-platform (Windows, macOS, Linux), cross-protocol (OSC, MIDI, ...). The libossia library has been ported to many creative coding tools (Ableton, Max/MSP, PureData, VVVV, Touch Designer, OpenFrameworks, Processing…). It simplifies connecting and controlling various digital production software together. Its main goals are to facilitate the development of time-centric interactive artworks and lower the barrier of entry to interactive media creation and authoring for emerging artists.

The *ossia score* software's execution engine is based on a dataflow architecture described in [3]. The user interface part leverages a modern C++ and Qt-based generic document framework which can be easily reused for other document-centric software. It features an extensible plug-in API, undo-redo with automatic recovery in case of crash, interface injection, serialization, selection handling and multiple document management. It is specifically well-tailored to hierarchical document structures and enforces strong typing practices.

This framework has been used in an unrelated software as a test of its flexibility: a point-and-click game editor (SEGMent, developed with Raphaël Marczak[3]).

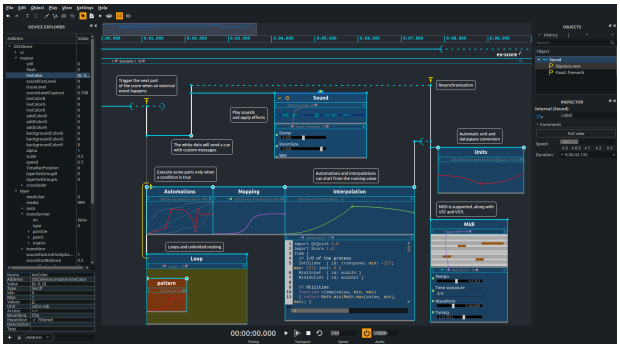


Figure 1: *ossia score*, the main software leveraging this framework

3. C++ JIT

We chose to extend *ossia score* with a C++ just-in-time compilation mechanism. The main motivations for this were:

- Using C++ allows reusing easily large amounts of existing code ; for instance digital signal processing libraries such as Gamma[12], KFR[4] or FFmpeg[5].
- Due to the amount of software built using C++, compiler optimisations for this language are still an active research topic [8, 9], which guarantees "free" performance improvements in the following years.
- *ossia score* was already integrating Faust, which itself uses LLVM, and thus acted as a gateway drug of sorts.

4. PLUG-IN AND PLUG-IN APIS

ossia score already provides multiple plug-in APIs: a simple API based on defining a unit generator with strong type-safety features relating to the input and output ports of the unit generator, and a low-level API which allows creating plug-ins that can modify every part of the *ossia score* software: menus, panels, etc.

The JIT system leverages the existing plug-in APIs: the same code can seamlessly be integrated either during the build of *ossia score*, or at run-time. We give thereafter a brief overview of these two APIs.

4.1. Safe process API

This API only gives the ability to provide a new unit generator to the system. Inputs, outputs and controls are given as C++ constant expressions, which generates the user-interface code at compile-time and guarantee type-safety. The necessary boilerplate being relatively low (for C++ code), it is viable to use in live-coding contexts. A specific unit generator, for now simply named "C++ Jit process" in the software, allows the user to input code using such API, which will be live-recompiled ; the corresponding node will be instantiated.

Algorithm 1 provides an example of a "gain" node, which has one audio and one floating-point input, one audio output, and applies the gain to the input.

Algorithm 1 : A naive gain implementation in the "safe" plug-in API. The inputs and outputs of the unit generator are declared in the `Metadata` struct. A compile-time mechanism ensures that the prototype of the `run` function conforms to the prototype, and that the types of the arguments are correct. This increases type safety at run-time when compared to the more traditional C-based solutions where the programmer has to manually cast the inputs of the unit generator into the correct type according to knowledge not part of the type system.

```
struct Node
{
  struct Metadata : Control::Meta_base
  {
    static const constexpr auto prettyName = "Gain";
    static const constexpr auto controls
      = std::make_tuple(Control::FloatSlider{"Gain", 0., 2., 1.});
    static const constexpr audio_in audio_ins[]{"in"};
    static const constexpr audio_out audio_outs[]{"out"};
  };

  using control_policy = ossia::safe_nodes::last_tick;
  static void run(
      const ossia::audio_port& p1, float g, ossia::audio_port& p2,
      ossia::token_request, ossia::exec_state_facade)
  {
    const double gain = (double)g;
    const auto chans = p1.samples.size();
    p2.samples.resize(chans);
    for (std::size_t i = 0; i < chans; i++)
    {
      auto& in = p1.samples[i];
      auto& out = p2.samples[i];

      const auto samples = in.size();
      out.resize(samples);

      for (std::size_t j = 0; j < samples; j++)
      {
        out[j] = in[j] * gain;
      }
    }
  }
};
```

[3] https://scrime.u-bordeaux.fr/Arts-Sciences/Projets/Projets/SEGMent2-Study-and-Education-Game-Maker

[4] https://www.kfrlib.com

[5] https://www.ffmpeg.org

4.2. General plug-in API

This API enables its user to introduce new elements in most parts of the software:

- New menus, panels, etc.
- Run-time additions to existing data types of the software.
- File loaders.
- Network and hardware protocols.

At the source code level, it mainly leverages the *Abstract Factory* design pattern. A plug-in can define a new interface, identified by an UUID. An example is given in algorithm 2:

Algorithm 2 : An example of interface definition in *ossia score*. This particular interface allows a plug-in to register the handling of new file types in the "Library" panel.

```
class LibraryInterface : public score::InterfaceBase
{
  SCORE_INTERFACE(LibraryInterface, "9b94d974-9f2d-4986-a62b-
        b69e51a4d305")
public:
  ~LibraryInterface() override;

  virtual QSet<QString> acceptedFiles() const noexcept;
  virtual QSet<QString> acceptedMimeTypes() const noexcept;

  virtual void setup(
      ProcessesItemModel& model
    , const score::GUIApplicationContext& ctx);
  virtual bool onDoubleClick(
      const QString& path
    , const score::DocumentContext& ctx);
  // ...
};
```

Plug-ins can then register implementations for these interfaces, which can be listed and accessed through a global context object.

The majority of the *ossia score* codebase is based on this API, the actual software being itself merely a set of plug-ins implemented on top of the base plug-in framework. The JIT extension discussed here is itself a plug-in [6].

The original plan for *ossia score* was to rely on this plug-in API to allow prebuilt extensions to be downloaded from a common repository. Due to the ongoing development of the software, no ABI (Application Binary Interface) stability guarantees are provided, which means that plug-ins must generally be recompiled against the source code of newer versions. This requires an extensive compilation architecture which could not only rebuild and publish new versions of *ossia score* but also the plug-ins regularly. Common service providers such as *Travis CI* and *Appveyor* do not provide enough capacity for this to be viable for an open-source, volunteer-led project.

Hence, the plan going forward is to distribute the plug-ins not included in the base software under source code form. The JIT system looks for addons on startup in the user library folder: for instance `~/Documents/ossia score library/Addons` and simply compiles all the source files of the addon together. This guarantees that API and ABI breakage do not cause subtle run-time errors since the add-ons are compiled against the exact source code that was used to build the software, the headers being shipped as part of the package: if the API has changed in a breaking manner, the add-on will not be compiled at all and the user warned.

[6] `https://github.com/OSSIA/score-addon-jit`

5. A CROSS-PLATFORM TOOLCHAIN

ossia score being a cross-platform software, it is necessary to ensure the same level of support on the three major operating systems: Windows, macOS and Linux. The endeavor was relatively straightforward on Linux thanks to the availability of the LLVM libraries and compilers in package managers. In particular, the Linux implementation of JIT compilation in *ossia score* is also able to use system libraries instead of the ones provided by the toolchain. The official release of *ossia score* is based on the AppImage mechanism which allows it to work on many distribution: as such, it is also necessary to build a recent toolchain to be able to target older systems, such as CentOS 7 or Ubuntu 12.04.

The complete toolchain, whose build scripts are available at `https://github.com/OSSIA/sdk` provides the following libraries:

LLVM 7.0.1 (8 svn on Windows due to previous versions not working) , Qt 5.12 , FFMPEG 4.1 , PortAudio , JACK headers , SDL2 , OpenSSL , Faust.

5.1. Uniform C++ standard library

The C++ parts of the toolchain are built against the *libc++* standard library implementation on all platforms. This is for two reasons: uniformity, and licensing. Using a single C++ standard library across all platforms guarantees less variance in behavior, which is still fairly common for instance across the various implementations in the implementation of standard algorithms, or complex libraries such as `<regex>`. Especially on Windows, the standard library headers provided as part of Visual Studio are not freely redistributable. This means that this would introduce an unacceptable dependency on a Visual Studio installation into *ossia score*. Hence, we use the system headers provided by the mingw-w64 project, along with the LLVM libc++ standard library. The build process implies a first build of the LLVM project, clang compiler and libc++ standard library, which are then used to boostrap a second set of LLVM libraries. This is needed due to the JIT implementation directly calling into LLVM's OrcJIT API: if we linked directly against the first set of LLVM libraries, there would be a standard library mismatch which would in the best case fail to link properly, and in the worst case fail at run-time.

The *llvm-mingw* project[7] greatly simplified the creation of the Windows toolchain.

5.2. macOS and rpath handling

macOS is special in that libc++ is the default C++ library implementation. There is no equivalent to MinGW in the Apple world: the only implementation of system headers is the one provided by Apple. Those are not under a free license, to the exception of the C standard library and Mach kernel headers.

In addition, the customized clang / libc++ provided by Apple is slightly out-of-date when compared to other platform's implementations and suffers from some artificial limitations: using various C++17 standard library types, such as `std::any`, `std::optional` or `std::variant` restricts the deployment to the latest in date version of macOS, 10.14, which is not acceptable for multimedia software users often restricted to older system versions for the sake of compatibility. The macOS version of the toolchain thus provides its own clang / libc++ build which overcomes this problem.

[7] `https://github.com/mstorsjo/llvm-mingw`

A custom-built clang-based toolchain on macOS will by default still link against the system libc++ implementation. The observed behavior is as follows:

- No arguments passed: the compiler hard-codes an absolute path to the system `/usr/lib/libc++.1.dylib`.
- `-L$SDK/lib -lc++ -lc++abi`: the compiler links the software to `@rpath/libc++.1.dylib`.

It is thus necessary to specifiy the rpath to get working binaries during development: `-L$SDK/lib -lc++ -lc++abi -Wl,-rpath,/sdk/lib`.

6. BENCHMARKING

We provide a few performance tests of the system: what advantages and what costs actually bring C++ JIT compilation. Benchmarks are run on two machines, both running Linux (Kernel: 4.20.8-arch1-1-ARCH):

- Machine 1: Intel(R) Core(TM) i7-6900K CPU @ 4.00GHz (Broadwell architecture, desktop).
- Machine 2: Intel(R) Core(TM) i7-8750H CPU @ 4.00GHz (Coffee Lake architecture, laptop).

6.1. Compile times

C++ is notorious for its slow compile times, due to large amounts of header files to include, and the cost of the template instantiation mechanism. More recent C++ standards being oriented towards compile-time computation of most values in a program also leads to an increase in compile times.

On the test machine, a simple node such as the one provided in 1 takes between 1.3 and 1.5 seconds to compile on an average of five runs. A generic test addon providing mock implementations of a few interfaces, comprised of 7 source files, 10 header files, for a total of 428 lines of code which themselves include part of the C++ standard library and Boost, takes between 4.5 and 5 seconds to compile on an average of five runs.

LLVM generates bitcode, which could be cached on-disk, and be used to make following start-ups faster. This optimization is not yet applied and a complete recompile cycle currently occurs for each addon on startup.

The current "interactive" performance characteristics, while much slower than what the Javascript interpreter provides, are thus still viable for some level of live-coding.

6.2. Run times: benchmarking gain adjustment

We discuss here the runtime improvements provided by the system. The following cases cases are tested:

- The *gain* node of algorithm 1 as provided pre-built in the *ossia score* binary, which must work on a variety of systems and thus is not optimized for any kind of vector instruction set outside of the x86-64 SSE2 baseline.
- The same gain node, passed in the system presented in this paper which operates at an `-Ofast -march=native` optimization level and is thus able to take into account the user's actual CPU features.
- A manually optimized version of the gain node, done with hand-written AVX intrinsincs.

We measure every time the time taken by the computation for various common buffer sizes. Figure 2 gives the measurements for the first machine, figure 3 for the second machine.

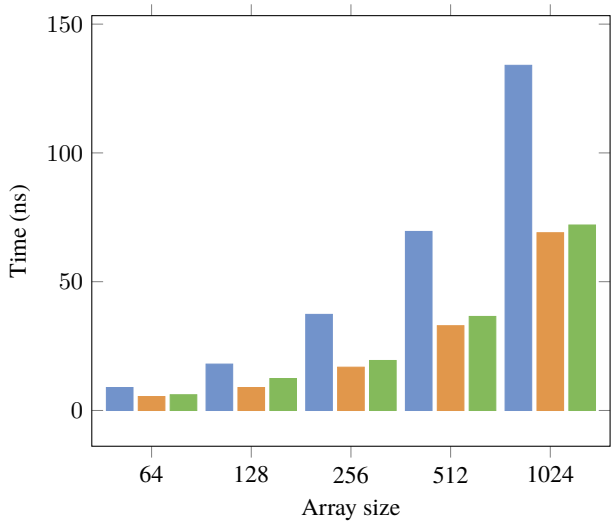


Figure 2: Broadwell CPU: average time in nanoseconds to compute a buffer. In blue: generic code with the default compilation settings. In orange: generic code while built with the JIT system. In green: manually-written AVX implementation.

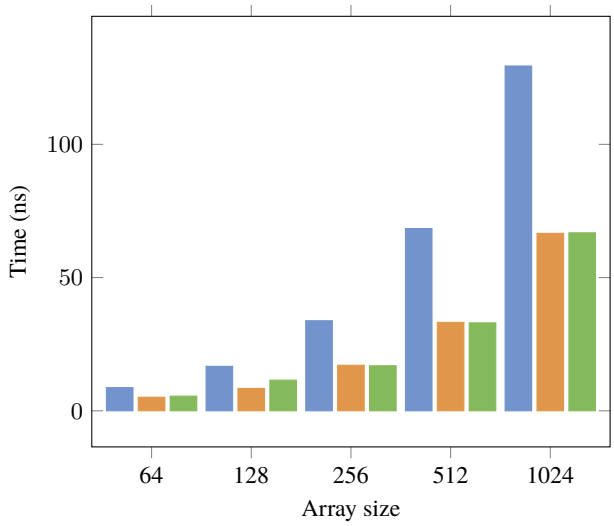


Figure 3: Results for the Coffee Lake CPU, following the same nomenclature than the Broadwell CPU.

Figure 4 presents the improvements between the two CPUs, in order to help the reader see the differences more clearly between figures 2 and 3.

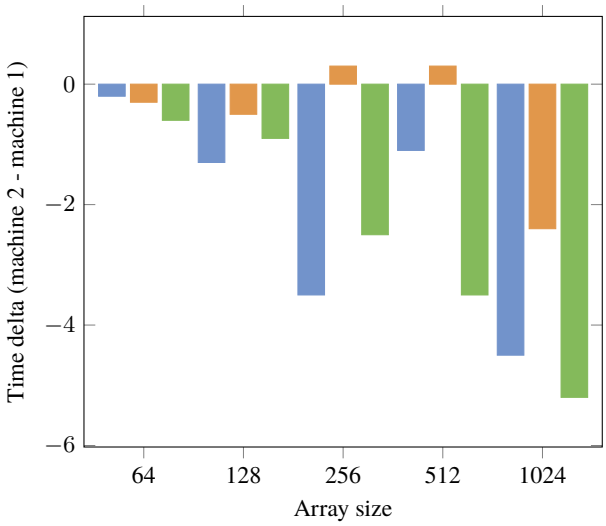


Figure 4: Performance difference between the Coffeelake and the Broadwell CPU: it is interesting to note that the buffer size heavily influences which workloads benefits the most from the CPU improvements.

6.3. Run times: benchmarking FFT

For this benchmark, we compare the run time of a Fast Fourier Transform algorithm implemented in the KFR library mentioned earlier. This library provides hand-optimized versions for many different instructions sets, ranging from SSE2 to AVX2. The results are presented in 1. The test is done on a large array: 16384 double-precision floating-point values.

Machine	Generic	JIT	Time saved
Broadwell	214 µs	144 µs	32.7%
Coffeelake	172 µs	107 µs	37.8%

Table 1: Performance increases yielded by using the proper instruction set.

6.4. Discussion

A few things are made apparent by the previous benchmarks:

- In simple cases, it is pointless to try to optimize better than what the compiler can: the manually-written AVX version is almost never faster than the simple for-loop version when optimized by the compiler.
- The improvement in that case is fairly expected: AVX is able to compute almost twice as many floats than SSE2 in the same time.
- In the more complex, hand-optimized case of the FFT, there are also important performance benefits.
- The C++ compile-times are certainly not negligible for large amounts of code. Potential paths for improvement could be the use of precompiled headers, or upcoming C++ modules.

In addition, we note that the system does not currently add any performance benefits – nor drawbacks – versus compiling the whole codebase at `-Ofast -march=native`. Thus, the system is mainly useful performance-wise in the case where the end-user is not able to rebuild the software himself. While on Linux systems this is generally not a problem (even though users may use old distributions with compilers unable to support recent editions of the C++ language required by *ossia score*), this is tremendously useful for Mac and Windows users where the default toolchain requires mutltiple gigabytes of disk space and takes a long time to install.

7. CONCLUSION

We presented the integration of a C++ just-in-time compilation system based on LLVM in an existing media authoring environment, *ossia score*.

There are multiple further steps that we would like to reach for the system:

- Correct live-reloading of addons. The main problem to handle is that a JIT-compiled addon may instantiate new objects in the system. These objects must be tracked, serialized and reloaded whenever the addon code change: else, due to the ABI of objects potentially changing, this will cause runtime crashes.
- Generation of cross-compiled code. An often requested feature for *ossia score* is to support embedded architectures. While the software already builds and run on such systems, it would be useful to generate a minimal executable for such platforms from a desktop machine, which only contains a given score with implementations optimized for the exact system being targeted.
- In longer time-scales, cross-unit-generator optimizations could be interesting: in particular, how can the system integrate with other languages also based on LLVM such as Faust ? The Mozilla team is currently researching cross-language inlining between C++ and Rust for instance. Combining multiple audio nodes written in different languages, and compile them together in a single dataflow graph may open further optimization opportunities.

Finally, the JIT denomination for the system could in practice be argued: since *ossia score* is itself an interpreter for a visual language, but the execution of the programs of this visual language are done only once every part of the system has been compiled to assembly: for reasons of safety, we prefer not to launch C++ compilations during the execution of a score, since it may seriously hamper the available performance of the system. The JIT process still allows this, but the user must be aware of the risks in doing so if the score already uses most of the machine's cores for instance.

8. ACKNOWLEDGMENTS

We would like to thank Thibaud Keller and the SCRIME & OSSIA teams for their tireless testing of new *ossia score* features, Martin Störsjo for the development of the llvm-mingw toolchain and Stefan Gränitz for simple examples on how to use LLVM's OrcJIT API.

References

[1] Nick Avramoussis et al. "A JIT expression language for fast manipulation of VDB points and volumes". In: *Proceedings of the 8th Annual Digital Production Symposium (DigiPro' 18)*. ACM. 2018.

[2] John Aycock. "A brief history of just-in-time". In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[3] Jean-Michael Celerier. "Authoring interactive media: a logical & temporal approach". PhD thesis. Bordeaux, 2018.

[4] Jean-Michaël Celerier et al. "OSSIA: Towards a Unified Interface for Scoring Time and Interaction". In: *Proceedings of the International Conference on Technologies for Music Notation and Representation (TENOR)*. Paris, France, 2015.

[5] Jean-Michaël Celerier, Myriam Desainte-Catherine, and Jean-Michel Couturier. "Graphical Temporal Structured Programming for Interactive Music". In: *Proceedings of the International Computer Music Conference (ICMC)*. Utrecht, The Netherlands, 2016.

[6] Théo De la Hogue et al. "OSSIA : Open Scenario System for Interactive Applications". In: *Proceedings of the Journées d'Informatique Musicale (JIM)*. Bourges, France, 2014.

[7] Chris Lattner and Vikram Adve. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*. IEEE Computer Society. 2004, p. 75.

[8] Juneyoung Lee et al. "Reconciling high-level optimizations and low-level code in LLVM". In: *Proceedings of the ACM on Programming Languages* 2.OOPSLA (2018), p. 125.

[9] Juneyoung Lee et al. "Taming undefined behavior in LLVM". In: *ACM SIGPLAN Notices* 52.6 (2017), pp. 633–647.

[10] Thor Magnusson. "Algorithms As Scores: Coding Live Music". In: *Leonardo Music Journal* 21 (2011), pp. 19–23.

[11] Yann Orlarey, Dominique Fober, and Stéphane Letz. "Faust: an efficient functional approach to DSP programming". In: *New Computational Paradigms for Computer Music*. Paris, France, 2007.

[12] Lance Putnam. "Gamma: A C++ sound synthesis library further abstracting the unit generator". In: *Proceedings of the Joint International Computer Music Conference (ICMC) / Sound and Music Computing Conference (SMC)*. Athens, Greece, 2014.

[13] Evgeniy Yur'evich Sharygin et al. "Dynamic compilation of expressions in SQL queries for PostgreSQL". In: *Proceedings of the Institute for System Programming of the Russian Academy of Sciences* 28.4 (2016), pp. 217–240.

BRINGING THE GRAIL TO THE CCRMA STAGE

Fernando Lopez-Lezcano
CCRMA
Stanford University, USA
nando@ccrma.stanford.edu

Christopher Jette
CCRMA
Stanford University, USA
jette@ccrma.stanford.edu

ABSTRACT

The Stage, a small concert hall at CCRMA, Stanford University, was designed as a multi-purpose space when The Knoll, the building that houses CCRMA, was renovated in 2003/5. It is used for concerts, installations, classes and lectures, and as such it needs to be always available and accessible. Its support for sound diffusion evolved from an original array of 8 speakers in 2005, to 16 speakers in a 3D configuration in 2011, with several changes in speaker placement over the years that optimized the ability to diffuse pieces in full 3D surround. This paper describes the evolution of the design and a significant upgrade in 2017 that made it capable of rendering HOA (High Order Ambisonics) of up to 5th or 6th order, without changing the ease of operation of the existing design for classes and lectures, and making it easy for composers and concert presenters to work with both the HOA and legacy 16 channel systems.

1. INTRODUCTION

We have been hosting concerts at CCRMA since it was created in the 70's. In 2009 we started expanding our concert diffusion capabilities while gearing up for the inaugural season of a new concert hall being built at Stanford, the Bing Concert Hall. In 2013 we were able to use our newly created GRAIL system (the Giant Radial Array for Immersive Listening) to diffuse concerts with out own "portable" speaker array with up to 24 speakers and 8 subwoofers arranged in a dome configuration for full 3D surround sound diffusion [1].

Figure 1: CCRMA Concert in the Bing Studio with the GRAIL

By 2011 our Listening Room Studio included a 22.4 speaker array in a full 3D configuration (with speakers below an acoustically transparent grid floor), which could accurately decode periphonic (full 3D) 3rd order Ambisonics. Our upgraded GRAIL concert diffusion system was also able to render up to 3rd order Ambisonics, or even 4th order if some errors in rendering were ignored. This was made possible by the publication of algorithms that allowed the design of HOA decoders for irregular arrays [2]. In particular, the release of the Ambisonics Decoder Toolkit software package written by Aaron Heller [3][4], which included software implementations of the aforementioned research, simplified the task of designing decoders. This work enabled the creation of successful diffusion strategies for irregular speaker placement in the Bing Concert Hall and its rehearsal space (the Studio), as well as other spaces. Both systems benefited from an open architecture based on the GNU/Linux operating system and many free audio software packages that, combined, allowed us to tailor the system to our specific needs.

We have curated many concerts with content of varied spatial resolution. As composers went on to create works requiring more speakers for a higher Ambisonic order decode, the limitations of our systems became apparent. While Ambisonics is well known for a graceful degradation of the spatial resolution when not enough speakers are available for the original order of the piece, the state of research and artistic creation was moving towards orders that were higher than what we could support.

1.1. From WFS tests to HOA in the Stage

In 2011 we bought 32 small speakers (Adam A3X) to create an experimental WFS array. Over the next few years we used it for demos and classes, but other than a couple of concert performances the system was used very sparingly. On the other hand, our Stage concert hall had a complement of 16 speakers and 8 subwoofers, which limited our ability to render full 3D HOA (we had been recently using a 32.8 system for our off-site concerts).

In an effort to upgrade our dedicated diffusion space at CCRMA, we proposed to re-purpose the "unused" speakers and add them to the existing Stage diffusion system. This addition would increase the total count of speakers to 48, and preliminary studies determined that we would be able to render up to 6th order Ambisonics quite accurately. Natasha Barret's research [5] points to diminishing returns in spatial performance for 7th and higher order decoding, so we felt confident that moving to a fifth or sixth order system would be adequate for our needs and a worthwhile upgrade.

The design and implementation of this upgrade ended up being anything but easy.

2. REQUIREMENTS

The existing system in the Stage consisted of 8 movable tower stands, each one housing a main speaker (four S3A and four P33 Adam high quality mid-field studio monitors) and a subwoofer (M-Audio

SBX10). In addition to those, we had 8 Adam P22 speakers hanging from the trusses and arranged as a ring of 6 with an additional two more overhead. All 16 speakers could be individually addressed from a Yamaha DM1000 mixer, with some limitations as the subwoofers were paired to the 8 main speakers - we used their internal crossovers - and could not be used by the upper 8 speakers.

The Stage is not only a concert hall, it is also regularly used for classes, lectures, demos and other events that do not need or want a high spatial resolution speaker array. In fact, the majority of users require access to just stereo playback. As the CCRMA concert events combine live performers, touring musicians and researchers, many concerts do not deal with 3D surround sound and use mostly stereo projection. The existing flexible 16 channel system allowed for creative diffusion using a combination of speakers and provided flexibility in which orientation the space could be used.

One of the key requirements for the upgrade was that the existing system and methods of operation would not be changed. Furthermore, the space sometimes is used to accommodate big audiences (for its size), so any addition to the Stage could not permanently encroach in the floor space available for setting up chairs for events.

These varied requirements complicated the design process in ways which we had not anticipated.

We were required to:

1. have a mode of operation that would keep the existing design, 8 main speaker and subwoofer towers plus 8 secondary speakers hanging from the ceiling trusses, all of them driven directly from our DM1000 digital mixer
2. not degrade the performance of the existing system in any way, including the low latency achievable with the digital mixer, appropriate for live performances
3. have a way to easily switch from the basic system to a fully expanded speaker array which added 32 speakers, all of them controlled through a single Linux based computer similar to the one managing diffusion tasks in our Listening Room [6][7]
4. have the ability to physically move the additional small speakers positioned at ear level out of the way, so that they would not interfere with the existing floor footprint of the diffusion system
5. easily switch between the two modes of operation, preferably with "one big switch" that would need no expertise from the operator
6. the system had to be "low cost"

This created a situation with many mutually incompatible system requirements from a design standpoint.

3. FEASABILITY TESTING

Before starting the upgrade a practical question had to be answered: were the tiny A3X speakers good enough (in quantity) to be able to produce enough SPL for a concert diffusion situation? Matt Wright and Christopher Jette organized a quick test session in which we installed 16 speakers in a ring at ear level (on top of chairs and plastic bins!) and drove them from our GRAIL concert control computer. This test was successful and confirmed that they were up to the task, but only if properly equalized, so we could go ahead with the upgrade.

4. LOCATION, LOCATION, LOCATION

Where and how to mount all speakers was a difficult task, made harder by the rectangular shape of the room and the presence of trusses that hold the cathedral-style ceiling. To arrive at a preliminary even distribution in space we used a simple successive approximation software that treats speaker locations as electrons that repel each other, and determines the approximate ideal locations of the speakers [8]. Additional constraints were introduced in the software to "fix" the position of the existing 16 speakers in space (remember that our design must be a superset of the existing system), and see where the rest of the speakers would fall.

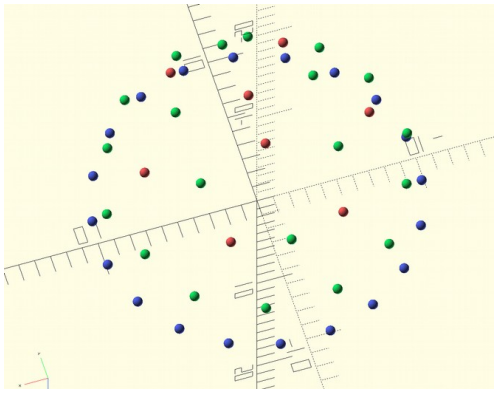


Figure 2: Ideal projection of speaker locations on a hemisphere (red dots: original upper 8 speakers, blue dots: ear level speakers)

A simple geometrical model of the Stage created in OpenSCAD [9] was used to project those ideal locations into the walls and ceiling of the Stage, to see where we might approximate the ideal locations in space with real mounting points. It was challenging to find locations which would not be shadowed by the ceiling trusses for most of the audience seating space, and in a couple of instances there was unavoidable shadowing that we had to ignore.

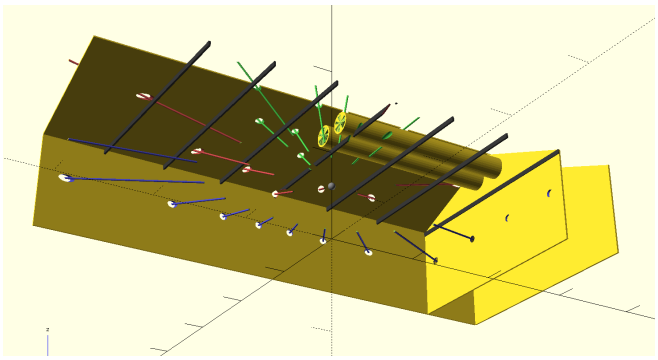


Figure 3: OpenSCAD model of the Stage (seen from below) with speaker location projections, the cylinders partially represent the A/C ducts, the black beams are the lower part of the trusses

We used ADT (the Ambisonics Decoder Toolkit)[3][4] as a design verification tool, in particular the energy and particle velocity graphs helped us determine if the proposed mounting locations for the speakers would provide uniform coverage for the desired Ambisonics orders (5ht and 6th order was the goal). Other diffusion

methods (VBAP, etc) would also benefit from a uniform spatial distribution of the speakers.

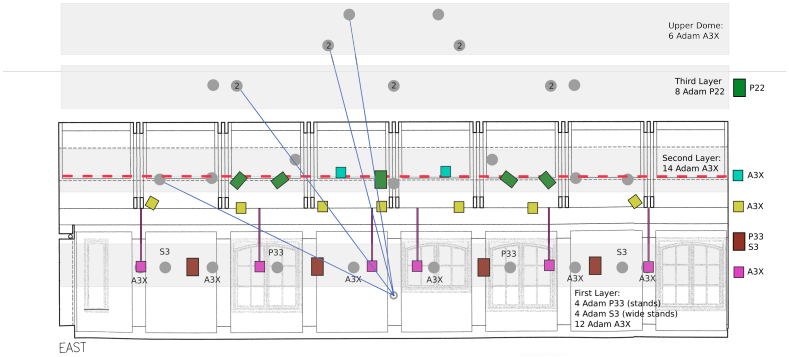


Figure 4: Side view of the Stage with speaker mounting points. Grey dots are ideal positions in a hemispherical dome, colored rectangles are the real positions

The final speaker configuration at which we arrived was an ear level ring of 20 speakers (the 8 original towers plus 12 additional A3X speakers), another ring of 14 A3X speakers mounted on the trusses (roughly 20 degrees in elevation above the first ring), and the original 8 speakers (roughly 20 degrees of elevation higher) plus 6 more A3X's distributed in the upper part of the dome. The 12 ear level speakers could not be mounted on stands that would take away floor space needed for seating, and had to be able to be moved out of the way when not in use. We installed a truss mounted rail system and designed telescoping mounts that could be switched between the normal listening position and a "parked" position where the 12 small speakers are moved next to the existing towers. The mechanical design took a long time and several prototypes were built and tested. Our final system features custom fabricated mounts made from 80/20 extruded aluminum profiles and hanging steel channel to facilitate rolling the speakers between locations.

5. DRIVING MANY SPEAKERS

One of the difficult aspects of the design process was finding an audio routing and distribution technology that would allow us to satisfy all the requirements within a reasonable amount of time and with the limited budget and manpower available to us. Furthermore, the full system needed to be controlled from a computer running GNU/Linux (like our Listening Room system), and Linux desktops and laptops should be able to connect to it for diffusion tasks.

For our GRAIL concert sound diffusion system we had been using a homebrew system which consisted of one half of a network snake (the Mamba box), plus some ingenious software in the form of a Jack[10] client (jack-mamba [11]), to transform it into a very reliable 32 channel D/A converter. While the system proved to be rock solid for our concerts, it was not really expandable in a way which could satisfy our requirements.

The first audio technology we explored was MADI. We had used RME MADI audio interfaces which had good driver support in Linux in our Listening Room system. For this 22.4 system we had to use two cards, one RME MADI and one RayDAT. This type of system could scale up to the number of inputs and outpus that we needed, but we could not find an easy way to control rerouting of connections to

Figure 5: Speaker mount

support both modes of operation. The only reasonable cost option we found was an RME MADI switching matrix, but switching between MADI scenes required several operations on the front control panel, and there was no option for remote software control which would have enabled us to design a separate simple to use interface.

Our experience with the ethernet based Mamba digital snake system suggested that a similar technology based on ethernet could be an answer to meet our requirements.

There are several protocols that rely on ethernet connections to transport audio and interconnect several audio interfaces together. The most widespread commercially so far has been Dante, but that was ruled out as the protocol specification is closed and proprietary, and there is no formal support for Linux. There is one company that offers a 128 channel ethernet card with associated Linux binary drivers, but there is no guarantee that this will be supported for future kernel upgrades and the card and driver combo is extremely expensive.

AVB (Audio Video Bridging) [12], on the other hand, is an open standard with a free software implementation embodied in the openAVNu project [13]. Regretfully not many manufacturers have used this standard for their products. One product manufacturer we considered was Motu, as their newer audio interfaces can be connected to each other through AVB and standard ethernet cables. Their internal configuration can be completely controlled through a built-in web server which makes it platform agnostic, and there is a published API that can use JSON http requests and OSC to remotely control all aspects of its operation. A Linux computer could control the full system without relying on proprietary software.

Regretfully the AVNu project does not yet include code for a complete Linux-based solution. It would be possible to create one, but that would require a substantial software development effort which was beyond the scope of the resources available to this project.

We bought a couple of interfaces for evaluation and experimented with using their USB interfaces. In the most desirable MOTU cards we found that the implementation of the USB2 class compliant driver was limited to 24 channels, which was much less than what we

needed (the cards were advertised as having 64 channel I/O through USB2, but that was only possible when using their proprietary driver). So we were at an impasse.

5.1. Firmware giveth...

Almost by chance we found an online reference to a "64 channel mode", and traced it back to a very recent firmware upgrade that added a mode selection configuration option to the USB audio interface. The new firmware allowed us to set the maximum number of channels handled by the USB class compliant driver to 64 if the sampling rate was limited to 44.1 and 48KHz, which was acceptable for our use case. This is beyond what the USB2 specification can do, but it performed well in tests under Linux, and allowed us to potentially address all speakers through the GRAIL control computer's USB2 interface, while multiple additional audio interfaces could communicate audio data through AVB. This new feature also would enable end users to interface with the finished diffusion system using another audio interface with its own USB2 interface. This would provide multiple entry points into the system using just USB2, making it easily usable by our users.

A firmware upgrade transformed the Motu hardware into a viable option. But what firmware can give, it can take away, as we will see...

5.2. Digital Mixer Mode

The first phase of the design centered around finding a configuration that could keep the old setup of DM1000 plus 8 main speakers operational with minimal changes. Some simple tests determined that routing the DM1000 to a 16A Motu interface through ADAT so it would drive the speakers (instead of the DM1000 driving them directly) would not change the latency of the system significantly. This 16A audio interface would also be the word clock master for the whole system, and this basic setup would depend on only the DM1000 and that interface being up and running to work.

This means that the 16.8 legacy system (we will call this the "Digital Mixer Mode") could be kept unchanged, and could be a subset of the full 48.8 system (the "OpenMixer Mode").

5.3. Routing the Subwoofers

There was a very long design detour that tried to use the internal crossover of the old subwoofers in "Digital Mixer Mode" as they were working fine and everybody wanted to keep their well known sound. We are going to skip those 4 months and jump straight into the design that incorporated new subwoofers much later.

The subwoofer upgrade proved to be a problem, both from the point of view of signal routing and from the specs that they had to meet. We wanted to have standalone crossovers when in "Digital Mixer Mode", and software crossovers implemented in the GRAIL control computer when in "OpenMixer Mode". We also wanted to have a rather high crossover frequency (originally 110Hz, currently about 90Hz) to minimize the cone excursion of the main speakers at low frequencies (they are mid-field monitors and almost too small for the space, but we love their very precise sound). And we wanted a low frequency limit of around 20Hz with enough power to fill the room without clipping or distortion.

The ideal subwoofer that would meet all our requirements does not exist (the details of why that is the case are beyond the scope of this paper). We ended up buying SVS SB4000 units, and not using the internal DSP processing included in the unit.

The only workable solution we found was to use external programmable crossovers when the system was operating in "Digital Mixer Mode". We used DBX 260 units and routed them through inputs and outputs of the same Motu audio interface used to drive the 8 main speakers (this back and forth tour added a tiny bit of latency). In "Digital Mixer Mode" the DBX crossovers are inserted into the signal path by the internal routing of the Motu audio interfaces, and in "OpenMixer" mode they are completely disconnected so that the GRAIL control computer can directly interface with speakers and subwoofers, and provide its own separate digital crossovers. In "Digital Mixer Mode" the signals going to the 8 main speakers are routed to the crossovers which split it between the main speakers and to the corresponding subwoofers, in "OpenMixer Mode" all speakers are mixed in to the 8 subwoofers. All the signal switching is accomplished using the routing matrix that is part of the Motu audio interfaces.

The use of external crossovers also allowed us to properly match phase at the crossover frequency and equalize the whole system in "Digital Mixer mode" for best performance, something we could not do before the upgrade.

Another 16A Motu interface drives the upper 8 speakers with signals that are sent from the digital mixer through AVB and the internal routing matrices of both audio interface cards.

The core system in "Digital Mixer Mode" consists of two Motu 16A cards, the DBX crossover units and the DM100 digital mixer. That not only keeps the same operational characteristics as before, but improves the system through better crossovers and speakers.

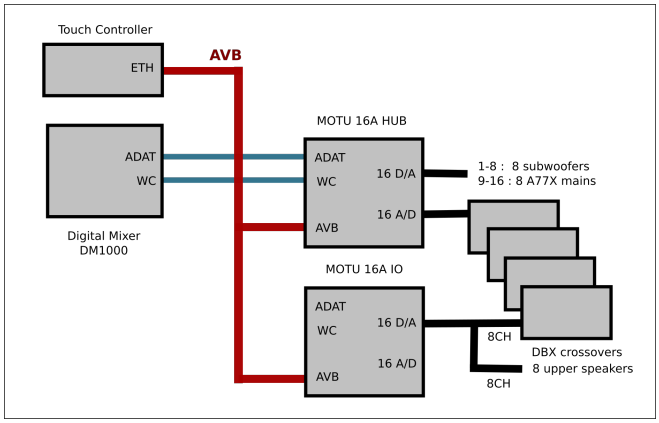


Figure 6: Signal routing in Digital Mixer Mode

5.4. And firmware taketh away...

In the middle of the design and implementation of the system we found that newer Motu interfaces no longer had the 64 channel mode configuration option. It turns out that Motu had "unspecified problems" with it, and removed the feature from their products through another firmware upgrade.

Suddenly the audio interfaces were useless for our purposes (24 channels instead of 64), with no fix coming from Motu, after all, they worked fine with their proprietary drivers. To make a long story short, we were able to downgrade the firmware to a version where that feature was still supported, and everything worked again. A not

very sustainable fix and a hack as we (and possibly random users of the system) have to ignore the constant reminders that a "software upgrade is available".

While software upgradable products offer useful flexibility, you never know when something you depend on might go away, or some new and exciting capabilities might be added, and in which order that might happen. That was not the last problem we had with Motu firmware versions.

5.5. OpenMixer Mode

With the core architecture now a working reality, we added three 24Ao Motu audio interfaces hidden in the ceiling trusses of the Stage to drive the additional 32 small speakers (two would have been enough, but using three made the wiring easier and wiring represented a large time expenditure in this upgrade). An additional 16A in the OpenMixer control computer rack (on casters) acted as the interface between the OpenMixer Linux control computer and the rest of the system, using a single USB2 interface. AVB streams are used to send and receive audio to all other Motu audio interfaces, and finally to all speakers, and changing the internal routing in the audio interfaces through JSON http calls configures the audio routing for the two main modes of operation.

An additional 24Ai audio interface in the OpenMixer system rack is the entry point in the system for connecting laptops and other computers for concert diffusion or other purposes (Windows, OSX and Linux are all supported). A single USB2 cable allows us to have up to 64 channels of input/output available, which is enough for our current needs. AVB and the internal routing of the interfaces is used to send signals around.

Yet another 16A audio interface is used to interface with our dedicated Linux desktop workstation which resides on another cart together with its display, keyboard and mouse. A total of 8 Motu audio interfaces interconnected through AVB make up the audio part of the diffusion system.

Three Motu AVB switches connect all the audio interfaces together, and the different racks and mobile units in the space are easily connected through long ethernet cables (one mobile rack for the digital mixer and associated equipment, another for the OpenMixer control computer and another one for the desktop computer). The use of ethernet means there is a **significantly** smaller cable count to manage 64 channels of audio.

5.6. Switching modes

The attentive reader might have noticed that switching between "Digital Mixer Mode" and "OpenMixer Mode" seems to be happening magically so far. While we do have a Linux control computer, we cannot rely on it for switching modes. The system should keep working even if the control computer is off, or if it breaks down.

A solution that has worked admirably well is to add yet another computer (as if the system was not complex enough). This additional computer is a RaspberryPI 3 with a touch panel, mounted right next to the digital mixer. It allows the user to switch sampling rates, switch between operating modes and even activate different options in "OpenMixer mode" (changing between the Direct and Ambisonics modes, selecting Ambisonics decoders, etc). It communicates through ethernet with all the Motu audio interfaces and the main OpenMixer control computer.

The OpenMixer control computer also has a touch display, and

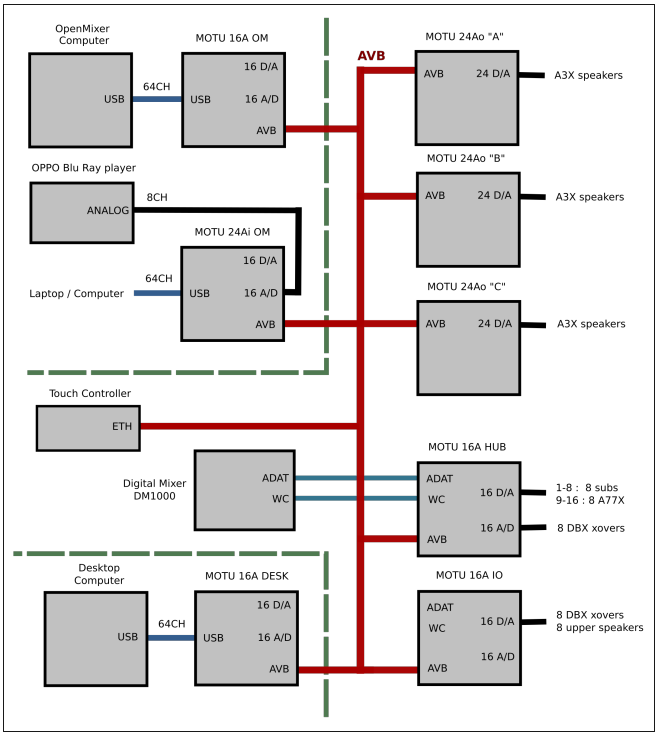


Figure 7: Signal routing in OpenMixer Mode

the software was designed so that either of them can be used to control the system and they stay synchronized with each other.

5.7. What? More Speakers?

Quite early in the implementation process Christopher Jette pushed for the immediate inclusion of something we had planned as a future expansion. In addition to the existing subwoofer and main speaker, the eight main towers would house 8 speakers almost hugging the ground. These speakers were included to help "pull down" the sound image, specially in the Ambisonics decoder modes. So our final speaker count is 56 speakers and 8 subwoofers, adding up to 64 individual outputs. We are maxed out.

5.8. Control Software

In "OpenMixer Mode" the Linux control computer (currently booting Fedora and running an optimized RT patched kernel) performs all internal DSP using SuperCollider[14] and its Supernova multi-core load-balancing sound server [15]. Jconvolver [16] is used for very efficient low latency partitioned convolution, and implements the digital loudspeaker correction filters. The software itself is conceptually simple, it provides for level and delay equalization of all speakers, digital crossovers (a combination of Linkiwitz Rayley [17] and Butterworth filters), routing control so that different sound sources (digital mixer, laptop, desktop) can be connected to the speakers, optional built-in Ambisonics decoders created with ADT [3][4](up to 6th order) and of course digital equalization of all speakers with convolution filters created from analyzing their measured impulse responses with the DRC (Digital Room Correction [18]) software package.

SuperCollider is started automatically on boot through a systemd

unit and takes care of orchestrating the rest of the system startup process. First, Jack [10] is started, then the SuperCollider program starts the Supernova sound server and its associated DSP software, two instances of Jconvolver, and finally everything is connected together using aj-snapshot and dynamically generated XML connection files. SuperCollider monitors all auxiliary programs, and restarts and reconnects them if they somehow fail.

The whole system is optimized for low latency, and currently runs with 128 frames per period (work is underway to get it to work at 64 frames per period, which would start approaching the performance of the digital mixer which runs with 64 frame blocks).

SuperCollider is also used for the touch graphical user interface in both the main computer and the small RaspberryPi switching appliance.

5.9. Calibration

For best performance the full speaker array is calibrated after the initial installation and when hardware changes are made. First elevation and azimuth angles for all speakers are measured, as well as the distances to the center of the space. These measurements are used to create the Ambisonics decoders for the main array and the subwoofers, and also to compensate for arrival times at the center of the space. After that we use Aliki [19] to measure the impulse response of the speakers, and that information is used to calculate convolution filters using DRC. Finally SPL measurements are done to compensate for small differences in speaker loudness in both direct and Ambisonics modes.

6. PROBLEMS AND CHALLENGES

While the selection of Motu products lead to a viable design, there are still occasional problems when using them on "unsupported platforms".

Occasionally an audio interface can disconnect from one or more of its AVB streams. The web interface shows them blinking and we have not found a way out of this other than rebooting both interfaces. After the reboot the connections are re-established automatically. We have not been able to find a way to reproduce this, and it only happens in the more complex Stage system we are describing in this paper (it has not happened, so far, in a far simpler system now running in our Listening Room). We have to do an thorough audit of the existing streams and only enable exactly what we need. This may be a problem solved in later firmware releases, but we are chained to older ones to retain the features that make the system possible in the first place.

In a different Studio in which we also deployed a single Motu interface we found another firmware related problem when using the class compliant driver under Linux. Suddenly inputs going into the computer through USB would switch channels in blocks of 8. What was coming through input 1 is suddenly in input 9, and so on and so forth. Again, downgrading to a previous firmware version fixes the problem (or using the proprietary driver). *Caveat emptor*.

In terms of the Linux control computer for the Stage system, the long term solution for interfacing with the audio interfaces is to use AVB streams directly. That would lift the 64 channel limitation (we of course would like to add a few more speakers), and hopefully make the system more reliable. The foundation of that is available in the OpenAVNu git repository but much work remains to be done (some preliminary tests managed to sync the Linux computer to the AVB clock, and get the system to recognize the existence of a Motu card).

6.1. Motu vs. Jack vs. PulseAudio

A weird feature of the Motu interfaces is that every time the sampling rate is changed (even if it is an internal change and the card is not slaved to an external clock) it takes the card a few seconds to acquire a "lock". During this time Jack can try to start, but at some point it decides that it can't, and fails.

This can lead to an endless loop of failed starts in the following scenario: assume the card is already running at 44.1KHz and we are trying to start Jack at 48KHz. Jack requests exclusive access to the card from PulseAudio and the request is granted. Jack tries to start but fails, because the card was running at 44.1KHz and it takes time to switch to 48KHz. After the attempt the card is switching to 48KHz, but when Jack quits it hands the card back to PulseAudio, which promptly resets its sampling rate to its default, 44.1KHz. And we are back where we started. There is no way to start Jack, unless PulseAudio is killed or its default sampling rate is changed to the one we want, or we tell it to ignore the card, which is not what we want to do.

If there is no change in sampling rate and Jack fails to start, waiting a few seconds and trying again succeeds.

To avoid this problem, in the control software for both the Listening Room and Stage Linux computers we use a JSON http call to check the lock status of the audio interface clock and delay the start of Jack until the sampling rate is locked.

7. CONCLUSIONS

The opening concerts of the newly upgraded Stage took place in October 4/5 2017, and the system performed very well (at the time we were still using the old subwoofers). Another round of upgrades in 2018 replaced the original subwoofers with newer ones, as outlined above, and also upgraded the main 8 speakers with newer A77X Adam monitors. The lower layer of speakers were repositioned at the bottom of the main towers, and the new subwoofers were stacked immediately above them (originally they had been reversed). A second round of successful concerts (our annual Transitions concerts) took place in October 2018 with the fully upgraded array. The full array has seen more use in the past year, with several concerts using it instead of what would have been stereo or quad diffusion.

We have outlined the design process of a complex Linux-based diffusion system, using off-the-shelf components and GNU/Linux for all the software components.

8. ACKNOWLEDGMENTS

This would have been impossible to accomplish without the support of CCRMA and its community. Many many hours of discussions made for a better system that satisfies all the use cases of the space. Endless critical listening tests honed the system into better and better sound quality. Many thanks to Eoin Callery for his contributions to the design and keen ears, and for keeping us grounded at all times (we tend to fly away). The whole project would not have happened had we not had Christopher Jette on the CCRMA Staff at the time. He pushed and worked and talked and discussed and designed and kept things going. Invaluable. Matt Wright, our Technical Director, also spent many hours helping with big and small details. Many students helped, in particular thanks to Megan Jurek,

Figure 8: Transitions 2018 concert

who spent many hours soldering many many small connectors, and routing what seemed like miles of cables. No audio would flow if not for her help. Jay Kadis, our audio engineer at the time, also spent quite a bit of time wiring DB25 connectors and cabling the main towers. Juan Sierra, one of our MA/MST students, was instrumental in properly phase matching of the new subwoofers with the main speakers and tuning the crossovers for best performance, the Stage sounds much better thanks to him. Carlos Sanchez, sysadmin and staff at CCRMA, designed and implemented the hardware and software that drives the touch interface that controls the whole system. And Constantin Basica, our new concert coordinator, has been helping visiting artists use the full system for much more interesting concerts over the past year. Many thanks to all involved, we can now do justice to many fantastic pieces from composers that tickle our ears with beautiful sounds arranged in space.

9. REFERENCES

[1] Fernando Lopez-Lezcano, "Searching for the grail," *Computer Music Journal*, vol. 40, no. 4, pp. 91–103, 2016.

[2] Franz Zotter and Matthias Frank, "All-round ambisonic panning and decoding," *J. Audio Eng. Soc*, vol. 60, no. 10, pp. 807–820, 2012.

[3] Aaron Heller, Eric Benjamin, and Richard Lee, "A toolkit for the design of ambisonic decoders," *Proceedings of the Linux Audio Conference 2012*, 2012.

[4] Aaron Heller and Eric Benjamin, "The ambisonics decoder toolbox: Extensions for partialcoverage loudspeaker arrays," *Proceedings of the Linux Audio Conference 2014*, 2014.

[5] Thibaut Carpentier, Natasha Barrett, Rama Gottfried, and Markus Noisternig, "Holophonic sound in ircam's concert hall: Technological and aesthetic practices," *Computer Music Journal*, vol. 40, no. 4, pp. 14–34, 2016.

[6] Fernando Lopez-Lezcano and Jason Sadural, "Openmixer: a routing mixer for multichannel studios," in *Proceedings of the Linux Audio Conference 2010*, 2010.

[7] Elliot Kermit-Canfield and Fernando Lopez-Lezcano, "An update on the development of openmixer," *Proceedings of the Linux Audio Conference 2015*, 2015.

[8] A. B. J. Kuijlaars E. B. Saff, "Distributing many points in a sphere," in *The Mathematical Intelligencer*, Volume 19, Number 1, 1997.

[9] "Openscad, the programmers solid 3d cad modeller," `http://www.openscad.org/`.

[10] Paul Davis, "Jack audio connection kit," `http://jackaudio.org/`, 2002.

[11] Fernando Lopez-Lezcano, "From Jack to UDP packets to sound and back," in *Proceedings of the Linux Audio Conference 2012*, 2012.

[12] "The avnu alliance (avb)," `https://avnu.org/`.

[13] "Openavnu git repository," `https://github.com/AVnu/OpenAvnu`.

[14] J. McCartney, "Supercollider: A new real-time synthesis language," in *Proceedings of the International Computer Music Conference*, 1996.

[15] Tim Blechmann, "Supernova: a multiprocessor aware real-time audio synthesis engine for supercollider," M.S. thesis, TU Wien, 2011.

[16] Fons Adriaensen, "Jconvolver, a convolution engine," `http://kokkinizita.linuxaudio.org/linuxaudio/`, 2006.

[17] Siegfried Linkwitz, "Active crossover networks for noncoincident drivers," in *Journal of the Audio Engineering Society*, Volume 24 Issue 1, 1976, pp. 2–8.

[18] Denis Sbragion, "Drc: Digital room correction," `http://drc-fir.sourceforge.net/`, 2002.

[19] Fons Adriaensen, "Aliki, an integrated system for impulse response measurements," `http://kokkinizita.linuxaudio.org/linuxaudio/`, 2006.

[20] Ingo Molnar and Thomas Gleixner, "Real-time linux, the preempt_rt patches," `https://wiki.linuxfoundation.org/realtime/start`, 2000.

Figure 9: Transitions 2017 concert

RENDERING OF HETEROGENEOUS SPATIAL AUDIO SCENES

Nicolas Bouillot, Michał Seta, Émile Ouellet-Delorme, Zack Settel, Emmanuel Durand

Société des arts technologiques [SAT]
Montréal, Canada
[nbouillot,mseta,eodelorme,zack,edurand]@sat.qc.ca

ABSTRACT

We present the ability of our Spatial Audio Toolkit for Immersive Environment (SATIE) to render simultaneously real-time audio scenes composed of various spatialization methods. While object oriented audio and Ambisonics are already included in SATIE, we present a prototype of a directional reverberation method based on Impulse Response computation and describe how this method will be included in SATIE.

1. INTRODUCTION

A growing number of computer music performance venues are now equipped with large loudspeaker configurations [1], and therefore provide new opportunities for artists using 3D audio scene environments for composition and sound design. This, along with the recent rise of affordable spatial audio recording devices and increased interest in virtual reality experiences, gives rise to a growing need of combining multiple spatialization methods: captures (live or not) made in different ambisonic formats, mono object-based audio sources as well as flexible & adaptable speaker configurations. We anticipate the evolution of spatial audio composition — targeting performing arts, installations or any other immersive experiences — involving different types of audio sources such as live audio capture, field recordings and synthetic audio, and where visual[2] and haptic[3] correlates with the audio part.

Moreover, innovation from the game industry is pushing forward virtual and augmented realities, approaching spatial audio with an object oriented manner: sources are *sound objects*, located in space and controlled with low level parameters such as gain, equalizer and spread. This approach, although effective for speaker array systems, is missing architectural acoustical responses and adapts poorly to non clearly located sound sources such as the sound of a river. The 3D graphic world is now entering audio and provides methods for the simulation of sound based on physics of soft body vibration and sound propagation [4]. Although such simulations are probably hard to achieve in real-time, simulations of acoustic responses of 3D environment may improve significantly the coherence of the integration of audio sources with the virtual space, while still allowing a real-time & 6-DoF navigation [5]. The use of ray tracing algorithms for real-time rendering is appropriate [6] and has the advantage of including the direction of the sound during auralization [7], allowing real-time calculation of directional sound reflections.

One of the main challenges today for spatial audio render is to support the multiplicity of the i) audio display methods, ii) spatial audio algorithms and iii) spatial audio authoring and 6-DoF navigation in spatial audio [8]. To date however, many existing real-time 3D audio scene rendering systems, such as COSM [9], Blender-CAVE [10], Spatium [11], Zirkonium [12], CLAM [13], 3Dj [14], Panoramix [15] and the spatDiff library [16] mostly focus on trajectory based composition with object oriented audio and sound fields with ambisonics. The challenge of navigating in heterogeneous spatial audio content is illustrated with Figure 1, where the spatial audio scene is constituted from 360° audio/video footage where the sound field captured using an ambisonic microphone[1] is mixed with synthetic audio is spatialized through an object oriented approach and correlated with 3D objects on screen (the white bubbles coming out from the white vase).

Figure 1: Example of an augmented reality application where a combination of several spatialization algorithms (ambisonics and object oriented audio): a 360° audiovisual capture is rendered simultaneously with synthetic objects, the bubbles coming out from the white vase.

In this paper, we present how our Spatial audio Toolkit for Immersive Environments (SATIE[2]) addresses the challenge of several approaches to audio scene rendering, possibly combining simultaneously object based audio, ambisonic formats and architectural based acoustical spatialization.

2. SATIE

The development of SATIE (with the SuperCollider language [17]) was first motivated by the need to render dense and rich audio scenes the Satosphere, a large dome-shaped audiovisual projection space at the Society for Art and Technology [SAT] in Montreal, and to compose real-time audio/music scenes consisting of hundreds of simultaneous sources targeting loudspeaker configurations of 32 channels or more, and sometimes with two or more different audio display systems [18]. In fact, SATIE easily adapts to different audio display configurations and supports plugins architecture which makes it easily extensible to new situations. As such, it fills the role of a rapid

[1] The Zylia ZM-1 microphone.

[2] https://gitlab.com/sat-metalab/satie, accessed Dec. 2018

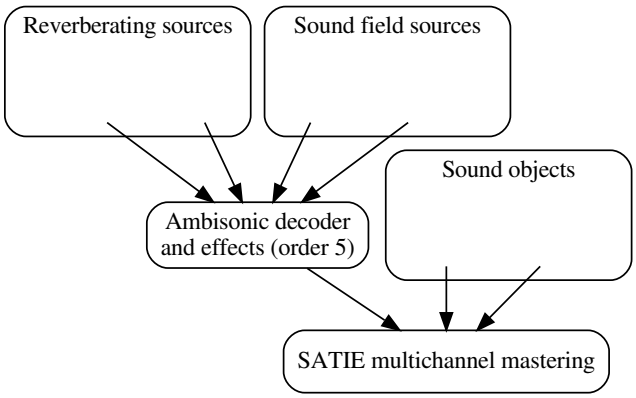


Figure 2: Example pipeline of a spatial rendering involving heterogeneous audio sources: reverberating sources, sound field sources and sound object.

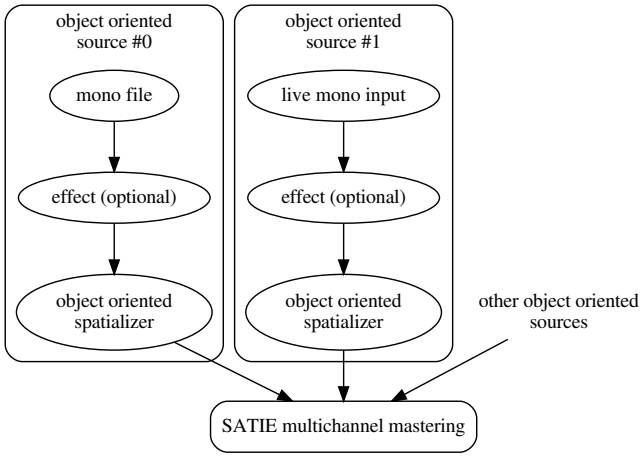


Figure 3: Internal pipeline for object oriented sound spatialization

prototyping tool for spatial audio composition.

Control of sound sources in SATIE is done through unified OSC [19] messages allowing for life management of each sound sources, along with (possibly custom) parameters control.

3. RENDERING METHODS

Facing a variety of approaches to composition with dense audio structures and a variety of audio displays, SATIE implements a flexible rendering pipeline allowing mixing of different audio input formats and multichannel mastering and is easily adaptable to various audio displays. We rely mainly on SuperCollider's *supernova* rendering engine for multi-threading operation. Consequently, we have access to parallel groups[20] which solve some real-time related issues with synth instantiating and bus allocation. SATIE structures different types of audio processors in layers, represented by a hierarchy of parallel groups (ParGroups):

- audio sources
- effects
- post-processors.

Audio sources are different types of mono or multichannel audio generators and players. On the second level are effects which usually do not generate sound but modify the signal of audio sources. Finally, post-processors are meant as mastering stage, where the final stages of DSP are done. In the actual implementation, the post-processors are divided in two groups: one for b-format signals and one for traditional mono/multichannel signals.

The signals between audio sources and effects pass through busses, i.e. the user allocates *auxiliary* busses and manages the bus access on both, the generator and effect side. If any post-processors are present, all signals are collected there, otherwise, they bypass directly to the spatializer. Multiple spatializers can be used, in which case SATIE will create appropriate number of output channels.

Figure 2 shows a rendering pipeline that combines object based audio sources, sound field sources and reverberating sources into heterogeneous mix.

3.1. Object Based Audio

Object audio (Figure 3) is what is most commonly used in various entertainment industries where a sound source has a clearly defined position within the coordinate system [21]. SATIE supports different types of object based audio sources, such as mono audio, mono live input sources and synthesized sounds [22]. The spatializers handling object audio expect azimuth, elevation and gain for panning each audio object.

SATIE was initially designed to render large numbers of mono audio sources, optionally with effects, to large multi-channel loudspeaker systems. Audio sources and effects can be placed in groups and controlled either per group or on individual basis. Similarly, spatializers take mono signals and place them on different channels according to azimuth, elevation and gain parameters. The post-processing audio object is comparable to mastering effects in a studio or live pipeline, typically limiting, compressing or normalizing signals.

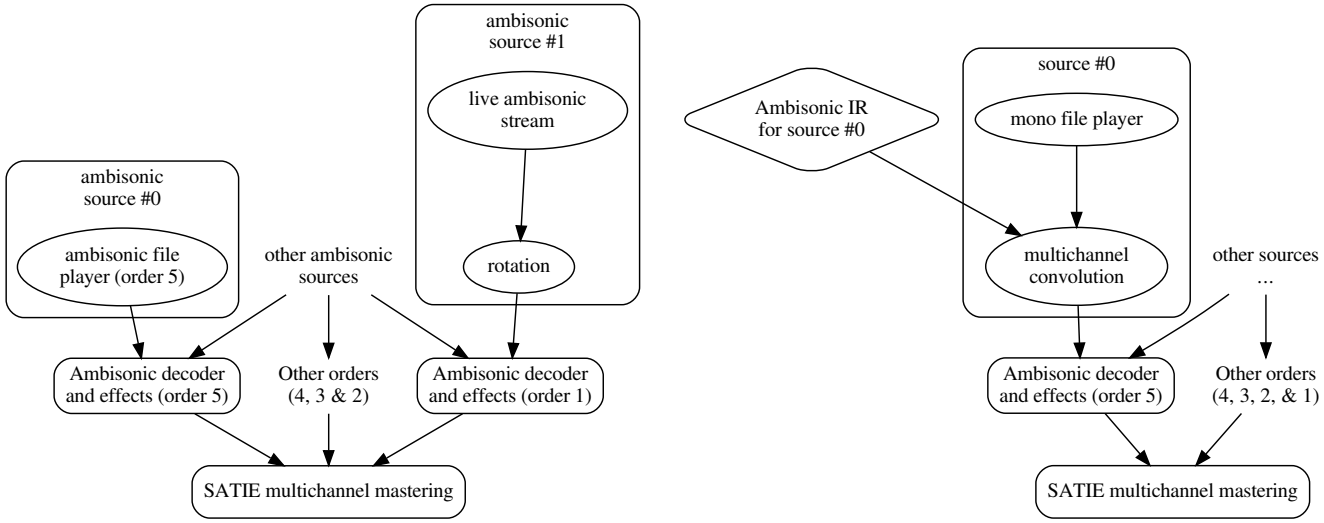
While all parameters (audio object specific as well as spatialization) can be modified either directly from the SuperCollider language, SATIE supports OSC and our preferred method is using a 3D engine for "volumetric" control of the sources as well as actual geometry computation. In line with this object based approach and load balancing physical computation we were able to use particle swarms of hundreds simultaneous sound sources.

3.2. Ambisonics

Ambisonic pipeline, implemented via SC-HOA plugins/quark [3] (Figure 4(a)) provides means to play multichannel files, live audio inputs, encode mono signals into b-format signals and transcode between different ambisonics formats (ACN and FuMa). It supports b-format up to order 5.

SATIE supports ambisonics with the same approach to signal path. The ambisonic audio input can be sent to ambisonic effects and post-processors such as rotation, mirroring, and beamforming filtering. The significant cost of ambisonic decoding is payed only once since not embedded in each ambisonic source pipeline, but rather at

[3] `https://github.com/florian-grond/SC-HOA`



(a) Pipeline for ambisonic sources.

(b) Pipeline for convolution based spatialization.

Figure 4: SATIE pipeline involving ambisonics

the post-processor stage. We can also transcode between different ambisonic orders.

3.3. Reverberating Sources with Convolution Reverb

Having various audio rendering methods driven by 3D engines opens doors to the desire of simulating acoustic spaces. Consequently, we have started developing a tool for real-time generation of impulse responses through ray tracing with the idea of integrating the IR workflow with SATIE. Figure 5(a) shows a screenshot of a real-time rendered frame where the listener is facing a sound source represented by a cube at the end of the hallway. Figure 5(b) shows a wireframe view of a simple model (not related to the picture on the left) showing what is actually going on. The black dots on the inner faces of the model represent the impact points of the rays on the walls of a 3D model. Sound sources and the listener are not shown, it simply shows a point cloud mapped on the model for reference. This implementation uses another custom software, VARAYS [4], which shares the 3D model with the 3D engine (in this case we're using EIS), receives the coordinates of the sound sources and the listener and writes IR files to disk. The IR files are read by SATIE which continuously replaces the buffer read by SuperCollider's PartConv UGen. A crude prototype of this process (using mono convolution) is demonstrated in the following video `https://vimeo.com/306202441`.

Besides mono IR, we can also generate Ambisonic IR (AIR), although at the time of the writing, this process has not yet been integrated into SATIE.

4. CONCLUSION

This paper outlined some of our approaches to heterogeneous audio scenes consisting of different types of audio input sources and multichannel displays. We described some SATIE functionalities with regard to heterogeneous spatial audio scenes. We have also described our approach to Ambisonic Impulse Response (AIR) in VARAYS in order to enable ambisonic acoustic simulation. VARAYS is still at very early stages of development, it needs proper support for material based diffraction and diffusion. Figure 4(b) shows the general workflow, where AIR is applied to a mono sound source and is spatialized using the usual SATIE pipeline. There is still some work left to do in order to fully integrate vaRays into SATIE pipeline (both IR and AIR). One of the areas to explore is in the interpolation of IR instances in order to compensate for real-time changes in the listener and the sound source location. This process can be mixed with types of rendering which provides sufficient creative liberty to the user. There is also some work left to provide IR and AIR to SATIE as files I/O are not the most optimal. We will be looking into sending OSC blobs. Another path would be sharing buffers between SATIE and vaRays using out shared memory library SHMDATA [5]. Another desired functionality is rendering VBAP spatialization into b-format signals.

5. ACKNOWLEDGEMENTS

This project would not be possible without the support of the Ministère de l'Économie et de l'Innovation du Québec. We also wish to thank Julien Wantz who spent time with us at the [SAT] Metalab, starting implementation of VARAYS, Mylène Pardoen for sharing the Paris 3D model used in Figure 5(b) and Florian Grond for his SuperCollider Higher Order Ambisonics library, SC-HOA.

6. REFERENCES

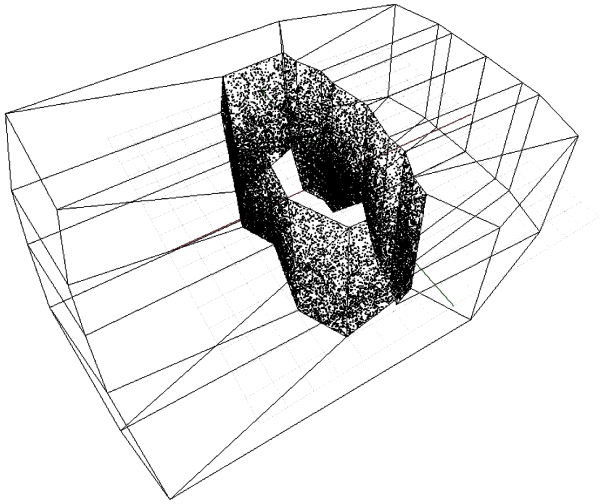
[1] N. Barrett, "A musical journey towards permanent high-density loudspeaker arrays," *Computer Music Journal*, vol. 40, no. 4, pp. 35–46, Dec 2016.

[2] Zack Settel, Nicolas Bouillot, and Michal Seta, "Volumetric approach to sound design and composition using SATIE:

[4] `https://gitlab.com/sat-metalab/varays`

[5] `https://gitlab.com/sat-metalab/shmdata`

(a) The cube floating at the end of the hallway in the 18th century Paris model represents a sound object. The image is from a prototype developed by Metalab using EIS for visual rendering and navigation, VARAYS for real-time impulse response processing and SATIE for audio spatialisation.



(b) Visualisation showing the impacts (black dots) of sound sources (not shown) on the walls of a 3D volume for a listener (not shown) placed inside the same volume. For this example, 2000 rays were thrown with a maximum of 3 reflections. A point cloud representing the impacts was saved by our software VARAYS and rendered in BLENDER.

Figure 5: Example of directional reverberation approach with our prototype based on conjoint use of SATIE and vaRays. We used the Bretez 3D of the 18th century.

a high-density 3D audio scene rendering environment for large multi-channel loudspeaker configurations," in *15th Biennial Symposium on Arts and Technology*, Ammerman Center for Arts and Technology at Connecticut College, New London, feb 2016, 8 pages.

[3] Nicolas Bouillot and Michał Seta, "A scalable haptic floor dedicated to large immersive spaces," in *Proceedings of The Linux Audio Conference*, March 2019.

[4] Jui-Hsien Wang, Ante Qu, Timothy R. Langlois, and Doug L. James, "Toward wave-based sound synthesis for computer animation," *ACM Trans. Graph.*, vol. 37, no. 4, pp. 109:1–109:16, July 2018.

[5] David Poirier-Quinot, Brian FG Katz, and Markus Noisternig, "EVERTIMS: open source framework for real-time auralization in architectural acoustics and virtual reality," in *Proceedings of Digital Audio Effects (DAFx)*, New York, NY, USA, 2017.

[6] Samuli Laine, Samuel Siltanen, Tapio Lokki, and Lauri Savioja, "Accelerated beam tracing algorithm," *Applied Acoustics*, vol. vol. 70, no. no. 1, pp. pp. 172–181, 2009.

[7] Mendel Kleiner, Bengt-Inge Dalenbäck, and Peter Svensson, "Auralization-an overview," *J. Audio Eng. Soc*, vol. 41, no. 11, pp. 861–875, 1993.

[8] Axel Plinge, Sebatian J. Schlecht, Oliver Thiergart, Thomas Robotham, Olli Rummukainen, and Emanuël A. P. Habets, "Six-degrees-of-freedom binaural audio reproduction of first-order ambisonics with distance information," in *Audio Engineering Society Conference: 2018 AES International Conference on Audio for Virtual and Augmented Reality*, Aug 2018.

[9] Graham Wakefield and Wesley Smith, "COSM: a toolkit for composing immersive audio-visual worlds of agency and autonomy," in *Proceedings of the International Computer Music Conference 2011*, University of Huddersfield, UK, aug 2011.

[10] David Poirier-Quinot, Damien Touraine, and Brian F.G. Katz, "BlenderCAVE: A multimodal scene graph editor for virtual reality," in *Proceedings of the 19th International Conference on Auditory Display (ICAD2013)*, Lodz, Poland, jul 2013, Georgia Institute of Technology & International Community for Auditory Display.

[11] Rui Penha and Joao Pedro Oliveira, "Spatium, tools for sound spatialization," in *Proceedings of the Sound and Music Computing Conference*, Stockholm, Sweden, 2013.

[12] Chikashi Miyama and Götz Dipper, "Zirkonium 3.1-a toolkit for spatial composition and performance," in *Proceedings of the International Computer Music Conference*, 2016, vol. 313, p. 312.

[13] Natanael Olaiz, Pau Arumi, Toni Mateos, and David Garcia, "3D-audio with CLAM and blender's game engine," in *proceedings of The Linux Audio Conference*, Parma, Italy, 2009.

[14] Andres Perez-Lopez, "3Dj: a supercollider framework for real-time sound spatialization," in *Proceedings of the 21th International Conference on Auditory Display (ICAD–2015)*, Graz, Austria, jul 2015.

[15] Thibaut Carpentier, "Panoramix: 3D mixing and post-production workstation," in *42nd International Computer Music Conference (ICMC)*, Utrecht, Netherlands, Sept. 2016.

[16] Jan C. Schacher, Chikashi Miyama, and Trond Lossius, "The spatdif library - concepts and practical applications in audio software," in *ICMC*, 2014.

[17] James McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, , no. 26, pp. 61–68, 2002.

[18] Zack Settel, Peter Otto, Michal Seta, and Nicolas Bouillot, "Dual rendering of virtual audio scenes for far-field surround multi-channel and near-field binaural audio displays," in *16th Biennial Symposium on Arts and Technology*, Ammerman Center for Arts and Technology at Connecticut College, New London, February 2018, 5 pages.

[19] M. Wright, "Open sound control 1.0 specification," Published by the Center For New Music and Audio Technology (CNMAT), UC Berkeley, 2002.

[20] Tim Blechmann, "Supernova, a scalabale parallel audio synthesis server for SuperCollider," in *Proceedings of the International Computer Music Conference 2011*, University of Huddersfield, UK, aug 2011.

[21] Ben Shirley, Rob Oldfield, Frank Melchior, and Johann-Markus Batke, "Platform independent audio," *Media Production, Delivery and Interaction for Platform Independent Systems: Format-Agnostic Media*, pp. 130–165, 2013.

[22] Nicolas Bouillot, Zack Settel, and Michal Seta, "SATIE: a live and scalable 3d audio scene rendering environment for large multi-channel loudspeaker configurations," in *New Interfaces for Musical Expression (NIME'17)*, Copenhagen, Denmark, 2017.

A JACK-BASED APPLICATION FOR SPECTRO-SPATIAL ADDITIVE SYNTHESIS

Henrik von Coler

Audio Communication Group
TU Berlin
voncoler@tu-berlin.de

ABSTRACT

This paper presents a real-time additive sound synthesis application with individual outputs for each partial and noise component. The synthesizer is programmed in C++, relying on the Jack API for audio connectivity with an OSC interface for control input. These features allow the individual spatialization of the partials and noise, referred to as spectro-spatial synthesis, in connection with an OSC capable spatial rendering software. Additive synthesis is performed in the time domain, using previously extracted partial trajectories from instrument recordings. Noise is synthesized using bark band energy trajectories. The sinusoidal data set for the synthesis is generated from a custom violin sample library in advance. Spatialization is realized using established rendering software implementations on a dedicated server. Pure Data is used for processing control streams from an expressive musical interface and distributing it to synthesizer and renderer.

1. INTRODUCTION

1.1. Sinusoidal Modeling

Additive synthesis is among the oldest digital sound creation methods and has been the foundation of early experiments by Max Mathews at *Bell Labs*. It allows the generation of sounds rich in timbre, by superimposing single sinusoidal components, referred to as partials, either in the time- or frequency domain. Based on the Fourier Principle, any quasi-periodic signal $y(t)$ can be expressed as a sum of N_{part} sinusoids with varying amplitudes $a_n(t)$ and frequencies $\omega_n(t)$ and an individual phase offset φ_n:

$$y(t) = \sum_{n=1}^{N_{part}} a_n(t)\ sin(\omega_n(t)\ t + \varphi_n) \tag{1}$$

In harmonic cases, which applies to the majority of musical instrument sounds, the partial frequencies can be approximated as integer multiples of f_0:

$$y(t) = \sum_{n=1}^{N_{part}} a_n(t)\ sin(2\ \pi\ n\ f_0(t)\ t + \varphi_n) \tag{2}$$

Although relative phase fluctuations are important for the perception [1], the original phase can be ignored in many cases, which is of benefit for manipulations of the modeled sound:

$$y(t) = \sum_{n=1}^{N_{part}} a_n(t)\ sin(2\ \pi\ n\ f_0(t)\ t) \tag{3}$$

Based on this theory, an algorithm for speech synthesis has been proposed by McAulay *et al.* [2]. For musical sound synthesis the algorithm has been added a noise component [3], resulting in the *sinusoids+noise* model. The signal is then modeled as the sum of the deterministic part x_{det} and the stochastic part x_{stoch}, also referred to as residual:

$$x = x_{det} + x_{stoch} \tag{4}$$

Modeling of residuals can for example be performed by approximating the spectral envelope using linear predictive coding [3] or a filter bank based on Bark frequencies [4]. The phase of the stochastic signal is random, in theory, and thus needs not be modeled. However, residuals usually are not completely random since they still contain information from the removed harmonic content.

In order to fully model the sounds of arbitrary musical instruments, a transient component x_{trans} is included [4] in the full signal model. This component captures plucking sounds and other percussive elements:

$$x = x_{det} + x_{stoch} + x_{trans} \tag{5}$$

Since the work presented in this paper focuses on the violin in legato techniques, the transient component can be neglected without impairing the perceived quality of a re-synthesis.

1.2. Spectral Spatialization

In electronic and electroacoustic music, the term *spectral spatialization* refers to the individual treatment of a sound's frequency components for a distribution on sound reproduction systems [5]. Timbral sound qualities can thusly be linked to the spatial image of the sound, even for pre-existing or fixed sound material. In the case of *spectro-spatial synthesis*, this process is integrated on the synthesis level,for example in additive approaches. This is not yet a common feature in available synthesizers, but several research projects have been investigating the possibilities of such approaches with applications in musical sound processing, sound design, virtual acoustics and psychoacoustics.

Topper *et al.* [6] apply additive synthesis of basic waveforms (square wave, sawtooth), physical modeling and sub-band decomposition in a multichannel panning system with real time, prerecorded and graphic control. Their system is implemented in MAX/MSP and RTcmix, running on both Mac and PC/Linux hardware with a total of 8 audio channels.

Verron *et al.* [7] use the *sinusoids + noise* model for spectral spatialization of environmental sounds. Each component can be synthesized with individual position in space on Ambisonics and Binaural systems. Deterministic and stochastic components are composed and added together in the frequency domain and subsequently spatially encoded with a filterbank. Control over the synthesis process is depending on the nature of the environmental sounds [8].

In the context of electroacoustic music, James [9] expands Dennis Smalley's concept of *spectromorphology* to the idea of *spatiomorphology*. *Timbre Spatialization* is achieved using terrain surfaces

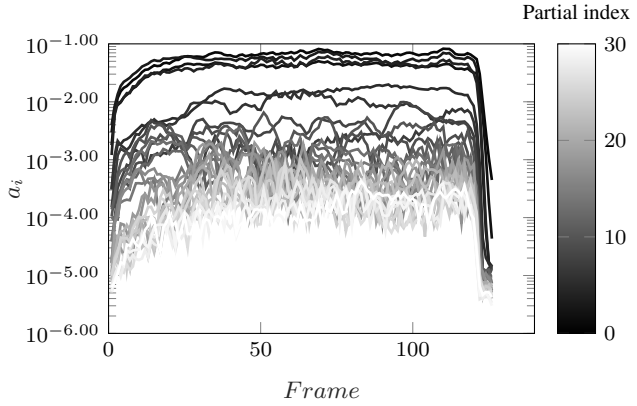


Figure 1: Partial amplitude trajectories of a violin sound

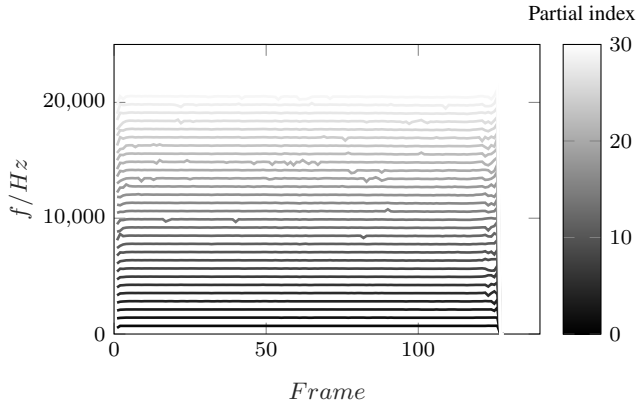


Figure 2: Partial frequency trajectories of a violin sound

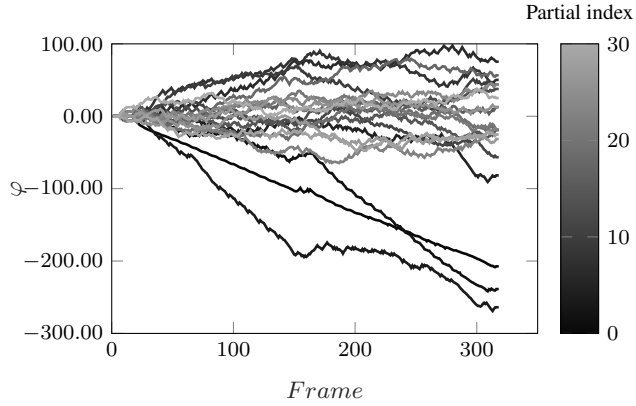


Figure 3: Unwrapped partial phases of a violin sound

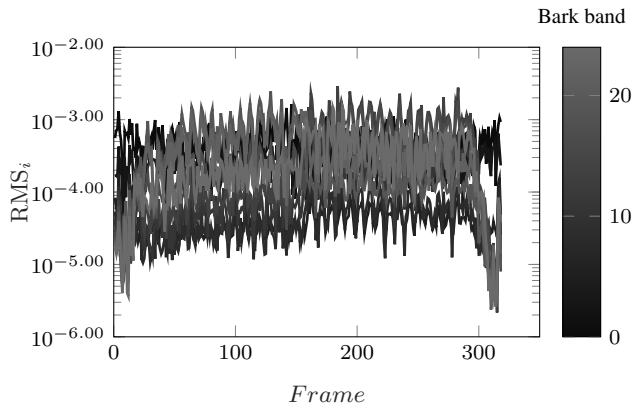


Figure 4: Bark band energy trajectories of a violin sound

and by mapping these to *spacio-spectral* distributions. Max-MSP is used for computing the contribution of spectral content to individual speakers with Distance-based amplitude panning (DBAP) and Ambisonic Equivalent panning (AEP) methods.

Spectral spatialization can also be used to synthesize dynamic directivity patterns of musical instruments in virtual acoustic environments. Since the directivity in combination with movement has a significant influence on an instrument's sound, this can increase the plausibility. Warusfel *et al.* [10] use a tower with three cubes, each containing multiple speakers, to spatialize frequency bands of an input signal for the simulation of radiation patterns.

1.3. The Presented Application

The presented application incorporates different synthesis modes, of which only the so called *deterministic mode* will be subject of this paper. In this basic mode, precalculated parameter trajectories, as presented in Sec. 2, are used for a manipulable resynthesis of the original instrument sounds.

The software architecture is designed to allow the use of additive synthesis, respectively of sinusoidal modeling, on sound field synthesis systems or other reproduction setups. This is achieved by providing individual outputs for all partials and noise bands in an application implemented as a JACK client, described in Sec. 3. Using JACK allows the connection of all individual synthesizer output channels to a JACK-capable renderer, such as the SoundScape Renderer (SSR) [11], Panoramix [12] or the HOA- Library [13]. Making each partial a single virtual sound source in combination with these rendering softwares, the spatial distribution of the synthesis can be modulated in real-time. Pure Data [14] is used to receive control data from gestural interfaces or to play back predefined trajectories for generating control streams for both the synthesizer and the spatialization renderer. A direct linkage between timbre and spatialization is thus created, which is considered essential for a meaningful *spectro-spatial synthesis*.

2. ANALYSIS

The *TU-Note Violin Sample Library* [15], [16], is used as audio content for generating the sinusoidal model. Designed in the style of classic sample libraries, this data set contains single sounds of a violin in different pitches and intensities, recorded at an audio sampling rate of 96 kHz with 24 Bit resolution.

Analysis and modeling is performed beforehand in Matlab, using monophonic pitch tracking and subsequent extraction of the partial trajectories by peak picking in the spectrogram. YIN [17] and SWIPE [18] are used as monophonic pitch tracking algorithms. Based on the f0-trajectories, partial tracking is performed with STFT, applying a hop-size of 256 samples (2.7ms) and a window size of

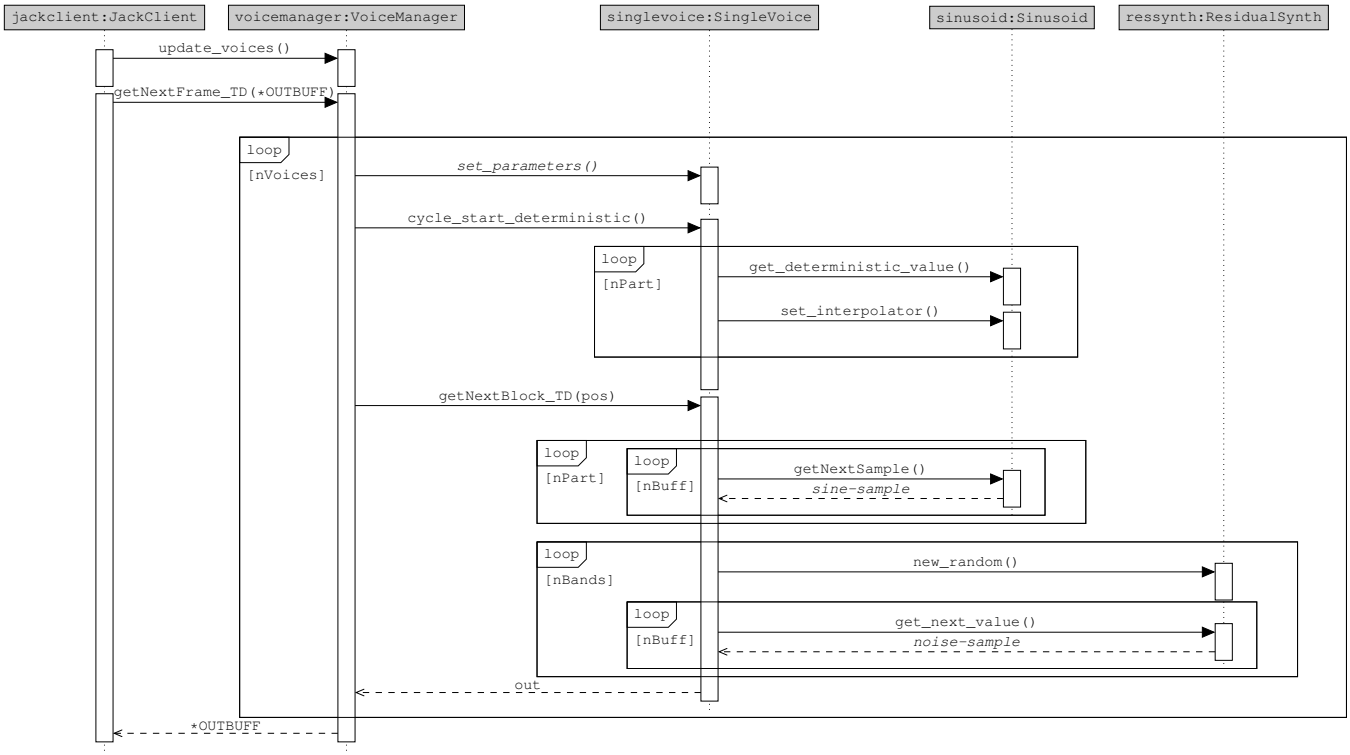


Figure 5: Sequence diagram for the jack callback function

4096 samples, zero-padded to 8192 samples. Quadratic interpolation (QIFFT), as presented by Smith *et al.* [19], is applied for peak parameter estimation of up to 80 partials. Due to the sampling frequency, the full number of partials is only analyzable up to the note D_5 (576.65 Hz)

By subtracting the deterministic part from the complete sound in the time domain, the residual signal is obtained. The residual is then filtered using a Bark scale filterbank with second order Chebyshev bandpasses and the temporal energy trajectories are calculated for the resulting 24 band-limited signals. At this point, a large amount of information is removed from the residual signal. Due to the shortcomings of the time domain subtraction method, the residual still contains information from the deterministic component. By averaging the energy over the Bark bands, this relation is eliminated.

Results of the synthesis stage are trajectories of the partial amplitudes, as shown in Figure 1, the trajectories of partial frequencies and phases, as shown in Figure 2, respectively Figure 3 as well as the trajectories of the Bark-band energies, illustrated in Figure 4. The resulting data is exported to individual YAML files for each sound, which can be read by the synthesis system.

3. SYNTHESIS SYSTEM

3.1. Libraries

The synthesis application is designed as a standalone Linux command line software. The main functionality of the synthesis system relies on the *JACK*[1] API for audio connectivity and the *liblo*[2], respectively the *liblo C++ wrapper* for receiving control signals. *libyaml-cpp*[3] is used for reading the data of the modeled sounds and the relevant configuration files. *libsndfile*[4] for reading the original sound files, as well as the *libfftw*[5] are included but not relevant for the aspects presented in this paper. Frequency domain synthesis and sample playback are partially implemented but not used at this point.

3.2. Algorithm

Both the sinusoidal and the noise component are synthesized in the time domain, using a non-overlapping method. For the sinusoidal component, the builtin `sin()` function of the `cmath` library and a custom lookup table can be selected. The choice does not affect the overall performance, significantly. The filter bank for the noise synthesis consists of 24 second order Chebyshev bandpass filters with fixed coefficients, calculated before runtime. The amplitude of each frequency band is driven by the previously analyzed energy trajectories.

During synthesis, the algorithm reads a new set of support points from the model data for each audio buffer and increments the position within the played note. Figure 5 shows a sequence diagram for the deterministic synthesis algorithm, starting at the JACK callback function, which is executed for each buffer of the JACK audio server. Since the synth is designed to enable polyphonic play, the voice manager object handles incoming OSC messages in the function `update_voices()` to activate or deactivate single voices.

[1] `http://jackaudio.org/`
[2] `https://github.com/radarsat1/liblo`
[3] `https://github.com/jbeder/yaml-cpp/`
[4] `http://www.mega-nerd.com/libsndfile/`
[5] `http://www.fftw.org/`

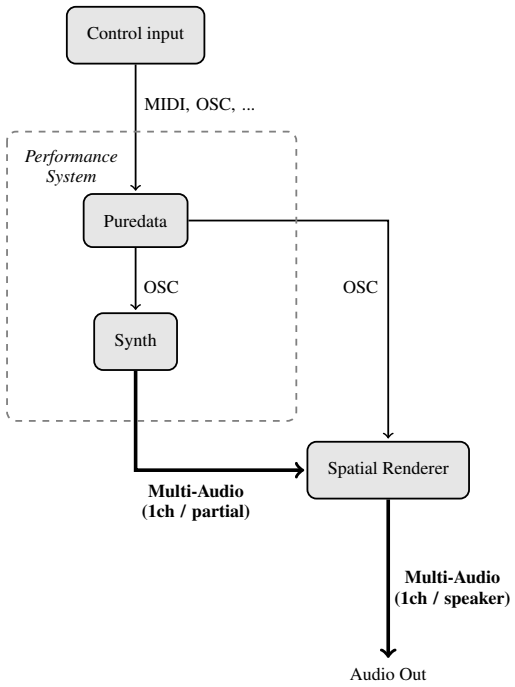


Figure 6: Combination of synthesizer and renderer on separate machines using Pure Data for synth configuration and parameter parsing

For the synthesis of mostly monophonic, excitation continuous instruments like the violin, the polyphony merely handles the overlapping of released notes. Subsequently, the voice manager loops over all active voices in the function `getNextFrame_TD()`, first setting the new control parameters for each voice.

In `cycle_start_deterministic()`, support points for all partial's parameters are picked at the relevant voice's playback position. These support points are then linearly interpolated over the buffer length in `set_interpolator()`.

Finally, in `getNextBlock_TD()`, each single voice generates the output for all sinusoids and all noise bands in two separate vectorizable loops, adding both to the output buffer.

3.3. Runtime Environment and Periphery

The runtime system for the synthesis is starting a JACK server with 48 kHz sampling rate, a buffer size of 128 samples and 2 periods per buffer. This results in $5.3\,\mathrm{ms}$ latency for the audio playback, which is within the limits for this synthesis approach. On an Intel(R) Core(TM) i7-5500U CPU @ 2.40GHz with disabled speed-stepping and a Fireface UFX, the JACK server is showing an average load of approximately $20\,\%$.

The interaction of the involved software components is visualized in Figure 6. For reasons of performance and increased flexibility in the studio, two separate machines are used for synthesis and spatialization. Connectivity between the systems is realized with MADI or DANTE, using individual channels for the 80 partials and 24 noise bands.

3.4. Control

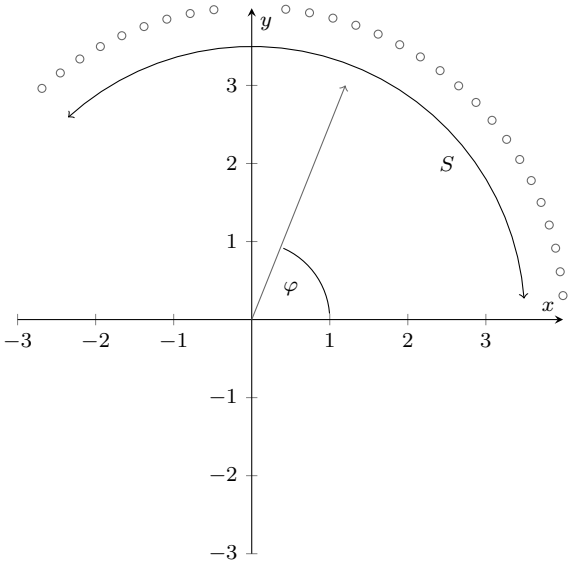


Figure 7: Spatialization scene in a 2D setup with 30 partials and their positions

The control data for the partial positions in the rendering software is not generated in the synthesis system at this point and is managed, externally. This offers more flexibility for testing different mappings at this stage of development. A Pure Data patch is used to receive incoming control messages, either from OSC or MIDI, and distribute them to the synthesizer and the spatialization software. For live performance, the patch receives continuous control streams for *pitch* and *intensity* from an improved version of the interface presented by von Coler *et al.* [20] and visualizes the sensor data. Pitch and intensity are forwarded to the synth, directly. Additionally, data from several Force Sensitive Resistors (FSR) and a 9 degrees of freedom IMU, which can be used for controlling the spatialization, is sent to the patch.

Figure 7 shows an example for a simple spatialization mapping on a 2D system. The absolute orientation of the IMU is used to control the general direction φ of the partial flock. A second parameter S, derived from the intensity and additional sensor data, controls the spread of the partials around this angle, depending on the partial index.

4. CONCLUSION

After significantly improving the performance of the synthesis system, the application can now be used with the full 80 partials and 24 Bark bands as individual outputs. Recent tests in combination with different spatial rendering softwares and different loudspeaker setups show promising results. However, the dynamic spatialization of such number of virtual sound sources and the resulting traffic of OSC messages is demanding for the runtime system. Using separate machines for synthesis and rendering reduces the individual load. The number of rendering inputs can also be reduced without limiting the perceived quality of the spatialization. Multiple partials may share one virtual sound source.

Next steps are now possible, which include the empirical investigation of mappings from controller sensors to both the spectral and spatial sound properties. This includes user experiments to evaluate different mapping and control paradigms, as well as perceptual measurements of the synthesis results.

5. ACKNOWLEDGMENTS

Thanks to Benjamin Wiemann for contributions to the project in it's early stage and to Robin Gareus for the help in restructuring the code and hencewith improving the performance.

6. REFERENCES

[1] T. H. Andersen and K. Jensen, "Importance and Representation of Phase in the Sinusoidal Model", *J. Audio Eng. Soc*, vol. 52, no. 11, pp. 1157–1169, 2004.

[2] R. McAulay and T. Quatieri, "Speech analysis/Synthesis based on a sinusoidal representation", *Acoustics, Speech and Signal Processing, IEEE Transactions on*, vol. 34, no. 4, pp. 744–754, 1986.

[3] X. Serra and J. Smith, "Spectral Modeling Synthesis: A Sound Analysis/Synthesis System Based on a Deterministic Plus Stochastic Decomposition ", *Computer Music Journal*, vol. 14, no. 4, pp. 12–14, 1990.

[4] S. N. Levine and J. O. Smith, "A Sines+Transients+Noise Audio Representation for Data Compression and Time/Pitch Scale Modi cations", in *Proceedings of the 105th Audio Engineering Society Convention*, San Francisco, CA, 1998.

[5] D. Kim-Boyle, "Spectral spatialization - an Overview", in *Proceedings of the International Computer Music Conference*, Belfast, UK, 2008.

[6] D. Topper, M. Burtner, and S. Serafin, "Spatio-operational spectral (sos) synthesis.", in *Proceedings of the International Computer Music Conference (ICMC)*, Singapore, 2003.

[7] C. Verron, M. Aramaki, R. Kronland-Martinet, and G. Pallone, "Spatialized additive synthesis of environmental sounds", in *Audio Engineering Society Convention 125*, Audio Engineering Society, 2008.

[8] C. Verron, G. Pallone, M. Aramaki, and R. Kronland-Martinet, "Controlling a spatialized environmental sound synthesizer", in *2009 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, IEEE, 2009, pp. 321–324.

[9] S. James, "Spectromorphology and spatiomorphology of sound shapes: Audio-rate aep and dbap panning of spectra", in *Proceedings of the International Computer Music Conference 2015*, 2015.

[10] O. Warusfel and N. Misdariis, "Directivity synthesis with a 3d array of loudspeakers: Application for stage performance", in *Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-01), Limerick, Ireland*, 2001, pp. 1–5.

[11] J. Ahrens, M. Geier, and S. Spors, "The SoundScape Renderer: A unified spatial audio reproduction framework for arbitrary rendering methods", in *Audio Engineering Society Convention 124*, Audio Engineering Society, 2008.

[12] T. Carpentier, "Panoramix: 3d mixing and post-production workstation", in *Proceedings of the International Computer Music Conference (ICMC)*, 2016.

[13] A. Sèdes, P. Guillot, and E. Paris, "The HOA library, review and prospects", in *International Computer Music Conference\ Sound and Music Computing*, 2014, pp. 855–860.

[14] M. S. Puckette, "Pure Data", in *Proceedings of the International Computer Music Conference (ICMC)*, Thessaloniki, Greece, 1997.

[15] H. von Coler, J. Margraf, and P. Schuladen, *TU-Note Violin Sample Library*, TU-Berlin, 2018. DOI: 10.14279/depositonce-6747.

[16] H. von Coler, "TU-Note Violin Sample Library – A Database of Violin Sounds with Segmentation Ground Truth", in *Proceedings of the 21st Int. Conference on Digital Audio Effects (DAFx-18)*, Aveiro, Portugal, 2018.

[17] A. de Cheveigné and H. Kawahara, "YIN, a Fundamental Frequency Estimator for Speech and Music", *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.

[18] A. Camacho, "Swipe: A Sawtooth Waveform Inspired Pitch Estimator for Speech and Music", PhD thesis, Gainesville, FL, USA, 2007.

[19] J. O. Smith and X. Serra, "PARSHL: An Analysis/Synthesis Program for Non-Harmonic Sounds Based on a Sinusoidal Representation", Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, Tech. Rep., 2005.

[20] H. von Coler, G. Treindl, H. Egermann, and S. Weinzierl, "Development and Evaluation of an Interface with Four-Finger Pitch Selection", in *Audio Engineering Society Convention 142*, Audio Engineering Society, 2017.

GPU-ACCELERATED MODAL PROCESSORS AND DIGITAL WAVEGUIDES

Travis Skare
CCRMA
Stanford University, USA
travissk@ccrma.stanford.edu

Jonathan Abel
CCRMA
Stanford University, USA
subsection abel@ccrma.stanford.edu

ABSTRACT

Digital waveguides and highly-resonant filters are potential fundamental building blocks of physical models and modal processors. What might a sound designer accomplish with a massive collection of these objects?

When the building blocks are independent, the overall system becomes highly parallel. We investigate the feasibility of using a modern Graphics Processing Unit (GPU) to run collections of waveguides and filters, toward constructing realtime collections of room simulations or instruments made up of many banded waveguides.

These two subproblems offer different challenges and bottlenecks in GPU acceleration: one is compute-bound while the other has memory optimization challenges.

We find that modern GPUs can run these algorithms at audio rates in a straightforward fashion–that is, sample-by-sample without needing to implement transforms that allow computation of subsequent time samples concurrently. While a fully-realized instrument or effect based on these building blocks requires additional processing and will have more data dependencies that reduce parallelism, we find that consumer-GPU-accelerated audio enables a scale of realtime models which would be intractable on contemporary consumer-CPUs.

1. INTRODUCTION

Potential applications for a large number of modal filters or digital waveguides include:

- A large collection of coupled acoustic spaces, for example an opera house with listening booths that may be seen as resonators, or the interior architecture of ancient Chavín[1].
- A virtual orchestra where we have many players, each using an instrument made up of several digital waveguides.
- A virtual reality simulation where a server may track room- and position-dependent modal reverberators for a number of participants on low-power client devices.
- A drum set made up of a couple dozen individual instruments, each using many modal filters.

While the first three ideas are hypotheticals enabled by having access to massively parallel filtering/waveguide systems, the fourth exists as a real-world proof of concept to synthesize a dozen modal cymbal models at realtime rates using a GPU. Active work is toward adding realtime controls for a performer.

1.1. Building Blocks: Modal Synthesis and Digital Waveguides

Modal synthesis involves determining the natural resonant modes of a vibrating object, and using the appropriate frequencies, amplitudes, and decay rates to build a system that simulates the original sound.

A filter bank of high-Q filters is often used for such sound synthesis, and is also the backbone of modal reverberators[2]. The more modes we can compute at realtime audio rates, the higher the fidelity of the sound, and the more sources or rooms we may model.

Digital waveguides[3] are efficient for simulation of traveling waves, and with scattering junctions and nonlinearities added, a wide range of physically-accurate bowed strings, brass, etc. may be simulated with robust realtime performance controls. Here, we are interested in working toward many virtual performers each playing an independent instrument (orchestra), or one performer given control over simultaneous but mostly independent "clusters" of waveguide-powered instruments, such as a virtual drum set with a large number of pieces.

Digital Waveguides may be implemented efficiently in the 1-dimensional case via a bidirectional delay line representing two traveling waves, plus filters to account for dispersion loss. These are the basic building blocks we seek to accelerate, noting that for more complex physical models we will add scattering junctions, additional filtering, and nonlinear elements incurring additional computation cost. In some cases, such as piano string modeling[4], some terms may be commuted, or combined with an impulse response, to add complexity to the overall model without scaling the overall steady-state computational cost.

1.2. GPU Acceleration for Audio Algorithms

For years, graphics processing units (GPUs) have supported both high-level realtime graphics APIs as well as lower-level, general-purpose computational APIs. GPU acceleration of audio synthesis and audio effect algorithms has been shown to yield substantial speedups on certain algorithms. GPUs advance in performance each generation in terms of parallel core count and base core speed, so we expect some previously intractable problems to become tractable over time.

Among papers in the literature:

Savioja et. al.[5] give an overview of potential audio tasks that may be accelerated via GPGPU programming at audio rate and reasonable buffer sizes for realtime performance. Sinusoid-based additive synthesis obtained 250x+ speedup over CPU implementations, FFTs running on a GPU were able to be eight times as long as those running on a CPU-based implementations, and FIR filters were able to be 130 times as long as their CPU counterparts. In [5] and [6], the authors showed it was possible to synthesize 1.9 million sinusoids in realtime, a 1300x speedup over a serial lookup table computation on one CPU. This was on a GPU that is six generations behind ours and three major GeForce architecture revisions behind our card[1]. And we note that our graphics card is itself now a generation and major architecture advance behind the times. This work results in a

[1] Fermi (GTX 480, 2010) → Kepler → Maxwell → Pascal (GTX 1080Ti, 2017); RTX cards released in 2018 use the Turing architecture.

realtime sound canvas with more "paint" than previously available (how might an artist use 1.9 million partials in additive synthesis?). The million sinusoids example also demonstrates that maximum performance requires tuning and knowledge of the specific underlying hardware.

Trebien et. al.[7] use modal synthesis to produce realistic sounds for realtime collisions between objects of different materials. Noting that IIR filters do not traditionally perform well on GPUs, due to dependence on prior state not mapping well to the parallel nature of GPUs, they introduce a transform to change this into a linear convolution operator and to unlock time-axis parallelism.

Belloch et. al. [8] accelerate IIR filters on the GPU directly by using the Parallel IIR representation. They achieve 1256 concurrent 256th-order IIR filters at audio rates and sub-millisecond latency at a 44.1kHz sampling rate.

Subsequently, Belloch covers GPU-accelerated massively parallel filtering in [9], and Belloch et. al.[10] leverage GPU acceleration to implement Wave Field synthesis on a 96-speaker array, with nearly ten thousand fractional-delay room filters with thousands of taps. The maximum number of simulated sound sources is computed for different realtime buffer sizes and space partitions; with a 256-sample buffer (5.8ms at 44.1kHz), between 18 and 198 real-time sources could be placed in the field.

Bilbao and Webb[11] present a GPU-accelerated model of timpani, synthesizing sound at 44.1kHz in a 3D computational space within and outside the drum. The GPU approach uses a matrix-free implementation to obtain a 30x+ speedup over a MATLAB CPU-, sparse-matrix-based prototype, and a greater-than-7.5x speedup over single-threaded C code baseline. The largest (and most computationally-expensive) drum update equation is optimized to 2.04 milliseconds per sample, where the bottleneck is a linear system update for the drum membrane.

Our area of study utilizes recursive filters and unfortunately optimizations of the million-sinusoinds and Parallell IIR filter works do not apply directly; we would like to be able to adjust parameters arbitrarily in realtime and at sample rate, which would require rerunning transformation code too often.

Still, our filter bank is expected to be highly parallel in terms of independence *between* the filters. We may have coupling between modes, but so long as it's limited, we can implement this in a way that is compatible with GPU programming ideas. We also do not need to implement arbitrary IIR filters as in Belloch et. al., but will be able to use special-purpose damped oscillation filters that only require a first-order complex update equation (see Section 2).

If GPUs have advanced enough in terms of increased clock rate, increased floating-point resources, and lower memory latency in the last few generations, we aim to compute filter and physical model updates sample-by-sample in realtime.

1.3. GPU Programming

Next, we present a brief overview of GPU programming, and note advantages and challenges versus programming for a general-purpose processor.

Various toolkits exist to develop GPU programs: two of the biggest are NVIDIA's CUDA for use with their graphics cards, and APIs implementing OpenCL, a more general heterogeneous computational framework. For the following investigation we use CUDA. If readers have any modern NVIDIA card, they may download the software developer kit at developer.nvidia.com.

When starting to port an algorithm to the GPU, we must consider if it has parallelism to leverage. NVIDIA coined the term "single instruction, multiple thread" (SIMT) as a variation on the "single instruction, multiple data" (SIMD) of vector processors and modern mainstream processors. If our work is a series of several different and dependent computations, we may not be able to achieve a speedup. If we can structure it as applying identical operations to many points, it is a good candidate for acceleration.

The core work unit in CUDA is a group of 32 threads, called a *warp*. Each thread in a warp may have its own values for local variables, but all threads in a warp will always run the same instruction simultaneously.

A warp is executed on a *Streaming Multiprocessor* (SM). Different graphics cards have different numbers of SMs; a low-power embedded device may have two while our graphics card used for the trials below has 28.

A trivial example task would be to take N integer inputs and double them.

There are two main steps involved in this task. First, we write a *kernel*, the code that will run on the GPU. This will accept an array of inputs; each thread will index into the array, find the element it is to double, multiply it by 2, and store it in an output array. Second, we write host (CPU) code that calls a CUDA function to send an input array to the GPU, execute the kernel, wait for the kernel to complete, and finally copy the output values back to the CPU, for example so we can save them to disk.

If we have 32 inputs to double, CUDA will execute our kernel code on one warp of 32 threads. All 32 threads in that warp execute in lockstep and run the same instructions, but obtain a different value of the array to double and a different output location to store the result. If we have only 15 inputs to double, this is not a problem. We will still run on one warp, and the 17 threads without any work to do effectively get a break (they technically are issued instructions but do not write to memory or compete for resources). If we have 33 inputs to double, we outgrow one warp. Threads will be grouped in one warp of 32 threads and a second warp of one solitary thread. More than one warp can run at a time, so it is very likely we will run in the same time as it took to run the 32 and 15 input cases.

A CUDA-enabled graphics card has some number of Streaming Multiprocessors (SMs). The product specifications for individual graphics card models and a capabilities table such as provided in the CUDA Programming Guide lets authors know how many threads may be in flight per SM, and how many may actually get run each cycle.

For example, our graphics card may have up to 64 warps assigned to each SM ($\rightarrow$ 64warps $*$ 32threads/warp $=$ 2048threads), though only 4 warps (128 threads) may be scheduled on each single clock cycle.

The programming guide lists other bottlenecks and numbers to consider. One piece of information very relevant to us is the number of simultaneous arithmetic operations available.

The graphics card we use is a consumer card meant for gaming. Some other cards (the NVIDIA TITAN for example) are targeted more for enterprise and scientific computing uses, albeit at a significantly higher price point.

We note the issue rate of floating-point operations from the programming guide:

This means that if we require 64-bit precision, our consumer card is more likely to be bottlenecked by this figure than the enterprise card in the lineup.

On the other hand, we note that our card has higher per-clock

Table 1: *Throughput of FP instructions (Results per Clock per SM)*

	Consumer	Enterprise
16-bit mult+add	2	128
32-bit mult+add	128	64
64-bit mult+add	4	32

32-bit floating-point throughput. Assuming that is sufficient precision for a problem, this means our card may execute 128 32-bit multiply-adds per SM $*$ 28 SMs $= 3584$ multiply-adds per clock. For reference, the card's core clock runs at 1.3-1.5GHz.

Thus far we've considered parallelism and availability of arithmetic units; we must also keep the memory hierarchy in mind.

Each SM has some number of registers. These are very fast. The compiler will attempt to use them for thread-local variables. On our card, there are 64,000 32-bit registers per SM.

Each *thread block* (user-defined organizational unit of threads, comprised of one or more warps) has some fast "shared" memory. This is 64KB per SM for our setup. We'll return to this later as an optimization.

All threads may access a card-wide pool of read-only constant memory.

All threads may also access a pool of global memory–11GB on our card. However, this is described as having roughly 100x the latency of shared memory or registers.

If a thread's local data will not fit in registers, the compiler may reduce parallelism or spill to "local memory." This technically lives in the slow global device memory pool, but is backed by a cache.

1.4. Development Approach

We take an iterative development approach, getting a basic algorithm working and then proceeding to tune it in stages. The CUDA toolkit contains IDE plugins and debugging tools, making it straightforward to analyze bottlenecks as we encounter them. The compiler will also be helping us along the way.

To set expectations, we know there will be overhead involved in transferring data between CPU and GPU, overheads in starting and stopping our kernel, and overhead introduced by the host operating system. We try to mitigate some of these, but some are unavoidable.

It is also important to note the significant effort that would be involved in moving from this proof of concept to a commercial DAW plugin. A hypothetical DAW is competing for CPU resources, will be using the GPU to render its GUI (our kernels can run alongside that with no issue, but there's still potential resource competition), and will force our choice of buffer size and latency.

1.5. Test Setup

The test setup consists of:

- GPU: An NVIDIA GeForce GTX 1080Ti, which is a consumer-grade graphics card, though a relatively high-level one.
- CPU: An Intel i5 3570K running at stock speed. We note this CPU is six generations old and a mid-level chip even in its generation, and newer CPUs may include newer vector instructions including AVX-512. However it is unlikely to bottleneck us, as it is used primarily for memory transfer and GPU kernel launches.
- RAM: CPU has 16GB, GPU has 11GB; neither will bottleneck us in these synthetic benchmarks.
- Storage: consumer SATA SSDs that will not be a bottleneck, especially since our tests should reside completely in RAM.
- OS and software: Development was cross-platform; kernels were written on Ubuntu Linux with Microsoft's open-source VSCode as a text editor and compiled using the CUDA Toolkit. During the memory optimization phase of the project, NVIDIA Nsight Visual Studio Edition on Windows was used for its "Next-Gen CUDA Debugger," though it is noted that the Linux/Mac Eclipse edition also contains an Eclipse-based profiler.
- Programs were compiled as 64-bit in case we use more than 4GB of RAM, possible with high buffer sizes and high numbers of parallel waveguides.

We discuss development of two algorithms: high-Q filters suitable for use in modal processors, and a simplified form of digital waveguides, running independently without scattering junctions and only a gain multiplier in the feedback loop. These two systems were developed simultaneously and do not depend on each other; we begin with the modal filter code since it is simpler, can essentially ignore the GPU memory hierarchy (everything besides output data fits in registers), and we estimate will be bottlenecked exclusively by the floating-point throughput of the graphics card, which makes it the easier of the two to optimize.

2. MASSIVE MODAL FILTER BANK

As described above, a modal filter bank used for synthesis, effects or reverberation consists of N resonant filters. We make the assumption that all the filters are uniform in construction and vary in parameters; a GPU can of course run multiple styles of filters in parallel, either through conditional execution or simultaneous kernel execution.

In practice, rapidly changing the coefficients on e.g. Direct-Form II filters may result in audible artifacts. In [12] Max Mathews and Julius Smith proposed a filter that is very-high-Q, numerically stable, and artifact-free, based on properties of complex multiplication.

This is suitable for modal synthesis and reverberators such as in [2]; the recursive update equation we need to implement is:

$$y_m(t) = \gamma_m x(t) + e^{(j\omega_m - \alpha_m)} y_m(t-1) \tag{1}$$

where:
$x()$ is an input or excitation signal.
ω_m is mode frequency m.
γ_m is a per-mode complex input amplitude gain.
α_m is a per-mode dampening factor.

This is straightforward to implement; the state we store for each mode is limited to the prior output $y_m(t-1)$, the parameters α_m, γ_m, and ω_m, even if only for intermediate computation. For simplicity we keep them all; noting that while complex values use two 32-bit registers each (four when using 64-bit precision), we likely have 255 registers per thread and have room to spare.

We benchmark three approaches:

When letting these resonating filters run as undamped oscillators, we are able to compute and reuse the complex exponential value, and only conditionally add the input term; with these simplifications we will require two floating-point multiplies per cycle. We create a benchmark to determine the number of such oscillators

we can run in parallel in realtime. We run two variations of this benchmark at different buffer sizes. A third benchmark simulates a pesrformance that modulates all the filters on every sample: we recompute the exponential term each time it is used, and look at the performance impact.

We move to benchmark those three approaches. In more detail:

Free-Run is the optimal case where the oscillators only need to update based on a complex multiplication of $y(t-1)$ with a static value of the complex exponential. A buffer size of 2,000 samples is likely larger than we'd want for realtime performance (45ms at 44.1kHz), but allows us to reduce kernel-switch overhead.

Small Buffer is identical to Free-Run, but with a buffer of 256 samples (5.8ms at 44.1kHz).

Continuous Modulation is our third approach, simulating gain parameters and frequencies changing continuously, requiring recomputing the complex exponential term with each sample update, in addition to performing the 2-multiply complex update of the filter state. This case uses the same 256-sample buffer as *Small Buffer*.

We measure the amount of time it takes to render ten seconds of 44.1kHz audio for N phasor filters in parallel. This means that benchmark runtimes over 10 seconds fall behind realtime performance, while values under 10 might be feasible. For each trial, the median of three runs was used; in practice we did not see large outliers in these tests.

Tabulated results are in Table 2; bold entries took less than ten seconds to compute and thus are candidates for realtime performance. In practice, we might want to avoid values under but close to ten seconds, due to system variance and unmeasured overhead of a DAW, OSC server, controller processing, etc. The same data is available as a plot in Figure 1, with a horizontal line representing realtime limits. In all graphs in this paper, lines between sample counts are present only to show trends, and we do not expect results for intermediate values of N to fall precisely on that line.

Table 2: *Time to run N filters for 10 seconds of Audio*

N Filters	Free-run	Small Buffer	Continuous Mod.
458752	**1.48**	**2.95**	**3.97**
917504	**2.63**	**4.18**	**5.54**
1835008	**4.85**	**7.17**	**8.49**
3670016	**9.23**	11.39	13.21

Some observations:

As these filters are completely independent, we achieve high utilization on the GPU and are only blocked on availability of floating-point units. All data is stored in registers and we avoid memory accesses, especially global memory accesses.

It is worth reiterating that this is benchmarking building blocks. We synthesize audio and copy it back to the host RAM, but additional logic is needed on the CPU to modulate parameters based on realtime user input or performance data and most likely to post-process the output with effects.

Using a smaller buffer incurs more cost, which can be 50% and even higher, percentage-wise, for low N. At very high N the effect is lower; we bottleneck on floating point unit availability in the large-buffer version, but have lower kernel launch overhead.

As a final observation on Table 2's data, the continuously-modulated version does not suffer as large a performance penalty as expected since it looks like we had some idle 32-bit floating-point units - they are not occupied every cycle. It also allows us to eliminate a conditional check since we always run that logic.

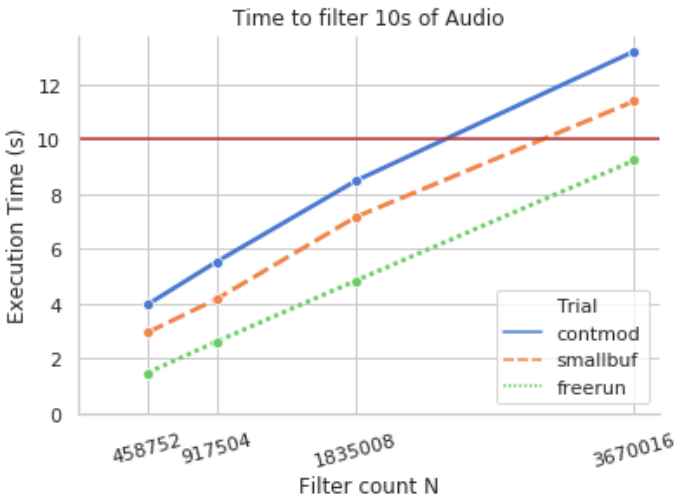


Figure 1: Time to run N filters for ten seconds of samples under different trials.

Moving forward, we benchmark the use of double-precision arithmetic. We made an alternate 64-bit kernel - basically swapping `cuDoubleComplex` in for the default `cuComplex`, which is by default `typedef`ed to be single-precision.

With a 256-sample buffer and continuously-changing parameters, and N=458,752 filters, it takes 19.49 seconds to render 10 seconds of audio. Our corresponding single-precision trial only took 3.97 seconds, so we note a 4.9x slowdown. As noted earlier, each SM on our GPU may only issue four 64-bit floating point multiply-adds versus 128 32-bit adds. As we did not achieve 100% utilization of the floating point units in prior benchmarks, we don't necessarily suffer a 32x (128/4) slowdown, but it is clear we are being bottlenecked by double-precision FPU availability with this configuration.

As we might expect, scaling down to 114,688 filters lowers resource contention enough to run within our realtime constraints (7.04 seconds to synthesize 10 seconds of audio). If we need the extra precision, that is likely still more than enough high-Q filters to enable some interesting instruments and effects, such as creating a virtual drum set with several thousand filters available to each instrument.

3. MASSIVE WAVEGUIDE "ORCHESTRA"

Next, we code up a kernel that performs the computations for a simple 1-D Digital Waveguide. This follows the description of the structure from Section 1.1; each thread owns a bidirectional delay line made up of continuous memory on its thread stack. This is used as a circular buffer, with an index value serving as a read/write head, and a multiplicative factor on feedback introduces dispersion loss. Note that for most waveguide-based physical models, additional code will be needed for scattering junctions, nonlinearities, etc., reducing our maximum throughput and complicating our kernel code, but parallel waveguides may be benchmarked as a starting point, to suggest an upper bound for performance.

As a baseline, we start with uniform waveguides of delay-length M=5000 samples in total[2], and process audio in 2000-sample chunks

[2]lengths in this benchmark represent the total length of the delay in the system; if building a waveguide from a bidirectional delay line, each delay

with each launch of the GPU kernel.

Because a buffer size of 2000 at 44.1kHz would be 45 milliseconds–longer than we'd like for interactive applications–we performed the same trial at 256 samples (5.8ms).

Then, because we expect to run out of thread registers and spill to expensive "local" memory (as above, really in the global pool) we run a second variant of waveguides of length 10 samples and a buffer of 2000 samples–arguably too much of a simplification, but this may be useful to establish a loose upper bound on performance.

In each trial, we compute the amount of time it takes to repeatedly run the kernel on the GPU and copy some data back out to the host.

The data copy is non-negligible overhead; assuming 1.8M waveguides and buffer of length 2000, we generate 13GB of audio data each kernel execution. As such, we first sum all the samples in each warp to reduce the overhead by a factor of 32 - still leaving us with a substantial amount of sound data to transfer across the PCI Express bus.

As in the high-Q filter benchmarks, we build N independent objects in parallel and measure the time it takes to synthesize ten seconds of sound at 44.1kHz.

Results are in Table 3. N is a multiple of 32 to ensure all warps are occupied. Bold entries take less than ten seconds to compute and thus ran faster-than realtime. A plot of the data is in Figure 2.

Table 3: *Time to generate 10s of Audio, Uniform Waveguides*

N DWGs	Baseline	Small Buffer	Short Waveguide
3584	**0.249**	**0.544**	**0.11**
14336	**0.522**	**0.811**	**0.272**
57344	**1.44**	**1.75**	**0.95**
114688	**2.79**	**3.09**	**1.83**
229376	**5.5**	**5.74**	**3.68**
458752	10.85	11.07	**7.28**
917504	21.14	20.24	14.59
1835008	42.611	49.72	29.152

We note some trends:

As expected, computing more waveguides requires more time. Scaling is sub-linear while growing at small N as we utilize more of the GPU in the parallel section of the benchmark ("for free"), but we still incur a cost for memory transfer of the outputs off the card, which itself scales linearly with N. The parallel sound synthesis portion of the program becomes linear with N as resources are exhausted; beyond this point we essentially are cycling through groups of warps serially.

Decreasing the buffer size from processing 45ms to 6ms of audio per kernel execution did not seem to affect the feasible N as much as anticipated. There is a notable 2x difference at small N but for both, the 458,000 waveguides trial was not feasible while the 230,000 waveguide trial used approximately 55% of the available time slice.

A variation of the trial using a shorter waveguide showed that through the range of our trial values of N, scaling is partially dependent on memory usage. As noted above, this is an experiment performed to validate that, as we might expect, longer-length delay lines may incur more computational cost. Of course, the delay line lengths used in practice will be defined by our physical model and sampling rate.

would have length $M/2$

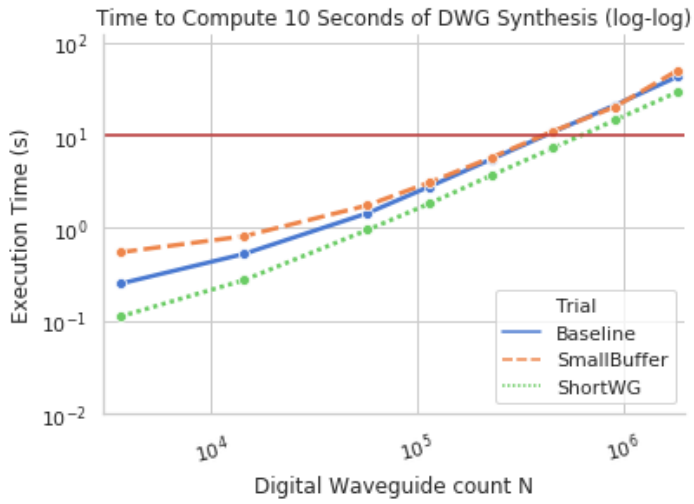


Figure 2: Time to run N waveguides for ten seconds of samples under different trials.

These preliminary benchmarks suggest a rough upper bound, so in our iterative development approach we return to coding, refining our kernel and un-relaxing some assumptions. Currently, all the waveguides have the same delay line length; this is unrealistic for real-world applications, so we next move to have waveguides play one of 1000 different pitches, by having each parallel digital waveguide own a delay line of different length. We use our baseline setup, and allocate the same blocks of memory as before, but have waveguide n be of length $128 + 5n \bmod 5000$. This means that all waveguides inside a warp will have different delay line lengths, and warps compute different values overall.

Results are in Table 4.

Table 4: *Time to generate 10s of Audio, "Baseline" uses same-length waveguides, "Differing" experiment uses heterogeneous waveguides. Slowdown Factor is the multiplicative performance penalty.*

N DWGs	Baseline(s)	Differ.Lengths(s)	Slowdown Factor
3584	**0.249**	**2.0**	8.03x
14336	**0.522**	**5.03**	9.63x
57344	**1.44**	19.72	13.69x
114688	**2.79**	39.41	14.12x
229376	**5.5**	78.41	14.26x
458752	10.85	156.08	14.38x

This is not ideal; we see a slowdown factor of 14x in our highly-parallel cases and went from supporting computation of 450 thousand simultaneous commuted waveguides to only 28 thousand. What changed? Two initial ideas come to mind:

Increased branch divergence: Some advice when writing GPU kernels is to avoid *branch divergence* where a portion of the threads in a warp take one path of an `if()` statement but others take the `else()`. This is because GPU threads do not have independent branching logic: the SIMT approach means that all threads execute each instruction in lockstep. In the case of an `if()` statement, threads evaluate the conditional and vote; if they are not unanimous, then both branches are executed in serial and threads ignore execu-

tion during the branch they did not take.

Looking at our code, the piece of our code that uses branching is trying to determine whether to loop in a delay line's circular buffer:

```
if (bufferIndex >= waveguideLength) {
  bufferIndex = 0;
}
```

The overhead of running both branches is minimal: we incur an extra instruction of setting a register to zero more often (1 cycle) and the else() branch is a no-op. In addition, GPUs have support for predicated instructions for short branches, which means this case is compiled to the non-branching code:

```
cond = bufferIndex >= waveguideLength
cond? bufferIndex = 0
```

This may be validated by looking at generated PTX pseudoassembly code, or using profiling tools to annotate branch divergence for each line of our source code after a test run.

The second thought of why we see slowdown when introducing heterogeneous delay line lengths is memory access patterns.

Our block of memory for waveguide state was defined to be of size `WARPSIZE*NWAVEGUIDES` by `BUFFERSIZE` rows. This means the N waveguides write to memory locations $0..N-1$ on the first sample, $N..2N-1$ on the second sample, etc.

Global memory access in CUDA is slow, but reads and writes may be coalesced; that is, if all threads in a warp are accessing data in the same aligned 128-byte block, only one to four line reads will need to occur (this is card-dependent). Newer cards have better caches, compiler optimizations, and runtime logic for global memory placement, but this is still worth considering.

In our case, consider we have 32 waveguides in a warp; these are of lengths $5000..5031$. During the first "trip" through the waveguide's circular buffer (first 5000 samples), memory is aligned as all waveguides index to the same offsets. Over the next several cycles through the waveguide, some waveguides will cycle earlier than others and eventually we will reach a state where we require simultaneous memory reads to 32 different lines, so slowdown will result.

Such memory accesses are cached, but with high numbers of waveguides we could easily evict old entries quickly. We open the CUDA Analysis tools, profile memory access, and find that this is indeed the case; Figure 3 shows lots of global memory accesses with only 2% hitting the L1 cache:

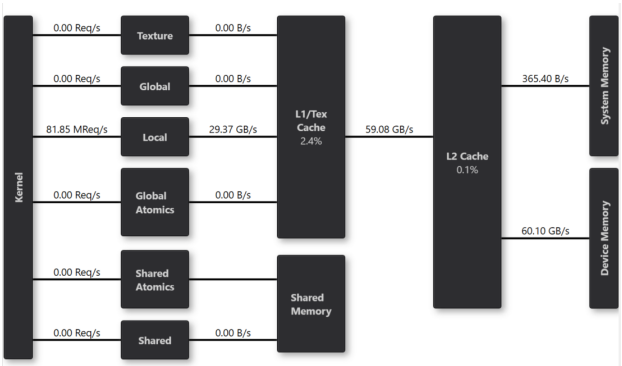


Figure 3: *CUDA memory profiler results. L1 cache hit rate is 2.4%*

To work past this slowdown, we propose two ideas:

3.0.1. *Synchronize on Cycle Point*

We could determine the longest waveguide in a block and ensure all waveguides' circular buffers loop at the same moment. For example, if we have guides of length 250 and 255, the former avoids writing to memory until our indexing counter loops back to index 1 of the array. The tradeoff is that we need to run more overall cycles in order to completely fill the output buffer from the shorter waveguide[3].

In a degenerate case, what if we have waveguides of length 5000 and 100? Only 2% of cycles are spent actually generating audio for the shorter waveguide with a naïve approach.

We could sort all waveguides by length globally, so that similarly-sized waveguides are in the same warp, to minimize the number of extra iterations–however this makes it much slower to later couple specific waveguides together, which we aim to do in a project that leverages this acceleration.

We could consider a middle ground where we have multiple eligible cycle points. Perhaps every 32 or 64 calculated samples, we could reset and unblock waveguides that are currently idle. This introduces a tradeoff between number of simultaneous memory accesses and number of cycles to compute at the end. On a positive note, in the case of coupling a busy memory controller with the "underpowered" 64-bit Floating Point unit on consumer cards would hide some of the drawback of each.

In the end we did not pursue this approach in depth; a second approach was more promising and produces more readable code:

3.0.2. *Shared memory*

Shared memory is a type of memory that belongs to a thread block and has much lower latency than global memory, but is only readable by threads within the block that owns it. This sounds great for our current use case and bottleneck. With our card's hardware, we have 96KB of shared memory available to each thread block. This means that we could choose, for example, two warps per block and have 1.5KB of RAM per thread, or 384 32-bit samples per thread.

We may wish to have longer waveguides than 384 samples, so we propose two workarounds:

- Lower utilization: Simply split the 1.5KB up among fewer threads, and use for example 24 out of 32 threads in a warp. While our overall utilization will be lower, the faster memory might save us enough time overall to run multiple copies of our work serially to increase N globally.
- Mixed-size waveguide groups: We could split the available shared memory such that, for example, the two shortest-length waveguides in a warp each donate half their buffer to the longest waveguide.

While the second approach may still seem problematic from a memory access point of view, the rules for shared memory access optimization are different than those for global memory access optimization. Shared memory on our GPU's architecture is grouped into 32 banks, and as long as two threads do not access the same bank at the same time (a *bank conflict*), we obtain full-speed access. On our card, access is actually done in two successive stages, each obtaining results for a half-warp, so the "donation" approach should be safe from slowdown as long as we can arrange memory accesses in time to have no bank conflicts. With simple donation schemes this is straightforward.

[3] we also note writes to that output buffer, previously perfectly aligned, are now unaligned themselves.

As a middle ground, we can sacrifice some compute utilization for larger buffer donations. Consider having each warp provide a developer a predefined "care package" of: (a) One "Extra Large" digital waveguide that occupies 4 banks of shared memory (length configurable, up to 1536 samples of delay), (b) Two "Large" waveguides owning 2 banks each (up to 768 samples each), and (c) 26 normal waveguides (up to 384 samples each).

This still lets us compute 28 waveguides per warp vs. the 32 we had before (87.5% effective utilization) but allows for lower frequency extension. We note that with such approaches we are becoming opinionated concerning the basic system building blocks; when we do this certain applications are enabled but we may block other applications.

We adjust our kernel to use shared memory. Due to the resource configuration of the graphics card and the dimensions of problem, we would use more shared memory than is available in each SM, so we must move from using 64 threads (2 warps) per block down to 32 threads (1 warp), at which point we are under the shared-memory-per-SM limit and our kernel can be scheduled. This serves as a reminder that GPU hardware is not as abstracted as we may be used to when coding for a CPU. In this particular case though, the "nvcc" compiler helpfully caught this at compile-time since it was an oversized static allocation, making for an easy fix.

We also include a quick performance gain of pinning memory on the host, accomplished by simply swapping `malloc` with the API call `cudaMallocHost`.

Results are in Table 5. As before, bolded entries are feasibly realtime. A plot of the same data is in Figure 4.

Table 5: *Time to generate 10 seconds of Audio, "Baseline" uses same-length waveguides, "Differing" experiments use heterogeneous waveguides with either global or shared memory.*

N DWGs	Baseline	differing: global mem...	...shared mem
3584	**0.249**	**2.0**	**0.58**
14336	**0.522**	**5.03**	**0.96**
57344	**1.44**	19.72	**1.82**
114688	**2.79**	39.41	**3.50**
229376	**5.5**	78.41	**6.79**
458752	10.85	156.08	12.59

To summarize: using shared memory allows us to make higher-waveguide counts tractable again. We can still run over a hundred thousand independently-sized waveguides with half of our cycles to spare for extending the algorithm.

At this point we have enough waveguides that we can spend some time thinking of creative applications for them. Those applications will certainly make them computationally more expensive by adding coupling, nonlinearities, modifiable tap points, fractional-length delays, etc.

4. AREAS FOR DEVELOPMENT

We stop here, but note there may still be room acceleration. For example, relatively new GPUs including ours have the ability overlap kernel executions with host/device memory transfers. If we were to double-buffer on the host and device, we can work on one array while the other transfers, and vice versa. This would help us especially at small N or if we wanted to copy hundreds of thousands of individual audio streams back to the host (skipping our current merge step where we sum them per-warp).

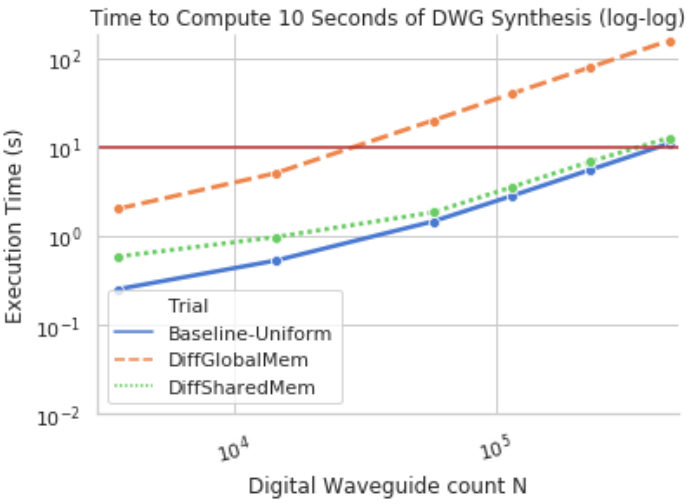


Figure 4: Time to run N heterogeneous waveguides for ten seconds of samples under different trials.

We have been discussing the general-purpose case of supplying waveguides of preconfigured lengths. With a pre-specified configuration, we could write tooling to efficiently group waveguide computations into a warp for maximum resource utilization.

In the modal filter bank, we note that we do not implement phase-correct input re-excitation which is a nice feature supported by these filters: we have logic to track zero-crossings but do not implement parameter updates from the host in a fashion that a "real" system would use. This is a simple and low-cost feature. Furthermore, it is likely that either CPU or GPU should interpolate parameters, which is work that is not being accounted for.

The high performance of these oscillators bodes well if we were to implement a massive collection of digital waveguide oscillators–another case requiring only a few variables and limited multiplies per cycle. It may be worth looking at algorithms that traditionally did well on VLSI architectures for use here, as the concept of many parallel independent instances of a module executing concurrently but varying on input data is shared between the two architectures.

5. CONCLUSIONS

We showed modern consumer GPUs may run high-Q phasor filters and 1D digital waveguides without needing to leverage parallelism across time. In particular, we showed that it is feasible to build a bank of several hundred thousand 1D waveguides, a hundred thousand 64-bit phasor filters with stable per-sample adjustments, or a few million phasor filters at 32-bit precision, all at 44.1kHz.

Reflecting on the the overall optimization development and debugging strategy used here: care must be taken to have the right number of warps grouped together into grids and blocks. Memory accesses should go to the fastest RAM possible, and we need to pay attention to memory alignment. While CPU code does benefit from similar optimizations, GPU algorithms rapidly fall in performance when parameters stray from the ideal range.

One other note around generalizing this code for end users: during development we consulted the capabilities of our particular graphics card several times, to see how many registers we have or to see the

various ways we can slice shared memory. While these parameters may be queried from the card at runtime and for the most part newer and more powerful GPUs contain a superset of old resources, this is not a guarantee, and for example if we tried to run our compiled waveguide binary on a GTX 480 from several generations back, it would fail to run because we request too much shared memory.

Still, optimization of these algorithms can be seen as an interesting puzzle; profiling tools make it easy to see where bottlenecks live (if not how to work around them), and it's fun to transpose an array or adjust memory layout and unlock a 10x speedup.

From a sound designer's point of view, being able to use so many of these building blocks at audio rates may allow for higher-fidelity physical models and modal effects, using commodity hardware that often sits idle while working with audio software.

6. ACKNOWLEDGMENTS

Special thanks to conference organizers, and to reviewers for helpful suggestions in improving background and presentation. Thanks to cited authors for providing foundational background for this work: demonstrating feasibility of parallel audio-rate algorithms on the GPU and describing an efficient, numerically stable high-Q filter.

7. REFERENCES

[1] Regina E Collecchia, Miriam A Kolar, and Jonathan S Abel, "A computational acoustic model of the coupled interior architecture of ancient chavín," in *Audio Engineering Society Convention 133*. Audio Engineering Society, 2012.

[2] Jonathan S. Abel, Sean Coffin, and Kyle Spratt, "A modal architecture for artificial reverberation with application to room acoustics modeling," in *Audio Engineering Society Convention 137*, Oct 2014.

[3] Julius O. Smith III, "Physical modeling using digital waveguides," *Computer music journal*, vol. 16, no. 4, pp. 74–91, 1992.

[4] Julius O Smith III and Scott A Van Duyne, "Commuted piano synthesis.," in *ICMC*, 1995.

[5] Lauri Savioja, Vesa Välimäki, and Julius O Smith, "Audio signal processing using graphics processing units," *Journal of the Audio Engineering Society*, vol. 59, no. 1/2, pp. 3–19, 2011.

[6] Lauri Savioja, Vesa Välimäki, and Julius O. Smith III, "Real-time additive synthesis with one million sinusoids using a gpu," *128th Audio Engineering Society Convention 2010*, vol. 1, 05 2010.

[7] Fernando Trebien and Manuel Oliveira, "Realistic real-time sound re-synthesis and processing for interactive virtual worlds," *The Visual Computer*, vol. 25, pp. 469–477, 05 2009.

[8] Jose Belloch, Balazs Bank, Lauri Savioja, Alberto Gonzalez, and Vesa Välimäki, "Multi-channel iir filtering of audio signals using a gpu," in *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, 05 2014, pp. 6692–6696.

[9] Belloch Rodríguez and José Antonio, *Performance Improvement of Multichannel Audio by Graphics Processing Units*, Ph.D. thesis, 2014.

[10] Jose A Belloch, Alberto Gonzalez, Enrique S Quintana-Orti, Miguel Ferrer, and Vesa Välimäki, "Gpu-based dynamic wave field synthesis using fractional delay filters and room compensation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 25, no. 2, pp. 435–447, 2017.

[11] Stefan Bilbao and Craig J Webb, "Physical modeling of timpani drums in 3d on gpgpus," *Journal of the Audio Engineering Society*, vol. 61, no. 10, pp. 737–748, 2013.

[12] Max Mathews and Julius O. Smith III, "Methods for synthesizing very high q parametrically well behaved two pole filters," in *Proceedings of the Stockholm Musical Acoustics Conference (SMAC 2003)(Stockholm), Royal Swedish Academy of Music (August 2003)*, 2003.

[13] NVIDIA Corporation, "NVIDIA CUDA toolkit documentation," `https://docs.nvidia.com/cuda/`, [Online].

CPU CONSUMPTION FOR AM/FM AUDIO EFFECTS

Antonio Jose Homsi Goulart
University of São Paulo
São Paulo, Brazil
ag@ime.usp.br

Marcelo Queiroz
University of São Paulo
São Paulo, Brazil
mqz@ime.usp.br

Joseph Timoney
Maynooth University
Maynooth, Ireland
joseph.timoney@mu.ie

Victor Lazzarini
Maynooth University
Maynooth, Ireland
victor.lazzarini@mu.ie

ABSTRACT

In this paper we present an assessment of the computational performance regarding the use of the AM/FM decomposition framework for the design and implementation of audio effects. The equations and intuitions are reviewed and audio examples are provided, alongside Csound code for real-time implementation. Two types of hardware and several computer music techniques were considered for the comparisons. We also introduce sqENVerb, a novel inexpensive reverb-enhancer effect.

1. INTRODUCTION

Following studies in areas like modulation vocoder [1] [2] [3] and modulation filtering [4] [5] [6], in our previous studies [7] [8] the non-coherent mono-component AM/FM paradigm was presented as a framework for the development of new audio effects. The theory was thoroughly revised and treated in [9], however, the computational effort required to run different types of effects was not addressed.

In this paper we present an assessment of the performance considering different computational systems and different audio processing techniques. Two kinds of computers were used, namely a RaspberryPi model 2B and a Lenovo ThinkPad x220. The former was chosen because it represents the category of low cost programming platforms, that can be used, among other applications, for audio processing; the later represents a more powerful and relatively popular computational system. Netbooks and old laptops might loosely fall in a category between these two examples. Notice also that many programming platforms similar to the Pi actually outperform it, in the same way that many computers assembled for gaming purposes outperform the ThinkPad. So the assessment presented here represents a somewhat conservative scenario; anything running satisfactorily on the Pi and ThinkPad should also run in these more powerful computers.

Beyond the CPU consumption, while our previous papers emphasised manipulations on the instantaneous frequency component of the AM/FM decomposition, now we also address an effect obtained by manipulating the envelope of the signal.

In Section 2 we will briefly review the AM/FM Hilbert-based framework and code for real-time implementation. In Section 3 a new reverb-like effect is introduced and evaluated with a brief objective assessment based on audio descriptors. Then we proceed in Section 4 to a presentation and discussion of the required computational power in order to run the AM/FM framework and effects. Finally, we conclude and point our current and future work. Audio examples will be referenced in the paper with the symbol [▶filename] and are available alongside Csound code for download[1].

2. THE AM/FM FRAMEWORK

The AM/FM decomposition unravels a signal $x(t)$ to a pair of components: an envelope $a(t)$ and an instantaneous frequency signal $f(t)$. Together these signals can modulate a sinusoid both in amplitude and frequency in order to obtain the original signal back, so

$$x(t) = a(t)\cos\left(\int_0^t f(\tau)d\tau\right). \tag{1}$$

We can also think of phasors and interpret the argument for the cosine as an instantaneous phase, which is given by regular increments (the sum represented by the integral) depending of the instantaneous frequency. For instance, a regular sinusoid is the projection on the x-axis of a phasor in which the increments are always the same (tied to its frequency).

In contrast to additive synthesis, where we think globally about the signal, the local aspect of the signals in the AM/FM framework tracks local dynamics in the envelope case, while the instantaneous frequency represents the frequency of a sinusoid that best fits the original signal at each instant.

One of the possibilities for implementing the decomposition is by means of an analytic signal

$$z(t) = x(t) + i\hat{x}(t), \tag{2}$$

where $i = \sqrt{-1}$ and $\hat{x}(t)$ is the Hilbert Transform of $x(t)$.

The Hilbert Transform shifts all the components in a signal by $90°$ [10], so it might be implemented by using a set of all-pass filters, as is done in the `hilbert` Csound opcode. The important characteristic of the analytic signal is the absence of the negative frequencies; its spectrum resembles the original spectrum of $x(t)$ on the positive frequencies, while the negative components are void, so

$$z(t) = \frac{1}{2\pi}\int_0^{+\infty} X(\omega)e^{i\omega t}d\omega, \tag{3}$$

[1] https://www.ime.usp.br/~ag/dl/lac19.zip

where $X(\omega)$ is the Fourier Transform of $x(t)$ [11]. In such a way we can interpret the analytic signal as a superposition of infinite phasors with different frequencies and radii, as shown in Figure 1.

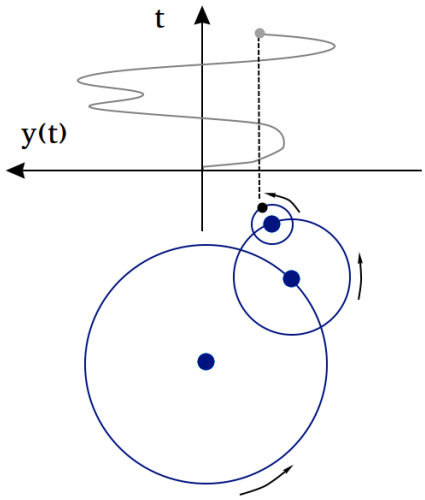


Figure 1: *Analytic signal as a superposition of phasors. The original signal is the projection of the analytic signal onto the real axis. Source: reproduced from [9]*

In Csound the AM/FM decomposition might be implemented with the following code:

```
opcode Udiff,a,a
 setksmps 1
 asig xin
 /* differentiation */
 asig diff asig
 ksig = downsamp(asig)
 /* phase unwrapping */
 if ksig >= $M_PI then
   asig -= 2*$M_PI
 elseif ksig < -$M_PI then
   asig += 2*$M_PI
 endif

 xout asig
endop

opcode AmFmAna,aa,a
  asig xin
  aim,are hilbert  asig /* xhat and x */
  a_am = sqrt(are^2 + aim^2) /* envelope */
  aph = taninv2(aim, are) /* inst. phase  */
                          /* inst. freq. */
  a_fm = Udiff(aph)*sr/(2*$M_PI)

  xout  a_am, a_fm
endop
```

Notice that the `hilbert` opcode is used in order to obtain the analytic signal, and also that the phase needs to be unwrapped. This opcode works in the time domain using 6th-order recursive filters to keep signals in quadrature. Alternatively, we could also employ the `hilbert2` opcode, which implements the same process using a frequency-domain approach implementing a finite impulse response filter (FIR) using a Fast Fourier Transform (FFT) algorithm. However, for this paper we have concentrated on using the former method due to the fact that the FIR approach introduces a latency between input and output that is proportional to the analysis window, and therefore it might not be as well suited to hard real-time applications. In the tests section, we will compare the costs of the time-domain AM/FM process against the application of FFT analysis-synthesis to a signal.

In order to design AM/FM effects we proceed to manipulations in $a(t)$ and/or $f(t)$ followed by a resynthesis step considering the modified signals, as represented in the following code:

```
opcode AmFmRes,a,aa
  a_am_p,a_fm_p xin
  xout a_am_p*cos(integ(a_fm_p)*2*$M_PI/sr)
endop
```

Notice that `a_am_p` and `a_fm_p` represent the potentially processed versions of the estimated `a_am` and `a_fm` (remember that Csound's audio variables names must start with "a").

3. SQENVERB: A NEW AM/FM EFFECT

In our previous papers different families of manipulations were described and thoroughly explained. For instance, the octIFer [8], a beautiful sounding octaver-like effect might be obtained by multiplying the instantaneous frequency signal by 0.5 [▶octifer-half] or even by 0.25 [▶octifer-quarter]. We emphasize, though, that these manipulations are not directly altering frequencies in the spectrum of the original signal, but are actually changing the increments that drive the phasor in the resynthesis process.

Now we describe an effect not yet considered in our previous studies. The manipulation is based on extracting the square root of the estimated envelope signal. The analytic signal envelope lies within the [0,1] range, and considering this interval as our domain for the square root function, we can affirm that the `sqrt` will always return values greater than the argument. Notice that

$$\frac{\sqrt{x}}{x} = \frac{1}{\sqrt{x}}, \tag{4}$$

so the closer the argument is to 0, the greater will be the relative gain. As a consequence, moments of low-intensity sound will be emphasized, leading to pronounced tails. Albeit reverberation is characterized by both early and late reflections [12], the reverberation is arguably more noticeable in the tail of the sound. In such a way the effect can be seen as a sort of compressor/expander [13] which in this case acts extending an already present reverberant tail in the sound.

Differently than a regular gain operation that multiplies the whole signal by the same amount, the square root application results in a selective gain along the signal duration, directly influencing its decay and thus the perception of length. In Figure 2 we can actually check the influence of the Root Mean Square in both the original signal [▶original] and the one with sqENVerb [▶sqenverb]. The RMS is an audio descriptor related to the perception of level in a sound.

As we would expect, the spectral information is not considerably altered by extracting the envelope's square root. In Figure 3 we can check the spectral centroid for the original audio and the sqENVerb edition. The spectral centroid [14] is an audio descriptor related to the perception of brightness in a sound. Both the RMS and spectral centroid evaluation were realized with the Essentia [15] library.

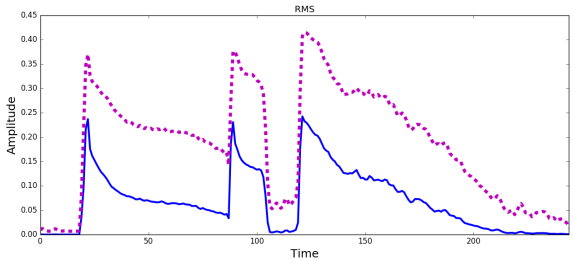


Figure 2: *RMS of dry (solid blue) and processed (dashed magenta) signals. Low intensity moments are greatly influenced by the square root operation.*

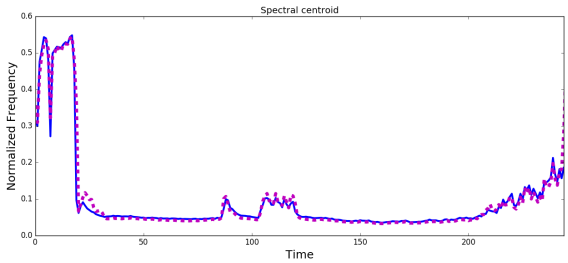


Figure 3: *Spectral centroid of dry (solid blue) and processed (dashed magenta) signals. The operation on the envelope does not considerably influence the spectral centroid.*

4. AM/FM CPU CONSUMPTION

The CPU consumption assessment in important both for artistic considerations (e.g. maximum tolerated latencies to avoid difficulties in musical performance) and also technical reasons (e.g. hardware sizing). In order to evaluate the computational effort to run the different effects, the software `time`[2] was used. It is executed from the shell with the command

```
time csound amfmdafx.csd
```

Here `amfmdafx.csd` refers to a Csound code with an AM/FM effect implemented. The default `time` execution returns three measures:

- real: total duration of the process under analysis;
- user: time taken to work directly on the process;
- sys: time taken to work on system tasks related to the process.

The CPU consumption is then given as $\frac{user+sys}{real}$.

The results[3] are shown in Table 1, which is divided in several parts:

- in the first part of the table some simple and inexpensive computer music tasks are evaluate just to set the scale for the comparisons;
- the second part shows the consumption for realising a FFT and an inverse FFT, considering different windows and hop sizes (shown as number of samples);
- the performance for classic octaver and reverb implementations are then shown;
- then the raw AM/FM framework performance is presented (decomposition followed by resynthesis, with no effects implemented);
- the second to last part shows the performance for some AM/FM effects explored in [9];
- the last part shows the performance considering the octIFer and the sqENVerb cases.

Table 1: *CPU consumption for different types of effects. *The RaspberryPi could not handle a 5000-sample long convolution reverb.*

	CPU consumption (%)	
	RaspberryPi 2B	ThinkPad x220
looped audio	9.49	4.69
clip distortion	10.77	5.06
FFT pair (1024/512)	26.35	8.52
FFT pair (1024/256)	37.38	10.37
FFT pair (1024/128)	45.76	12.31
FFT pair (512/256)	26.05	8.68
FFT pair (512/128)	33.68	10.47
FFT octaver (1024/128)	48.00	12.41
convolution reverb 2500	88.98	14.73
convolution reverb 5000	–*	22.97
simulation reverb	26.13	7.67
AM/FM framework	25.23	7.53
AM/FM IF filtering	29.72	7.73
AM/FM IF compression	33.21	7.78
AM/FM IF modulation	29.23	7.92
AM/FM octIFer	29.87	7.81
AM/FM sqENVerb	28.51	7.77

The FFT algorithm [16] is used widely for the design of audio effects, therefore we adopt it here as a benchmark against which we can measure the computing costs of the AM/FM framework. From the table we can check that both the FFT and AM/FM schemes are computationally accessible, and also that the AM/FM framework is lighter than the FFT/iFFT in all its cases. Another observation is that, in both frameworks, the implementation of a manipulation in the alternative domain does not cause a large increase in the CPU consumption, in comparison to the case where the raw frameworks are applied without any actual effects.

The octIFer effect delivers a high quality sonority [▶octifer-half] for a cost considerably lower than the classic contender [▶octaver], bearing good resemblance in the sonority.

The sqENVerb effect shows a similar consumption in relation to the simulated reverb case [▶simu-reverb], and a huge economy in relation to the convolution reverb. We emphasize that the 5000-sample impulse response convolution [▶conv-reverb5000] required almost twice CPU as the heaviest FFT case; it was not even possible

[2] `http://manpages.ubuntu.com/manpages/xenial/man1/time.1.html`

[3] In order to give more meaning to the numbers, the hardware specifications are: RaspberryPi 2B / quad-core ARM Cortex-A7 @ 900 MHz 32 bits / 1 GB SD-RAM @ 400 MHz / Raspbian / Csound 6.08; ThinkPad x220 / dual-core i5-2520M @ 2.5 GHz 64 bits / 8 GB RAM DDR3 @ 1333 MHz / Debian / Csound 6.09.1. The sample rate considered was always 44100 samples per second.

to run in the RaspberryPi, so another IR with 2500 samples was considered [►conv-reverb2500]. The sonority obtained in this case was not bad, but such a limitation might be questionable, and even with the short IR the Raspberry CPU was almost entirely taken. All the tested AM/FM examples leave considerable CPU headroom so other effects might be applied concurrently.

5. CONCLUSIONS

In this paper we presented, for the first time, a computational performance assessment of the AM/FM audio effects framework. The new AM/FM effect sqENVverb was also developed and compared to the established reverb techniques.

All the examples we explored are based on the non-coherent mono-component Hilbert Transform case of AM/FM decomposition. Different techniques for the decomposition are available, and richer scenarios might also be considered, for instance a filter bank framework, where the dry signal is separated in bands and the subsequent decomposition and processing are applied individually to each band, increasing the computational cost.

The AM/FM decomposition takes the signal to an alternative representation, where even subtle modifications in the envelope or instantaneous frequency signals might result in deep effects after the resynthesis.

The means by which both the octIFer and the sqENVerb effects emulate the octaver and reverb effects might not be orthodox, but the sonorities obtained in both cases resemble the classic techniques, at a considerably lower computational cost. The octIFer sound is quite similar to the classic octaver, and the sqENVerb works fine as a reverb, albeit lacking any control besides a dry/wet mix parameter (which is actually extremely efficient for tuning a reverb).

While it is true that powerful computational systems are increasingly available at decreasing cost, low-consumption algorithms will always be on demand: draining the battery of devices like tablets or smartphones with audio effects might not bring a good user experience; contemporary small single-board computers are still very limited in processing power; old laptops and netbooks, nowadays usually discarded, can instead be harnessed as terrific multi effect pedals.

Plugins for the octIFer and sqENVerb are currently being developed, to be released as open-source software.

6. ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001.

7. REFERENCES

[1] Sascha Disch and Bernd Edler, "An amplitude and frequency-modulation vocoder for audio signal processing," in *Proceedings of the International Conference on Digital Audio Effects (DAFX-08)*, Espoo, Finland, September 2008.

[2] Sascha Disch and Bernd Edler, "An iterative segmentation algorithm for audio signal spectra depending on estimated local centers of gravity," in *Proceedings of the International Conference on Digital Audio Effects (DAFX-09)*, Como, Italy, September 2009.

[3] Sascha Disch and Bernd Edler, "An enhanced modulation vocoder for selective transposition of pitch," in *Proceedings of the International Conference on Digital Audio Effects (DAFX-10)*, Graz, Austria, September 2010.

[4] S. Schimmel and L. Atlas, "Coherent envelope detection for modulation filtering of speech," in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, 2005.*, March 2005, vol. 1, pp. 221–224.

[5] Qin Li and Les Atlas, "Over-modulated am-fm decomposition," in *Proceedings of the SPIE - Advanced Signal Processing Algorithms, Architectures, and Implementations*, Bellingham, WA, 2004, pp. 172–183.

[6] P. Clark and L. Atlas, "Time-frequency coherent modulation filtering of nonstationary signals," *IEEE Transactions on Signal Processing*, vol. 57, no. 11, Nov 2009.

[7] Antonio José Homsi Goulart, Joseph Timoney, Victor Lazzarini, and Marcelo Queiroz, "Psychoacoustic impact assessment of smoothed AM/FM resonance signals," in *Proceedings of the Sound and Musical Computing Conference*, Maynooth, Ireland, July 2015.

[8] Antonio José Homsi Goulart, Joseph Timoney, and Victor Lazzarini, "AM/FM DAFx," in *Proceedings of International Conference on Digital Audio Effects (DAFx)*, Trondheim, Norway, December 2015.

[9] Antonio José Homsi Goulart, Marcelo Queiroz, Joseph Timoney, and Victor Lazzarini, "Interpretation and control in AM/FM-based audio effects," in *Proceedings of International Conference on Digital Audio Effects (DAFx)*, Aveiro, Portugal, September 2018.

[10] Stefan Hahn, *Hilbert Transforms in Signal Processing*, Artech House, Norwood, MA, 1996.

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

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

[13] Udo Zölzer, Ed., *DAFx: Digital Audio Effects*, Wiley & Sons, 2nd edition, 2011.

[14] James Beauchamp, "Synthesis by spectral amplitude and 'brightness' matching of analyzed musical instrument tones," *J. Audio Eng. Soc*, vol. 30, no. 6, 1982.

[15] Dmitry Bogdanov, Nicolas Wack, E. Gómez, Sankalp Gulati, Perfecto Herrera, O. Mayor, Gerard Roma, Justin Salamon, J. R. Zapata, and Xavier Serra, "Essentia: an audio analysis library for music information retrieval," in *International Society for Music Information Retrieval Conference (ISMIR)*, Curitiba, Brazil, 04/11/2013 2013, pp. 493–498.

[16] Julius Smith, *Spectral Audio Signal Processing*, W3K Publishing, 2011.

FORMALIZING MASS-INTERACTION PHYSICAL MODELING IN FAUST

James Leonard and Jérôme Villeneuve

Univ. Grenoble Alpes, CNRS, Grenoble INP*, GIPSA-lab,
38000 Grenoble, France
* Institute of Engineering Univ. Grenoble Alpes
james.leonard@gipsa-lab.grenoble-inp.fr

Romain Michon,[1,2] *Yann Orlarey*[1] *and Stéphane Letz*[1]

[1]GRAME-CNCM, Lyon (France)
[2]CCRMA, Stanford University (USA)
michon@grame.fr

ABSTRACT

This paper presents recent work conducted on the integration of mass-interaction physical models in the FAUST programming language. After a brief introduction to mass-interaction networks, FAUST, and previous works on this topic, we present a simple modeling framework, a FAUST code generator and its associated library, allowing to implement 1D mass-interaction models. In addition to the open-source tool itself, this research offers a perspective on formalizing arbitrarily large networks of bidirectional feedback couplings and state-space models in FAUST, through routing patterns. We finish with a set of examples, and discuss future perspectives and challenges.

1. INTRODUCTION

For several decades, physical modeling has been used to synthesize audio by means of simulating the behaviour of vibrating objects. A panoply of methods have been proposed over the years, from lumped discrete models [1], to Waveguides [2], to large scale Finite Difference schemes [3], that have gained in popularity with the increase of computing power. Creating a model of a mechanical instrumental system can be simpler than explicitly formulating the signal that it produces (as sound properties emerge from the physical conditions of the matter) and offers direct means for control and interaction, either by simulating musical gestures or by coupling the user and the virtual object, for instance using haptic technologies [4].

FAUST [5] is a functional programming language for real-time Digital Signal Processing (DSP) with a strong focus on the design of synthesizers, musical instruments, audio effects, etc. The FAUST compiler can be used to "translate" a FAUST program to various non-domain-specific-languages such as C++, C, JAVA, JavaScript, LLVM bit code, WebAssembly, etc. Thanks to a wrapping system, code generated by FAUST can be easily compiled into a variety of objects ranging from audio plug-ins to standalone applications, smartphone apps, web apps, etc.[1] This mechanism also makes it possible to add MIDI, OSC, polyphony, etc. support to any FAUST-generated program.

1.1. Mass-Interaction Physical Models

Pioneered in artistic applications by the CORDIS-ANIMA system [1] at ACROE, mass-interaction physical modeling allows to formulate physical systems in the form of lumped networks, composed of two main components: masses, representing material points in a given space (1D, 2D, 3D) with a given inertial behaviour, and interactions, each representing a specific type of physical coupling (i.e., visco-elastic, collision, non-linear, etc.) between two mass elements. Mass-interaction systems are now used in a variety of contexts (musical & other), partly for the fact that arbitrarily complex virtual objects can be described simply as a construction of elementary physical components. A basic model is shown in Figure 1.

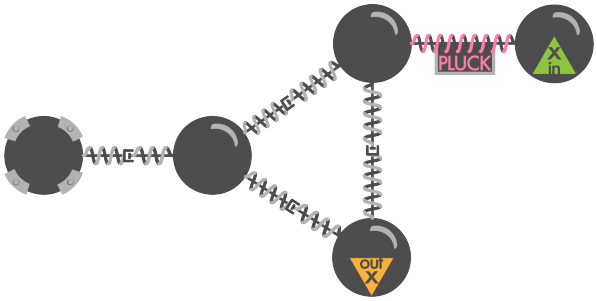


Figure 1: *Topological representation of a mass-interaction model. Here, a fixed point (represented on the left) is connected to a triangle composed of masses and dampened springs. An input module interacts with the top mass through a non-linear pluck interaction.*

Unlike FDTD methods [3], creating physical models with this formalism avoids the need to explicitly define a mathematical model (partial difference equations systems, boundary conditions, etc.) for a given physical structure beforehand. Therefore, it lends itself particularly well to iterative and exploratory design of "physically plausible" virtual objects, grounded in the laws of Newtonian physics but not necessarily limited to the mechanical constraints of the real world.

Mass-interaction physical models can contain anything from a couple of physical elements to tens or hundreds of thousands of them. Assembling and configuring the models element by element can be very time consuming. To this end, user-friendly modeling environments have been proposed, namely GENESIS [6] (and more recently *Synth-A-Modeler* [7]) for 1D audio applications. The former offers high level tools for generating topological structures, and analyzing/tuning physical constructions through modal analysis [8].

1.2. Current State of Physical Modeling in Faust

Various projects have been using FAUST to implement physical models of musical instruments.

The FAUST-STK [9] is a complete re-implementation of the waveguide and modal models of the Synthesis ToolKit (STK) [10]. It also contains various models from the *Soundius Project*. [2]

[1]The FAUST website contains an exhaustive list of all the FAUST targets: https://faust.grame.fr.

[2]Unfortunately, there is no documentation/publication on this project yet.

Julius O. Smith implemented a series of waveguide meshes that landed in the FAUST libraries[3] but that were never documented/published.

More recently, the FAUST physical modeling toolkit [11] was introduced. It is based on a library allowing for the implementation of bi-directional block diagrams in FAUST and containing a wide range of musical instrument parts that can be assembled in a modular way. It also comes with `mesh2faust`[12], a tool to generate modal physical models compatible with the FAUST physical modeling library using Finite Element Analysis (FEA).

The work presented in this paper was partly inspired by Ed Berdahl's *Synth-A-Modeler* [7] (which itself direcly draws upon CORDIS-ANIMA [1] and GENESIS [6]). This environment allows for the implementation of hybrid models combining mass-interaction systems with waveguide models using a graphical user interface (GUI). *Synth-A-Modeler* is based on a series of FAUST libraries and generates custom FAUST programs corresponding to the models implemented in the GUI. While it successfully combines various types of modeling techniques at a high level and facilitates their control using custom haptic interfaces such as the FireFader [13], it has, to our knowledge, never been used to implement large scale mass-interaction models.

Our proposed approach does not aim to supplant Berdahl's; rather, from a similar starting point it questions how the FAUST language's versatility can be used to formalize arbitrarily large mass-interaction models – and more generally speaking complex feedback networks – in a direct, concise and clear manner.

2. MASS INTERACTION PARADIGM IN FAUST

Before getting into implementation specifities, this section presents the basics of mass-interaction networks, in the case of 1D systems, in which all masses vibrate along a single z axis. These models are sometimes referred to as "zero-D", since they are purely topological and contain no direct geometrical information. First, discrete-time mass and interaction physical algorithms are presented and assembled into an explicit computational scheme.

Then, relying on a matrix-based representation of the topological network, we present a generic FAUST architecture that implements this computational scheme.

2.1. Discrete-Time Physical Algorithms

Below, we present finite difference implementations of two of the most basic elements in a mass-interaction network: punctual masses and springs.

2.1.1. Discrete-time implementation of a punctual mass

The motion equation for a continuous time mass is given by Newton's second law:

$$f = m.a = m\frac{d^2x}{dt^2} \tag{1}$$

Where f is the force applied to the mass, m is its inertia a its acceleration, and x its position. Applying the second-order central difference scheme, with the sampling interval noted ΔT, a discrete equation of the mass can be formulated as follows:

$$f(t) = m.\frac{x(t+\Delta T) - 2x(t) + x(t-\Delta T)}{\Delta T^2} \tag{2}$$

[3] `https://github.com/grame-cncm/faustlibraries`

Equation (2) can be normalized to unity, and rearranged in order to express the mass' position update scheme (discrete-time positions and forces are noted X and F) :

$$X_{(n+1)} = 2X_{(n)} - X_{(n-1)} + \frac{F_{(n)}}{M} \tag{3}$$

With M, the discrete time inertial parameter defined as :

$$M = \frac{m}{\Delta T^2} \tag{4}$$

Hence, the basic discrete-time mass module produces new position data based on its current position, previous position, the "discrete-time" mass parameter M, and the sum of forces applied to the mass from the previous interaction computation step.

The initial position $X_{(0)}$, delayed initial position $X_{(-1)}$ (which infers initial velocity) and initial force $F_{(0)}$ must be supplied at the start of the computation.

2.1.2. Discrete-time implementation of a dampened spring

The elastic force applied by a linear spring with a stiffness k and a resting length of $l_0 = 0$ connecting a mass $m2$ at the position x_2 to a mass $m1$ at the position x_1 is given by Hookes law:

$$f_{s1\to 2} = -k.(x_2 - x_1) \tag{5}$$

The exact equivalent of this equation in discrete time is :

$$F_{s1\to 2(n)} = -K.(X_{2(n)} - X_{1(n)}) \tag{6}$$

Where the discrete-time stiffness parameter $K = k$. The friction force applied by a linear damper with a damping parameter z connecting the same two masses is :

$$f_{d1\to 2} = -z.\frac{d(x_2 - x_1)}{dt} \tag{7}$$

Using the Backward Euler difference scheme, the frictional force can be formulated in discrete-time as :

$$f_{d1\to 2}(t) = -z.\frac{(x_2(t) - x_1(t)) - (x_2(t-\Delta T) - x_1(t-\Delta T))}{\Delta T} \tag{8}$$

Which after normalization becomes :

$$F_{d1\to 2(n)} = -Z.((X_{2(n)} - X_{2(n-1)}) - (X_{1(n)} - X_{1(n-1)})) \tag{9}$$

With the discrete time inertial parameter Z defined as :

$$Z = \frac{z}{\Delta T} \tag{10}$$

The global equation of the force applied by the dampened spring is composed of F_s and F_d :

$$\begin{aligned} F_{(n)} = &- K.(X_{2(n)} - X_{1(n)}) \\ &- Z.((X_{2(n)} - X_{2(n-1)}) - (X_{1(n)} - X_{1(n-1)})) \end{aligned} \tag{11}$$

It is applied symmetrically to each mass (Newton's third law):

$$\begin{aligned} F_{2\to 1(n)} &= -F_{(n)} \\ F_{1\to 2(n)} &= +F_{(n)} \end{aligned} \tag{12}$$

2.1.3. *Discrete mass - dampened spring - fixed point oscillator*

A linear harmonic oscillator is obtained by combining equations (3) and (11), in the case where X_1 is a fixed point set to $X_{1(n)} = 0$, $n \in \mathbb{Z}$. This results in :

$$X_{(n+1)} = \left(2 - \frac{K+Z}{M}\right).X_{(n)} + \left(\frac{Z}{M} - 1\right).X_{(n-1)} + \frac{F_{(n)}}{M} \quad (13)$$

Since the basic oscillator is a very common element in modeling, the integrated form given in (13) can be implemented in the form a specific mass-type module (although it is identical to assembling a mass, dampened spring and a fixed point).

2.1.4. *Generalization*

Any element in a mass-interaction model follows the basic template of the elements described above. More complex interactions stem from conditional statements (e.g. springs only active during interpenetration of two material points, as in visco-elastic collisions) or dynamic stiffness or damping parameters that depend on the position and/or velocity of the connected material points (e.g. through non-linear lookup tables, such as in plucking or bowing interactions [14]).

It is important to note that the M and Z parameters are dependent on the sampling interval. Hence, the oscillatory behaviour of physical models will be dependent on the sampling rate of the simulation.

2.2. Computation Scheme

Computing a mass-interaction model consists in calculating the *mass-type* and *interaction-type* algorithms in a closed loop. The explicit time step increment is carried by the masses, as shown in the discrete-time equation (3). The interactions in themselves are delayless operations, but can be computed since their output is fed back into the masses for the next calculation step (cf. Figure 2). In other words, calculating a step of real-time audio requires to run all the masses' algorithms once, then all the interactions' algorithms.

2.3. Representing the Topological Network

The topological connections of a mass-interaction model can be formalized as a routing matrix of dimensions $J \times 2K$, where J is the number of material elements (or *M points*) in the network, and K is the number of interactions (each interaction module has two connections - or *L points* in the usual terminology[1]) :

$$\begin{array}{c} \\ m_1 \\ m_2 \\ \vdots \\ m_j \end{array} \begin{array}{c} \begin{array}{ccccc} i_{0\_l1} & i_{0\_l2} & \dots & i_{k\_l1} & i_{k\_l2} \end{array} \\ \begin{pmatrix} 1 \text{ or } 0 & 1 \text{ or } 0 & \dots & \dots & 1 \text{ or } 0 \\ 1 \text{ or } 0 & \dots & \dots & \dots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 1 \text{ or } 0 & \dots & \dots & \dots & 1 \text{ or } 0 \end{pmatrix} \end{array} \quad (14)$$

Each column in the matrix must have a single connection set to 1 and all others to 0, as an *L point* only connects to a single *M point* (partially connected interactions are not allowed). On the other hand, a material point could be connected to any number of interactions in a given model (many connections set to 1 for a single line).

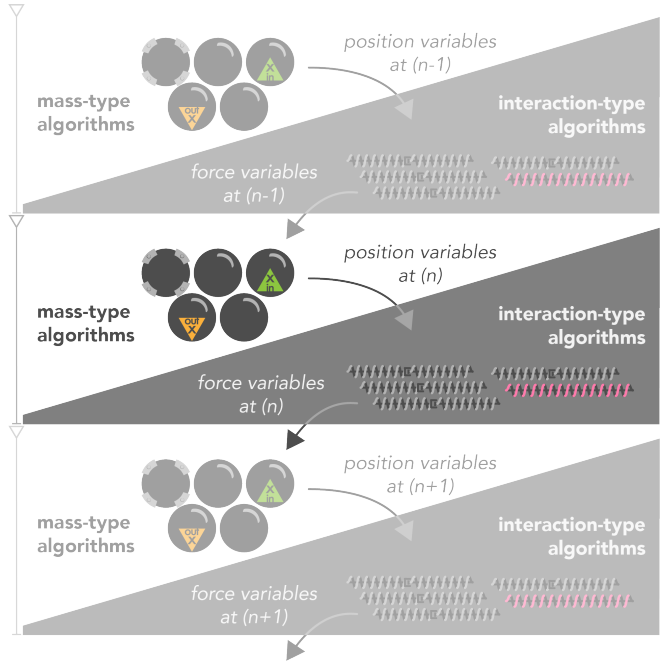


Figure 2: *Computation cycles of the model presented in Figure 1. At each time step, the mass-type algorithms are first computed using the forces calculated in the previous step, then the interaction-type algorithms are computed using the new positions.*

As an example, (15) presents the routing matrix for the topological structure shown in Figure 1. The material elements (fixed point, three masses and a position input module) are represented vertically and the *L points* of the four springs and the non-linear interaction are represented horizontally.

The closed-loop physical calculation scheme performed by FAUST is shown in Fig. 3. On the left, a LINKTOMASS connection function routes the force feedback signals produced by the interactions based on the routing matrix (thus calculating the sum of forces for each mass). The new positions of the material elements modules are then calculated. These positions are then fed into a MASSTOLINK connection function, that routes the signals to all of the concerned interactions. Finally, the pairs of force signals produced by the interactions are fed back for the next calculation step.

Position and force inputs are directly incorporated into the LINKTOMASS function, so that they are applied to the correct input module. Similarly, modules whose positions are observed as audio outputs are simply added as extra signals at the end of the MASSTOLINK function.

2.4. FAUST Implementation of Mass and Interaction Elements

The `mi.lib` library contains the FAUST implementation of most elementary mass-type elements (i.e., masses, fixed points, oscillators, etc.) and link-type elements (i.e., springs, collisions, non-linear plucking / bowing, etc.). Since the implementations are similar, we will explicit only the two simplest and most common elements below: the mass and the spring.

$$\begin{array}{c} \\ s \\ m_0 \\ m_1 \\ m_2 \\ in \end{array} \begin{array}{c} \begin{array}{cccccccccc} r_{0\_1} & r_{0\_2} & r_{1\_1} & r_{1\_2} & r_{2\_1} & r_{2\_2} & r_{3\_1} & r_{3\_2} & nl_1 & nl_2 \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \tag{15}$$

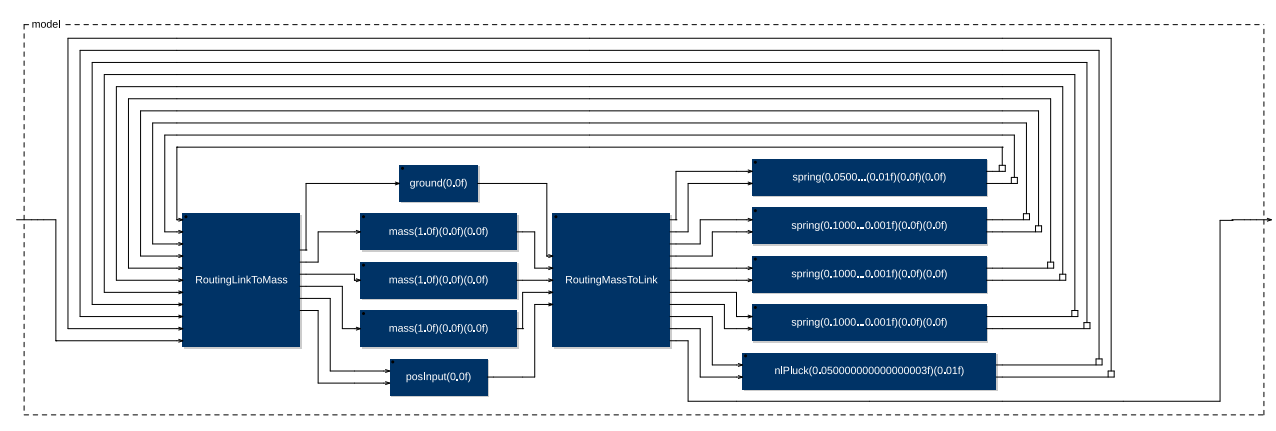


Figure 3: FAUST-*generated diagram corresponding to the model presented in Fig. 1.*

2.4.1. Mass

The discrete-time algorithm of a basic mass module described in (3) can be easily expressed with `letrec` environment expression in FAUST:

```
mass(m,x0,x1) = equation
with{
    A = 2;
    B = -1;
    C = 1/m;
    equation = x
    letrec{
        'x = A*(x : initState(x0)) +
             B*(x' : iniState((x0,x1))) +
             *(C);
    };
};
```

Listing 1: *The discrete-time mass algorithm in* FAUST.

The module takes an input signal (the sum of all forces fed back through the interaction feedback loop and routing function) and produces a position output. The initial position and delayed position of the module are dealt with using the `initState` function, which initializes the first step with the correct values.

2.4.2. Dampened spring

Similarly, a visco-elastic spring expressed in FAUST is shown in Listing 2. Interaction modules such as the spring take two input signals (the positions of the masses connected together by the link) and produce two identical and opposite force signals.

Attention must be paid to the correct initialization of velocity based interactions, especially when the initial position or speed of the masses is non-zero. To this end, the delayed initial positions of the two connected mass elements are supplied as arguments to the interaction function, which initializes them with the `initState()` function.

```
spring(k,z,x1r0,x2r0,x1,x2) =
    k*(x1-x2) +
    z*(
        (x1 - (x1' : initState(x1r0))) -
        (x2 - (x2' : initState(x2r0)))
    )
    <: *(-1),_;
```

Listing 2: *The discrete-time dampened spring algorithm in* FAUST.

3. CREATING MODELS WITH MIMS

MASS INTERACTION MODEL SCRIPTER[4] is a simple graphical or command-line tool written in Python to generate structured FAUST code from a textual description of a physical model.

Models are described in a format similar to the PNSL language [15]: each physical element has a specific label, specific physical parameters and/or initial conditions, etc. Parameters can be added to this description and shared by any number of physical modules, allowing global variation of the physical attributes (i.e., stiffness, damping, mass, etc.) of a subset of modules in real-time.

MIMS' `physics2faust` tool compiles the model by :

- parsing all of the physical modules and noting any specific elements (i.e., position or force inputs, audio outputs, etc.)
- creating the routing matrix and translating it into the two dual FAUST routing functions.

[4] https://github.com/mi-creative/MIMS

- ordering the resulting data into the output *.dsp* file. "Placeholder" functions are created for position / force inputs, allowing the user to describe his input functions directly in the FAUST code.

```
# Define global parameter attributes
@m_K param 0.1
@m_Z param 0.001

@nlK param 0.05
@nlScale param 0.01

# Create material points
@m_s0 ground 0.
@m_m0 mass 1. 0. 0.
@m_m1 mass 1. 0. 0.
@m_m2 mass 1. 0. 0.

# Create and connect interaction modules
@m_r0 spring @m_s0 @m_m0 0.05 0.01
@m_r1 spring @m_m0 @m_m1 m_K m_Z
@m_r2 spring @m_m1 @m_m2 m_K m_Z
@m_r2 spring @m_m2 @m_m0 m_K m_Z

# Inputs and outputs
@in1 posInput 0.
@out1 posOutput @m_m2

# Add plucking interaction
@pick nlPluck @in1 @m_m1 nlK nlScale
```

Listing 3: MIMS *description for the model presented in Fig. 1.*

The graphical UI version of MIMS also provides basic tools for generating certain categories of physical structures (i.e., strings, membranes, etc.) and performing modal analysis of linear structures.

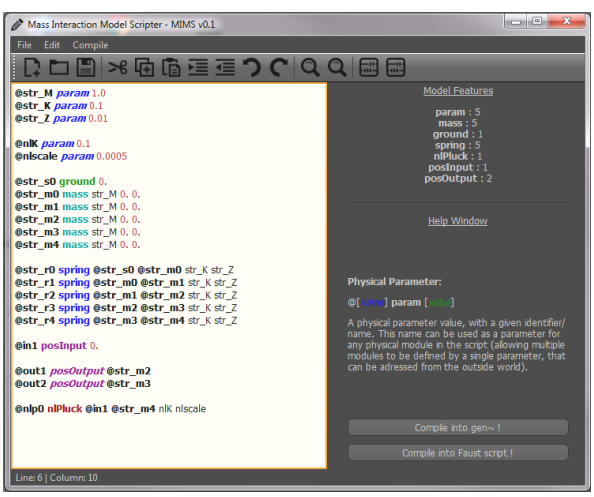


Figure 4: *The MIMS model editor prototype.*

The FAUST code generated from the model in Code Listing 3 is presented in Code Listing 4. The only hand-written element is the `inPos` function, that adds a graphical slider to control the position of the input mass. The control-rate output of the slider is smoothed to avoid artifacts.

```
import("stdfaust.lib");
import("mi.lib");

inPos = hslider("pos",1,-1,1,0.0001) : si.
    smoo;

OutGain = 10.;

m_K = 0.1;
m_Z = 0.001;
nlK = 0.05;
nlScale = 0.01;

model = (
    RoutingLinkToMass:
        ground(0.),
        mass(1.,0., 0.),
        mass(1.,0., 0.),
        mass(1.,0., 0.),
        posInput(0.) :
    RoutingMassToLink :
        spring(0.05,0.01, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        spring(m_K,m_Z, 0., 0.),
        nlPluck(nlK,nlScale),
    par(i, 1,_)
)~par(i, 10, _): par(i, 10,!), par(i,  1, _)
with{
RoutingLinkToMass(l0_f1,l0_f2,l1_f1,l1_f2,
    l2_f1,l2_f2,l3_f1,l3_f2,l4_f1,l4_f2,in1)
    = l0_f1, l0_f2+l1_f1+l3_f2, l1_f2+l2_f1+
    l4_f2, l2_f2+l3_f1, l4_f1, in1;
RoutingMassToLink(m0,m1,m2,m3,m4) = m0, m1,
    m1, m2, m2, m3, m3, m1, m4, m2,m3;
};
process = inPos : model: *(OutGain);
```

Listing 4: MIMS *description for the model presented in Figure 1.*

4. EXAMPLES AND EVALUATION

The basic *mi_faust* package contains several examples of virtual instruments and use-cases of mass-interaction physics in FAUST. All of these examples can be compiled and executed directly as web applications via the FAUST online editor,[5] with generic user interfaces. They can also be found as pre-compiled web-apps on the *mi_faust* project web-page.[6]

- `IPlayTheTriangle`: the demonstration model discussed previously in this paper (Figure 1).
- `PolyTriangle`: the same model (with a direct force impulse applied instead of a pluck system), using FAUST's ability to automatically handle polyphonic voice allocation for MIDI instruments.
- `PluckedHarmonics`: a 150-mass string terminated by two fixed points. The first position input allows plucking the

[5] https://faust.grame.fr/tools/editor/
[6] https://faust.grame.fr/community/made-with-faust/mi-faust

string, and three others are used to press down lightly on the string at specific areas in order to bring out natural harmonics.

- `BowedString`: a bowed string, using the *nlBow* interaction. The user can control bow pressure and velocity, as well as the stiffness of the string.
- `LargeTriangleMesh`: a big triangular mesh, fixed at one summit, excited by a plucking system and damped by user input.
- `Resonator`: the audio input is fed into one end of a resonating physical model. The user can alter the properties of the resonator.
- `PhysicalLFO`: Using a physical model with slow dynamics as a control variable for another synthesis process. Here, the wave propagation observed along a very loose string is used to modulate the amplitude of a white noise source, generating AM modulation going from complex patterns at the onset to quasi-sinusoidal modulation as the higher modes of the string decay.

In addition to these examples, two large structures (a 20 by 30 mass mesh: `20x30mesh` and a 1000 mass string: `1000 massString`) were created for model complexity tests. The bench test results in Table 1 show the compile time and CPU load for various models. Large routing functions result in slower compilation, and maximum complexity is reached for approx. 1800 physical elements. Overall, fairly complex models run well, with a reasonable CPU load.

5. FUTURE WORKS

5.1. Faust

FAUST proves to be well adapted to implement mass-interaction physical models. The combination of connection matrices and of the use of the `letrec` environment expression allowed us to seamlessly implement the various elements of `mi.lib`. However, this raised some issues that will need to be solved in the future. They are presented below.

5.1.1. Specifying Initial States in `letrec`

The `letrec` environment expression doesn't allow us to specify an initial state (i.e., the value of $y(n-1)$, $y(n-2)$, etc. at $n = 0$). We got around this problem by implementing the `initState` function which requires some unneeded computation. Hence, `letrec` could be modified to allow this type of expression to be written (rewriting Code Listing 1):

```
equation = x
letrec{
    x' = x0;
    x'' = x1;
    'x = A*x +B*x' + *(C);
};
```

We believe that this would significantly reduce computation for large scale models.

5.1.2. Optimizing Routing Matrices

The current "bare bone" implementation of connection matrices (e.g., `RoutingLinkToMass` in Code Listing 3) is hard to solve by the FAUST compiler, preventing large models to be generated (see §4). This could be solved by turning this operation into a primitive of the language. Compilation time would be significantly reduced since pattern matching [5] wouldn't be involved to solve this type of expression.

6. CONCLUSIONS

In this paper, we have presented early results of formal integration of 1D mass-interaction physical modeling into the FAUST environment, resulting in a new library. The MIMS and *physics2faust* tools allow to automatically generate FAUST dsp code for complex topological models, by expliciting the routing scheme for the model's position and force signals. Several basic models have been implemented and benchmarked, showing promising results. Furthermore, FAUST's capabilities offer an efficient solution for playing several dynamically allocated and parameter-mapped instances of a physical model across large ranges. More generally, this work extends beyond mass-interaction modeling and explores the possibilities for describing complex feedback networks and state space-models in FAUST.

7. REFERENCES

[1] Claude Cadoz, Annie Luciani, and Jean Loup Florens, "Cordis-anima: a modeling and simulation system for sound and image synthesis: the general formalism," *Computer music journal*, vol. 17, no. 1, pp. 19–29, 1993.

[2] Julius O. Smith, "Physical modeling using digital waveguides," *Computer Music Journal*, vol. 16, no. 4, pp. 74–91, Winter 1992.

[3] Stefan Bilbao, *Numerical Sound Synthesis: Finite Difference Schemes and Simulation in Musical Acoustics*, John Wiley and Sons, Chichester, UK, 2009.

[4] James Leonard, Nicolas Castagné, Claude Cadoz, and Annie Luciani, *The MSCI Platform: A Framework for the Design and Simulation of Multisensory Virtual Musical Instruments*, pp. 151–169, Springer International Publishing, Cham, 2018.

[5] Yann Orlarey, Stéphane Letz, and Dominique Fober, *New Computational Paradigms for Computer Music*, chapter "Faust: an Efficient Functional Approach to DSP Programming", Delatour, Paris, France, 2009.

[6] Nicolas Castagné and Claude Cadoz, "Genesis: a friendly musician-oriented environment for mass-interaction physical modeling," in *ICMC 2002-International Computer Music Conference*. MPublishing, 2002, pp. 330–337.

[7] Edgar Berdahl, "An introduction to the Synth-A-Modeler compiler: Modular and open-source sound synthesis using physical models," in *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, USA, May 2012.

[8] Jérôme Villeneuve and Claude Cadoz, "Understanding and tuning mass-interaction networks through their modal representation," in *40th International Computer Music Conference/11th Sound and Music Computing Conference*, 2014, pp. 1490–1496.

Model Name	N. Masses	N. Springs	FAUST Comp. Dur.	CPU Load
`1000massString`	1000	1002	-	-
`20x30mesh`	598	1151	20.576s	45%
`BowedString`	150	152	1.962s	14%
`IPlayTheTriangle`	3	5	0.029s	1%
`LargeTriangleMesh`	324	901	12.083s	48%
`PhysicalLFO`	10	12	0.032s	1%
`PluckedHarmonics`	150	152	2.192s	14%
`PolyTriangle`	3	5	0.027s	1%
`Resonator`	30	32	0.056s	4%

Table 1: *Number of masses and springs, compilation duration, and CPU load of the examples. Measurements were made on a Lenovo ThinkPad X1 Carbon with the following configuration: Linux Manjaro, Intel i7-7500U 4 cores at 2.7GHz, 16GiB of RAM, sampling rate of 48KHz, buffer size of 256 samples. Programs were compiled as ALSA applications with a GTK interface using* `faust2alsa`.

[9] Romain Michon and Julius O. Smith, "Faust-STK: a set of linear and nonlinear physical models for the Faust programming language," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011.

[10] Perry Cook and Gary Scavone, "The Synthesis Toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC-99)*, Beijing, China, 1999.

[11] Romain Michon, Julius O. Smith, Chris Chafe, Ge Wang, and Matt Wright, "The faust physical modeling library: a modular playground for the digital luthier," in *Proceedings of the 1st International Faust Conference (IFC-18)*, Mainz (Germany), 2018.

[12] Romain Michon, Sara R Martin, and Julius O Smith, *MESH2FAUST: a Modal Physical Model Generator for the Faust Programming Language - Application to Bell Modeling*, Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2017.

[13] Edgar Berdahl and Alexandros Kontogeorgakopoulos, "The firefader: Simple, open-source, and reconfigurable haptic force feedback for musicians," *Computer Music Journal*, vol. 37, no. 1, pp. 23–34, 2013.

[14] James Leonard and Claude Cadoz, "Physical modelling concepts for a collection of multisensory virtual musical instruments," in *Proceedings of the Conference on New Interfaces for Musical (NIME15)*, Baton Rouge, USA, May 2015.

[15] Nicolas Castagné, Claude Cadoz, Ali Allaoui, and Olivier Tache, "G3: Genesis software environment update," in *ICMC 2009*. MPublishing, 2009, pp. 407–410.

ARE PRAAT'S DEFAULT SETTINGS OPTIMAL FOR INFANT CRY ANALYSIS?

Giulio Gabrieli

Psychology Program, School of Social Sciences
Nanyang Technological University, Singapore
GIULIO001@e.ntu.edu.sg

Wan Qing Leck

Psychology Program, School of Social Sciences
Nanyang Technological University, Singapore
LECK0006@e.ntu.edu.sg

Andrea Bizzego

Department of Psychology and Cognitive Science
University of Trento, Italy
andrea.bizzego@unitn.it

Gianluca Esposito

Psychology Program, School of Social Sciences
Nanyang Technological University, Singapore
Department of Psychology and Cognitive Science
University of Trento, Italy
gianluca.esposito@ntu.edu.sg
gianluca.esposito@unitn.it

ABSTRACT

In recent years, the number of studies investigating possible non-invasive health screening techniques for infants have increased exponentially. Amongst those, one of the most prominent is health screening based on the acoustic investigation of infant cry. Clinicians involved in the field moved from visual inspection of the audible spectrum to automatized analysis of cry samples using computer software. A software that has been more widely adopted in recent years is Praat, a free software designed for speech analysis. Unfortunately, the software's default settings are not suitable for investigation of cry samples, yet rarely used settings are reported in final manuscripts. In this article, we tested 4 different computer generated signals, with frequency features comparable to cry frequencies, and 3 real cry samples using both Praat's standards and tuned settings. Our results highlight the importance of properly tuning software's parameters when expanding their field of usage, and provide a starting point for the development of optimal Praat algorithm's parameters selection for cry analysis.

1. INTRODUCTION

Screening of infants' health statuses can lead to early recognition of developmental pathologies, this allows clinicians to define an intervention program, which can lead to enhanced outcomes when adopted in earlier stages of life. Among infants' health screening methods, non-invasive techniques received the highest level of attention within the community of pediatricians and researchers. Starting from the second half of the Twentieth Century, researchers investigated several possible ways to identify different pathologies and developmental issues through non-invasive methods.

For example, pulse oximetry, a non invasive technique that measures the amount of oxygenated and deoxygenated hemoglobin in blood by mean of infrared light, has been widely tested for early screening of congenital heart defects in asymptomatic newborn babies [1, 2, 3, 4, 5]. Recently, in a review by Thangaratinam et al. [5], authors compared the overall sensitivity of this method and false-positive ratio against other screening techniques, including prenatal ultrasounds and routine physical exams [6, 7]. [what are the results?] One of the techniques in which researchers' interest increased exponentially during the last sixty years is the empirical analysis of infant cry [8, 9]. Acoustical properties of infant cry have been associated with different developmental pathologies, including Autism Spectrum Disorders (ASD), Sudden Infant Death Syndrome (SIDS), hearing impairments and unilateral cleft lip and palate (UCLP) [10, 11, 12].

1.1. Properties of Infant cry

Cry sound utterances are produced by the larynx during the expiratory phase of respiration. Pressure differences of air streams flowing through the larynx cause vocal folds to open and close rapidly, from about 250 to about 550 times per second in healthy infants [13, 14, 15, 16, 17, 18, 19]. This ratio of vibration is defined as fundamental frequency (F_0) [20, 21]. Position of the vocal folds is modulated by central nervous system (CNS), and therefore activity of the vocal folds can be used to estimate an infant's developmental status. Moreover, the lower vocal tract produces different sound characteristics, including the loudness of the expiratory phase.

The upper vocal folds concur instead in the production of higher frequencies, resonants of the fundamental frequency [22, 23]. During the first two years of life, an infant's body evolves. The vocal tract shapes during this period, and therefore acoustical properties of cry vocalizations changes accordingly. [24, 25, 26].

Research studies conducted on infants suffering from pathological conditions highlighted a positive shift in the spectrum of cry frequency properties, as compared to those of healthy infants. For example, investigation of infants at high risk of developing ASD disorders showed that the fundamental frequency of their cry vocalization can be higher than 700 Hz [27]. Analogous, F_0 collected from vocalization of infants suffering from colic were significantly higher than those collected from healthy infants [28].

1.2. Cry analysis

In a typical cry experiment, audio recordings are collected by inducing infants to cry using a specific paradigm, or trigger (e.g. pain caused heel prick test [29]). Collected samples are then preprocessed to increase the signal to noise ratio.

During the 1960s, when systematic analysis of infant cry began, researchers relied on visual inspection of spectrograms [30, 31]. With the advent of more powerful computing devices, techniques and algorithms employed in cry analysis became more sophisticated, producing more accurate and useful results.
Because of the similarities between infant vocalization and adult voice, cry researchers adopted software designed for speech analysis. One of the software most widely used within the field is Praat, a free software developed by Paul Boersma and David Weenink, specifically designed for acoustic analysis of adult voice [32]. In the last 18 years, Praat has been used in 41.3% (N=36) of the articles published within the field during this period (N=87), detailed information about the software in use is provided[8, 9]. Despite being a robust tool for speech analysis, Praat's default parameters are not suitable for accurate analysis of cry samples. In this article we discuss the role of Praat in cry analysis, highlighting the reasons for which standard settings are not suitable and provide suggestions on how to apply it successfully on cry samples.

2. PRAAT

Praat features a graphical user interface that fits the needs of different researchers, from phoneticians to musicians and biologists involved in the acoustic analysis of animal vocalizations. Written in C and C++, Praat provides tools for analysis of signals' pitch (F_0) and formants in audio signals. Not only that, Praat comes with a picture tool which produces high-quality graphics ready to be used in manuscripts and dissertations.
The software uses a general purpose scripting language that can be used to automatize the analysis of multiple files, allowing for fast processing of large amount of auditory samples [32].
Praat implements an auto-correlation algorithm for pitch analysis. According to Boersma, the applied algorithms is not only more accurate than other frequency-based pitch detection procedures, but is also less dependent on the length of selected window and more resistant to rapid shifts and external noise [33].

2.1. Praat settings

In this work, settings have been verified on Praat version 6.0.43 (8 September 2018), running on a Linux machine (Linux Mint 19 Tara x86_64, Kernel: 4.15.0-42-generic).

2.1.1. Pitch

Default pitch settings point the algorithm to search for F0 in the frequency range that goes from 75Hz to 500Hz. As introduced above, healthy infant cry's fundamental frequency usually lays between 250 and 550 Hz, with the latter higher in sick infants.
With those settings, there are at least two possible situations in which Praat cannot identify the real fundamental frequency value:

- *F_0 is above the upper cutoff:* In this situation, Praat will identify a wrong value (lower) for the fundamental, or provide no pitch information within a window.
- *F_0 lays between the cutoff values but a strong noise with a frequency between 75 and 250Hz is present*: In the situation where a strong periodical noise is recorded within the signal, such as the presence of a split-system air-conditioner within the recording environment [34], it is possible that the software identifies this lower frequency as the real fundamental, especially when this noise is about half of the real fundamental frequency.

2.1.2. Formants

Standard formant settings are used to obtain up to 5 formants with a frequency lower than 5500Hz. The GUI returns $n-1$ formants' frequency values, where n is the number of formants indicated within the settings.

3. ANALYSIS OF COMPUTER GENERATED SIGNALS

To better illustrate pitch and formant extraction errors, we tested Praat with standard and cry suitable settings on a set of computer generated signals with a specific F_0, to which white noise was added. Formants (N=5), with a frequency of about $F_0 * (n+1)$ and decreasing amplitudes [35, p. 306] have been added to the generated signals. For half of the files, noise at a specific frequency band, close to $F_0/2$. was added. Four different signals of 5s length have been generated. Audio files and the source code written in Python which were used to generate those signals are available online [1]. Used frequency values for F_0, formants and, where added, $F_0/2$, are reported in Table 1 (*Real*). To verify the validity of generated files, a visual inspection of the spectrum was conducted. Frequency peaks are shown in parenthesis in Table 1.
Using Praat we extracted value of Pitch and Formants at $t = 2.5s$, using both Praat's standard (*Praat S.*) and cry-optimized (*Praat O.*) settings:

- **Pitch:**
  - Pitch range (Hz) = 250.0 - 800.0 Hz
- **Formants:**
  - Maximum formants (Hz) = 4500.0 Hz

Fundamental frequencies and formant have also been verified by visual inspection of the spectrogram using Audacity version 2.2.1 and the following settings:

- **Algorithm:** Spectrum
- **Window size:** 1024
- **Function:** Hanning window
- **Axis:** Logarithmic frequency

Pitch and formants frequency obtained using the two set of settings, and their Mean Absolute Percentage Error (*MAPE*) are reported in Table 1.

4. DISCUSSION

As described above and demonstrated by analysis on simple computer generated samples, Praat's default settings are not suitable for the analysis of infant cry. In example *A.wav*, F_0 is located between the pitch cutoff values and no periodic noise was added. We can observe, that parameter optimization led to a general improvement of formant estimation, with the MAPE drastically reduced.
Similarly, in example *B.wav*, where F_0 was still between pitch cutoff values and periodic noise was above the lower cut-off with standard settings but not parameters optimized, the latter configuration

[1] `https://github.com/ABPLab/Praat-LAC2019`

granted a better recognition of the fundamental as well as of the formant.

In example *C.wav*, F_0 was higher than the upper cut-off for the pitch of Praat's standard settings. Here, pitch recognition identified the wrong peak as the signal pitch. This situation did not occur when parameters were optimized and the higher cut-off was increased. This is especially important when working with pathological infants or where the risk of developmental pathology is high, and therefore acoustic properties of cry are expected to differ from those of healthy infants.

Finally, as shown with file *D.wav*, when the presence of periodic noise was at about half of the fundamental frequency (with a high fundamental frequency), it led the software to a recognition error even with optimized parameters. This did not happen when the spectrum was visually inspected, since it was clear that the amplitude of $F_0/2$ was lower than the amplitude of the peak of F_0, as visible in Figure 1

Parameter tuning sharpens extracted features, but because of the properties of cry, researchers still have to pay special attention to obtained values, as well as to the quality of collected data.

Generally, we can expect Praat with standard settings to perform poorly when employed in infant cry studies, because of the complexity of the signal itself and of the presence of external noise. In the next section, we will shows the performances of Praat on real cry sample, using both the standard and optimized settings.

5. ANALYSIS OF REAL CRY SIGNALS

In order to provide a demonstration of Praat's performance on real cry samples, we analyzed infant utterances from a public dataset [36]. More specifically, we assessed the first three utterances from the file "BabyCrying2.wav", therefore named here as "Utterance1", "Utterance2" and "Utterance3".

F_0 and formants have been first obtained by visual inspection with Audacity, using the same configurations used to obtain the spectrum of computer generated signals. Because of the properties of cry, reported value are the mean values of a whole utterance. Frequencies' peaks are reported in Table 2. Then, each utterance have been analyzed in Praat, using both the default settings (*Praat S.*) and our suggested settings (*Praat O.*). For each pair of file and settings, we estimated the Mean Absolute Percentage Error using as actual value the peak obtained manually in Audacity by visual inspection of the spectrum. Pitch and formants frequency values and MEAP per file and settings are reported in Table 2.

6. DISCUSSION

As shows in Table 2, the difference in the estimated MEAP of investigates samples follows what have been shown for computer generated signals in Table 1. Similarly to the previous examples, the higher the formant number, the higher the difference between the peak detected in Praat or by visual inspection.

With an average reduction in the estimated MEAP of 18.4%, a fast optimization of pitch and formant detection parameters demonstrated to be helpful in increasing the accuracy of estimated features. As demonstrated by our examples, differences in the used settings can result in a large variance in estimated frequency values. Because of that, we expect researchers involved in cry studies to tune the software properly and to report used settings in final manuscripts. Unfortunately, this is not the case: only in 12 out of 36 studies in which the researchers used Praat, details about the used settings were provided [8, 9].

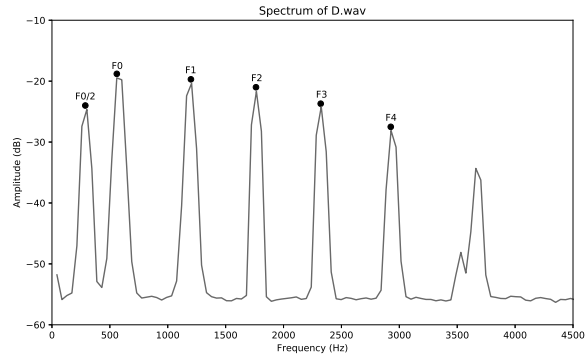


Figure 1: *Spectrum of D.wav, extracted using Audacity. Pitch (F_0), formant (F_1,F_2,F_3,F_4) and periodic noise ($F_0/2$) have been labelled accordingly.*

7. CONCLUSIONS

In this work, we demonstrated the different level of performance that Praat, an open source software designed for speech analysis, can achieve when used with infant cry samples when the parameters are or aren't tuned. In the first part of this work, we generated different acoustic signals with features similar to those of real cry samples. Generated files have been analyzed first by visual inspection, then using Praat standard settings and finally by fine tuning the algorithms' parameters. The performance of the sofware has been evaluated using the Mean Absolute Percentage Error (MAPE). In the second part of this work, we applied the same procedure to a set of real cry utterances. Our results show that Praat standard settings are not suitable for the analysis of cry signal, and therefore the software should not be employed in cry studies without tuning. Researchers have to carefully examine collected data, to ensure that no external sources of periodic noises are recorded within the signals. Furthermore, because of the high inter-individual variability of cry properties, it may be advisable to tune pitch and formant extraction settings according to the investigated participants and their health statuses. We advise researchers of the field to test Praat's parameters with more complex and extreme cry sounds so as to identify the extent to which the software can be correctly integrated in cry studies.

8. REFERENCES

[1] Gary G Berntson, J Thomas Bigger, Dwain L Eckberg, Paul Grossman, Peter G Kaufmann, Marek Malik, Haikady N Nagaraja, Stephen W Porges, J Philip Saul, Peter H Stone, et al., "Heart rate variability: origins, methods, and interpretive caveats," *Psychophysiology*, vol. 34, no. 6, pp. 623–648, 1997.

[2] Jonathan D Reich, Sean Miller, Brenda Brogdon, Jennifer Casatelli, Timothy C Gompf, James C Huhta, and Kevin Sullivan, "The use of pulse oximetry to detect congenital heart disease," *The Journal of pediatrics*, vol. 142, no. 3, pp. 268–272, 2003.

Table 1: *Real and Praat's estimated values for generated acoustic signal properties. For Praat estimation, standard (Praat S.) and optimized (Praat O.) settings were used and values where computed at t=2.5s. In parenthesis are values obtained by visual inspection of the spectrum generated with Audacity version 2.2.1. For each pair of file and settings, the Mean Absolute Percentage Error (MAPE) has been calculated.*

	A.wav			**B.wav**		
F_n	Real	Praat S.	Praat O.	Real	Praat S.	Praat O.
$\mathbf{F_0}$	440.0 (444)	448.2	448.0	440.0 (444)	223.0	447.3
$\mathbf{F_1}$	867.0 (882)	881.6	667.7	873.0 (882)	864.7	777.7
$\mathbf{F_2}$	1339.0 (1384)	1704.9	1367.4	1338.0 (1384)	1688.4	1470.5
$\mathbf{F_3}$	1752.0 (1768)	2655.9	1945.4	1755.0 (1768)	2573.1	2078.3
$\mathbf{F_4}$	2196.0 (2208)	4444.7	2719.0	2190.0 (2208)	3916.1	2753.1
$\mathbf{F_5}$	2641.0 (2756)			2621.0 (2756)		
$\mathbf{F_{0/2}}$	-			213.0 (216)		
MAPE		36.9%	12.3%		40.4%	13.3%

	C.wav			**D.wav**		
F_n	Real	Praat S.	Praat O.	Real	Praat S.	Praat O.
$\mathbf{F_0}$	575.0 (581)	293.4	588.7	575.0 (581)	293.6	293.5
$\mathbf{F_1}$	1166.0 (1131)	932.4	712.1	1169.0 (1190)	1030.9	688.4
$\mathbf{F_2}$	1714.0 (1769)	1176.2	1411.0	1707.0 (1769)	1898.7	1397.6
$\mathbf{F_3}$	2305.0 (2321)	2617.7	2110.4	2292.0 (2321)	2765.6	2105.0
$\mathbf{F_4}$	2889.0 (2942)	3578.0	2880.4	2884.0 (2942)	3695.7	2890.0
$\mathbf{F_5}$	3455.0 (3676)			3433.0 (3676)		
$\mathbf{F_{0/2}}$	-			306.5 (288)		
MAPE		27.6%	13.5%		24.1%	23.3%

[3] Enrico Rosati, Giovanna Chitano, Lucia Dipaola, Claudio De Felice, and Giuseppe Latini, "Indications and limitations for a neonatal pulse oximetry screening of critical congenital heart disease," *Journal of perinatal medicine*, vol. 33, no. 5, pp. 455–457, 2005.

[4] Anna Truzzi, Peipei Setoh, Kazuyuki Shinohara, and Gianluca Esposito, "Physiological responses to dyadic interactions are influenced by neurotypical adults' levels of autistic and empathy traits," *Physiology & behavior*, vol. 165, pp. 7–14, 2016.

[5] Shakila Thangaratinam, Kiritrea Brown, Javier Zamora, Khalid S Khan, and Andrew K Ewer, "Pulse oximetry screening for critical congenital heart defects in asymptomatic newborn babies: a systematic review and meta-analysis," *The Lancet*, vol. 379, no. 9835, pp. 2459–2464, 2012.

[6] Alex R Kemper and Gerard R Martin, "Screening of newborn babies: from blood spot to bedside," *The Lancet*, vol. 379, no. 9835, pp. 2407–2408, 2012.

[7] E Garne, C Stoll, and Mog Clementi, "Evaluation of prenatal diagnosis of congenital heart diseases by ultrasound: experience from 20 european registries," *Ultrasound in Obstetrics and Gynecology: The Official Journal of the International Society of Ultrasound in Obstetrics and Gynecology*, vol. 17, no. 5, pp. 386–391, 2001.

[8] Giulio Gabrieli, Li Ying Ng, Giulia Scapin, Marc H. Bornstein, and Gianluca Esposito, "Acoustic analysis of infants' cry: A review and analysis of available research," *The Journal of the Acoustical Society of America*, Manuscript submitted for publication.

[9] Gianluca Esposito and Gabrieli Giulio, "Replication data for: (acoustic analysis of infants cry: A review and analysis of available research)," Dataset published on DR-NTU (Data), 2019.

[10] Gianluca Esposito, Noboru Hiroi, and Maria Luisa Scattoni, "Cry, baby, cry: Expression of distress as a biomarker and modulator in autism spectrum disorder," *International Journal of Neuropsychopharmacology*, vol. 20, no. 6, pp. 498–503, 2017.

[11] Sebastian Möller and Rainer Schönweiler, "Analysis of infant cries for the early detection of hearing impairment," *Speech Communication*, vol. 28, no. 3, pp. 175–193, 1999.

[12] Kathleen Wermke, Christine Hauser, Gerda Komposch, and Angelika Stellzig, "Spectral analysis of prespeech sounds (spontaneous cries) in infants with unilateral cleft lip and palate (uclp): a pilot study," *The Cleft palate-craniofacial journal*, vol. 39, no. 3, pp. 285–294, 2002.

[13] Heidi Elisabeth Baeck and Marcio Nogueira de Souza, "Longitudinal study of the fundamental frequency of hunger cries along the first 6 months of healthy babies," *Journal of Voice*, vol. 21, no. 5, pp. 551–559, 2007.

[14] Harvey R Gilbert and Michael P Robb, "Vocal fundamental frequency characteristics of infant hunger cries: birth to 12 months," *International journal of pediatric otorhinolaryngology*, vol. 34, no. 3, pp. 237–243, 1996.

[15] Robin Prescott, "Infant cry sound; developmental features," *The Journal of the Acoustical Society of America*, vol. 57, no. 5, pp. 1186–1191, 1975.

Table 2: *Real and Praat's estimated values for generated acoustic signal properties. For Praat estimation, standard (Praat S.) and optimized (Praat O.) settings were used and values where computed at t=2.5s. In parenthesis are values obtained by visual inspection of the spectrum generated with Audacity version 2.2.1. For each pair of file and settings, the Estimated Mean Absolute Percentage Error (Estim. MAPE) has been estimated using as Actual value the peak highlighted by Audacity trough visual inspection.*

	Utterance1.wav			**Utterance2.wav**			**Utterance3.wav**		
F_n	Visual Insp.	Praat S.	Praat O.	Visual Insp.	Praat S.	Praat O.	Visual Insp.	Praat S.	Praat O.
$\mathbf{F_0}$	531	380.5	477.3	450	438.2	434.0	415	270.6	434.7
$\mathbf{F_1}$	904	1019.5	927.1	873	847.1	762.4	716	852.3	752.0
$\mathbf{F_2}$	1343	2078.0	1637.8	1341	1566.0	1350.8	957	1755.9	1325.9
$\mathbf{F_3}$	1797	2832.4	2691.8	1810	2816.8	2494.5	1616	2756.4	2454.6
$\mathbf{F_4}$	2271	3685.5	3288.1	2285	3935.1	3363.8	2313	3874.2	3114.2
Estim. MAPE		43.2%	25.8%		30.0%	20.4%		55.1%	27.0%

[16] William C Sheppard and Harlan L Lane, "Development of the prosodic features of infant vocalizing," *Journal of Speech, Language, and Hearing Research*, vol. 11, no. 1, pp. 94–108, 1968.

[17] Hartmut Rothgänger, "Analysis of the sounds of the child in the first year of age and a comparison to the language," *Early Human Development*, vol. 75, no. 1-2, pp. 55–69, 2003.

[18] Tanja Etz, Henning Reetz, Carla Wegener, and Franz Bahlmann, "Infant cry reliability: acoustic homogeneity of spontaneous cries and pain-induced cries," *Speech Communication*, vol. 58, pp. 91–100, 2014.

[19] Katarina Michelsson, Kenneth Eklund, Paavo Leppänen, and Heikki Lyytinen, "Cry characteristics of 172 healthy 1-to 7-day-old infants," *Folia phoniatrica et logopaedica*, vol. 54, no. 4, pp. 190–200, 2002.

[20] Katarina Michelsson and Oliver Michelsson, "Phonation in the newborn, infant cry," *International journal of pediatric otorhinolaryngology*, vol. 49, pp. S297–S301, 1999.

[21] Linda L LaGasse, A Rebecca Neal, and Barry M Lester, "Assessment of infant cry: acoustic cry analysis and parental perception," *Mental retardation and developmental disabilities research reviews*, vol. 11, no. 1, pp. 83–93, 2005.

[22] W Tecumseh Fitch and Jay Giedd, "Morphology and development of the human vocal tract: A study using magnetic resonance imaging," *The Journal of the Acoustical Society of America*, vol. 106, no. 3, pp. 1511–1522, 1999.

[23] Kathleen Wermke, W Mende, C Manfredi, and P Bruscaglioni, "Developmental aspects of infant's cry melody and formants," *Medical Engineering & Physics*, vol. 24, no. 7-8, pp. 501–514, 2002.

[24] Houri K Vorperian, Ray D Kent, Mary J Lindstrom, Cliff M Kalina, Lindell R Gentry, and Brian S Yandell, "Development of vocal tract length during early childhood: A magnetic resonance imaging study," *The Journal of the Acoustical Society of America*, vol. 117, no. 1, pp. 338–350, 2005.

[25] James F Bosma, "Anatomic and physiologic development of the speech apparatus," *The nervous system*, vol. 3, pp. 469–81, 1975.

[26] C Manfredi, L Bocchi, S Orlandi, L Spaccaterra, and GP Donzelli, "High-resolution cry analysis in preterm newborn infants," *Medical engineering & physics*, vol. 31, no. 5, pp. 528–532, 2009.

[27] Lisa M Unwin, Ildiko Bruz, Murray T Maybery, Victoria Reynolds, Natalie Ciccone, Cheryl Dissanayake, Martha Hickey, and Andrew JO Whitehouse, "Acoustic properties of cries in 12-month old infants at high-risk of autism spectrum disorder," *Journal of autism and developmental disorders*, vol. 47, no. 7, pp. 2108–2119, 2017.

[28] Philip Sanford Zeskind and Ronald G Barr, "Acoustic characteristics of naturally occurring cries of infants with "colic"," *Child development*, vol. 68, no. 3, pp. 394–403, 1997.

[29] Per Runefors, Einar Arnbjörnsson, G Elander, and K Michelsson, "Newborn infants' cry after heel-prick: analysis with sound spectrogram," *Acta Paediatrica*, vol. 89, no. 1, pp. 68–72, 2000.

[30] Alejandro Rosales-Pérez, Carlos A Reyes-García, Jesus A Gonzalez, Orion F Reyes-Galaviz, Hugo Jair Escalante, and Silvia Orlandi, "Classifying infant cry patterns by the genetic selection of a fuzzy model," *Biomedical Signal Processing and Control*, vol. 17, pp. 38–46, 2015.

[31] O Wasz-Höckert, TJ Partanen, V Vuorenkoski, K Michelsson, and E Valanne, "The identification of some specific meanings in infant vocalization," *Experientia*, vol. 20, no. 3, pp. 154–154, 1964.

[32] Paul Boersma and Vincent Van Heuven, "Speak and unspeak with praat," *Glot International*, vol. 5, pp. 341–347, 2001.

[33] Paul Boersma, "Accurate short-term analysis of the fundamental frequency and the harmonics-to-noise ratio of a sampled sound," in *Proceedings of the institute of phonetic sciences*. Amsterdam, 1993, vol. 17, pp. 97–110.

[34] Malcolm J Crocker, Jorge P Arenas, and Rajeev E Dyamannavar, "Identification of noise sources on a residential split-system air-conditioner using sound intensity measurements," *Applied Acoustics*, vol. 65, no. 5, pp. 545–558, 2004.

[35] Tony L Sahley and Frank E Musiek, *Basic Fundamentals in Hearing Science*, Plural Publishing, 2015.

[36] Gianluca Esposito, "Replication data for: (where the baby cries matter: context effect in men and women)," Dataset published on DR-NTU (Data), 2019.

ISOCHRONOUS CONTROL + AUDIO STREAMS FOR ACOUSTIC INTERFACES

Max Neupert

The Center for Haptic Audio Interaction Research
Weimar, Germany
max@chair.audio

Clemens Wegener

The Center for Haptic Audio Interaction Research
Weimar, Germany
clemens@chair.audio

ABSTRACT

An acoustic interface (also: *hybrid controller*) is presented. By tapping, scratching, rubbing, bowing, etc. on the surface, excitation signals for digital resonators (waveguides, lumped models, modal synthesis and sample convolution) are created in synchronicity with augmenting control signals. It is described how a direct acoustic excitation delivers an intimate and intuitive interaction. Questions are raised about which protocols to use for isochronous audio and control transmission as well as file formats. Standardization of such protocols is desirable for future hybrid instruments with analog interfaces. A first step towards standardization is made with the publication of our implementation.

1. INTRODUCTION

Recent developments in the musical instrument controller market follow the demand for more expressive and continuous control. At the same time more computing power allows for expensive synthesis methods so that more parameters can be made use of as a continuous stream of control data in several degrees of freedom.

1.1. Keys or silicone?

A keyboard of the MIDI standard is generally sufficient to generate the parameters for a simple electronic representation of a piano. Mod-wheel and pitch-bend only extended this affordance mildly. For instruments with a continuous articulation like wind and string instruments the single parameter velocity is inadequate. When *Yamaha* came out with the CS-80 in 1977 it pioneered after-touch on every key and laid the foundations for a class of 'extended keyboards' such as the *Haken Continuum* [1], McPherson's *TouchKeys* [2] and the *Seaboard* [3] by *Roli*. All these instruments make multiple parameters per key available continuously. A standardization effort of these parameter streams lead to the MIDI Polyphonic Expression (MPE) specification. Jones' *Soundplane* [4] and Linn's *Linnstrument* likewise belong to this group of instruments but do away with the traditional (and some may say reactionary) piano key layout.

1.2. Exciting audio

A full audio signal is offering even more expression compared to just control-rate parameters. Therefore, contact microphones (piezoelectric sensors) have become a staple of electro-acoustic exploration. They have also found their way in commercial music instruments, but mostly as cheap threshold trigger pads delivering way below their potential. Only a handful of commercially available instruments, namely *Korg's Wavedrum* (1994), Zamborlin's *Mogees* [5] (2014) and the ATV *aframe* [6] (2017) have put them to much more adequate use by feeding the excitation signal into a digital resonator. In the context of research a variety of implementations for experimental and affordable instruments with acoustic interfaces have been proposed. From ceramic tiles as a source for percussive sounds [7], to acrylic sheets instead of guitar strings [8], [9] or intricate prototypes with vibration insulated pads for eight fingers [10].

1.3. Marrying control and exciter

Miller's tiles [7] and Momeni's *Caress* [10] consider the processing of the contact microphone as sufficiently expressive. Cook's *Nukulele* [11] combines two sensors, one at audio rate and one at control rate, to create the affordance of an Ukulele which is played with both hands on different positions of the instrument. As one would with a guitar, a hand controls the parameters while the other provides an excitation signal. Former is the control rate input and latter the audio rate input.

The *Kazumi* by Zayas is an instrument which combines capacitive sensing and piezoelectric microphones on the same surface [12]. It features seven separate faces in a prismatic heptagonal shape. Each of the faces has a copper capacitive sensing layer which divides it into six areas from bottom to tip, combined with a piezo mic underneath.

We want to augment the sound signal with additional parameters, so we simultaneously track the **position** of touch on the surface. This way we make a second hand for generating parameters obsolete. (Figure 1) Our implementation creates a percussive instrument which can be hit, but also can be melodic and played in continuous gestures by rubbing, scratching, or bowing on its edge.

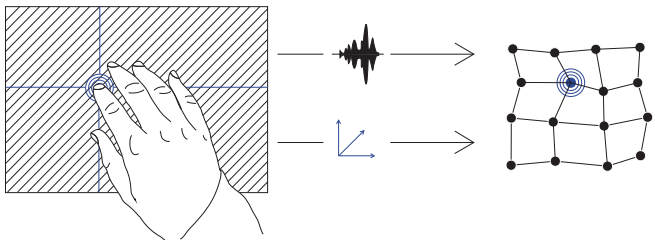


Figure 1: *Hybridity of audio and control data*

1.4. Instrument versus controller

Great effort has been put into abstracting controller hardware to become universal input devices for software instruments. The generic controller is an interface to change parameters on the synthesizer in which the actual sound is generated. In our instrument it's not so easy to define where the controller ends, and the instrument starts. Cook writes that "...many of the striking lessons from our history of intimate expressive musical instruments lie in the blurred boundaries between player, controller, and sound producing object." [11].

In our instrument we are using the actual audio signal from the surface which then is fed into a digital filter on the computer. In effect, a significant component of the final sound is defined by the spectrum and gesture of the excitation signal. While in the literature the term 'hybrid controller' is found [9] we prefer to describe the *Tickle* as an 'acoustic interface'. In our opinion 'hybridity' is too generic and there is no declaration of its components, while 'acoustic interface' adds clarity to its nature.

2. THE TICKLE

The following section describes the components of the instrument.

2.1. Hardware

The case is made of bent steel with wooden side panels. Its top surface is a printed circuit board and has a capacitive touchpad, three endless rotary encoders with associated RGB LED and up/down buttons (for transposition or other parameters). On the back are six ports:

1. External in (if plugged-in it mutes the built-in sensor)
2. CV out Y axis (0-4 V)
3. CV out X axis (0-4 V) or note
4. Host (micro-USB port)
5. Gate or envelope (0-5 V)
6. Excitation (audio signal)

2.2. Surface

After a brief evaluation of piano key layouts and variations thereof [13] it was concluded, that a piano key layout is contradictory to the intended interaction with the instrument. A hexagon pattern was chosen to have equal distanced and sized[1] segmentation without empty spaces on the surface. It is also found in other electronic instruments and controllers, for example, the *Synderphonics Manta* [14]. From the 8-Bit resolution in X and Y axis we can calculate in which of the 14 hexagons printed on the surface a touch occurred. The capacitive touch sensing is single-touch, so polyphony cannot be achieved by simultaneous touches. A two or more point gesture will produce erroneous ghosting touch points and thus needs to be avoided while playing. However, with voice allocation we can let one touch resonate while a new touch gets its own resonator, so subsequent touch events may have overlapping resonances.

2.3. Material and Texture

To create an acoustic excitation signal we rely on a hard material that captures the spectra of different gestures. In addition to the rigidity of the material, a textured surface is essential to create enough noise when rubbed and wiped. Silicone surfaces are not suitable for our application since they absorb too much of the subtle interaction.

2.4. Residual and Resonance

Generally, we want the physical surface of the instrument to resonate as little as possible, so that we can feed the dry residual signal of the touch gesture (rub, scratch, hit, flick, bow etc.) as excitation signal into a digital resonator (See also [7]). This way the full power of physical modeling synthesis algorithms may be accessed. The practice of sending generated noise-bursts or clicks into digital resonators which can be found in literature for physical modeling and which is still the standard in many soft- and hardware implementations is crippling the true potential of such algorithms.

[1] except for the hexagons at the edges

2.5. Synthesis

For the sound synthesis we employ techniques of digital reverbrators which at their heart are delay lines, feedback and filters. They can be understood as modeled simulations (waveguides and mass-spring models) of the physics happening in real instruments as described by Smith [15]. These models can be generated with Berdahl and Smith's *Synth-A-Modeler* compiler [16] which has received a graphical interface with Vasil's *SaM-Designer* [17]. *Synth-A-Modeler* generates *Faust* code which can be compiled in a variety of other formats such as a *Pure Data* external. With the *Pure Data* object `pmpd~` from Henry's PMPD [18] library which creates static mass and spring models, we achieved nice sounding string, plate, and gong topologies. However, we are not aiming for perfect recreations of classic instruments, our interest lies in the exploration of synthetic sounds with an acoustic and intimate level of control. Algorithms such as the nested comb filter delay as described by Ahn and Dudas [19] prove interesting and fun to interpret with our instrument while being surprisingly cheap to compute. We can employ our acoustic interface to excite *extended, hybrid* and *abstract cyberinstruments* as described by Kojs et al. [20]. Convolution methods with samples can be useful to digital Foley artists to articulate a sample in a plenitude of variations.

Figure 2: *The Tickle instrument*

2.6. Software Architecture and Code

Our hardware is based on a Cypress PSoC 5 microprocessor and runs a firmware which is digitizing the capacitive sensing surface and the signal from the piezoelectric sensor. It communicates to a custom kernel driver which is then communicating to user-space software like our *Pure Data* external or a VST-plugin. Our kernel driver for

Linux as well as the *Pure Data* external are published under a free license. A repository of the source[2] is available (mirrored on github[3]).

2.7. Drivers and Communication

A great challenge was to transmit control rate signals married to a stream of audio with a stable latency and reliable offset to each other. The capacitive sensing reports every 4 ms a position while the audio streams with a sample rate of 48 kHz and a block size of 64 samples. Currently the user-space software is expected to match these settings to work reliably. We wrote our own Linux kernel driver receiving this isochronous stream of control and audio rate signals via USB from the device.

3. STANDARDS FOR TRANSMISSION AND STORAGE

We believe that acoustic interfaces will soon become a category of their own and manufacturers will introduce hybrid controllers to the market. To make these new devices work with synthesis software there will have to be a standardization effort for interoperability. McMillen and Thew published a proposal on how to send sound spectrum information over MIDI and OSC [21]. However many questions are yet to be answered about which format and standard should be used for audio and data. A plethora of further questions arise when thinking about a possible integration of a track with control and audio-as-synthesis-source into a DAW. With this publication and the open source driver we wish to start a discussion about possible open standards for transmission, storage, and integration of analogue interfaces into the creative workflow of musicians.

3.1. Specifications for the Driver

Our aim is an isochronous transfer of data and audio rate signals with minimal latency, and more importantly, with little jitter [22]. The touch position data needs to be present before the audio arrives to be able to tune the synthesis. There can't be any variation to the offset between signal and data. The audio stream doesn't need to be continuous; it could start on the touch event and end with it. In a future polyphonic version, several audio streams could exist in parallel. The implementation could be a data protocol with (multichannel-) audio streaming segments on demand, as well as an continuous audio stream with additional data interwoven. The touch events should refer to a specific sample in the audio, possibly with a timestamp. Other interface data like extra knobs, faders, potentiometers or rotary encoders don't need this precision in timing.

3.2. Surveyed Communication Protocols

We've considered different established and experimental protocols. Each was evaluated against the aforementioned goals.

1. A **kernel module** driver was our choice, as it gives us the maximum amount of control to make sure it meets our criteria. However, it needs an installation procedure. On *Windows* and *Mac OS* the operating system vendor restricts who can distribute kernel modules, in fact we have paid *Apple* and applied for kernel signing and are still waiting for any response after 5 months. On Linux, Secure Boot needs to be deactivated or the kernel extension manually signed. A custom kernel driver means additional development overhead and for the customer the fear that the device will be rendered useless if support ends.
2. **Audio spectrum data** (via MIDI or OSC). Another approach would be to break down the audio into metadata and then send this over established protocols like MIDI or OSC which would allow for a partial reconstruction. This was proposed in the aforementioned draft by McMillen [21]. We dismissed this approach because we see it as necessary to include a full audio stream to reduce the latency required for the analysis of such descriptive meta information. It also creates a computational overhead on both, the sending and receiving device.
3. **Audio and MIDI Class Compliant** drivers are a viable alternative. It's possible to use one USB connection providing two virtual devices, an audio interface, and a HID or a MIDI device. Using standards means compatibility, no driver installs and continuous support. However, it's not guaranteed that latency and offset will be consistent. Another problem lies in limitations of popular proprietary DAWs like *Ableton Live*, which will only allow the use of one sound card at a time. Assuming that the sound synthesis happens in a plugin of the DAW, this restriction would block the plugin to access the audio device.
4. **Control Data as Audio Signal.** Control data may be sent as signals at audio rate, not unlike control voltage in synthesizers or upsampled sensor output in Wessel's *Slabs*, which features 96 channels of audio [23]. It could also be encoded as frequencies and later be decoded with a Fourier transformation like the *Nuance* as described in Michon et al. [24].
5. **MIDI 2.0** There is no indication that MIDI 2.0, which is currently in prototyping stage at the *MIDI Manufacturers Association*, will include the feature to send audio streams for acoustic interfaces.

This list claims no completeness, for example we have not surveyed protocols like *Ultranet* or AVB. It's likely we have overlooked something and there may be a sensible solution to our problem already available.

4. FUTURE WORK

Future research may be conducted to implement the following features to the instrument: 1. **Multi-touch** to relieve from ghosting issues when two fingers touch the surface simultaneously. It also allows for **polyphony** later on. 2. **Pressure sensing** [25] either for every point or at least globally for the whole surface. 3. **Haptic feedback** is challenging to implement due to the feedback into the sensor, but can give the user a much more intense sense of reality. The *Lofelt Basslet*[26] is a good demonstration of such a device. 4. Integrated sound synthesis either implemented by a) **analog circuitry** or b) an **embedded computing** platform, for example the *Bela* board [27]. 5. **Playful interfaces** to manipulate mass-spring models in real-time as seen in Allen's *Ruratae* [28].

[2] Source code: https://gitlab.chair.audio/explore/projects
[3] Github mirror: https://github.com/chairaudio

Figure 3: *One of the more unconventional and unintended ways to play the Tickle*

5. CONCLUSIONS

Our instrument *Tickle* combines several well-known techniques and technologies which on their own are not new. Touch pad, contact microphone, and physical modeling synthesis have been around for decades. However, in their combination they synergize to a powerful intuitive instrument which allows for a natural and intimate [29] interaction with precise and reproducible control over sound. Feeding an analogue excitation signal into a (digital) resonator can create familiar as well as alien sounds. Sounds which either behave like instruments we know: Violin, guitar, snare drum, cymbal, gong, marimba, etc., or sounds which are distinctly synthetic but have an analogue touch to it.[4]

With this paper we hope to have shown the necessity of sample accurate, low latency and jitter free communications for acoustic interfaces and started a discussion on how to achieve it.

6. ACKNOWLEDGMENTS

Our gratitude belongs to Philipp Schmalfuß for coming up with new synthesis techniques as well as Björn Kessler, Joachim Goßmann, Richard Dudas, Sven König for valuable input and feedback. Extra thanks to Brian Bixby for the copy editing and the Sündermann-Oeft family for hosting us during LAC19. We also would like to thank the following institutions: The ESF of the European Union which had funded us for one year through the EXIST! program, as well as *Bauhaus-Universität Weimar* for access to workshops and network.

[4] A playlist of video demonstrations with the instrument can be found on our website https://chair.audio

7. REFERENCES

[1] Lippold Haken, Ed Tellman, and Patrick Wolfe, "An Indiscrete Music Keyboard," *Computer Music Journal*, vol. 22, no. 1, pp. 30, 1998.

[2] Andrew McPherson, "TouchKeys: Capacitive Multi-touch Sensing on a Physical Keyboard," in *Proceedings of the 2012 International Conference on New Interfaces for Musical Expression*, Michigan, 2012.

[3] Roland O. Lamb, *The Seaboard: discreteness and continuity in musical interface design*, Dissertation, Royal College of Art, London, June 2014.

[4] Randall Evan Jones, "Intimate Control for Physical Modeling Synthesis," M.S. thesis, University of Victoria, Victoria, 1993.

[5] Bruno Zamborlin, *Studies on customisation-driven digital music instruments*, Dissertation, Goldsmiths, University of London and Université Pierre et Marie Curie - Paris 6, London, Oct. 2014.

[6] "Product Development Story: ATV aFrame – A Next-Generation Electric Instrument Created through a Fusion of an Acoustic Instrument, DSP Technologies, and Traditional Japanese Craftsmanship," *ICON*, Sept. 2016.

[7] Miller Puckette, "Infuriating Nonlinear Reverberator," in *Proceedings of the 2011 International Computer Music Conference*, Huddersfield, 2011, p. 4, International Computer Music Association.

[8] Daniel Schlessinger and Julius O. Smith III, "The Kalichord: A Physically Modeled Electro-Acoustic Plucked String Instrument," in *Proceedings of the 2009 International Conference on New Interfaces for Musical Expression*, Pittsburgh, June 2009, p. 4.

[9] Romain Michon and Julius O. Smith III, "A Hybrid Guitar Physical Model Controller: The BladeAxe," in *Proceedings of the 2014 International Computer Music Conference*, A. Georgaki and G. Kouroupetroglou, Eds., Athens, 2014, p. 7, International Computer Music Association.

[10] Ali Momeni, "Caress: An Enactive Electro-acoustic Percussive Instrumentfor Caressing Sound," in *Proceedings of the 2015 International Conference on New Interfaces for Musical Expression*, Baton Rouge, May 2015, NIME '15, pp. 245–250.

[11] Perry R. Cook, "Remutualizing the Musical Instrument: Co-Design of Synthesis Algorithms and Controllers," in *Proceedings of the SMAC 2003*, Stockholm, Aug. 2003, p. 4.

[12] Charles Gantt, "Design Challenge Project Summary: Kazumi," Apr. 2016.

[13] Max Neupert, "Optimizing chromatic Keyboards for small, non-tactile Surfaces," in *Proceedings of Korean Electro-Acoustic Music Society's 2017 Annual Conference*, Gwangju, Oct. 2017, pp. 29–31.

[14] Jeff Snyder, "Snyderphonics Manta Controller, a Novel USB Touch-Controller.," in *Proceedings of the 2011 International Conference on New Interfaces for Musical Expression*, Oslo, 2011, pp. 413–416.

[15] Julius O. Smith III, "Physical Modeling Synthesis Update," *Computer Music Journal*, vol. 20, no. 2, pp. 44, 1996.

[16] Edgar Berdahl and Julius O. Smith III, "An Introduction to the Synth-A-Modeler Compiler: Modular and Open-Source Sound Synthesis using Physical Models," in *Proceedings of the 2012 Linux Audio Conference*, Stanford, Apr. 2012, pp. 93–99, CCRMA, Stanford University.

[17] Edgar Berdahl, Peter Vasil, and Andrew Pfalz, "Automatic Visualization and Graphical Editing of Virtual Modeling Networks for the Open-Source Synth-A-Modeler Compiler," in *Haptics: Perception, Devices, Control, and Applications*, Fernando Bello, Hiroyuki Kajimoto, and Yon Visell, Eds., Cham, 2016, pp. 490–500, Springer International Publishing.

[18] Cyrille Henry, "Physical modeling for pure data (PMPD) and real time interaction with an audio synthesis," in *Proceedings of Sound and Music Computing 2004*, Paris, Oct. 2004.

[19] Jae-Hyun Ahn and Richard Dudas, "Musical Applications of Nested Comb Filters for Inharmonic Resonator Effects," in *Proceedings of the 2013 International Computer Music Conference*, Perth, 2013, vol. 2013, pp. 226–231, International Computer Music Association.

[20] Juraj Kojs, Stefania Serafin, and Chris Chafe, "Cyberinstruments via Physical Modeling Synthesis: Compositional Applications," *Leonardo Music Journal*, vol. 17, pp. 61–66, Dec. 2007.

[21] Keith McMillen and Barry Threw, "Acoustic Instrument Message Specification," June 2014.

[22] Robert H. Jack, Tony Stockman, and Andrew McPherson, "Effect of latency on performer interaction and subjective quality assessment of a digital musical instrument," in *Proceedings of the Audio Mostly 2016 on - AM '16*, Norrkoping, Sweden, 2016, pp. 116–123, ACM Press.

[23] Adrian Freed, "David Wessel's Slabs: a case study in Preventative Digital Musical Instrument Conservation," in *Proceedings of Sound and Music Computing 2016*, Hamburg, Aug. 2016, p. 5.

[24] Romain Michon, Julius O. Smith III, Chris Chafe, Matthew Wright, and Ge Wang, "Nuance: Adding Multi-Touch Force Detection to the iPad," in *Proceedings of Sound and Music Computing 2016*, Hamburg, Aug. 2016, p. 5.

[25] Adam R Tindale, Ajay Kapur, George Tzanetakis, Peter Driessen, and Andrew Schloss, "A Comparison of Sensor Strategies for Capturing Percussive Gestures," in *Proceedings of the 2005 International Conference on New Interfaces for Musical Expression*, Vancouver, 2005, vol. 2005, p. 4.

[26] Amir Berrezag, "US Patent 2017/0019008 A1: Vibrating Actuator," Jan. 2017.

[27] Andrew McPherson and Victor Zappi, "An Environment for Submillisecond-Latency Audio and Sensor Processing on BeagleBone Black," in *Audio Engineering Society Convention 138*, Warsaw, May 2015.

[28] Andrew S. Allen, *Ruratae: A physics-based audio engine*, Dissertation, University of California, San Diego, San Diego, 2014.

[29] David Wessel and Matthew Wright, "Problems and Prospects for Intimate Musical Control of Computers," *Computer Music Journal*, vol. 26, no. 3, pp. 11–22, 2002.

GAME|LAN:CO-DESIGNING AND CO-CREATING AN ORCHESTRA OF DIGITAL MUSICAL INSTRUMENTS WITHIN THE FAB LAB NETWORK

Alexandros Kontogeorgakopoulos

Cardiff Metropolitan University
Cardiff School of Art and Design
Cardiff, UK
akontogeorgakopoulos@cardiffmet.ac.uk

Olivia Kotsifa

Decode Fab Lab
Athens, GR
olivia@decodefablab.com

ABSTRACT

This paper presents an ongoing project focused on the co-design and co-creation of a small orchestra of digitally fabricated digital musical instruments (DMIs) based on the Bela board, an open-source embedded computing platform. The project took place in Fab Labs, an international network of digital fabrication laboratories[1]. The orchestra, named *Game|Lan*, is inspired by the traditional Indonesian Gamelan ensembles, their music and philosophy. The project aims to explore the capabilities of the Fab Lab network which runs on an open-access, open-source and open-hardware ethos, for a distributed project of this type. The aspiration is to create an original orchestra for non-musicians, which offers the rich collective experience of being in a music group and explore it as a medium for social interaction. This paper presents the first results of the research project which took place is three Fab Labs in South America and it focuses on the process and the development of the project.

1. INTRODUCTION

In the last two decades, a large number of digital musical instruments have been developed by the sound and music computing community [1],[2]. The international conference for New Interfaces for Musical Expression [2], annually hosts numerous music technology research projects related to musical expression and to digital luthiers. However, very few projects are designed and made by participatory methods and techniques. The Input Devices and Music Interaction Laboratory at McGill University has co-developed the McGill Digital Orchestra which involved collaboration between researchers, composers and performers. More recently, the Augmented Instruments Laboratory at Queen Mary University of London, has started developing a research trajectory related to participatory design and co-design of digital musical instruments. [3],[4].

This paper presents the process of development of a digital musical instrument with participatory design and creation methods: brainstorming sessions, workshops, hands-on experimentation etc. Different approach has been adopted for each stage of the project depending on the resources and research area of each Lab. Focus was given equally to the physical body of the instrument as well as its electronic and digital component where an embedded computing platform for low-latency audio was used and programmed. The sound synthesis algorithms have been designed and developed as an iterative process; it was not possible to employ true participatory techniques in this case as the participants had no necessary experience or necessary skills in music signal processing.

[1] https://fablabs.io/labs/map
[2] http://www.nime.org/

The first section of the paper gives and overview of the open design, co-design and co-creation culture and the Fab Lab network. Section two presents the concept behind this project and outlines the basic idea behind the orchestra, the requirements and constraint of the approach. Finally, section three focus on the design and the making of the instrument during the residencies that the authors had in three Fab Labs in South America.

2. CO-DESIGN AND DIGITAL FABRICATION

2.1. Fab Labs

In the recent years the maker movement has started emerging, in part because of people's need to engage passionately with objects in ways that make them more than consumers [5],[6]. Particularly the Digital Fabrication Laboratories, so called Fab Labs, form part of a larger "maker movement" of high-tech do-it-yourselfers, who are democratising access to the modern means to make things [7],[8].

Fab Labs are often seen as open-innovation contexts in which lead users can develop innovation that may become commercial solutions from which companies can profit. But they may also be seen as platforms for broader participation and new ways of collaborative engagement in design and innovation, pointing at alternative forms of user-driven production [9].

The reason why Fab Labs were chosen over other type of makerspaces is the fact that the philosophy of the Fab Lab Network is the collaboration between its Labs. The fact that each Fab Lab has to share same machines and processes allows for information, projects and people to move freely between them. Also, fabricating the instrument with the principles and practices of a Fab Lab means that anyone can download the open designs, customise them if they need to and fabricate them in any Fab Lab around the world.

2.2. Co-Design and Co-Creation

Co-design is being used as an umbrella term for participatory design and collaborative design. Participatory Design, seen as design of Things, has its roots in the movements toward democratisation of work places in the Scandinavian countries. In the 1970s participation and joint decision-making became important factors in relation to workplaces and the introduction of new technology [10]. Co-design breaks the rules between the traditional designer-client relationship and allows for creative contribution to design decisions. Without excluding the designers in the process, it recognises the important role of the users' participation in the design decisions, as experience experts. This research uses the method of participatory design, a human centered design approach that attempts to involve users and experts to assist in the design process in order to ensure the usability

of the product design[11]. The authors have applied and adapted the Participatory Design methods in the Fab Lab environment depending on each user group. Participatory research methods[12], [13] that involve hands on processes and Fab Lab principles both take the same approach of testing feasibility in all stages of work. The authors followed the five stage design thinking model proposed by the Hasso-Plattner Institute of Design at Stanford (d.school). The research was therefore conducted in 5 steps: empathise, define, ideate, prototype, test [3] [4]. For the first two steps a mind map was drawn on a whiteboard, as qualitative data collection tool for generating ideas.

3. CONCEPT

The concept of the project was to co-design and co-fabricate locally a series of elegant and simple to use embedded digital musical instruments for non-musicians. The aim is to create a small orchestra similar to the philosophy of the Gamelan Orchestra [14] and explore it as a medium for social interaction. The percussion-type instruments would be plug-and-play and easy to perform creatively without necessarily any musical background. It is worth mentioning that most Gamelan ensembles, especially in the UK, allow people of all ages and abilities to take part. Both authors of the paper were part of the Cardiff Gamelan ensemble and found very inspiring this fact which eventually constituted one of the main reason to approach the Game|Lan project orchestra in a similar way [5].

A very important aspect of the project was its participatory character and ethos. The instruments had to be co-designed and co-created locally, in Fab Labs. Each Fab Lab with its particular focus, skills and expertise, would contribute to the project accordingly. The authors planned to visit three to four Fab Labs in South America and work for a short period of approximately one week with the makers, engineers, entrepreneurs and designers in their premises.

Moreover, it is worth mentioning that this is a mobile project and follows the authors' idea of "how to make almost anything while travelling". The authors wanted to test how feasible is to do creative work while travelling, following a digital nomads lifestyle [6]. Every single destination would serve as a source of inspiration and every Lab would contribute uniquely to the realisation of the project. Ideally each Lab would develop its own instrument, aligned to its local culture and geographical location. This idea was proven to be too ambitious for the time spent in each Lab and although many prototypes were fabricated in each place, one final instrument was produced at the very end of the trip.

Material and technical-wise, the project had to be digitally fabricated, with open design files and with the machines and technologies shared within the Fab Lab network: 3D printers, CNC machines, laser cutters, high resolution milling machines for printed circuit board milling, electronics and microprocessors. Since the majority of the Fab Labs do not focus on DMIs, the authors had to provide the necessary embedded computing platforms for the development of low-latency audio applications. For that reason, the Bela board has been chosen, an open-source embedded computing platform and Pure Data visual programming language [15]. Other alternative platforms more widespread in the Fab Lab community such as the Arduino with the ATMega328 chip or the ATtiny microcontroller were not appropriate even with extra boards to support audio input and output. The Raspberry Pi could be an alternative but it would also need other peripherals [16].

For the sound creation component of the instrument, the intention was to design and develop a simple sound synthesis system, which would generate timbres and sequenced music material that would be mapped intuitively to the physical interface. Since the performers wouldn't be musicians it was important to make it easy to them to create quite rich musical output with simple gestures.

4. PROCESS

The co-design and co-fabrication sessions of the project were carried out in three Fab Labs in South America: The Fab Lab in the University of Chile in Santiago, the Fab Lab Lima in Peru and the Fab Lab of the National University of Colombia Medellin. It is worth noting that these three sessions, were very different in nature and approach. Furthermore, the participants were not researchers from the DMI community nor were they professional instrument players or digital luthiers, but mainly active members of the Fab Lab network and the Maker movement. That was not necessary a complication in the co-creation process since the instrument addressed this type of performers. Below it is presented chronologically how each Fab Lab contributed to the project and how the authors approached the collaboration with the teams in each location.

Fab Lab - University of Chile

Fab Lab U. de Chile [7]is housed in the Engineering School of Universidad de Chile in Santiago. The Fab Lab quickly embraced the Game|Lan project idea and invited us to work with three of their core team, to discuss our ideas on the physical and digital interaction, form, fabrication method and electronic design.

After having presented the idea and discussed the available resources, the authors collected the information from the mind-maps and started drawing out all important points as discussed with the team onto a whiteboard (see figure 1).The points proved to be our compass for agreeing on a good size, form and interaction; decisions that were made collectively. The figure below shows how the team defined some parameters that would be followed throughout the project. It was equally important to embed the Fab Lab ethos into the project, the mobile nature of the instrument, the electronics restrictions, the aesthetics and the Gamelan philosophy.

Further to the research and decisions taken by the team, the instrument had an approximate size of 250x150x150mm with an enclosure that would fit the microcontroller, battery and sensors. The first prototype was done on day three and from there on, we could easily test the interaction. The decision taken was that different faces would allow for a certain tilting of the instrument which would work well with the physical and digital interaction.

The physical structure of the musical instrument embedded sensors, very simple signal conditioning circuits and a small single-board computer for audio and sensor signal processing. The Fab Lab community commonly uses the Arduino board or directly its ATMEGA single-chip microcontroller which unfortunately does not allow on board audio processing. As mentioned on section 3, the Bela board has been chosen for its audio specifications and because it is very well integrated with Pure Data, a very well known open-source programming language for computer music applications that

[3] http://www.nime.org/

[4] https://www.interaction-design.org/literature/article/5-stages-in-the-design-thinking-process

[5] http://artsactive.org.uk/2018/02/09/cardiff-gamelan-community-group/

[6] https://nomadlist.com/

[7] http://www.fablab.uchile.cl/

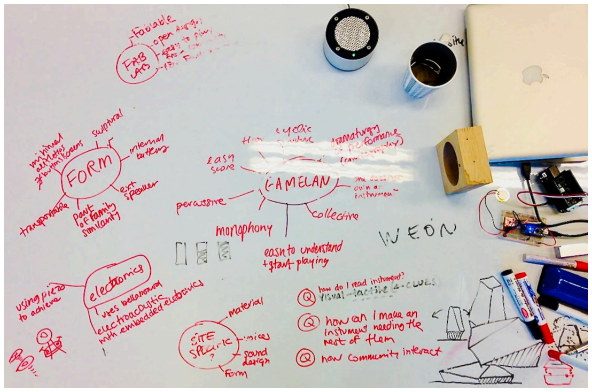


Figure 1: *Points to be considered during design decisions*

is aligned with the open-source philosophy of the Fab Lab community.

In our prototypes in Santiago, the team used a two-axis accelerometer, a piezoelectric sensor and three reed switches. An algorithm was developed in order to detect the active face of the polyhedron from the readings of the accelerometer and accordingly influence the signal processing algorithms. The piezoelectric sensor was measuring pressure on the faces of the instrument which was used either as an audio input or as trigger of samples. The reed switches and the three magnets acted as a 3-bit digital input signal that affected the settings of the instrument. All these electronic components were soldered on a perforated board. An electronic engineer from the local team helped with the electronic development and started programming for the first time in Pure Data.

One of the concepts in Santiago that the team developed, was to have an ensemble of maximum eight reconfigurable, modular and interchangeable instruments. During the music performance, the players would mix the top with the bottom parts of their instrument in order to increase the dramaturgy and the physicality of the performance. This gesture would change the settings of the instrument such as the timbre family or the sequenced music patterns triggered by the performers. The reed sensors mentioned above where used for that reason.

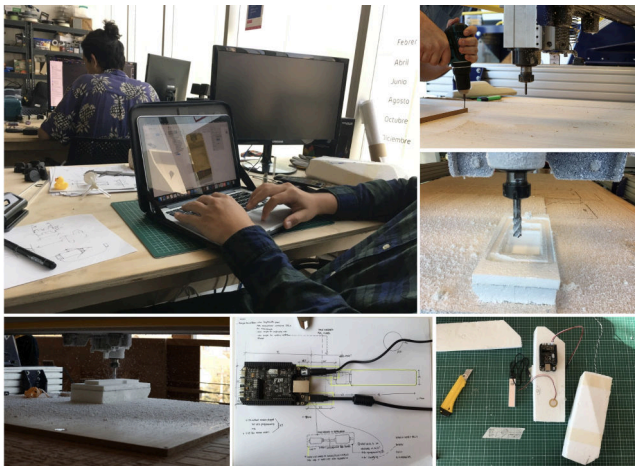


Figure 2: *Co-design and prototyping in Fab Lab U.de Chile*

In the first prototype, the instrument was sampled-based, playing back randomly a collection of samples coming from the same family of sounds. That was enough in order to test the interaction design and study how feasible was for the performers to play the instrument together. A simple score system was devised , similar to the Gamelan Kepatihan notion, where the number would indicate the face to be slapped. The first author was part of the Gamelan orchestra in Cardiff in UK for five years and he was aware of the level of difficulty of performing music with this type of notation. As already mentioned before, one of the main reason why the Gamelan philosophy was adopted for this project was the quick access the beginner performers have, to play notated music within the context of an orchestra. The score was briefly tested with non-musicians in Santiago and was confirmed that learning curve is very smooth and beginners could easily engage with that type of orchestra. More information on the process can be found on the authors' website [8]

Fab Lab - Fab Lab Lima

Fab Lab Lima[9] is a community Fab Lab therefore rather than working with the Fab Lab team, we organised a workshop open to the public with knowledge in either a design related field or electronics, programming or fabrication. We spent two days with a multidisciplinary group of participants with diverge backgrounds ranging from architecture to mathematics, biology, art, electrical engineering, civil engineering as well as members of the community interested in the project. Each one chose to contribute to one of the three areas of interest as designed by the authors: instrument form and design, electronics and programming and 3d prototyping in collaboration with the design group. During the time in Fab Lab Lima the authors repeated the last 3 stages: ideate, prototype, test.

On the second day of the workshop we experimented with different materials and processes as textiles and weaving, parametrically designed forms and 3d printing etc. Moreover, the electronics were further developed and a PCB board was designed according to the circuit developed in Santiago, Chile. More information on the process can be found on the authors' website [10]

The rest of the time we worked in the Lab refining the interaction design and programming it in Pure Data. Different sound synthesis algorithms where programmed there and presented to the participants. One interesting one, passed the audio signal coming directly from the piezoelectric sensor to a bank of parallel band-pass filters. The central frequency and the Q factor of the filters was mapped to the orientation of the body of the instrument and the performers by tilting it could generate a variety of unexpected sonic textures such as rain drops.

Fab Lab - National University of Colombia Medellin

Fab Lab UNAL [11] is in Medellin, in the Arts and Architecture School of the National University of Colombia. During our week in the Fab Lab, we worked with the Lab's team to co-design a parametric [12]shape for the instrument and fabricate the result in wood. Parametric design and CNC milling was this Lab's strongest asset so we experimented with both.

[8] https://www.stiwdioeverywhere.com/2018/04/20/making-in-fab-lab-u-de-chile/

[9] https://www.fablabs.io/labs/fablablima

[10] https://stiwdioeverywhere.com/2018/05/09/making-in-fab-lab-lima/

[11] https://www.fablabs.io/labs/fablabUNmedellin

[12] https://www.grasshopper3d.com/

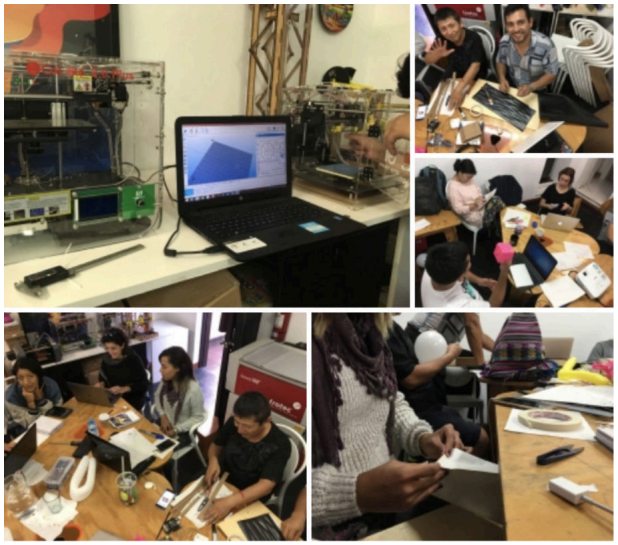


Figure 3: *Co-design and prototyping in Fab Lab Lima*

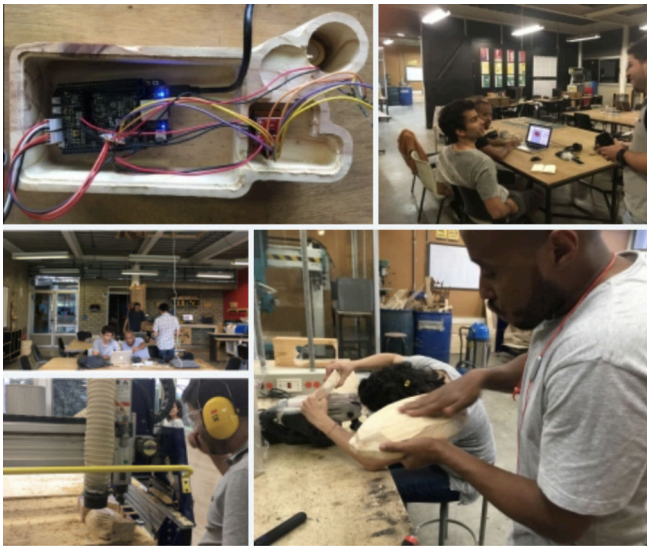


Figure 4: *Making in Medellin*

The team in Colombia had a particular interest in the digital fabrication aspect of the project, testing different types of wood for the end result. Oak, eucalyptus and pine were available to use at the Lab, and after testing the weight, acoustical properties and the milling bits to be used in each case, the team decided to use pine for the two-part instrument as illustrated in figure 5. We made three prototypes out of pine wood to test the size, ergonomics and wood texture and acoustics. The authors decided to repeat stages 3,4 and 5 of the methodology: ideation, prototype and testing. Without a major change in the ergonomics of the instrument, the final result was slightly bigger than the size agreed in Fab Lab U. de Chile, simply because the geometry generated by the algorithm was more complicated. The bottom part enclosed the electronics circuits and had 6 main faces that were used to produce different sounds depending on which angle the performer would decide to tilt it at. More information on the process can be found on the authors' website [13]

The circuit diagram and PCB layout for through-hole components designed in Peru was given to the team for milling. Unfortunately due to software implications, the drivers of the milling machine were not working and there was no alternative way of producing the board with a process used by the Fab Lab community. The widely known etching technique is not supported by the Fab Lab network which is focused to more computer-aided-manufacturing approaches.

For the sound generation part of the instrument, a different approach closer to algorithmic composition has been explored and produced higher lever of musical material. A number of short musical phrases were composed or generated algorithmically, which could be repeated and triggered interactively by the performers. Each face of the polyhedron triggered a different phrase randomly or in a predefined order. Musical parameters of the phrase such as its tempo and dynamics were mapped to the orientation of body. The performers could articulate the phrases, control how many times they are repeated and when they will start playing. This procedure was inspired by *In C* by *Terry Riley*.

[13] https://stiwdioeverywhere.com/2018/05/21/making-in-fab-lab-unal-medellin/

Figure 5: *Two part CNC milled prototype in Medellin*

5. CONCLUSIONS

The GamelLan project was an interesting experiment, trying to match the participatory approach in design and fabrication with the culture of the digital nomads.The different teams have managed to develop one finalised instrument and equally importantly to share knowledge, skills and ideas beyond their cultural barriers. The authors were flexible and worked with each Lab in a different way, respecting the diversity within the Fab Lab network. Unfortunately, there was no time left to experiment musically or perform with the instrument. Upon reflection, there are a few areas for improvement and points to consider for others who decide to do a similar project:

1. It was not an easy task to accomplish especially while travelling. The authors spent 8 days working in Fab Lab U.de Chile and managed to go through all stages of the design. In the other two locations they had to spend less time.
2. An ambitious project that would normally take a certain of amount of time in one's local Fab Lab, may take up to three times more time in other places especially when one is not familiar with the local settings. This does not apply for smaller projects or projects in collaboration with university students.

3. The Fab Labs' website that shows the location, machines and activity of each Fab Lab in the world needs an update: not all places were active or had the equipment needed and this cut the project short.

Despite the points above, the authors managed to gather an important body of knowledge related to the project, a series of alternative design ideas fabrication methods. The important points highlighted during the first days of the project in Fab lab U. de Chile set the rules, the design values to be followed. This part proved to be vital to the project, not only during the first week in Chile, but throughout the whole duration of the project. The participants whether this was in Colombia or Peru, understood and respected the decisions that were taken collectively by the first team in Chile. It was difficult for the participants to make sure they would address all the points when co-designing and prototyping the instruments in each place, however they happily accepted the challenge. There were always points where new decisions were discussed and tested; this gave a sense of empowerment and ownership in each place.

The overall challenge of co-creation, especially when not all participants have collaborated before, may delay the final result. However, each person's knowledge, ideas, or experiences added significant value to the project. Co-creation in spaces like the Fab Labs seems to come naturally by its members and the authors are optimistic that there will be more examples in the future.

This is work in progress; future work includes improved, longer in duration workshops where one instrument per location will be fabricated. All designs and music scores are to be uploaded on a web-based hosting service for version control such as GitHub so they are accessible to the community and step by step instructions and documentation of the fabrication are to be shared on the authors' website. Moreover a series of concerts are envisaged that could take place remotely as network performances or in the International Fab Lab conferences.

6. ACKNOWLEDGEMENTS

We would like to thank the Fab Lat Network for facilitating our communication with the regional Fab Labs. Special thanks to the manager and team in Fab Lab U.de Chile Danisa Peric, Gonzalo Olave, Joaquin Rosas, the manager of Fab Lab Lima Beno Juarez and the participants of the Fab Lab Lima workshop, and the Fab Lab UNAL manager and team Juan Gutierrez and Dani Asprilla.

7. REFERENCES

[1] Eduardo Reck Miranda and Marcelo M. Wanderley, *New digital musical instruments: control and interaction beyond the keyboard*, vol. 21, AR Editions, Inc., 2006.

[2] Alexander Refsum Jensenius and Michael J. Lyons, Eds., *A NIME Reader: Fifteen Years of New Interfaces for Musical Expression*, Current Research in Systematic Musicology. Springer International Publishing, 2017.

[3] Luca Turchet, Andrew McPherson, and Mathieu Barthet, "Co-design of a Smart Cajón," *Journal of the Audio Engineering Society*, vol. 66, no. 4, pp. 220–230, Apr. 2018.

[4] Jack Armitage and Andrew McPherson, "Crafting Digital Musical Instruments: An Exploratory Workshop Study," in *Proceedings of the 2018 conference on New Interfaces for Musical Expression*. 2018, New Interfaces for Musical Expression.

[5] Dale Dougherty, "The maker movement," *Innovations: Technology, Governance, Globalization*, vol. 7, no. 3, pp. 11–14, 2012.

[6] Chris Anderson, *Makers: The New Industrial Revolution*, Random House Business, London, Apr. 2013.

[7] Neil Gershenfeld, "How to Make Almost Anything," *Foreign Affairs*, , no. November/December 2012, Nov. 2012.

[8] Neil Gershenfeld, *Fab: The Coming Revolution on Your Desktop-from Personal Computers to Personal Fabrication*, Basic Books, New York, NY, new ed edition edition, Feb. 2007.

[9] Pelle Ehn, Elisabet Nilsson, Richard Topgaard, Anders Emilson, and Per-anders Hillgren, *Making Futures: Marginal Notes on Innovation, Design, and Democracy*, MIT Press, 2014.

[10] Erling Bjögvinsson, Pelle Ehn, and Per-Anders Hillgren, "Design things and design thinking: Contemporary participatory design challenges," *Design Issues*, vol. 28, no. 3, pp. 101–116, 2012.

[11] Mieke van der Bijl-Brouwer and Kees Dorst, "Advancing the strategic impact of human-centred design," *Design Studies*, vol. 53, pp. 1–23, Nov. 2017.

[12] Clay Spinuzzi, "The methodology of participatory design," *Technical communication*, vol. 52, no. 2, pp. 163–174, 2005.

[13] Elizabeth B.-N. Sanders and Pieter Jan Stappers, "Probes, toolkits and prototypes: three approaches to making in codesigning," *CoDesign*, vol. 10, no. 1, pp. 5–14, 2014.

[14] Richard Pickvance, *A Gamelan Manual: A Player's Guide to the Central Javanese Gamelan*, Jaman Mas Books, London, May 2005.

[15] Giulio Moro, Astrid Bin, Robert H. Jack, Christian Heinrichs, and Andrew P. McPherson, "Making high-performance embedded instruments with Bela and Pure Data," in *Proceedings. of the International Conference of Live Interfaces*, 2016.

[16] Edgar Berdahl and Wendy Ju, "Satellite CCRMA: A Musical Interaction and Sound Synthesis Platform.," in *NIME*, 2011, pp. 173–178.

FINDING SHIMI'S VOICE: FOSTERING HUMAN-ROBOT COMMUNICATION WITH MUSIC AND A NVIDIA JETSON TX2

Richard Savery
GTCMT
Georgia Institute of Technology, USA
rsavery3@gatech.edu

Ryan Rose
GTCMT
Georgia Institute of Technology, USA
rrose37@gatech.edu

Gil Weinberg
GTCMT
Georgia Institute of Technology, USA
gilw@gatech.edu

ABSTRACT

We present a novel robotic implementation of an embedded linux system in Shimi, a musical robot companion. We discuss the challenges and benefits of this transition as well as a system and technical overview. We also present a unique approach to robotic gesture generation and a new voice generation system designed for robot audio vocalization of any MIDI file. Our interactive system combines NLP, audio capture and processing, and emotion and contour analysis from human speech input. Shimi ultimately acts as an exploration into how a robot can use music as a driver for human engagement.

1. INTRODUCTION

The field of robotics depends on embedded hardware and software for real-time computational tasks such as kinematics, computer vision, and sensor data processing. For many of these tasks, state-of-the-art performance depends on computationally heavy deep learning techniques. Embedded computing devices have only recently been developed with the GPUs necessary to perform complex deep learning inference in real-time. One such device is the NVIDIA Jetson TX2, an embedded system-on-module that runs Linux on a quad-core ARM processor, and features an 8GB GPU built on NVIDIA's Pascal architecture. This powerful and energy-efficient device greatly expands the capabilities of robots and other embedded applications alike through its ability to run both high CPU and GPU tasks, such as artificial neural networks, deep learning, and signal processing.

This project uses the Jetson TX2 to run a musical robot companion named Shimi (Figure 1). Shimi moves with five degrees of freedom, and can play audio out of two speakers on either side of its head. Additionally, Shimi features a 4-microphone array on its underside. Prior to being run by the Jetson TX2, Shimi was controlled with an Android smartphone and an Arduino Mega.

The purpose of Shimi is to explore novel ways in which humans can communicate with artificial intelligence (AI) agents. Many modern AIs attempt to replicate communicative patterns of humans as closely as possible, using state-of-the-art text-to-speech procedures and complex mechanical operation to try and convince users that they interact with a human-like device, not a computer or a robot. This can quickly lead to the "uncanny valley" psychological phenomenon, where the small differences between an AI and a real human evoke a deeply unsettling feeling. In this project, the authors embrace the non-human robotic identity of Shimi and explore methods of communication using Shimi's limited range of motion and music, in place of verbal language. This is realized through a voice generation system that utilizes deep learning to respond to human speech in an emotionally relevant manner, and a gesture generation system that uses both quantified emotion and Shimi's musical voice to craft robotic body language using Shimi's five degrees of freedom.

Figure 1: The musical robot companion Shimi.

2. RELATED WORK

Prior work on Shimi focused first on utilizing the sensors and computational power of a smartphone to explore the possibilities of personal robotics in a cost-effective way [1]. The research in this study also provided inspiration for life-like gestures, taking cues from animation. Other work on Shimi explored expressing emotion through gesture, informed by observations of human movement and emotion from Darwin [2, 3]. Others have used the Laban Effort System in gesture generation, specifically in low degree of freedom robots such as Shimi [4]. Additionally, speech analysis as input to gesture generation has been used for robot communication in many cases such as *Kismet* [5].

Music as a vector for emotion has been demonstrated in numer-

ous studies, with comprehensive research exploring what emotions can be perceived or induced through music, what musical features encode emotion, and how music expresses or induces emotion [6]. Studies have shown clear correlations between musical features and movement features, suggesting that a single model can be used to express emotion through both music and movement [7]. Additionally, humans demonstrate patterns in movement that is induced from music [8].

3. TECHNICAL DESCRIPTION

3.1. Voice System

3.1.1. Input Analysis

Shimi analyzes incoming audio streams using a combination of natural language processing (NLP) and raw audio analysis. Shimi features a Seeed Studio ReSpeaker Mic Array v2.0 [1], a four-microphone array with on-board processing that combines each microphone stream and denoises the recording, emphasizing voice signals. No additional processing of input signals was added after the ReSpeaker processing, other than down-mixing to a single channel. Using the open-source hotword detection library `Snowboy`[2], Shimi responds to the phrase "Hey Shimi," and begins recording input audio. The Python phrase detection library `speech_recognition`[3] is then used to capture one phrase of raw audio.

Incoming audio is analyzed using the valence arousal model, whereby valence is the measure of the positivity or negativity of an emotion, and arousal is the measure of the energy of an emotion[9]. Raw audio analysis is used to find the arousal level, pitch, intensity and onsets. To do this we utilized `Parselmouth`[4], a Python library built on `Praat`[5]. We created custom metrics to analyze the input level based on analysis of the Ryerson Audio-Visual Database of Emotional Speech and Song (RAVDESS) data set [10]. RAVDESS includes 7356 audio files by 24 actors, each rated with an emotion independently validated by 10 participants. Our metrics were based on pitch contours and intensity levels found in the recordings. Figure 2 and 3 show analysis of the phrase *the dogs are sitting by the door* from the data set. Our metrics to measure arousal use the variety, level and standard deviation in intensity and the range, contour and standard deviation of pitch.

To measure valence we use the Natural Language Toolkit (`NLTK`) [11], a suite of Python modules for NLP. We calculate valence using a built in naïve bayes classifier trained on the `NLTK` data set of tagged phrases from social media. We also use the `NLTK` library for statement classification.

3.1.2. Shimi's Emotion

Shimi maintains its own emotional state through each communication, tracked through a position in valence and arousal. Valence and arousal are both measured between -1 and 1. The current model gradually shifts the valence level towards that of the user while mirroring the arousal of the user. A negative valence statement from the user will cause Shimi to respond in a sad tone. Following positive statements from the user will gradually move Shimi towards positive responses. When starting Shimi begins with a valence of 0.5, equating to slightly happy.

3.1.3. MIDI Dataset and Phrase Generation

To control Shimi's vocalizations we generate MIDI phrases that then drive the synthesis and audio generation described below and lead the gesture generation. For this purpose we created our own data set of MIDI files tagged with a valence and arousal quadrant. We collected MIDI files from eleven improvisers around the United States. Each was told to record MIDI phrases between 100ms and 6 seconds with each phrase assigned one of the quadrants from the valence/arousal model. They also recorded phrases that they believe represented a question, an answer to a question, a greeting and a farewell. Improvisers were told to record between 50 to 200 samples of each category. To restrict the data each phrase could only contain velocity values at the start of a note and no MIDI data outside pitch, velocity and rhythms were included in training (i.e. no expressive modulations).

As the data set was created by many improvisers we created a second process to confirm the validity of the collected files. This was done through a comparison of the pitch, velocity and contour variation between the new MIDI data set and the RADVESS data set. Figure 3 and Figure 4 present the an example of the variance in the data-set between different emotions (blue is pitch, orange is intensity, placed over a spectrogram). Any MIDI file that varied too far from the features of RADVESS was removed from the data set. Table 1 shows the final amount of files used for Shimi's phrase generation. The RADVESS data set does not include greetings, farewells, questions or answers and due to their limited use in Shimi's interaction we did not post process these phrases.

Table 1: *Shimi Emotional MIDI Data set*

$PhraseType$	MIDI Samples	Post Process
$VA1(Happy)$	895	400
$VA2(Angry)$	1042	621
$VA3(Sad)$	980	567
$VA4(Calm)$	700	385
$Greetings$	655	655
$Farewell$	895	895
$Question$	901	901
$Answer$	778	778

To generate phrases for Shimi vocalizations, we choose to use a data driven generative method. We also considered using the samples recorded by improvisers directly, however we wanted to aggregate the features created by all improvisers and develop a system that allowed limitless variability. Having chosen to use deep learning a relatively simple Long short-term memory, recurrent neural network (LSTM RNN) was implemented in Keras over Tensorflow as has been previously presented [12][13]. This type of neural network is useful for this task as it is sequential and considers parts of its input as it creates output, encouraging the creation of musical phrases. The data set was first transposed into all twelve keys, to avoid a need to identify a key center. Eight different versions of the network were trained, one for each tagged component of the data set. This was done with the goal of a faster run time.

[1] http://wiki.seeedstudio.com/ReSpeaker_Mic_Array_v2.0/
[2] https://snowboy.kitt.ai/
[3] https://github.com/Uberi/speech_recognition
[4] https://github.com/YannickJadoul/Parselmouth
[5] http://www.fon.hum.uva.nl/praat/

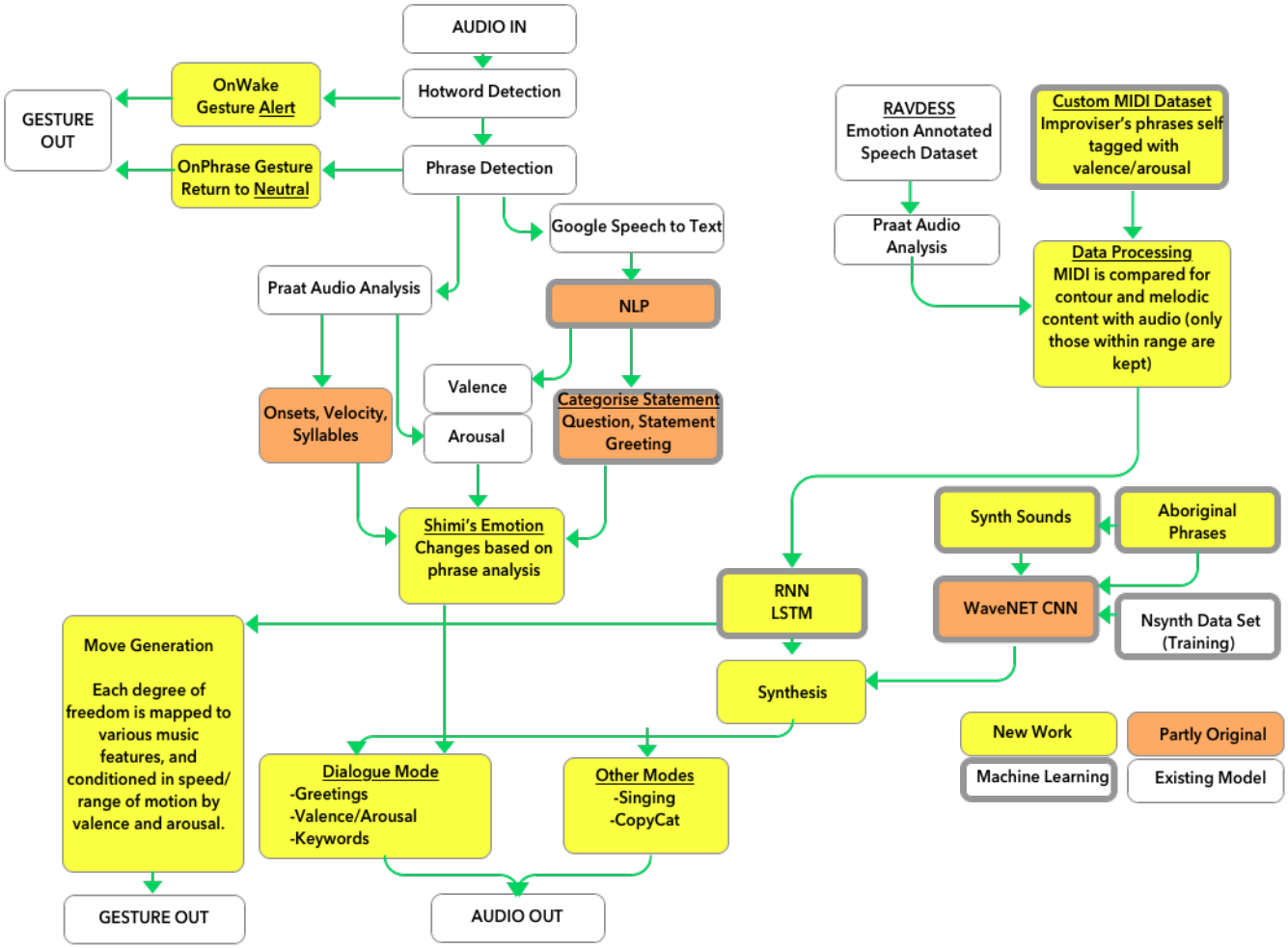


Figure 2: Shimi System Overview.

3.1.4. Audio Creation and Synthesis

MIDI phrases are fed to a new synthesis system created for Shimi. To generate vocalizations that focuses on emotions devoid of all semantic meaning, we chose to construct a new vocabulary. Shimi's vocabulary is built upon phonemes from the Australian Aboriginal language Yuwaalaraay a dialect of the Gamilaraay language. Originally ideas explored real-time implementations of deep learning raw audio synthesis, however it quickly became apparent that this would add unacceptable amount of latency to the system. In our testing even with large compromises in bit rate we were never able to achieve less than a 1 to 5 ratio of processing sound (1 second took 5 seconds to process). Instead of real-time synthesis we compromised by interpolating 28 language samples with four different synthesizer sounds, manually created by the authors. For each sound three different intensity levels were recorded at two different octaves, giving a total of 672 wave samples each 500 ms long. Our final interpolation was done using a modified version of NSynth[14], trained on the NSynth data set. Sounds are played back using a synthesis engine that time stretches and pitch shifts the wave samples to match the incoming MIDI file.

3.2. Gesture System

Much like in human communication, Shimi's gestures are tightly coupled with speech [15]. The voice system produces three outputs: an audio file of Shimi's speech, the MIDI musical representation of the audio, and quantitative measures of Shimi's current emotion. The latter two outputs are the inputs to a rule-based generative gesture system, which controls synchronized playback of gesture with the generated audio.

The first step in gesture generation is musical feature extraction from the MIDI representation of Shimi's speech. Using the Python libraries `pretty_midi`[6] and `music21`[7], musical features such as tempo, range, note contour, key, and rhythmic density are obtained. These features are used to create mappings between Shimi's voice and movement; for instance, pitch contour is used to govern

[6]https://github.com/craffel/pretty-midi
[7]https://github.com/cuthbertLab/music21

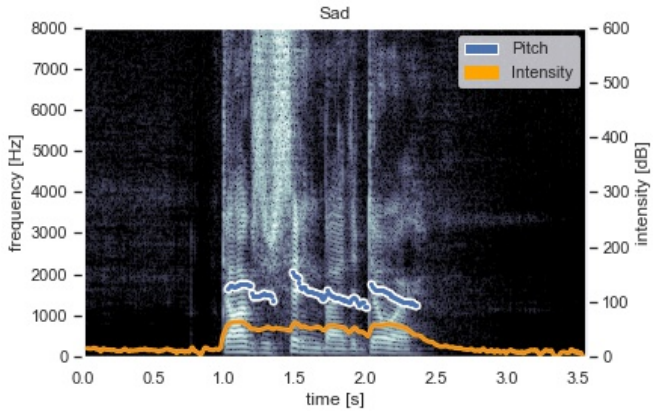


Figure 3: Sad Speech Intensity and Pitch

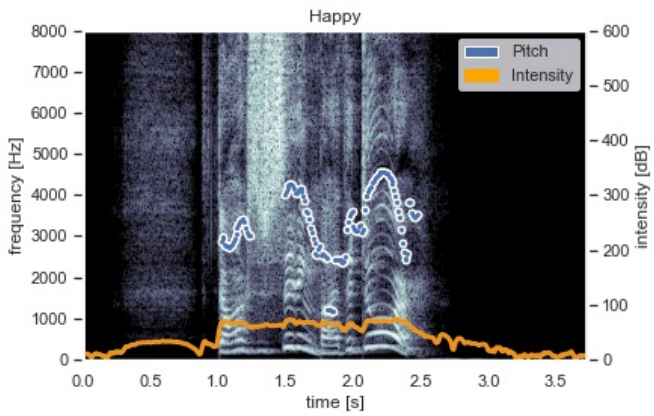


Figure 4: Happy Speech Intensity and Pitch

Shimi's torso forward-and backward movement. Other mappings include beat synchronization across multiple subdivisions of the beat in Shimi's foot, and note onset-based movements in Shimi's up-and-down neck movement. These mappings are based on research investigating correlative features in music and musically-induced movement [8, 7, 16].

The next step uses the emotion state of Shimi to condition Shimi's movement. Emotion is provided to the system in the form of continuous-valued valence and arousal. These values are then used to condition the musical mappings formed previously. In general, arousal is used to restrict or expand range of motion, and valence is used to govern the amount of motion Shimi exhibits, though exact usage varies for each degree of freedom.

In addition to musical and emotional mappings, some degrees of freedom are interdependent. For example, as Shimi's torso moves forward, Shimi's head naturally moves forward and toward the ground. This affects where Shimi is looking, so it is important to consider Shimi's torso position when generating neck up-and-down movement. To accommodate this, the movement paths of Shimi's degrees of freedom are generated sequentially and in full, before being actuated together in synchronization with the audio of Shimi's speech. This is implemented using the built-in `threading` library in Python, with each degree of freedom being associated with one Python thread responsible for sending motor control commands across the duration of the gesture.

The motors used in Shimi are Dynamixel MX-28 actuators produced by Robotis. They feature built-in controllers, allowing for closed-loop control through half-duplex UART serial communication. While the MX-28 motors allow for both reading and writing of position and speed, the half-duplex nature of their communication introduces latency when reading and writing to multiple motors at once, at a resolution high enough for smooth movement. To generate rigorously timed gestures, we do not read Shimi's motors write to them as infrequently as possible. This minimizes any latency inherent in the transmission of data to the motors. For smooth and natural-looking movement, the velocity curve of a gesture is most important. As such, position of Shimi's motors is only ever set when direction of movement changes, and velocity changes are set as frequently as possible without accruing latency. Setting position once and defining the velocity curve allows for control of both when Shimi reaches a certain position, and how Shimi gets there.

Gestures, then, are defined as sequences of movements to a position over a specified time. To facilitate programmatic gesture generation, a collection of velocity curves have been implemented to provide styles of movement. The simplest is a constant velocity, where velocity is the distance of the movement over its duration (Figure 5). This style looks the most stereotypically "robotic", as the motors can accelerate from rest to max velocity much faster than a human can.

Previous work on Shimi introduced a velocity curve that features a constant acceleration until the midpoint of the gesture, then a constant deceleration [1]. This works particularly well for single movement or broad gestures, and looks the most realistic when compared with human motion (Figure 5).

In the context of a multi-move gesture, however, accelerating and decelerating every movement becomes unnatural, as multi-movement human gestures do not come to rest bewteen each move. Thus, a constant acceleration (or deceleration) and constant velocity curve can cap both ends of a gesture. An example of the acceleration variety is shown in Figure 5.

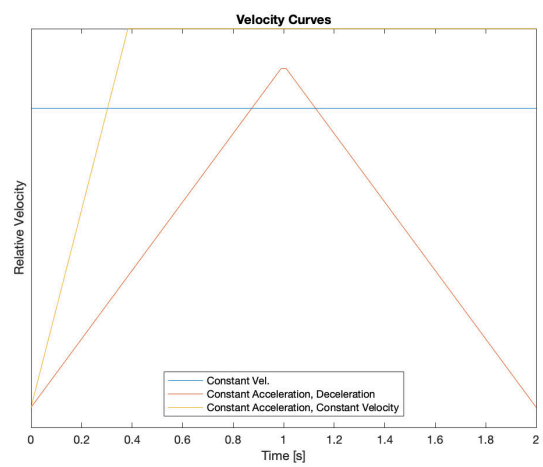


Figure 5: Graphs of the velocity curves used for Shimi movements.

In addition to the movement sequencing method of gesture generation, a different method of recording and playing back gestures is being explored. This method requires physically moving Shimi's

limbs in a desired gesture while the motors continuously record position and speed as fast as possible. After recording, the captured positions and speeds can be used to actuate the gesture on Shimi on demand, resulting in a highly detailed and smooth gesture. While this method results in the most nuanced and expressive gestures, there are difficulties in playing back recorded gestures accurately in time with the way they were recorded. The time taken to read a motor's position and speed varies, resulting in playback that is not aligned with the recording. This timing behavior makes synchronization with speech, which is a necessity for Shimi, very difficult. More research on ways to align these types of gestures with audio is being explored.

4. APPLICATIONS AND FUTURE WORK

This work has described Shimi's ability to generate musical and gestural responses to human speech input that attempts to replicate the emotion conveyed in a spoken phrase. These short form interactions provide insight into how robots can express emotion and communicate with music. A next step in communication will be seeing how accurately Shimi can imitate a phrase, both vocally and, more importantly, emotionally. We are also interested in expanding Shimi's musical phrases to include more languages and improvisers of different origins.

Shimi originated as a musically-intelligent speaker dock, and the work presented here can extend to more musical applications as well. One possibility is as a nuanced music recommendation system. In this system a human would ask Shimi if they would like a song, and Shimi would reply with a vocalization and gesture demonstrating an opinion of that song. This way of expressing opinion can be much more detailed than the thumbs up/thumbs down of many music service providers today. Another engaging musical experience furthers a previous goal of the Shimi project: to enjoy one's music alongside a human listener. Now that Shimi has a voice, the ability to dance along with one's music can incorporate singing along as well. This could also lead to Shimi as a robotic performer, listening to human performers and improvising alongside as a vocalist.

5. ACKNOWLEDGMENTS

Thanks to Matthew Kaufer and Yashveer Singh.

6. REFERENCES

[1] Guy Hoffman, "Dumb robots, smart phones: A case study of music listening companionship," in2012 IEEE RO-MAN: The 21st IEEE International Symposium on Robot and Human Interactive Communication, Paris, sep 2012, pp. 358–363, IEEE

[2] Charles Darwin and Phillip Prodger, The expression of the emotions in man and animals, Oxford University Press, USA, 1998.

[3] Mason Bretan, Guy Hoffman, and Gil Weinberg, "Emotionally expressive dynamic physical behaviors in robots,"International Journal of Human-Computer Studies, vol. 78, pp. 1–16,jun 2015

[4] Heather Knight, "Expressive motion for low degree-of-freedom robots," Jul 2018. https://pdfs.semanticscholar.org/69e2/dc8eab578131a536396a812a2306b6796477.pdf

[5] Cynthia Breazeal and Lijin Aryananda, "Recognition of Affective Communicative Intent in Robot-Directed Speech," p. 20, 2002. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.471.7147rep=rep1type=pdf

[6] Patrik N. Juslin and John A. Sloboda, "15 - music and emotion," in The Psychology of Music (Third Edition), Diana Deutsch, Ed., pp. 583 – 645. Academic Press, third edition, 2013

[7] Beau Sievers, Larry Polansky, Michael Casey, and Thalia Wheatley, "Music and movement share a dynamic structure that supports universal expressions of emotion,"Proceedings of the National Academy of Sciences, vol. 110, no. 1, pp. 70–75, jan 2013

[8] Petri Toiviainen, Geoff Luck, and Marc R Thompson, "Embodied Meter: Hierarchical Eigenmodes in Music-Induced Movement,"Music Perception: An Interdisciplinary Journal, vol.28, no. 1, pp. 59–70, sep 2010.

[9] James A. Russell, "A circumplex model of affect, "Journal of Personality and Social Psychology, 1980. https://www.researchgate.net/publication/235361517_A_Circumplex_Model_of_Affect

[10] Steven R Livingstone and Frank A Russo, "The Ryerson Audio-Visual Database of Emotional Speech and Song(RAVDESS): A dynamic, multimodal set of facial and vocal expressions in North American English, "PLOS ONE, vol. 13,no. 5, pp. 1–35, 2018

[11] Steven Bird, Ewan Klein, and Edward Loper, Natural Language Processing with Python, O'Reilly Media, Inc., 1st edition, 2009.

[12] Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks,"Web Page, 2015

[13] Richard Savery and Gil Weinberg, "Shimon the Robot Film Composer and DeepScore", "Proceedings of Computer Simulation of Musical Creativity, August 2018, Dublin, Ireland. http://galapagos.ucd.ie/wiki/pub/OpenAccess/CSMC/Savery.pdf

[14] Jesse Engel, Cinjon Resnick, Adam Roberts, Sander Dieleman, Douglas Eck, Karen Simonyan, and Mohammad Norouzi, "Neural Audio Synthesis of Musical Notes with WaveNet Autoencoders, "CoRR, vol. abs/1704.0, 2017.

[15] David McNeill, How language began: Gesture and speech in human evolution, Cambridge University Press, 2012.

[16] Birgitta Burger, Suvi Saarikallio, Geoff Luck, Marc R. Thompson, and Petri Toiviainen, "Relationships Between Perceived Emotions in Music and Music-induced Movement,"Music Perception: An Interdisciplinary Journal, vol. 30, no. 5, pp.517–533, June 2013.

A SCALABLE HAPTIC FLOOR DEDICATED TO LARGE IMMERSIVE SPACES

Nicolas Bouillot

Société des arts technologiques [SAT]*
Montréal, Canada
nbouillot@sat.qc.ca

Michał Seta

Société des arts technologiques [SAT]
Montréal, Canada
mseta@sat.qc.ca

ABSTRACT

We present a haptic floor composed of tiles with independently controllable vertices and designed to cover arbitrary large flat surfaces. We describe the signal distribution architecture, based on SATIE, our spatialization engine and SWITCHER, our low latency and multichannel streaming engine. The paper also provides a description of several approaches of content authoring when such a floor is deployed in an immersive space. These approaches emphasizes the correlation among immersion modalities such as continuous localization of sound from the speaker system to the floor and continuous physical effect from the video projection to the floor.

1. HAPTIC FLOOR FOR LARGE IMMERSIVE SPACES

We question and experiment the extension of a large immersive space with a haptic floor covering the entirety of the surface. Largely motivated by augmentation of artistic venues with audience, we are interested in those with i) the ability to offer immersive listening for a group of people, ii) visual immersion, and iii) floor space allowing for a case-by-case configuration of audience position (sitting, standing, lying down) and along with configuration of performance space.

Not limited to the above characteristics, the haptic floor prototype we propose has a scalable design and a flexible authoring possibility, targeting tight relation with the audiovisual effect in the venue. Indeed, as demonstrated by the scientific literature, augmentation of immersion with haptics could increase the perception of virtual environments by an audience, more specifically when combined with immersive sounds and visuals. Vection[1], for instance, when stimulated by the actuators placed between the ground and the feet of a sitting subject, is obtained in a shorter time and with more intensity when the haptic feedback (constant frequency sinusoidal vibrations) is applied[1]. Similar results are obtained for a standing posture, without any impact on the impression of presence[2]. Haptic feedback can contribute to increased perceptive sensibility of an individual[3, 4]. The above cited research, however concern experiences involving individual users. From this point of view, developing a floor simulating haptic feedback of walking on particular ground textures[5], such as snow, offers an opportunity for a collective sensory experience.

Unfortunately the above cited research does not apply directly to our work. The devices employed in the previous research consider pre-determined posture of the subject, particularly in the case of walking. Other approaches and applications remain to be explored that offer creators an immersive and flexible space where different experiences can be quickly prototyped.

As our primary source of inspiration, our immersive space called the Satosphere (see figure 1) is a large dome-shaped audiovisual projection space offering the view of the horizon (floor to ceiling projection) and 360° at the same time. Over 11 meters high and 18 meters in diameter, the Satosphere is equipped with 157 loudspeakers grouped into 31 adjacent clusters on the dome's surface, and with 8 video projectors that distribute the video image across the dome's surface. Other venues around the world provide a listening environment for spatial audio, where our research could apply. The CUBE [6] of the Virginia Tech is an immersive space dedicated to sound. It offers a significant spatial resolution with its 124 audio channels and provides for audio spatialization techniques, including movement capture. The ALLOSPHERE [7] offers 360° vertical and horizontal immersion. The lack of a floor is compensated by a bridge, allowing to go to the center and experiment with different data visualization strategies and audio-visual immersive compositions. Unfortunately, this constraints the viewer to assume standing position and move only around the narrow bridge. Another example, in France, ESPACE DE PROJECTION at IRCAM provides rotating panels to offer several acoustic profiles[8]. It hosts 75 speakers arranged in a cube. Finally, the team at the Center for Computer Research in Music and Acoustics at Stanford University has developed the GRAIL[9], a system of 32 speakers and 8 subwoofers that can be deployed in different locations, such as outdoors, concert halls or studios. There are other venues equipped with speaker setups that accommodate listening to spatial music, a non-exhaustive list provides[10]: the IEM-CUBE and MUMUTH in Graz, the MULIT in Bergen, the MOTION LAB in Oslo, the SPACE in Pesaro and the DIGITAL MEDIA CENTER THEATRE in Bâton-Rouge.

Our work on the haptic floor is, to our knowledge, unique in treating the question of creation and reproduction of content for haptic feedback in the context of immersive space for groups of participants in non-specific postures (see figure 1). In this case, the device is designed to cover arbitrarily large surfaces, its hexagonal shape (see figure 2) allows for easy assembly and fitting into large, flat surfaces, such as the Satosphere's floor.

In this paper, we present the prototype of a floor built to provide haptic feedback during experiences designed to posture agnostic content created for large immersive spaces (Figure 1). Our prototype represents one segment of a device that could cover arbitrarily large floor surfaces and is driven by audio signals delivered via local network. It is integrated into our spatialization software, SATIE[11], therefore it is tightly coupled with the immersive content. This is illustrated by our demonstration video showing our floor integrated in an immersive space[2].

* Society for Art and Technology [SAT] is an artist center in Montreal specializing in dissemination of art made with new technologies with the focus on immersive arts and experiences.

[1] Illusion of self-motion

[2] See our demonstration video (accessed Dec. 2018): https://vimeo.com/290925507

(a) The audience can freely move around the space and interact with telepresent space or simply sit around the center (Miscible by Maotik and Manuel Chantre

(b) The audience is lying down and fills most of the floor space (Plateaux by Vincent Brault, Owen Kirby and Vincent Martin)

Figure 1: Examples of scenographies in our immersive space, illustrating the need for a haptic floor to be posture-agnostic, group-friendly and correlated with audiovisual displays.

2. OUR HAPTIC FLOOR PROTOTYPE

As seen in Figure 2, the hardware prototype is only one portion of the projected floor, consisting of a single hexagon divided in six triangles forming a mesh. An actuator is placed at each of the 7 vertices and is controlled individually with an audio signal ranging between 0 and 100 Hz for a height amplitude of 38.1 mm. The shape has been designed to easily scale up to the surface of larger space by multiplying the hexagonal components.

The signal distribution pipeline (Figure 3) consists of an audio renderer (a computer with ubuntu Linux 18.04) equipped with an appropriate audio I/O and a set of Raspberry Pis running Raspbian. Each RasPi is required to control three actuators (the *250i* model from our partner D-Box[3]). The audio renderer is based on a SuperCollider script combining specific signal processing and our spatialization engine SATIE described in more detail in section 3.1.

The audio renderer runs SATIE[4], our spatialization engine that provides for use of multiple spatializers in parallel[12]. In this case, one rendering is performed for entire haptic floor, along with the existing 8-speaker audio display (or the dome). The coupling of the haptic floor and speaker system allows for keeping some coherence in experience design, thanks to an internal per-rendering handling of the same OSC [13] message.

The audio renderer can handle a variety of inputs. Direct audio signals correspond to sound objects in SATIE that can be spatialized. OSC messages are interpreted in two possible ways, one with the SATIE protocols that allows for sound object control (location, spread, etc) and the other by controlling position of each actuator independently.

Audio spatialization is handled via an audio interface wired to the speaker setup. The haptic floor is handled via LAN connecting Raspberry Pi devices, each talking to a custom USB audio interface which controls up to 3 actuators (where one actuator affects one vertex of the floor's "mesh"). The latter allows for a flexible increase in number of floor subparts, adding *just* more Raspberries in the network.

On the software side, SATIE handles all input cases (audio and control signals) and performs spatialization for both the physical speaker system and the haptic floor. The audio signals destined for the traditional speakers are handled directly with the audio interface. The audio signals that control the haptic floor need to be sent over network to the Raspberry Pis. Low latency streaming from the audio renderer to the Raspberries is achieved using SWITCHER[5], our multichannel and low latency streaming engine. The transmission of audio streams from SATIE to switcher is done through the jack server.

3. AUTHORING FOR HAPTIC FLOOR

The challenge lies in designing haptic content that is appropriate to the type of immersive experience, and more particularly when sound and graphics are involved. Here follow some use cases where haptic floor control can be correlated with immersive content:

- locate the sound in the floor in order to continue a sound trajectory
- propagate waves from a sea displayed on screen to mechanical waves on the floor
- ripple effect as well as delivering of different types of haptic content to different areas of the floor at the same time, corresponding to drops falling from the sky
- control vibration of the floor according to the sound played

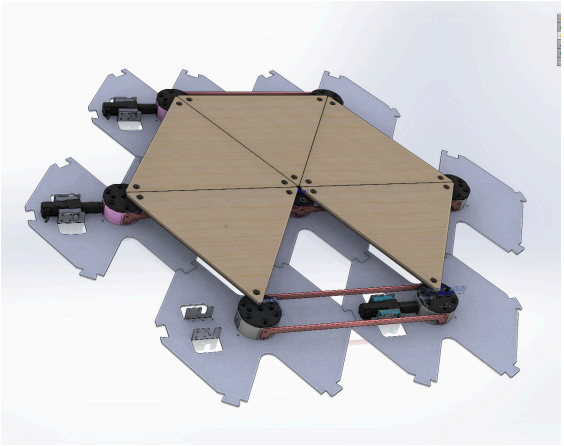
3.1. Audio spatializer based rendering

SATIE [11, 14], written in the SuperCollider language [15], provides rendering of virtual audio scenes spatialized over many audio chan-

[3] D-Box is a company which designs, manufactures, and markets actuators intended mainly for the entertainment and industrial simulation markets. `https://www.d-box.com/en`, accessed Dec. 2018

[4] `https://gitlab.com/sat-metalab/satie`, accessed Dec. 2018

[5] `https://gitlab.com/sat-metalab/switcher`, accessed Dec. 2018



(a) Concept drawing of one subpart of a larger haptic floor. 7 actuators (two of them, the black cylinders, can be seen on the left) are controlling each vertices independently.

(b) Our haptic floor prototype with a cube shaped 8 speaker system. We use the SATIE dual rendering feature in order to provide continuous spatialization among the speaker system and the haptic floor.

Figure 2: Hardware design of a floor subpart. The current prototype is only one section of the target haptic floor and.

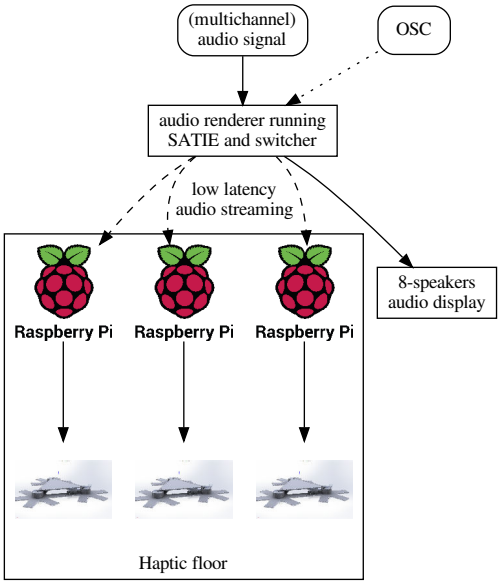


Figure 3: Distribution pipeline. Our prototype, in addition to : use three Raspberry Pis and three D-Box interfaces, each one controlling three actuators.

nels. We were able to build upon our previous experience with near-field/far-field audio rendering[12] and tackle the floor as another audio display because the provided actuators transforms digital audio signals to mechanical movement. This approach provides a few benefits. First of all, the synchronization of audio signals, after compensation for the delay between audio displays, is handled by SATIE and does not require any other work. Secondly, haptic content creation can be approached in parallel with audio creation and spatialization design.

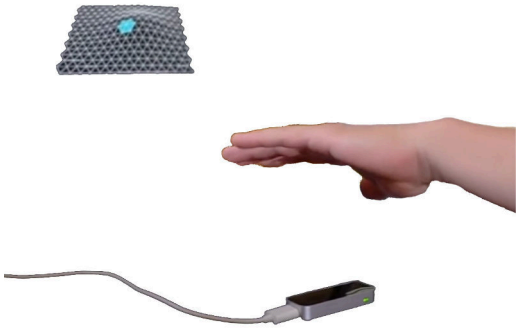
We have experimented with different approaches to spatial audio such as VBAP, ambisonics and a crude equal power panning. All types of audio spatialization work well and the choice of approach will depend on the desired effect and audio content. We also applied an envelope tracker filtering in order to convert audio signals into signals compatible with our actuators that respond well to frequencies between 0 and 100 Hz.

Moreover, since SATIE can handle many types of control inputs (audio, USB, network), it can still assist in delivering synchronized audible and haptic audio signals. Additionally we can take advantage of SuperCollider's powerful synthesis and DSP capacity to experiment and design audio signals suitable for the haptic floor with a lot of flexibility. Finally, the spatialization of audible audio signals and audio delivered to the haptic floor can be completely independent, which also offers the necessary creative freedom.

As one of our first experiments, in collaboration with D. Andrew Stewart[6], we explored the use of the floor prototype driven by 8 discreet channels of analog audio from an electronic performance instrument based on Omnisphere VST plugin, controlled by Karlax controller. The audio channels were spatialized as 8 independent sound objects on the octophonic, cube-shaped speaker layout, each acting on the floor depending on the spatialization parameters. We have experimented with OSC messages sent from Karlax directly to SATIE as well as via our interactive creation tool for immersive spaces, EIS[16]. Through this short experimentation with live performance, we found that using the sound of the instrument to drive

[6] `http://dandrewstewart.ca/`, accessed Dec. 2018

(a) Live control from Ableton (concept and development from Mourad Bennacer). The orientation from the device (left) is applied to the floor (right).



(b) Mapping with a 3D Mesh: the blue mesh (upper left) is a virtual representation of the haptic floor and part of a larger mesh. Here the hand detection with Leap motion (bottom) allows for *touching* the floor (Concept and development from Sébastien Gravel and Vincent Brault).

Figure 4: Authoring content for the haptic floor with other software (Ableton and Touch Designer) using an OSC protocol allowing real-time control of each actuator independently.

the movement and the texture of the floor creates a deep sense of coherence. In fact, placing performers on the floor can have its benefits.

3.2. Other approaches

The other approaches we describe here are based on the control of each actuator independently from Open Sound Control[17]. Messages from external software are composed of several float values ranging from -1 to 1, each one being the desired height of the corresponding actuator. Along with our audio spatializer based rendering, they are illustrated in our demonstration video[2].

The first one, illustrated in Figure 4a, is a live control from Ableton. The basic protocol has been implemented in Ableton where Ableton specific creations can be used in order to create a tight relation between the sound and/or an Ableton controller with the haptic floor. This allows, for instance, to program automated floor vibrations synchronized with the audio track. In our demonstration video, a simple mapping from the orientation of an accelerometer-equipped device with the orientation of the haptic floor has been implemented.

The second approach is a mapping of the floor with a 3D Mesh (Figure 4b) in a 3D software. Accordingly, any physics or interaction applied in the virtual environment becomes a source of vibration possibly applied to the floor. This provides the potential for strong correlation of the visual with the floor. With this approach for instance, sea waves from a simulation can be displayed from the screen with a consistent continuity in the floor. In our demonstration video, a hand tracking system, the Leap Motion, is used in order to move a virtual hill along a planar mesh.

4. CONCLUSION & NEXT STEPS

This paper has presented our experience with distribution architecture of a scalable haptic floor targeting posture-agnostic multi-person immersive spaces. The floor is scalable in space thanks to its triangular shape allowing for unlimited tiling. Its signal distribution scalability is ensured using low latency multichannel streaming to Raspberry Pis, each one dedicated to groups of three actuators.

Surprisingly, experiments with our prototype have pointed us towards an uncharted territory of haptic feedback, both from technological and creative points of view. This led us to describe in this paper a set of methods for authoring content for floor-involved immersive content: i) using the floor as an additional "audio display" driven by audio and/or using multi-speakers spatialization algorithm and ii) producing content from other software, including 3D graphic engine, with the help of a basic OSC protocol providing independent control of each actuator height.

Our next steps will be targeting experiments with a larger scale floor covering our dome with approximately 200 actuators. Along with physical design and construction methods, we will go forward with improvement of authoring methods for group of users and validation of the architecture scalability.

5. ACKNOWLEDGEMENTS

This project would not be possible without the support of the Ministère de l'Économie et de l'Innovation du Québec and D-Box. We also wish to thank the following people who were involved in making and documenting this prototype: Sebastien Gravel, Vincent Brault, Mourad Bennacer, Generique Design, Louis-Philippe St-Arnault, Luc Martinez, Jean-François Menard, Michel Paquette, Jérémie Soria and Giacomo Ferron. We thank Sébastien Roy for the pictures in Figures 1a and 1b.

6. REFERENCES

[1] I. Farkhatdinov, N. Ouarti, and V. Hayward, "Vibrotactile inputs to the feet can modulate vection," in *2013 World Haptics Conference (WHC)*, April 2013, pp. 677–681.

[2] Ernst Kruijff, Alexander Marquardt, Christina Trepkowski, Robert W. Lindeman, Andre Hinkenjann, Jens Maiero, and Bernhard E. Riecke, "On your feet!: Enhancing vection in

leaning-based interfaces through multisensory stimuli," in *Proceedings of the 2016 Symposium on Spatial User Interaction*, New York, NY, USA, 2016, SUI '16, pp. 149–158, ACM.

[3] Alexandre Gardé, Pierre-Majorique Léger, Sylvain Sénécal, Marc Fredette, Elise Labonté-Lemoyne, François Courtemanche, and Jean-François Ménard, "The effects of a vibrokinetic multi-sensory experience in passive seated vehicular movement in a virtual reality context," in *Extended Abstracts of the 2018 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, 2018, CHI EA '18, pp. LBW091:1–LBW091:6, ACM.

[4] J. Plouzeau, J. Dorado, D. Paillot, and F. Merienne, "Effect of footstep vibrations and proprioceptive vibrations used with an innovative navigation method," in *2017 IEEE Symposium on 3D User Interfaces (3DUI)*, March 2017, pp. 241–242.

[5] Yon Visell, Bruno L. Giordano, Guillaume Millet, and Jeremy R. Cooperstock, "Vibration influences haptic perception of surface compliance during walking," *PLOS ONE*, vol. 6, no. 3, pp. 1–11, 03 2011.

[6] Eric Lyon, Terence Caulkins, Denis Blount, Ivica Ico Bukvic, Charles Nichols, Michael Roan, and Tanner Upthegrove, "Genesis of the cube: The design and deployment of an hdla-based performance and research facility," *Computer Music Journal*, vol. 40, no. 4, pp. 62–78, 2016.

[7] A. Cabrera, J. Kuchera-Morin, and C. Roads, "The evolution of spatial audio in the allosphere," *Computer Music Journal*, vol. 40, no. 4, pp. 47–61, Dec 2016.

[8] T. Carpentier, N. Barrett, R. Gottfried, and M. Noisternig, "Holophonic sound in ircam's concert hall: Technological and aesthetic practices," *Computer Music Journal*, vol. 40, no. 4, pp. 14–34, Dec 2016.

[9] Fernando Lopez-Lezcano, "Searching for the grail," *Computer Music Journal*, vol. 40, no. 4, pp. 91–103, 2016.

[10] N. Barrett, "A musical journey towards permanent high-density loudspeaker arrays," *Computer Music Journal*, vol. 40, no. 4, pp. 35–46, Dec 2016.

[11] Zack Settel, Nicolas Bouillot, and Michal Seta, "Volumetric approach to sound design and composition using SATIE: a high-density 3D audio scene rendering environment for large multi-channel loudspeaker configurations," in *15th Biennial Symposium on Arts and Technology*, Ammerman Center for Arts and Technology at Connecticut College, New London, feb 2016, 8 pages.

[12] Zack Settel, Peter Otto, Michal Seta, and Nicolas Bouillot, "Dual rendering of virtual audio scenes for far-field surround multi-channel and near-field binaural audio displays," in *16th Biennial Symposium on Arts and Technology*, Ammerman Center for Arts and Technology at Connecticut College, New London, February 2018, 5 pages.

[13] M. Wright, "Open sound control 1.0 specification," Published by the Center For New Music and Audio Technology (CNMAT), UC Berkeley, 2002.

[14] Nicolas Bouillot, Zack Settel, and Michal Seta, "SATIE: a live and scalable 3d audio scene rendering environment for large multi-channel loudspeaker configurations," in *New Interfaces for Musical Expression (NIME'17)*, Copenhagen, Denmark, 2017.

[15] James McCartney, "Rethinking the computer music language: SuperCollider," *Computer Music Journal*, , no. 26, pp. 61–68, 2002.

[16] François U. Brien, Emmanuel Durand, Jérémie Soria, Michał Seta, and Nicolas Bouillot, "In situ editing (EiS) for fulldomes," in *23rd ACM Symposium on Virtual Reality Software and Technology (VRST)*, Gothenburg, Sweden, nov 2017.

[17] Matthew Wright, Adrian Freed, et al., "Open soundcontrol: A new protocol for communicating with sound synthesizers.," in *ICMC*, 1997.

MIDIZAP: CONTROLLING MULTIMEDIA APPLICATIONS WITH MIDI

Albert Gräf

IKM, Music-Informatics
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

ABSTRACT

The paper introduces midizap, a new Linux utility to interface MIDI controllers with multimedia applications such as audio and video editors or computer music programs. midizap is a heavily modified version of Eric Messick's ShuttlePRO program. Its purpose is to translate MIDI controller input to commands (either MIDI or X11 keyboard and mouse events) which the application understands. Configurations are simple text files, no programming skills are required. There's also an Emacs mode to help creating and testing these configurations. Jack session and MIDI patchbay functionality is available as well, making it easy to manage separate midizap instances for different controllers and applications.

1. INTRODUCTION

These days, MIDI controllers are typically USB class devices which can be connected to a Linux computer without requiring any special hardware or drivers. Also, they're often much cheaper than specialized gear for specific uses such as photo and video editing. So wouldn't it be nice if we could just use whatever MIDI controller we have for controlling our favorite multimedia applications? The problem is, while DAW and DJ programs typically have extensive and customizable MIDI interfaces built into them, other applications may not offer any MIDI support at all, or only recognize a particular set of MIDI messages. Thus we often have to translate the MIDI input from the controller to whatever keyboard or MIDI commands the application understands, and we'd like to be able to do this without having to modify the target application.

I was surprised to find that on Linux apparently there's no simple and practical solution for this problem yet. There is the Ctlra and Mappa software from the OpenAV project [1], but it is still under development and only readily supports a handful of devices and applications right now, which means that adding a new controller or application likely requires a fair amount of C programming. A popular commercial program in this realm is the Bome MIDI translator [2], but it's only available for Mac and Windows.

Another interesting utility is Eric Messick's ShuttlePRO program [3] which targets the Contour Design "Shuttle" devices [4] designed for video editing. These devices don't speak MIDI, but Messick's program is free (GPL) software, works on Linux, and includes the necessary code to recognize applications by their window name and translate device input to X11 keyboard and mouse events. Adding Jack MIDI support to it seemed to be a piece of cake, so that's what I set out to do. The first result of this side project was a fork of the ShuttlePRO program which improves the original program in some ways and adds Jack MIDI output [5]. The next obvious step then was to replace the Shuttle input with Jack MIDI input, giving birth to the midizap program as it stands now [6].

In the following sections, we discuss midizap's most important features and some typical uses. For lack of space, this description is necessarily somewhat terse and incomplete, but should give the interested reader an idea of what capabilities the program offers and when you might want to use it. More details can be found on the Github project page or in midizap's extensive manual.

2. TRANSLATION SYNTAX

As with the ShuttlePRO program, midizap's configuration is a simple text file which is divided into sections for different applications. A sample configuration is provided in /etc/midizaprc, you can copy this to create a .midizaprc file in your home directory and edit it there as needed. You can also run midizap with any other configuration file by specifying the name of the file on the command line. A collection of configurations for various purposes (mostly Mackie emulations for different devices) can be found in the examples folder in the sources.

The configuration language is line-oriented, each line is either a *section header* or a *translation rule*. The hash sign # at the beginning of a line or after whitespace starts a comment. Each section starts with a header of the following form, specifying a section name and a regular expression pattern:

[*name*] *pattern*

The section name is only used in diagnostic messages and can essentially be chosen freely. It is the regular expression pattern which actually determines whether the translations in the section are active at any given time. To these ends, midizap matches the pattern against the `WM_CLASS` and `WM_NAME` properties of the currently selected X application window. The latter is what is actually visible in the window title, while the former is an internal property which identifies the type of application window.[1] The regular expression pattern can also be omitted, in which case the translations will always be active. Such "default" sections are to be placed near the end of the file, and their translation rules will be used as fallback translations when none of the other translation sections in the configuration match the selected application window.

The section header is followed by a list of (zero or more) *translation rules* describing the translations which should be active for the given application. These just list MIDI messages and their translations in a human-readable symbolic format. Each translation rule must be on a line by itself and consists of a single left-hand side symbol denoting the MIDI message to be translated, followed by the right-hand side which is a list of zero or more symbols specifying the MIDI messages and/or keyboard and mouse commands to be output. It thus takes the following general form:

input $output_1$ $output_2 \ldots$

[1] You can find out about the `WM_CLASS` and `WM_NAME` properties of a window with the xprop program, or invoke midizap with the `-dr` debugging option to have it print this information. midizap will try to match both by default, but you can tell it explicitly to only match class or title by prefixing the pattern with the `CLASS` or `TITLE` token, respectively.

Here is a simple example:

```
[Terminal] CLASS ^(.*-terminal.*|konsole|xterm)$
F5    XK_Up
F#5   "pwd"
G5    XK_Down
G#5   "ls"
A5    XK_Return
```

This defines a list of translations for some common types of terminal windows, as specified in the section header on the first line. The input messages are listed on the left and the corresponding keyboard output on the right. Here we map a few notes in the middle octave to the cursor up and down and return keys, as well as some frequently used shell commands. The bindings above will let you operate the shell from your MIDI keyboard when the keyboard focus is on a terminal window. To make this work, you'll first have to connect your MIDI controller to midizap's MIDI input port, e.g., using a Jack MIDI patchbay program like QjackCtl. You then click on the desired terminal window and start entering notes on the MIDI keyboard to have the corresponding commands sent to the selected window.

It is important to note here that, like the ShuttlePRO program, midizap will only ever send keyboard and mouse commands to the *currently selected window* or, more precisely, the window which has the keyboard focus. The selected window also determines which section of translation rules is currently active. Thus you have to make sure that you first click on the right application window before you can go on sending keyboard and mouse commands to it. (In contrast, MIDI commands can be sent to any application as long as it is connected to midizap's MIDI output, see below.)

Let's now have a closer look at the syntax of translation rules. The precise syntax is a bit intricate, so we have to refer the reader to the EBNF grammar in Appendix A for details. But we will try to at least sketch out the most important elements in what follows. The first token of a translation rule (the left-hand side) denotes the MIDI message to be translated, which is followed by an output sequence (the right-hand side) consisting of MIDI messages or X key and mouse events. There can be any number of these, and you can freely mix MIDI messages and X events on the output side.

The `XK` symbols indicate X key codes and must be denoted exactly as they appear in the /usr/include/X11/keysymdef.h file. A string enclosed in double quotes is simply a shorthand for a sequence of X key events.[2] Besides the key codes from the header file, there are also some special tokens to denote mouse button and scroll wheel events (`XK_Button_1`, `XK_Scroll_Up`, etc.).

MIDI note messages are denoted in a symbolic format that will be familiar to musicians: a note letter (`A` to `G`) is followed by an optional accidental (`#` or `b`) and an octave number. By default, `C5` denotes middle C, but the octave numbering can be changed with a directive in the configuration file. Other kinds of (non-system) MIDI messages are denoted using short mnemonics: `KP:`*note* (aftertouch a.k.a. key pressure for the given note); `CC`*n* (control change for the given controller number); `PC`*n* (program change for the given program number); `CP` (channel pressure); and `PB` (pitch bend). These can all be followed by a dash and the MIDI channel (the default MIDI channel being 1).

In the example above, all note messages are interpreted as key events, having an "on" and "off" status: the key goes "down" when a note-on message is received, and goes "up" again when the corresponding note-off message (or a note-on with zero velocity) arrives. We also call this a *key translation*. These work in the same way as in the ShuttlePRO program; e.g., in the above example, the `XK_Up` key is pressed when the note-on for `F5` is received, and won't be released until the corresponding note-off is detected. If there's more than one key in the output sequence, as with the double-quoted strings in the example, each key will normally be released before the next one is pressed, and only the last key in the sequence will be held until the note-off is received. There are also some special suffixes for key specifications (`/D`, `/U`, `/H`) which indicate keys to be held and released explicitly or at the end of the sequence; we refer the reader to the documentation for details.

As another, more practical example, here are some bindings for the Kdenlive and Shotcut video editors mapping some keys and the big jog wheel on a Mackie-compatible device to some common video editing functions:

```
[Kdenlive/Shotcut] CLASS ^(shotcut|kdenlive)$

# playback controls
A#7 XK_space    # Play/Pause
A7  "K"         # Stop
G7  "J"         # Rewind
G#7 "L"         # Forward

# replace/drop (sets in and out points)
D#7 "I"         # Set In
E7  "O"         # Set Out

# left/right cursor movement
D8  XK_Home     # Beginning
D#8 XK_End      # End

# the jog wheel moves left/right by single frames
CC60< XK_Left   # Frame reverse
CC60> XK_Right  # Frame forward
```

The last two rules for the jog wheel show an example of a *data translation* which translates incremental changes in the extra data byte of a message to corresponding X key presses. For ordinary (absolute) control changes these take the form `CC`*n*`-` and `CC`*n*`+`, where *n* denotes the controller number, and the `-` or `+` flag the direction of the change. However, here we employed the special `<` and `>` suffixes which indicate a *relative* change in "sign-bit" encoding [7], which is commonly used with encoders (knobs or wheels which can be turned endlessly in either direction). In either case, the up or down output sequence is emitted for each unit change in the parameter. You can also scale these responses by adding suitable step sizes on the left-hand or right-hand side of the translation rules; again we refer the reader to the documentation for details.

The rules we've seen so far all translate MIDI to X key events. midizap can also work as a MIDI mapper which translates MIDI input to MIDI output. This is useful if the target application supports MIDI, but needs the controller input to be remapped to MIDI commands it understands. The following example lets you play a little drumkit on a General MIDI (GM) synthesizer like Fluidsynth, by remapping some of the white keys in the 4th octave to a few drum notes on MIDI channel 10 (the GM drum channel). We also threw in a rule to remap the modulation wheel (`CC1`) to the volume controller on MIDI channel 10 (`CC7-10`).[3]

[2] In the current implementation, this only works with printable ASCII characters which can be mapped 1-1 to X11 key codes. Otherwise explicit key codes must be used.

[3] The notation `CC1=` being used here provides a shorthand for two data

```
[MIDI]
C4    C3-10
D4    C#3-10
E4    D3-10
F4    D#3-10
CC1=  CC7-10
```

Note that we placed the MIDI translations into a special `[MIDI]` section here. This is a default section reserved for applications accepting MIDI input. To make this work, you will have to invoke the midizap program with the `-o` option. This enables the `[MIDI]` section and equips midizap with an additional MIDI output port which can be connected to the target application (like Fluidsynth in this example). As long as your translations only output MIDI messages, you then don't have to worry about keyboard focus, as the application will receive all data from midizap through the MIDI connection (in fact the application does not need to have any X window at all in this case).

The above example does a simple 1-1 mapping of MIDI events, but in general the output sequence may consist of as many MIDI messages of as many different types as needed, and you can also mix MIDI and X keyboard and mouse output if you want. An interesting use case for MIDI translations is Mackie emulation which we'll discuss in Section 4.

3. GETTING STARTED

Before we explore some of midizap's more advanced features, let us quickly go over the mundane technicalities of using midizap.[4] midizap is a command line application, so you typically run it from the terminal. However, it is also possible to launch it from your Jack session manager (see Section 5 below) or from your desktop environment's startup files once you've set up everything to your liking. In addition, for Emacs users there's a midizap mode which makes it very easy to edit and test your midizap configurations. It does syntax highlighting, auto-completion of keywords, and also lets you launch midizap in an Emacs buffer; please check the midizap-mode.el file in the sources for details.

midizap uses Jack for its MIDI input and output, so you'll need to be familiar with Jack. We recommend using a Jack front-end like QjackCtl which makes setting up Jack and doing MIDI connections much easier. You'll also need an ALSA-Jack MIDI bridge in order to expose the ALSA sequencer ports as Jack MIDI ports, so that the MIDI inputs and outputs of your controller and other non-Jack MIDI applications can be connected to midizap. Jack's built-in bridge will work for this purpose (in the QjackCtl setup, select `seq` as the MIDI driver), or you can use Nedko Arnaudov's a2jmidid utility [8]. The latter is easier to use with Jack2, and will work with Jack1 as well.

Running just `midizap` without any arguments launches midizap with the default configuration and a single Jack MIDI input port which you'll have to connect to your MIDI controller. To utilize MIDI output, run `midizap -o`; as already mentioned, this equips midizap with an additional Jack MIDI output port to be connected to the MIDI application you wish to control. You can also run midizap with any other configuration file by simply specifying the name of the file on the command line. There are a number of other options and configuration file directives which let you set the Jack client name, number of input and output ports and the desired MIDI connections; see Section 5.

Moreover, midizap offers a fair amount of debugging options which will be very helpful when you start developing your own configurations. A good set of options to start with is `-drkm`; `r` prints the class names and titles of selected windows which is useful to determine which regular expressions to use in the section headers; `k` prints out recognized translations so that you can check that midizap is actually picking the right translation rules for some given MIDI input; and `m` activates midizap's built-in MIDI monitor which prints out recognizable MIDI input in the same syntax that's used in the configuration file, which makes it easy to figure out which MIDI messages you may want to create translations for.

The default configuration is really just an example, to help you get started. You can either edit that file or create your own configuration. To start from a clean slate, create an empty file in a text editor, say myconfig.midizaprc, and invoke midizap on it. The file will be reloaded whenever you save it, so you can just keep on adding translation sections and rules and try them out immediately, without having to restart the program. If you're an Emacs user, you will find midizap's Emacs mode most convenient to do all this.

Let's walk through a simple example to show how this works. We'll use the Shotcut video editor (`https://www.shotcut.org/`) for illustration, so let's assume that you've already launched Shotcut and loaded a video file in it. Next, make sure that Jack is running, create the myconfig.midizaprc file, run `midizap -drkm myconfig .midizaprc`, and connect your controller to midizap's MIDI input. With the Shotcut window selected, wiggle one of the controls on your MIDI gear; I'll take the modulation wheel as an example. In midizap's output you should now see something like:

```
Loading configuration: myconfig.midizaprc
[0] CC1-1 value = 40
no translation found for Untitled - Shotcut
(class shotcut)
```

This tells you the class name (`shotcut`) of the application window, as well as the name of the incoming MIDI message (`CC1-1`, which can also be abbreviated as `CC1` in the configuration, as 1 is the default MIDI channel). Having identified the application and the MIDI message we'd like to translate, we can now edit our configuration in the myconfig.midizaprc file accordingly. Let's add the following section header and translations, and save the file:

```
[Shotcut] CLASS ^shotcut$
CC1- XK_Left
CC1+ XK_Right
```

midizap should automatically reload the file. Moving the modulation wheel again (with the Shotcut window still selected) will now change the playback position in Shotcut, while the translations we just added are printed by midizap.

4. ADVANCED USES

One particularly interesting use case for MIDI translations is the emulation of Mackie controllers. The *Mackie control protocol* (MCP) has become a de facto standard for DAW programs, because it allows the various track parameters to be mapped without requiring any manual setup.[5] Also, many Mackie-compatible devices offer *feed-*

translation rules `CC1-` and `CC1+` with the same right-hand side `CC7-10`.

[4] We don't discuss installation here, which is very easy and, besides the X libraries, only needs very few dependencies which should be readily available on all Linux distributions; details can be found in the README file.

[5] Although MCP is widely used, there doesn't seem to be a publicly accessible specification of the protocol anywhere. A partial description can be found at `http://www.jjlee.com/qlab/MackieControlMIDIMap.pdf`.

back, i.e., the ability to display current parameter values and other kinds of status information using LEDs, motor faders, scribble strips and the like, which makes them very convenient to use.

Some MIDI controllers have a built-in MCP mode, but many don't. Thus it is tempting to employ midizap to emulate this mode. Even if a device already offers MCP, it may be lacking some features; this is true especially for some of the cheaper and/or smaller devices like the Behringer X-Touch Mini. In such cases midizap may be used to beef up the device's capabilities and/or modify its bindings so that they better suit your workflow.

Emulating MCP usually requires remapping some or all of the MIDI messages of the device, on both input and output (if the device offers some feedback capabilities). Especially the feedback part often poses some challenges. The purpose of this section is to dive into some of midizap's more advanced features catering to these use cases, using MCP emulation as a running example. Of course, these features may also be helpful in other situations calling for complicated translations.

4.1. Shift State

One issue we often face right away when designing a Mackie emulation is the number of available controls. For instance, your device might only provide you with 8 faders which must then be used to emulate both the volume and the panning controls of a Mackie controller. Or it may not have enough buttons for all the special MCP functions that you need. In such cases it is useful to designate a special shift key on the device which lets you switch between different functions of the available controls.

midizap provides a special `SHIFT` token for this purpose which can be used anywhere on the right-hand side of a translation. This token doesn't produce any output, it merely toggles an internal bit indicating the current shift status. This is often used in a key translation as follows:

```
D8 SHIFT
```

Now, midizap will go into shift mode whenever the device generates the note `D8` (which happens to be the shift key on an AKAI APCmini device, cf. Fig. 1(8); but any available button-like control will do). Pressing the `D8` key again disables shift mode. Thus the above rule implements a "CapsLock"-style shift button. You can also do an ordinary shift button as follows:

```
D8 SHIFT RELEASE SHIFT
```

Here, the `RELEASE` token indicates an explicit release sequence which will be invoked as soon as the `D8` key is released (i.e., the corresponding note-off is received). Hence pressing this key now toggles on the shift status, and releasing it immediately toggles it off again, just like an ordinary shift key on a computer keyboard.

Having defined the shift key, we can now use its current status in other translations. The `^` character, when used as a prefix on the left-hand side of a translation, tells midizap that the translation should only be valid in shifted state. Thus we can now have two different rules associated with each incoming MIDI message, depending on the current shift status, effectively giving us about twice as many controls as we had before.

Let's take the AKAI APCmini as an example again. We can map the first eight faders `CC48` to `CC55` on this device, cf. Fig. 1(4), to the MCP encoders `CC16` to `CC23` in shifted mode as follows:[6]

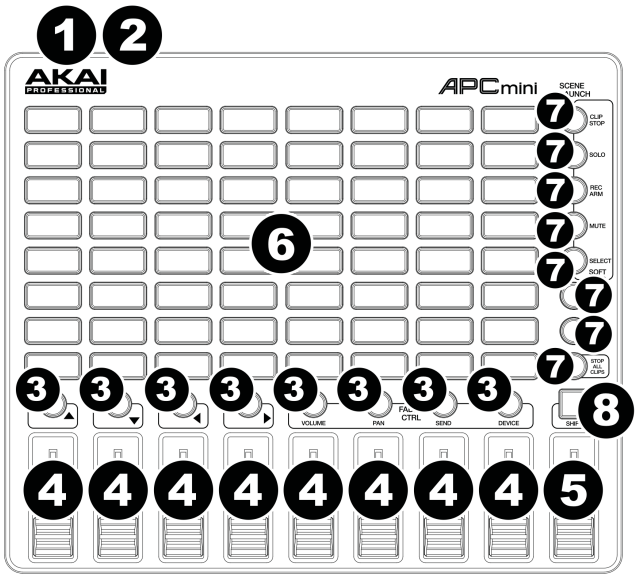


Figure 1: AKAI APCmini [9, p. 5].

```
^CC48= CC16~
^CC49= CC17~
...
^CC55= CC23~
```

The above translations will only be executed in shifted mode (i.e., by holding the designated shift key while operating the faders). In unshifted mode, the faders are still available to be mapped, e.g., to the MCP volume controls (`PB-1` to `PB-8`). For instance:[7]

```
CC48= PB[128]-1
CC49= PB[128]-2
...
CC55= PB[128]-8
```

You will find very similar rules in the APCmini.midizaprc example distributed with midizap. We've only sketched out the use of a single shift key here, but midizap actually supports up to four different shift states, which are denoted `SHIFT1` to `SHIFT4`, with the corresponding prefixes being `1^` to `4^`. The `SHIFT` token and `^` prefix we've seen above are in fact just shortcuts for `SHIFT1` and `1^`, respectively. Thus midizap lets you have up to five different "layers" of MIDI assignments (1 unshifted and 4 shifted states), which will hopefully be enough for most purposes.

4.2. Feedback

Some MIDI controllers have motor faders, LEDs, etc., requiring feedback from the application. To accommodate these, you can use the `-o2` option of midizap (or the `JACK_PORTS 2` directive in the midizaprc file, cf. Section 5), to create a second pair of MIDI input and output ports. Use of this option also activates a second MIDI default section in the midizaprc file, labeled `[MIDI2]`, which

[6] Note that the MCP encoders use relative values in sign-bit encoding; the ~ suffix on the output `CC` messages indicates that these messages should be converted to that special encoding.

[7] The `[128]` suffix on the `PB` output messages denotes a scale factor here, which scales up the 7 bit `CC` range to the 14 bit range of a pitch bend.

is used exclusively for translating MIDI input from the second input port and sending the resulting MIDI output to the second output port. The control output from the application is then connected to midizap's second input port, and midizap's second output port to the input of the controller, so that the feedback from the application passes through midizap on its way back to the controller.

If all this has been set up properly, MIDI feedback will eliminate most problems with controls being out of sync with the application. midizap has some built-in logic to help with this. Specifically, the current state of controls received from the host application via the second input port will be recorded, so that subsequent MIDI output for data translations on the first output port will use the proper values for determining the required relative changes. We refer to this as *automatic feedback*. Some devices may provide you with sign-bit encoders which don't need any kind of feedback for themselves. In this case the automatic feedback will be all that's needed to keep controller and application in sync, and you don't even have to write any translation rules for the feedback; just enabling the second input port and hooking it up to the application will be enough.

Other controls such as motor faders will require explicit translation rules for the feedback in the `[MIDI2]` section, however. In the simplest case these may just be the inverse of the rules in the `[MIDI]` section. For instance, if the APCmini had motor faders (it doesn't), we might use rules like the following to translate MCP feedback about the fader positions back to the device:

```
PB[128]-1= CC48
PB[128]-2= CC49
...
PB[128]-8= CC55
```

Translations can also generate their own feedback. To these ends, any MIDI message on the right-hand side of a translation can be prefixed with the `!` character (or the `^` character, which works in an analogous fashion, but has some special logic for dealing with shift keys built into it). This outputs the message as usual, but flips the output ports, so that the message will go to port 2 in a forward translation destined for port 1, and vice versa to port 1 in a feedback translation (in the `[MIDI2]` section) destined for port 2. We call this *direct feedback*. For instance, we can equip the `D8` shift key from the previous subsection with direct feedback as follows:

```
D8 SHIFT ^D8 RELEASE SHIFT ^D8
```

This might then light up the LED of the corresponding button when pressing and turn it off again when releasing the key.

Please note that any kind of controller feedback which goes beyond direct feedback requires that the target application already provides some level of MIDI feedback on its own. midizap is *not* capable of reading the internal state of a non-MIDI application by some other magical means.

4.3. Mod Translations

Most of the time, MIDI feedback uses just the standard kinds of MIDI messages readily supported by midizap, such as note messages which make buttons light up in different colors, or control change messages which set the positions of motor faders. However, there are some encodings of feedback messages which combine different bits of information in a single message, making them difficult or even impossible to translate using the simple kinds of rules we've seen so far. midizap offers a special variation of data translations to help decoding such messages. We call them *mod translations* (a.k.a. "modulus" translations), because they involve operations with integer moduli which enable you to both calculate output from input values in a direct fashion, *and* modify the output messages themselves along the way.

One important task, which we'll use as an example below, is the decoding of meter (RMS level) data in the Mackie protocol. There, each meter value is represented as a channel pressure (`CP`) message whose value consists of a mixer channel index 0..7 in the "high nibble" (bits 4..6) and the corresponding meter value in the "low nibble" (bits 0..3). We will show how to map these values to notes indicating buttons on the AKAI APCmini (Fig. 1). Mod translations aren't limited to this specific use case, however; similar rules will apply to other kinds of "scrambled" MIDI data.

In its simplest form, a mod translation looks as follows (taking channel pressure as an example):

```
CP[16] C0
```

In contrast to the simple kinds of data translations we've seen so far, there's no increment (`+` or `-`) flag here, so the translation does *not* indicate an incremental change of the input value. Instead, mod translations always work with *absolute* values, and the step size on the left-hand side is treated as a *modulus* to decompose the input value into two separate quantities, *quotient* and *remainder*. Only the latter becomes the value of the output message, while the former is used as an offset to modify the output message.

In order to describe more precisely how this works, let's assume an input value v and a modulus k. We divide v by k, yielding the quotient (offset) $q = v \operatorname{div} k$ and the remainder (value) $r = v \bmod k$. E.g., with $k = 16$ and $v = 21$, you'll get $q = 1$ and $r = 5$ (21 divided by 16 yields 1 with a remainder of 5). The calculated offset q is then applied to the note itself, and the remainder r becomes the velocity of that note. So in the example above the output would be the note `C#0` (`C0` offset by 1) with a velocity of 5. On the APCmini, this message will light up the second button in the bottom row of the 8x8 grid in yellow.

Mod translations are midizap's swiss army knife for dealing with complicated translations. There are also some special elements in the MIDI syntax which can be used in mod translations to make them even more flexible:

- The *empty modulus* bracket, denoted `[]` on the left-hand side of a mod translation, indicates a default modulus large enough (16384 for `PB`, 128 for other messages) so that the offset q always becomes zero and the translation passes on the entire input value as is.
- The *transposition* flag, denoted with the `'` (apostrophe) suffix on an output message, reverses the roles of q and r, so that the remainder becomes the offset and the quotient the value of the output message.
- The *change* flag, denoted with the `?` suffix on an output message, only outputs the message if there are any changes in offset or value.
- *Value lists*, denoted as lists of numbers separated by commas and enclosed in curly braces, provide a way to describe *discrete mappings* of input to output values. The input value is used as an index into the list to give the corresponding output value, and the last value in the list will be used for any index which runs past the end of the list. There are also some convenient shortcuts which let you construct these lists more easily: repetition $a{:}b$ (denoting b consecutive a's) and enumeration

a-b (denoting $a, a \pm 1, \ldots, b$, which ramps either up or down depending on whether $a \leq b$ or $a > b$, respectively).

We can't go into all of this here, so we have to refer the reader once again to the manual for details. But here's how we can use a single mod translation to map MCP meter feedback onto the APCmini's topmost five button rows, turning them into a colorful meter display:

```
CP[16] C2{0,1} G#2{0:3,1} E3{0:6,1} C4{0:9,5} G#4{0:12,3}
```

To understand how this works, one must know that the buttons of the 8x8 grid, cf. Fig. 1(6), can be lit up by sending them the appropriate note messages. Rows number 4 to 8 (counting from the bottom) start at notes `C2`, `G#2`, `E3`, `C4` and `G#4`, respectively. The velocities of the notes indicate the colors (0 means off, 1 green, 5 yellow, and 3 red). The rule above will thus light up buttons in different rows in different colors (depending on the low nibble of the channel pressure value), and in different columns (depending on the high nibble of the channel pressure value which ranges from 0 to 7 and indicates the mixer channel).

Mod translations are surprisingly versatile and can be used for various different purposes. In particular, they can also be called as *macros* from other translations. This adds a (rather rudimentary) programming facility to the configuration language, which isn't needed very often, but gives you some extra rope to tackle complicated translations. We won't go into this here, so please check the manual for details and many more examples.

4.4. Pass-Through

There are some situations in which it may be possible to keep most of the controller input and pass it through unchanged. In particular, this case arises in Mackie translations for devices which already support MCP, but might need some minor touches here and there to make them work exactly the way you want.

For instance, Behringer's X-Touch Mini (Fig. 2) is a fairly nice device with its eight encoders providing LED feedback, but its MCP mode is somewhat lacking. One thing that many users of the device complain about is that it doesn't have any keys for changing mixer banks. But in fact the device *has* two "layer" keys on the right which seem ideal for that purpose; alas, the Behringer engineers decided to have them assigned to some other less important MCP functions instead. With midizap it's very easy to fix this shortcoming, by just reassigning the two keys to the much wanted bank change keys:

```
C7   A#3 # BANK LEFT
C#7  B3  # BANK RIGHT
```

We still need to make sure that everything else is passed through unchanged. The most convenient way to do this is to just add the `PASSTHROUGH` directive to the configuration. You can place this anywhere, but it's most convenient to have this kind of stuff at the beginning of the configuration file, before the first translation section. The directive tells midizap to pass a message from the input to the output port if it doesn't have an explicit translation for that message. So the final configuration will look like this:

```
PASSTHROUGH

[MIDI]
C7   A#3 # BANK LEFT
C#7  B3  # BANK RIGHT

[MIDI2]
```

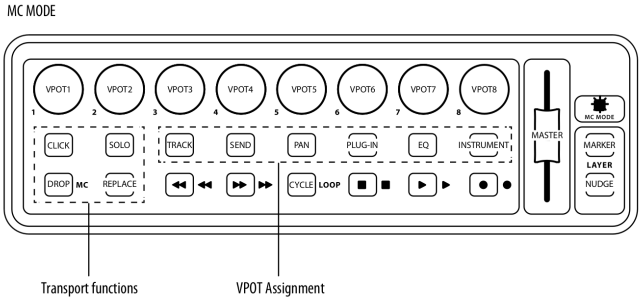


Figure 2: X-Touch Mini [10, p. 16].

```
# feedback for the BANK LEFT/RIGHT buttons
A#3  C7
B3   C#7
```

Here we also added two more translations in the `[MIDI2]` section so that the feedback for the two remapped buttons works as expected. To finish off that little example, you may want to add a few more directives, so that midizap automatically creates the feedback port and auto-connects to the right device and applications; we will discuss these in the next section. You can also find an enhanced version of this example in the sources (XTouchMini.midizaprc), which adds many other useful MCP functions.

Please note that the `PASSTHROUGH` directive only applies to normal (non-system) messages. In some cases it will be necessary to also pass on system messages, such as system exclusive, which can be done with the `SYSTEM_PASSTHROUGH` directive. System exclusive messages are used in MCP to set the contents of the scribble strips. The X-Touch Mini doesn't have these, but other devices like the X-Touch One do, and will thus need system pass-through to function properly (see the XTouchONE.midizaprc example in the sources).

5. JACK INTERFACE

There are some additional directives and corresponding command line options to configure midizap's Jack setup in various ways. If both the command line options and directives in the midizaprc file are used, the former take priority, so that it's possible to override the configuration settings from the command line. Note that all these options can only be set at program startup. If you later edit the corresponding directives in the configuration file, the changes won't take effect until you restart the program.

5.1. Client Setup

The `-j` option and the `JACK_NAME` directive change the Jack client name from the default (`midizap`) to whatever you want it to be. To use this option, simply invoke midizap with `-j` followed by the desired client name, or put a directive like the following into your midizaprc file:

```
JACK_NAME "midizap-XTouchMini"
```

This option is useful, in particular, if you're running multiple instances of midizap with different configurations for different controllers and/or target applications, and you want to have the corresponding Jack clients named differently, so that they can be identified more easily.

We've already seen the `-o` option which is used to equip the Jack client with an additional output port. This can also be achieved with the `JACK_PORTS` directive in the midizaprc file, as follows:

```
JACK_PORTS 1
```

The given number of output ports must be 0, 1 or 2. Zero means that MIDI output is disabled (which is the default). You may want to use `JACK_PORTS 1` if the configuration is primarily aimed at doing MIDI translations, so you'd like to have MIDI output enabled by default. `JACK_PORTS 2` or the `-o2` option indicates that *two* pairs of input and output ports are to be created. As already discussed in Section 4, the second port is typically used to deal with controller feedback from the application.

Not very surprisingly, at least one output port is needed if you want to output any MIDI at all; otherwise MIDI messages on the right-hand side of translations will be silently ignored.

5.2. MIDI Connections

Setting up all the required connections for the Jack MIDI ports can be a tedious and error-prone task, especially if you have to deal with complex setups involving feedback and/or multiple midizap instances. It's possible to automatize the MIDI connections, e.g., with QjackCtl's persistent MIDI patchbay facility, but this is often inconvenient if you need to accommodate multiple midizap configurations and you already have a complicated studio setup (or indeed a bunch of them) which you don't want to mess with.

Therefore midizap offers its own built-in patchbay functionality using the `JACK_IN` and `JACK_OUT` directives which let you specify the required connections in the configuration itself. The port number is tacked on to the directive, so, e.g., `JACK_IN2` connects the second input port. If the port number is omitted then it defaults to 1, so both `JACK_OUT1` and just `JACK_OUT` connect the first output port. The directive is followed by a regular expression to be matched against the Jack MIDI ports of your devices and applications. For instance, the following lines connect midizap to an X-Touch Mini device on one side and Ardour's Mackie control port on the other. (This kind of setup is rather typical for configurations involving feedback. For simple setups just specifying the `JACK_IN` and `JACK_OUT` directives is often sufficient, or even just `JACK_IN` if the target application isn't MIDI-capable.)

```
JACK_IN1  X-TOUCH MINI MIDI 1
JACK_OUT1 ardour:mackie control in
JACK_IN2  ardour:mackie control out
JACK_OUT2 X-TOUCH MINI MIDI 1
```

A connection will be established automatically by midizap whenever a MIDI port belonging to another Jack client matches the regular expression, as well as the port type and I/O direction. This also works dynamically, as new devices get added and new applications are launched at runtime. Only one directive can be specified for each port, but since midizap will connect to all ports matching the given regular expression, you can connect to more than one application or device by just listing all the alternatives. For instance, to have midizap's output connected to both Ardour and Pd, you might use a directive like:

```
JACK_OUT1 ardour:MIDI control in|Pure Data Midi-In 1
```

All matches are done against full port names including the *client-name*: prefix, so you can specify exactly which ports of which clients should be connected. However, note that in contrast to the QJackCtl patchbay, midizap does substring matches by default, so that, e.g., just "`MIDI control`" would match *any* Ardour MIDI control port, in any instance of the program (and also ports with the same name in other programs). If you want to specify an exact match, you need to use the `^` and `$` anchors as follows:

```
JACK_OUT1 ^ardour:MIDI control in$
```

5.3. Jack Sessions

midizap also supports Jack session management which provides a convenient alternative way to launch your midizap instances. Once you've finished a configuration, instead of running midizap manually each time you need it, you just invoke it once with the right command line options, and use a Jack session management program to record the session. The session manager can then be used to relaunch the program with the same options later.

Various Jack session managers are available for Linux, but if you're running QjackCtl already, you might just as well use it to record your sessions, too. QjackCtl's session manager is available in its Session dialog. To use it, launch midizap and any other Jack applications you want to have in the session, and then hit the "Save" button in the Session dialog to have the session recorded. Now, at any later time you can rerun the recorded session with the "Load" button in the same dialog, and your most recent sessions are available in the "Recent" menu from where they can be launched quickly.

6. CONCLUSIONS

I hope that you'll enjoy using midizap for your MIDI mapping needs as much as I do. I'd like to emphasize, however, that midizap is nothing more (and nothing less) than a simple and practical solution to a nagging problem that I have run into time and again (as presumably many Linux MIDI users do). midizap has its limitations, and it is definitely *not* intended as a replacement for more ambitious projects. Ctlra [1] along with its Mappa component takes a much higher-level approach based on the idea of abstracting device interfaces so that basically any Ctlra client can be used with any Ctlra-supported device. This promises to scale much more easily, but it will take its time to gather a critical mass of supported devices and applications.

In the meantime we now have midizap which is a much more modest design, but can make any MIDI controller work with pretty much any application out there, as long as the application can be controlled with keyboard and/or MIDI commands. And you don't need to be a computer expert to use it; if you know how to use Jack, a text editor, and the command line, you're good to go.

Contributions are welcome; in particular, we're looking for interesting configurations to be included in the distribution. I consider midizap itself finished at this point (ports, bugfixes and feature creep notwithstanding), but one area which could still be simplified is the configuration process. While experienced Linux users may actually prefer the textual interface that midizap provides (especially when using midizap's Emacs mode), editing configuration files and watching debugging output in a terminal can be a bit daunting. So a GUI-based configuration front-end (maybe something along the lines of existing MIDI learn facilities) might be in order here.

As Ctlra matures, another interesting possibility is to have a direct interface between Ctlra and midizap at some point. It's already possible to run midizap and Ctlra's daemon program in concert, but tighter integration could be achieved, e.g., by adding a Ctlra backend to midizap.

7. ACKNOWLEDGMENTS

This program wouldn't exist without Eric Messick's prior work, so a big thank you goes out to him. Thanks are also due to Harry van Haaren for helping me with Ctlra, which is used in the NI Maschine Mk3 Mackie emulation distributed with midizap. Last but not least, I'd also like to thank the reviewers for helpful comments.

8. REFERENCES

[1] Harry van Haaren, "openAV-Ctlra: A plain C library to program with hardware controllers," `https://github.com/openAVproductions/openAV-Ctlra`, Sept. 2018.

[2] "MIDI Translator Pro | Bome Software," `https://www.bome.com/products/miditranslator`.

[3] Eric Messick, "ShuttlePRO: User program for interpreting key, shuttle, and jog events from a Contour Design ShuttlePRO v2," `https://github.com/nanosyzygy/ShuttlePRO`, Sept. 2018.

[4] Paul White, "Contour Designs Shuttle Pro v2," `https://www.soundonsound.com/reviews/contour-designs-shuttle-pro-v2`, Oct. 2016.

[5] Albert Gräf, "ShuttlePRO fork," `https://github.com/agraef/ShuttlePRO`, Sept. 2018.

[6] Albert Gräf, "midizap: Control your multimedia applications with MIDI," `https://github.com/agraef/midizap`, Oct. 2018.

[7] "Binding Jump Prevention and Relative (Rotary) Encoder Support - Cantabile - Software for Performing Musicians," `https://www.cantabilesoftware.com/guides/controllerEncoding`.

[8] Nedko Arnaudov, "a2jmidid," `https://repo.or.cz/a2jmidid.git`.

[9] AKAI Professional, "APC mini - User Guide," `http://www.akaipro.com/products/pad-controllers/apc-mini`.

[10] Behringer, "X-TOUCH MINI Quick Start Guide," `https://media.music-group.com/media/PLM/data/docs/P0B3M/X-TOUCH%20MINI_QSG_WW.pdf`.

A. CONFIGURATION SYNTAX

```
config     ::= { directive | header | translation }
header     ::= "[" name "]" [ "CLASS" | "TITLE" ] regex
translation::= midi-token { key-token | midi-token }

directive  ::= "DEBUG_REGEX"|"DEBUG_STROKES"|"DEBUG_KEYS" |
               "DEBUG_MIDI" | "MIDI_OCTAVE" number |
               "JACK_NAME" string | "JACK_PORTS" number |
               "JACK_IN" [number] regex |
               "JACK_OUT" [number] regex |
               "PASSTHROUGH" [ number ] |
               "SYSTEM_PASSTHROUGH" [ number ]

midi-token ::= msg [ mod ] [ steps ] [ "-" number] [ flag ]
msg        ::= ( note | other | "M" ) [ number ]
note       ::= ( "A" | ... | "G" ) [ "#" | "b" ]
other      ::= "CH" | "PB" | "PC" | "CC" | "CP" | "KP:" note
mod        ::= "[" [ number ] "]"
steps      ::= "[" number "]" | "{" list "}"
list       ::= number { "," number | ":" number | "-" number }
flag       ::= "-" | "+" | "=" | "<" | ">" | "~" |
               "'" | "?" | "'?" | "?'"

key-token  ::= "RELEASE" | "SHIFT" [ number ] |
               keycode [ "/" keyflag ] | string
keycode    ::= "XK_Button_1" | "XK_Button_2" | "XK_Button_3" |
               "XK_Scroll_Up" | "XK_Scroll_Down" |
               "XK_..." (see /usr/include/X11/keysymdef.h)
keyflag    ::= "U" | "D" | "H"
string     ::= '"' { character } '"'
```

SC-HACKS: A LIVE CODING FRAMEWORK FOR GESTURAL PERFORMANCE AND ELECTRONIC MUSIC

Iannis Zannos

Department of Audiovisual Arts
Ionian University, Corfu, Greece
zannos@gmail.com

ABSTRACT

This paper presents a library for SuperCollider that enables live coding adapted to two domains of performance: telematic dance with wireless sensors and electroacoustic music performance. The library solves some fundamental issues of usability in SuperCollider which have been also addressed by the established live-coding framework JITLib, such as modifying synth and pattern processes while they are working, linking control and audio i/o between synths, and generation of GUIs. It offers new implementations, which are more compact and easy to use while emphasizing transparency and scalability of code. It introduces binary operators which when coupled to polymorphism facilitate live coding. Several foundation classes are introduced whose purpose is to support programming patterns or commonly used practices such as the observer pattern, function callbacks and system-wide object messaging between language, server processes and GUI.

The use of the library is demonstrated in two contexts: a telematic dance project with custom low-cost movement sensors, and digital implementations of early electroacoustic music scores by J. Harvey and K. Stockhausen. The latter involves coding of a complex score and generation of a GUI representation with time tracking and live control.

1. BACKGROUND

1.1. Bridging Live Coding and Gestural Interaction

The performance practice known as *live coding* emerged from the ability of software to modify state and behavior through the interactive evaluation of code fragments and to synthesize audio at runtime. As a result, several programming environments and technologies supporting live coding have been developed in the past 20 years, such as *SuperCollider*[1], *Impromptu*[2], *ChucK*[3] , *Extempore*[4], *Gibber*[5], and others. It has been noted, however, that such environments and practices suffer from a lack of immediacy and those visible gestural elements that are traditionally associated with live performance [6]. Recent research projects attempt to re-introduce gestural aspects or to otherwise support social and interactive elements in musical performance using technologies associated with live coding ([7], [8], [9], [10]). Amongst various types of gestural interaction, dance is arguably the one least related to textual coding. Few recent studies exist which prepare the field for bridging dance with coding ([11]). The challenges in this domain can be summarized as the problem of bridging the symbolic domains of dance and music notation and the subsymbolic numerical domain of control data streams input from sensors. This also implies translating between continuous streams of data and individual timed events, possibly tagged with symbolic values. This is a technologically higly demanding task which is subject of research in various gestural interface applications. The work related in the present paper represents an indirect and bottom-up approach to the topic, based on DIY and open source components and emphasizing transparency and self-sufficiency at each step. It does not address the task of gesture recognition, but rather it aims at supporting live coding in conjunction with dancers and instrumental performers. Ongoing experiments together with such performers, are helping to identify low-level tasks and features which are essential for practical work. This type of work is purely empirical, and tries to identify useability criteria purely from practice, rather than to develop features that are inferred from known interaction paradigms in other related domains. At this stage of the project it is still too early to formulate conclusions from these experiments. Instead, this paper concentrates on the fundamentals of the implemenation framework on which this work is based. These are readily identifiable and their potential impact on further development work as well as experiments are visible. This paper therefore describes the basic principles and design strategy of the *sc-hacks* library, and discusses its perceived impact on performances. Finally, it outlines some future perspectives for work involving data analysis and machine learning.

1.2. Live Coding Frameworks in SuperCollider

1.2.1. Types of Live Coding Frameworks

Live Coding libraries can be divided into two main categories depending on the level of generality of their implementation and their application scope. First, there are libraries which extend *SuperCollider* usage in order to simplify the coding of very behaviors or features which are very common in performance, but are otherwise inconvenient to code in *SuperCollider*. To this category belongs the JITLib framework. JITLib (Just-In-Time programming Library) has been around since at least August 2006, with an early version since ca 2000 [1] and is very widely used in the community, being the de-facto go-to tool for live coding in *SuperCollider*. The second category consists of libraries that concentrate on specialized usage scenarios and attempt to create domain-specific mini-languages for those scenarios on top of *SuperCollider*. Such are: *IXI-Lang* (a sequencer / sample playing mini-language by Thor Magnusson [12]), *SuperSampler* (a polyphonic concatenative sampler with automatic arrangement of sounds on a 2-dimensional plane, by Shu-Cheng Allen Wu [13]), and *Colliding* (An "environment for synthesis-oientd live coding", simplifying the coding of Unit Generator graphs, by Gerard Roma [14]). Finally, *TidalCycles* by Alex McLean [15] should be mentioned, which develops its own live coding language based on Haskell and focussing on the coding of com-

[1] See https://swiki.hfbk-hamburg.de/MusicTechnology/566 (accessed 20-December-2018)

plex layers of synchronized beat cycles with sample playback and synthesis, and uses the SuperCollider synthesis server as audio engine.

1.2.2. sc-hacks *Objectives and Approach*

sc-hacks belongs to the first category of frameworks, and its initial motivation was partly to implement some of the solutions of JITLib in more robust, simple, and general ways. In parallel, inspiration from *ChucK*'s => operator led to the development of a minimal extension of the language based on 4 binary operators (`+>`, `<+`, `*>` `*<`), which, coupled with polymorphism, permit simplified and compact coding of several common sound-structure coding patterns. Furthermore, the implementation of some basic programming patterns [2] opened new possibilities for the creation of GUI elements which update their state. This led to a proliferation of GUI building and management facilities and resulted in several interfaces for live coding tasks, such as a code browser based on the concept of code snippets, a browser for editing and controlling the behavior of named players holding synth or pattern items, and shortcuts for building GUI widgets displaying values of parameters controlled by OSC, MIDI or algorithmic processes. Finally, ongoing experiments with dancers and instrumentalists are giving rise to new interface and notation ideas. The current focus is on building tools for recording, visualising and playback of data received from wireless sensors via OSC, in order to experiment with the data in performance, and to apply machine-learning algorithms on them.

2. APPROACH

2.1. Players and Player Environments

JITLib addresses four fundamental problems in coding for concurrent sound processes: (a) Use of named placeholders for sound generating processes, (b) managing the control parameters of processes in separate namespaces, (c) modifying event-generating algorithmic processes (known in SuperCollider as *Patterns*) on the fly and (d) interconnecting audio signals between inputs and outputs of synth processes. Sc-lib offers alternative solutions to these problems which present advantages, described in the following sections:

2.1.1. Named placeholders: -def classes vs. Player class

To use a name as placeholder for a synth process in order to start, stop or modify the process on the fly, JITLib introduces the *[X-]def* convention, i.e. it defines a number of classes which act as named containers for different types of processes (Synths: `Ndef`, Tasks: `Tdef`, Patterns `Pdef`, etc.). Sc-hacks uses a single `Player` object class instead. A Player instance can play a Synth or a Pattern depending on the type of source which it is asked to play, i.e. synth definition, synth function, or event-stream generating instance (see for example code below 3). This provides greater flexibility and simplicity in the coding of synth processes over JITLib.

2.1.2. Separate parameter namespaces: ProxySpace vs. Nevent

A significant innovation introduced by JITLib consisted in the concept of a ProxySpace, that is, a namespace that can function as the current environment. ProxySpace is based on EnvironmentRedirect, a Class which holds a Dictionary and ensures that a predefined custom function is executed each time that a value is stored in one of the keys of the Dictionary. Sc-hacks defines a subclass of EnvironmentRedirect similar to ProxySpace, but defines a custom function that provides extra flexibility in setting values which is useful during performance in accessing control parameters. This enables keeping track of which parameter refers to which process, storing parameter values between subsequent starts of a process belonging to a player, and updating GUI elements to display values as these change. Additionally, sc-hacks makes the environment of the player current after certain operations, in order to make the current context the one normally expected by the performer. This however is not always a secure solution. For this reason, the target environment can be provided as adjective argument in binary operators involving players, which ensures that code will work as expected even when changing the order of execution of code in irregular manner.

2.1.3. Modifying event generating processes on the fly

Event generating algorihm processes are implemented in SuperCollider through class Pbind. Pbind takes an array of keys and associated streams as argument and creates a Routine that calculates parameters and event types for each set of keys and values obtained from their associated streams, and schedules them according to the duration obtained from the stream stored under the key `dur`. The implementation of Pbind allows no access to the values of each event, i.e. it is not possible to read or to modify the value of a key at any moment. Furthermore, it is not possible to modify the structure of the dictionary of keys and streams while its event-generating process is playing. This means that Pbind processes cannot be modified interactively while they are playing. In order to circumvent this limitation, a number of techniques have been devised which require to add code for any key that one wishes to read or to modify. JITLib uses such techniques and also provides a way to substitute a Pbind process while it is running with a new one, thereby indirectly allowing modification of that process. Sc-hacks provides a new approach for playing event-generating processes, which uses the same Event-playing mechanism as Pbind, but grants both read and write access to the data which generate the event stream, and thus permits modification of the generating key-stream collection on the fly. This radically simplifies the task of modifying event generating processes while they are playing. For example, adding or substituting key-value stream pairs to a process while it is playing can be achieved simply by sending the corresponding key-stream pairs as events to the same player, as shown in the following code 1.

```
(dur: 0.1) +> \mystream;
// Substitute duration stream:
(dur: [0.1, 0.2].prand) +> \mystream;
// Add degree stream:
(degree: (-10..10).prand) +> \mystream;
```

Figure 1: *Adding and substituting key streams to event generators.*

2.1.4. Interconnecting audio signals

The task of connecting the output of one audio process with the input of another audio process is complicated in SuperCollider by the requirements (a) to specify the bus which will carry the signal to be shared and (b) to ensure that the synth reading from the signal will be placed *after* the bus which is writing to the signal in the execution

[2] See for example the Observer pattern: `https://en.wikipedia.org/wiki/Observer_pattern` (accessed 20-December-2018)

order of the synth engine (scsynth). The implementation of the solution in JITLib involves several classes with several instance variables and hundreds of lines of code and defies description within the scope of the present paper. Additionally, coding the configuration of one-to-many or many-to-one interconnections of audio i/o between synth processes can be both verbose and complex, as witnessed for example in exchanges on the SuperCollider mailing list such as this one: `https://sc-users.bham.ac.narkive.com/PAapaSaM/many-to-one-audio-routing-in-jitlib` (accessed 20-December-2018). Sc-hacks introduces a new solution which permits simpler coding and guarantees persistence of established configurations even when the server is rebooted during a work session. The implementation is based on mechanisms for hierarchical namespaces and function callback implemented in sc-hacks through two new classes discussed below: *Registry* and *Notification*. The coding of one-to-many and many-to-one connections is exemplified through the following code 2:

```
// many - to - one interconnection
\source1 *> \fx1;
\source2 *> \fx1;
// one - to - many interconnection
\source3 *< \fx2;
\source3 *< \fx3;
```

Figure 2: *Interconnecting audio signals.*

Note that no additional coding is required if using the default input and output parameter names `\in` and `\out` and number of channels (1). PersistentBusProxy is used to specify custom parameter names and channel numbers. The operator `@` can optionally be used as shortcut to create PersistentBusProxy instances.

2.2. Binary operators

The primary coding strategy of *sc-hacks* for sound processes is built around a small number of binary operators. Each operator encapsulates a group of actions on sound objects such as synthesis parameters, player objects holding single synths or synth processes, busses, buffers, midi or osc control instances. The operators are:

left operand	operator	right operand
source	+>	player
source	*>	player
parameter	<+	value
parameter	*<	value

2.2.1. *+> : Play source in player*

The +> plays the *source* in the *player*. The source can be the name of a synthesis definition as symbol, a synthesis function, or an event. For example the code in 3 can be evaluated line-by-line to play in the player named `'example'` in sequence a synth using SynthDef named `'default'`, a Unit Generator Synth Graph containing a Sine Oscillator, an empty event with default parameters (degree: 0, dur: 1), an event with duration 0.1, and an event with degree a pattern using a brownian stream with values between -10 and 10 and maximum step 2. Sending different types of sources (synthdef names, synth functions, events) to the same player will replace the previous source with the newest one. Sending `nil` stops the player.

```
\default +> \example; // play synthdef
{ SinOsc.ar(440, 0, 0.1) } +> \example;
() +> \example; // play event
(dur: 0.1) +> \example; // modify event
(degree: [-10, 10, 2].pbrown) +> \example;
nil +> \example // stop player;
```

Figure 3: *Player operator +>.*

Additionally, sc-hacks permits one to browse the code executed for each player on a dedicated GUI (similar to operations on *Shreds* in the miniAudicle GUI of *ChucK*), to edit existing code and resend it to the player, and to start or stop a player by clicking on its name in the list of existing players, as shown in Figure 4. The list of evaluated code strings is permanently saved on file for each session.

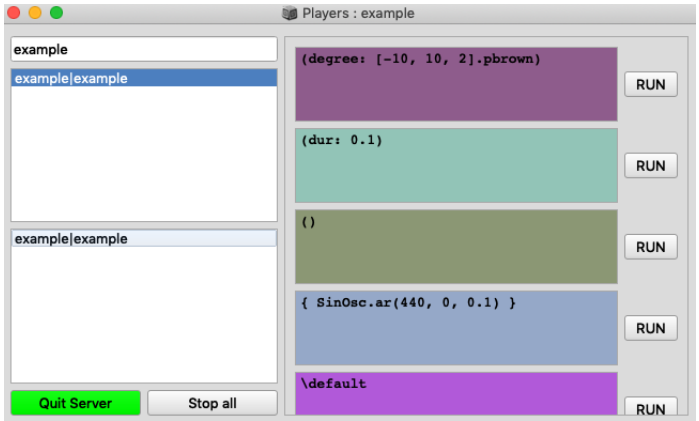


Figure 4: *Player GUI.*

2.2.2. **> : Advanced operations on player argument*

The *> operator takes different meanings depending on the type of the right operand, as follows:

type of left operand	action
Event	set parameter values without starting events
Function	Play function as routine in environment
Symbol	Add receiver as audio source to argument
PersistentBusProxy	Add source with custom i/o mapping

2.2.3. *<+ : Set or map parameter*

The <+ operator acts on the parameter named by the receiver (left operand) depending on the type of the argument (right operand), as follows:

type of right operand	action
Integer or Float	Set parameter value
Symbol	Map parameter to named control bus
Envelope	Map parameter to envelope signal
Function	Map parameter to Synth Function output
MIDI	Bind parameter to MIDI input
OSC	Bind parameter to OSC input

The parameter named by the left operand belongs by default to the current environment. In order to specify a different environment, one can name the environment as an adverb to the binary operator using standard SuperCollider syntax, e.g.: `\freq <+.myenvir 660`.

2.2.4. *<+ : One-to-many audio i/o interconnections*

The *< operator, in analogy to *>, is used to create one-to-many i/o interconnections, that is, to connect the audio output from one Player to the inputs of several different Players.

2.3. Fundamental Classes

To implement the above features, sc-hacks introduces classes which implement pattern-language-like features that enable functionality across a wide variety of tasks such as storing and retrieving single instances in tree data structures (Registry Class), updating state of concerned items in response to changes (Notification Class), and enforcing sequential order of execution in asynchronous calls to the server when booting, loading synthdefs and loading or initializing audio buffers (ActionSequence Class). These classes formed the backbone for rapid creation of custom extensions to the library to meet needs of performance requirements described in the next section. These results are encouraging indications that the library will serve as framework to develop more ambitious applications in the next stages of this work.

3. APPLICATIONS

3.1. Telematic Dance

Sc-hacks was first used in a telematic dance project whose goal is to enable dancers to perform together concurrently in different cities by sharing data from motion sensors sent via OSC over the internet [16]. Sensors were constructed using LSM9D0 motion sensor modules and Feather Huzzah ESP8266 wifi modules from Adafruit, and connected to SuperCollider via micro-osc package on micropython. Several sessions with dancers in Tokyo, Athens and Corfu served to experiment with different sound synthesis algorithms and to test the usabiity of the interface and algorithms for dance improvisation. The results were generally more encouraging than expected, except in Corfu where the dancers showed a more cerebral approach emphasizing control over the sound result rather than free exploration of the sonic landscape through movement.
A significant new turn in the development of the library was prompted during the initial tests for remote collaboration performed during a workshop organized at the University of Manchester by Prof. Ricardo Climent for the EASTN-DC EU-Culture program. This showed the need for distributing versions of the library to different remote partners, using different custom settings for each partner. Opening files in the SuperCollider IDE in order to select and execute appropriate code segments was soon proven to be impractical under the pressed time circumstances of preparing the test within a large scale workshop and awkward time-zone difference between the partners involved. Thus, a plug-and-play solution had to be devised, or at least one that relied on selecting options from menus or lists and clicking on buttons rather than opening files and executing code. This gave rise to a new interface as a GUI for selecting and evaluating snippets of code contained within files within subfolders of a global "Snippets" folder 5. The scheme has since served for the archival of experiments and performances, facilitating easy overview and reuse of past code. It is furthermore integrated for use with EMACS as primary IDE for SuperCollider, with automatic updates of code between EMACS and the SuperCollider based GUI.
Two further features were necessary for the experiments with dancers. First, a GUI that displays OSC data as they are received, and second a mechanism that scales and assigns incoming OSC data to the desired parameters. The following code shows how to generate a gui that displays data changes for a set of named parameters. Updates are displayed whenever a parameter is changed, independently of the source of the change (i.e. automated algorithm, evaluation of code, MIDI or OSC input).

```
\lsm1.v(
        \dur.slider([0.1, 12], \lsm1),
        \pos.slider([0.0, 1.0], \lsm1),
        \rate.slider([0.2, 15], \lsm1),
        \gps.slider([0.5, 20.0], \lsm1),
        \pan.slider([-1, 1.0], \lsm1),
        \amp.slider(\amp, \lsm1)
);
```

The GUI in figure 6 was generated by the code above.
Following example shows how to scale data input from OSC messages and to assign them to named parameters in a specified environment `'lsm1'`.

```
\dur <+.lsm1
   '/gyroscope1'.osc(0, [-40, 40], [0.01, 12.5]);
\pos <+.lsm1
   '/gyroscope1'.osc(1, [-20, 40], [0.0, 1.0]);
\rate <+.lsm1
   '/gyroscope1'.osc(2, [-20, 40], [0.1, 15]);
\gps <+.lsm1
   '/magnetometer1'.osc(0, [-1.0, 0.5],
                           [0.2, 15]);
\pan <+.lsm1
   '/magnetometer1'.osc(1, [-0.25, 0.25],
                           [-1, 1]);
\amp <+.lsm1
   '/magnetometer1'.osc(2, [-0.05, 0.25],
                           \amp);
```

The above features are only the beginning. As experiments with dancers have shown, other GUIs and coding schemes are needed to facilitate adjustment of the responsiveness of the sensors and adaptation of their sound control aspects during performance. In this respect a considerable amount of work is still required.

3.2. Coding Electroacoustic Music Performances

A second test scenario was provided through the collaboration with Dan Weinstein, a concert cellist specializing in contemporary music performance with good knowledge of contemporary audio tools in Linux. Mr. Weinstein selected two pieces from the early repertory of electroacoustic music scored for tape recorder: Jonathan Harvey's "Ricercare una melodia" and Karlheinz Stockhausen's *Solo 19*. Both pieces had to be coded in SuperCollider and rehearsed within one week during a residency of Mr. Weinstein in Corfu, leading to a public performance of the pieces. The time constraints were critical because the pieces were both complex and demanding in terms of score interpretation, following and coordination. The Stockhausen

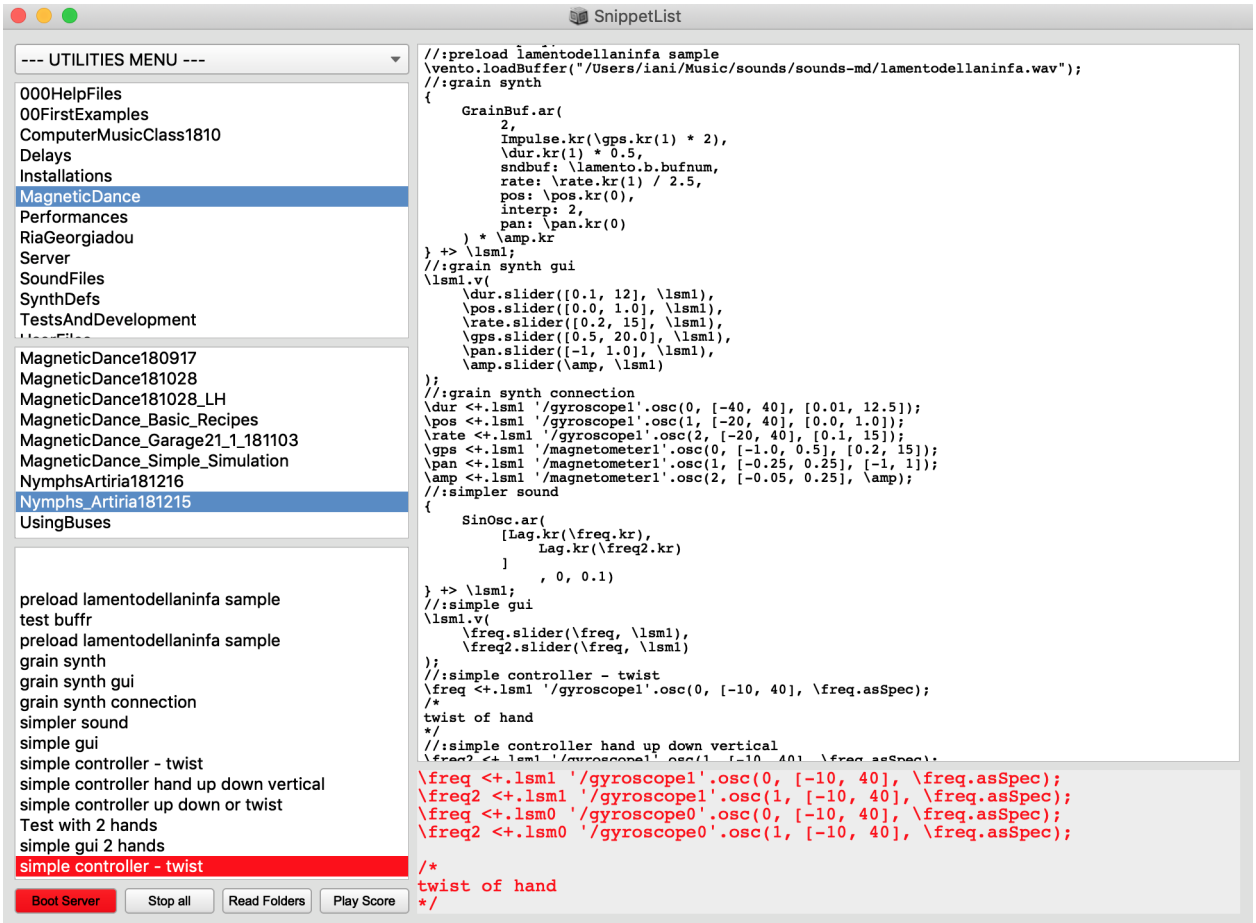


Figure 5: *Snippet List GUI.*

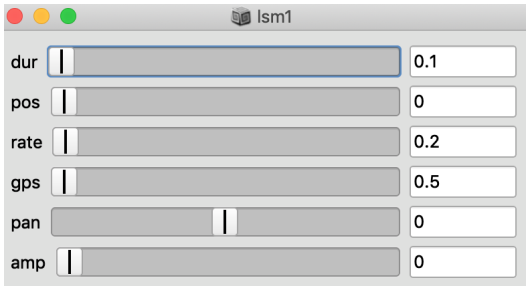


Figure 6: *Grain Control GUI.*

piece proved to be especially difficult as it is initially scored for 4 assistants in the electronic part, where each assistant is assigned control of the recording, playback and feedback levels of two tape recording channels with varying loop durations between sections, using two potentiometers. To execute this with a single performer on the computer, the slider actions as well as the loop duration changes had to be automated according to the indications in the score. Even under these circumstances, an ideal faithful performance was impossible, because each of the 6 levels demanded constant adjustment according to the actual level of the instrumental performer, and each transition had to be timed manually to prevent abrupt noticeable changes. Still, this proved to be a fruitful exercise in creating a user interface and coding the entire score, consisting of 6 different realization versions. It resulted in a compact coding scheme for durations of prescribed length (see 7 for the notation of the first version - Formschema I, and 8 for its translation into GUI and automated performance). This notation mechanism can in the future be repurposed as a type of beat sequencing notation similar to this found in ixilang or TidalCycles (although the Cycle scheme of Tidal has other features which go beyond the scope of the present discussion).

4. CONCLUSIONS AND FUTURE WORK

Sc-hacks is a general purpose extension to SuperCollider, and the intense use of several binary operators may raise doubts about its legibility or the general validity of its design priorities. However, stress-testing sc-hacks through collaborations with dancers and instrumentalists has shown its strong potential to solve diverse and demanding problems under time pressure, and furthermore has provided indications of its scalability in terms of coding various features. This indicates that it is a suitable platform for further work, and it is hoped that it will serve as a tool for addressing questions of machine listening in live performance as well as other advanced topics.

```
    *formschemaI {
        ^StockhausenSoloFormschema(
            [  // numperiods, duration per period
                [11, 6],
                [8, 14.2],
                [7, 19],
                [6, 25.3],
                [9, 10.6],
                [10, 8]
            ], thisMethod.name.asString
        )
        .loadPeriodStates(
"
X__X__XXX__|XXXXXXX_|X__X___|X__X__|_X_X_X___|X_X__X_X__
_XX_XX__XXX|XXXXXXX_|XXX_XXX|_XX___|XXXXX_XX_|_X__X_X_X_
_XXXX__XXXX|_XXXXXXX|_X__XXX|_X__XX|__XX__XX_|_XXXXX__X_
__X__XX__XX|_XXXXXX_|_XX__XX|__X___|_XXXXXXXX|__X____XX_
_XX_XX__XX_|_XXX_XXX|XXX_XX_|_XX_XX|X_XXX_XXX|_X_XX_X_XX
__XX__XX__X|_X_XXX_X|___X_XX|__XX__|_X___X__X|X_XX_X_XXX
"
        );
    }
```

Figure 7: *Code for Formschema I of Stockhausen Solo 19.*

Recording data received from sensors is a first priority in the project. A first prototype has been implemented using the built-in archival facilities of SuperCollider. A second implementation is currently under development, which will record data into multichannel audio signal buffers, and employ an extra channel to record the time interval between receipt of successive OSC messages. Based on this, and using the existing graphic visualization facilities of SuperCollider for audio signals, a functionality similar to the MuBu tools from IRCAM [3] is envisaged. In collaboration with PhD students working on Machine Learning, it is planned to use this for further research.
In parallel, work is being done to connect data sent over the internet in remote performances, and in developing a performance repertory with instrumental soloists interested in improvisation with live electronics. In both these cases, the most serious challenge consists in making the software stable and easy to use enough to be able to release it to non-specialist performers for work in real-world creative events without the need of specialized technical assistance to run it. This remains a major driving factor and design guideline in developing this software. At the same time it is expected that these requirements will help create best practice solutions that constitute the wider contribution of this project. In this sense, the present project is placed within the scope of efforts for developing contemporary languages of notation for performance practice that have lasting impact on the community and its aesthetics.

5. REFERENCES

[1] S. Wilson, D. Cottle, and N. Collins, Eds., *The SuperCollider Book*, MIT Press, 2011.

[2] A. Sorensen, "Impromptu: An interactive programming environment for composition and performance," *Proceedings of the Australasian Computer Music Conference*, 01 2005.

[3] A. Kapur, P. Cook, S. Salazar, and G. Wang, *Creating music with ChucK*, Manning, 2015.

[4] A. Sorensen, *Extempore: The design, implementation and application of a cyber-physical programming language*, Ph.D. thesis, The Australian National University, 2018.

[5] C. Roberts, M. Wright, and J. Kuchera-Morin, "Music programming in gibber," in *Proceedings of the 2015 International Computer Music Conference*, pp. 50–57. 01 2015.

[6] D. Stowell and A. McLean, "Live music-making: A rich open task requires a rich open interface," in *Music and Human-Computer Interaction*, pp. 139–152. Springer, 2013.

[7] S. Salazar, "Searching for gesture and embodiment in live coding," in *Proceedings of the International Conference on Live Coding*. 2017.

[8] G. Wang, G. Essl, J. Smith, S. Salazar, P. Cook, R. Hamilton, and R. Fiebrink, "Smule= sonic media: An intersection of the mobile, musical, and social," in *Proceedings of the International Computer Music Conference*, p. 16–21. 2009.

[9] S. Salazar and J. Armitage, *Re-engaging the Body and Gesture in Musical Live Coding*, 2018, [Online; accessed 20-December-2018], `https://embodiedlivecoding.github.io/nime2018-workshop/workshop-paper.html`.

[10] J. Armitage and A. McPherson, "The stenophone: live coding on a chorded keyboard with continuous control," in *Proceedings of the International Conference on Live Coding*. 2017.

[11] K. Sicchio, "Hacking choreography: Dance and live coding," *Computer Music Journal*, vol. 38, no. 1, pp. 31–39, 2014.

[12] T. Magnusson, "The ixi lang: A supercollider parasite for live coding," in *Proceedings of the International Computer Music Conference*, pp. 503–506. 2011.

[13] A. W. Shu-Cheng, "Supersampler: A new polyphonic concatenative sampler synthesizeer in supercollider for sound motive creating, live coding, and improvisation," in *Proceedings of the International Computer Music Conference*. 2017.

[14] G. Roma, "Colliding: a supercollider environment for synthesis-oriented live coding," in *Proceedings of the 2016 International Conference on Live Interfaces*. 2016.

[15] A. McLean and G. Wiggins, "Tidal – pattern language for the live coding of music," in *Proceedings of the 7th Sound and Music Computing Conference*. 2010.

[16] I. Zannos and M. Carle, "Metric interweaving in networked dance and music performance," in *Proceedings of the 15th Sound and Music Computing Conference*, pp. 524–529. 2018.

[3] `http://forumnet.ircam.fr/product/mubu-en/` (accessed 20-December-2018)

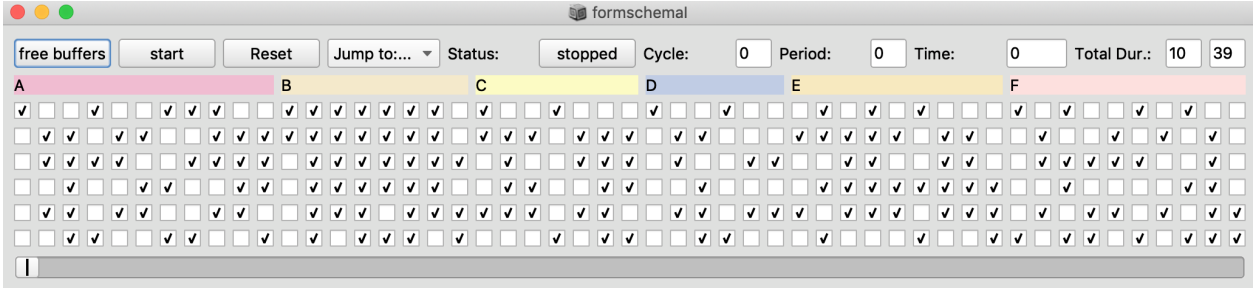


Figure 8: *GUI for Formschema I of Stockhausen Solo 19.*

BIPSCRIPT: A DOMAIN-SPECIFIC SCRIPTING LANGUAGE FOR INTERACTIVE MUSIC

John Hammen

bipscript.org
Berkeley, California, USA
jhammen@j2page.com

ABSTRACT

Bipscript is a domain-specific scripting language designed to make it easier to create interactive music. The base language is the Squirrel scripting language which has been complemented with a standard class library containing audio-specific domain objects. This API provides methods for creating, scheduling and handling events of various types including MIDI, OSC and extracted features of audio streams. A single-threaded programming model with asynchronous event handling is familiar to web developers but atypical in music DSLs. Scripts are executed by a command line interpreter with tight integration to the audio system.

1. INTRODUCTION AND DESIGN GOALS

The Bipscript project began as an attempt to implement a musical "bot" application, with the base functionality of existing auto-accompaniment software augmented with a high degree of interactive functionality. The goal was software that would output appropriate MIDI sequences in real-time based on external inputs, most notably data from human performers. The emphasis was on tempo-driven music with tight integration to a local transport.

Design goals did not include specialization on any particular style of music, nor any specific assumptions on how external inputs would affect generation of MIDI sequences, leaving these decisions to the configuration of a particular piece.

As the options for configuration grew it became clear the easiest way to express the behavior of a particular musical part would be via an imperative language with the abilities to directly receive relevant input from external sources, and use this information algorithmically to sequence MIDI notes.

2. FEATURES

2.1. Squirrel

The scripting language itself is the Squirrel language [1]. From the Squirrel website:

"Squirrel is a high level imperative, object-oriented programming language, designed to be a light-weight scripting language that fits in the size, memory bandwidth, and real-time requirements of applications like video games."

These attributes and the associated predictability in run-time behavior make Squirrel an ideal language also for the real-time demands of audio applications.

Bipscript builds on top of Squirrel by adding a class library API containing audio domain-specific classes and a custom transport-aware interpreter that allows for event handling.

2.2. Class Library

The Bipscript class library API features objects representing plug-ins, mixers, and system inputs and outputs of various types, predominately audio, MIDI and OSC [2]. These objects can be connected programmatically to create complex networks of the different protocols (see Figure 1)

Events of any applicable type can be scheduled to occur on any node in the network, in particular code can generate and output timed MIDI and OSC sequences. Event handlers can be registered to fire on particular events including features extracted from audio streams.

Additional classes allow the use of textual specification of musical score data in the scripts using ABC notation [3], Music Macro Language [4] or a MIDI tablature format based on common drum tablature.

```
local midiInput = Midi.Input("myinput")

local synth = Lv2.Plugin("urn:example:a-synth")
synth.connectMidi(midiInput)

local effect = Lv2.Plugin("urn:ardour:a-reverb")
effect.connect(synth)

local mainOutput = Audio.StereoOutput("main")
mainOutput.connect(effect)
```

Figure 1: Creating Connections.

2.3. Threading and Context Model

Scripts including event handlers run in a single execution thread with a single global context. Instructions in a script will be executed sequentially until the main body of the script completes. At this point any event handlers that were registered by the script will execute as needed in the same thread and scope with direct access to all variables defined in the main body of the script.

This programming model is analogous to traditional JavaScript development in a web browser where the main body of the script registers event handlers that are then executed in the same thread within the same page context. In both cases execution is single-threaded and non-blocking in favor of asynchronous event handling.

Although many objects such as plugins will participate in the audio client's process thread the script thread itself is separate from the process thread and not subject to its programming limitations on e.g. memory allocation and system I/O.

2.4. Transport-Aware Interpreter

The command-line interpreter is tightly integrated with the system transport, on Linux provided by the Jack Audio Connect Kit [5]. Scripts can request to be the transport master via the API but are not required to do so when there is an external master present to provide position information.

Whether or not the script itself is the transport master, the interpreter will act as a sequencer for any events scheduled by the script, playing them in time with the transport and reacting to any arbitrary external transport position changes including looping.

3. COMPARISON WITH OTHER PROJECTS

3.1. Synthesis Languages

There exist several DSLs for music creation with large and active communities, for instance Pure Data [6] and SuperCollider [7] among others. These environments differ from each other, for instance Supercollider is a traditional text-based programming language while Pure Data is a visual language.

There is a however a common emphasis on sound design with code libraries of elements representing oscillators, filters and other signal generation capabilities. Bipscript currently offers no such objects in its standard class library instead offering hosting capabilities for 3rd party sound generation and effects plugins.

The emphasis instead is on timed music which has led to an API built around handling events and just-in-time sequencing using data structures representing e.g. MIDI notes and mutable patterns (groups of notes). In contrast most music DSLs produce timed music at a lower level by alternating between immediate sound-generation instructions and some variation of a system "sleep" command.

3.2. Other Open Source Projects

Other comparisons can be drawn to some of the many projects arising from the community of open source audio software on Linux and elsewhere:

One of the most feature-rich open source audio applications is the Ardour DAW [8], which in recent versions has a large number of the C++ implementation classes exposed as Lua objects [9] giving a scripting environment incorporating much of Ardour's MIDI and DAW functionality. This differs from a more traditional script language-plus-interpreter environment in that scripts are executed as callbacks in application-specific contexts, each with their own scope and applicable model objects.

Another project using Lua is the Moony Lv2 plugins [10]. Taking advantage of Lua's real-time performance, scripts are run directly in the process thread and thus allow manipulation of e.g. MIDI messages as they pass through the plugin. but are bound by all the standard real-time limitations of running in the process thread.

LuaJack is a Lua binding library for Jack [11]. Scripts written in LuaJack and Bipscript have a visual similarity due to the similarity between Lua and Squirrel and the fact the LuaJack and Bipscript API are wrapping some of the same objects, e.g. system ports. However as a language binding LuaJack does not include an interpreter nor an object API beyond directly exposing the Jack client API.

4. IMPLEMENTATION

The command line executable that functions as an interpreter to execute scripts was written in C++ and runs on the Linux operating system with the Jack Audio Connection Kit as a run-time dependency.

The standard Squirrel implementation is intended to be embedded and was used as the basis of the command line interpreter.

The interpreter also acts as a standard audio client, opening and connecting system audio and MIDI input/output ports and hosting plugins as specified by the executing script.

Scripts are loaded and run in a dedicated execution thread separate from the application's audio process thread. Any event generated by the script is appended to an applicable lock-free queue that is consumed by the process thread.

Script objects that hold scheduled events will participate in the audio process thread to pull events from the queues and emit them at the appropriate position in a running transport with sample-level accuracy. Synchronization between the script and process threads allows the script execution to properly respond to arbitrary transport location changes.

A set of bindings was created for the class library, generated from a high level API description to interface the C++ implementations of the standard library classes and methods to the Squirrel engine via its stack-based API.

The object implementations make use of reliable third party code where possible via both embedded code and dynamically linked libraries. The current implementation makes uses of popular projects such as abcmidi, liblo and libsndfile.

5. USE CASES

Now existing in a basic implementation, Bipscript can be used for the following use cases:

5.1. Dynamic Accompaniment

An example "Robot Jazz Band" was built [12], showing the script implementation of 3 related bots (playing acoustic bass, piano and trap drum samples) that play a dynamic sequence based on rhythmic probabilities in a jazz "swing" pattern coupled with a given input chord progression. All players take an input parameter of "intensity" playing louder and busier vs. softer and sparser on a measure by measure basis. The main script connects an audio onset de-

tector to a system audio input and continuously updates this intensity variable for all players based on the number of onsets received.

This example shows a standard programming model for creating a script with this kind of interactive behavior:

- The main script instantiates plugins and needed audio and MIDI connections (see Figure 1) and schedules any static parts of the score
- Event handlers listen to human performers via e.g MIDI, OSC and/or audio features and update an internal state (see Figure 2)
- Scheduled methods read this state and use as input in calculating a short output sequence (e.g. a beat or bar at a time)

The result is a dynamic and reactive auto-accompaniment system written in a relatively few lines of code compared to general purpose programming or even other music DSLs. The simple example works as expected in practice but leaves open many paths for future development in creating more complex interactive scripts.

```
local controlPower = 0
controlInput.onControl(function(cc, pos) {
  if(cc.controller() == 15) {
    controlPower = cc.value() * 100 / 127
  }
})
```

Figure 2: Sample Code from the Robot Jazz Band Demo.

5.2. Utility Scripts for Live Performers

In many traditional live musical projects there is no need for dynamically generated sequences, especially those where human performers are playing from a static score. In such an environment there is still a use for certain computer-aided functions such as triggering samples in time or adding a “click track” or other timed audio cues not heard by the audience. These functions can be built in custom scripts with relatively few lines of code.

5.3. Live Coding

Another prospective use is that of live coding. Those functions of the command line interpreter that allow for developer convenience may also be useful in a live coding situation, e.g. the ability of the script to be loaded and unloaded dynamically including while the transport is running. This use case is as of yet mostly unexplored

6. FUTURE WORK

With the completion of a basic proof-of-concept interpreter and class library the main focus of the project remains stabilizing the standard API and improving the basic tool implementations especially with an eye to reliability in live settings. To this end recent work has been done in the area of unit and functional testing for testing scripts as well as the interpreter itself.

Much of the API design going forward should be based on feedback from those who use these tools in a live production setting.

7. REFERENCES

[1] Demichelis, A. 2016. Squirrel – The Programming Language. http://www.squirrel-lang.org/

[2] Wright, M. 2002. Open sound control 1.0 specification.

[3] Walshaw, C. 2018. abc notation. http://abcnotation.com/

[4] Nakamura, S. 2015. A tiny MML parser. *Cubeatsystems.com* https://www.cubeatsystems.com/tinymml/

[5] Davis, P. 2003. Jack audio connection kit. http://jackaudio.org/

[6] Puckette, M. 1970. Pure Data: another integrated computer music environment.

[7] McCartney, J. 1996. SuperCollider. *supercollider.github.io*

[8] Davis, P. 2018. Ardour: the Digital Audio Workstation. http://www.ardour.org

[9] Manual.ardour.org 2018. The Ardour Manual. http://manual.ardour.org/lua-scripting/

[10] Portner, H. 2018. Moony - realtime Lua as programmable glue in LV2. http://open-music-kontrollers.ch/lv2/moony/

[11] Trettel, S. 2018. LuaJack Reference Manual. *Stetre.github.io*, https://stetre.github.io/luajack/doc/index.html

[12] Hammen, J. 2016. Robot Jazz Band Example. bipscript.org, http://www.bipscript.org/en/examples/robotjazz

Switching “Le SCRIME” over to Linux as a complete novice

Thibaud Keller [1] Jean-Michaël Celerier[1,2]

[1]SCRIME / LaBRI [2]OSSIA Team

First steps into the community

SCRIME : Studio for Creation and Research in Computer Science and Experimental Music.

- Created in collaboration with the LaBRI (Bordeaux Computer Science lab) and the Music Conservatory in Bordeaux in 1996.
- Multimedia tools Pioneer, SCRIME played an important role in the development of the *Lemur* control surface and what became the “meta-sequencer” *ossia score*
- Retained until now a “conventional” set of tools consisting mainly of *Cycling 74' Max/MSP*, *Avid Pro Tools* and *Cockos Reaper* running on *Apple macOS*

The decision to switch over to Linux was taken at the end of 2017, shortly after the team changed and a Dome of 19.2 speakers for 3D sound rendering was set-up. Even though this task was undertook by someone new to Linux, the fact that these various changes happened all at once provided the intermission necessary to familiarize with the Linux environment and reevaluate the studio's main tools.

The community

The *Ubuntu Studio* distribution was chosen as a starting point for the overview of Linux multimedia production it offers, as well as its ease of use. On top of the *Ubuntu* community being very helpful, additional online resources made a real difference during the adjustment period:

- `linuxmao.org`, a french forum for all things music on Linux, curated by many *LibraZik* contributors as well as *Ubuntu Studio* and *KXStudio* users.

LinuxMAO.org
La musique en toute liberté !

- Echoplex Media `echoplexmedia.com`, recording studio and podcast providing basic tutorials for *Ubuntu* installation, sound and radio production with Linux audio tools
- Joe Collins `ezeelinux.com`, very helpful and accessible tutorials and scripts for different distributions.
- Benjamin Caccia `bcacciaaudio.com`, practical tips and tricks for Linux audio production.
- The Arch Wiki `wiki.archlinux.org`, for everything else, an ultimate resource if only a little too technical at times for first time Linux users.

If the readily available alternatives to commercial software have yet to find total acceptance with local composers, switching over to Linux proved fruitful when welcoming computer-science students for early 2018 internships. Each either working on our machines or on their preferred *Debian* flavors : these few months where rich in exchange and compatible contributions.

Putting it together

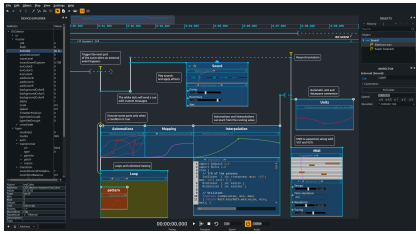
The hardware used is a set of old (2009 - 2013) Mac machines. At the time of the first installation, the latest *Ubuntu Studio* release was 17.04 beta. The ability to start over from scratch and re-install the system quickly proved useful, especially when working with a distribution's beta release, beta-testing various software and having to replicate the exact same set-up on multiple computers.

- After experimenting with *CloneZilla*, the best solution turned out to be the combination of `apt-clone` (to save and re-install all installed packages) and archives of the home directory compressed with tar. This method offers the comfort to experiment with successive “breakings” and restoring of the configuration. We were able to recover very quickly and identify culprits in scripts installing faulty headers.
- Audible xruns accumulate constantly if Wifi connection isn't disabled. Broadcom cards seem to be regularly causing CPU interruptions and to this day, no solution have been found. Identifying this issue actually took some time as WiFi seemed a very unexpected source of xruns.

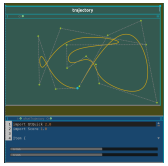
The OSSIA project

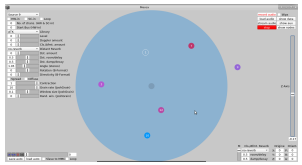
OSSIA : Open Software System for Interactive Applications (`ossia.io`).

- Bindings for a number of open source environment (SuperCollider, Pure Data, OpenFrameworks...).
- Faust, VST and LV2 support.
- OSCQuery, developed in collaboration with VDMX, Vezér and MadMapper, allows for automatic detection of OSC namespaces. Implemented with WebSockets, an OSCQuery server can communicate natively with a web page.
- Scriptable protocols with Javascript: HTTP, WebSockets, Mapper, Serial port.
- Protocols in the works at the moment include Artnet, Wiimote and gamepad support.



The Mosca case





Multimedia production on Linux differs mainly with commercial operating systems by its “decentralized” aspect. Whereas commercial environments attempt to be all-encompassing while remaining closed black boxes, the great number of independent modules and the freedom to have them interact encourages a more specialized and distributed approach.

Further development of the *Mosca* project at SCRIME exemplifies this practice. Initially created by Iain Mott, *Mosca* is a *SuperCollider* class for real time 3D sound processing. It's many features rely on the interconnection between several different environments:

ossia score [2] for interactive sequencing and control
AmbiDecoderToolBox [4] for direct rendering on speaker Arrays
Ardour [3] for editing and prototyping (connected through MMC)
Aliki [1] for creating and editing room impulse responses
Arduino for head tracking and control with gesture

Compatibility

- Focusrite, MOTU and RME suport with either USB or FireWire
  ALSA drivers for MADIFace Pro, Fireface UFX, Scarlett 2i2...
  FFADO drivers for 828 Mk2, Ultralight Mk3, Fireface 800...
- Digidesign C24 control through OSCQuerry.
- Digidesign 002 and Mbox suport (originaly restricted *Pro Tools*).

Acknowledgements

Many thanks to Simon Archipoff and Raphael Marczak for their support and suggestions, and to Arthur Liefhooghe for his contributions!

[1] Fons Adriaensen. “Acoustical impulse response measurement with ALIKI”. In: *Linux Audio Conference Proceedings*. 2006, p. 9.

[2] Jean-Michael Celerier. “Authoring interactive media: a logical & temporal approach”. PhD thesis. Bordeaux, 2018.

[3] Paul Davis et al. *Ardour digital audio workstation*. 2012.

[4] AK Heller and EM Benjamin. “The ambisonic decoder toolbox”. In: *Linux Audio Conference,(Karlsruhe)*. 2014.

SOUNDPRISM: A REAL-TIME SOUND ANALYSIS SERVER AND DASHBOARD

Michael Simpson
ITP
New York University
michael.simpson@nyu.edu

ABSTRACT

The paper presents SoundPrism[1], a real-time sound analysis server and dashboard. SoundPrism collects real-time sound data from a microphone or PCM and performs analysis which can be broadcast in near real-time for use by external applications. A dashboard interface visualizes the collected data in a matrix allowing more useful streams to be easily identified. This is useful in applications where mapping higher level sound events to another form of output desired. The tool offers particularly interesting new possibilities for musicians and audio-visual artists. The design, implementation, and conclusions will be discussed within this paper.

1. INTRODUCTION

SoundPrism is a software project that aims to simplify and expand the ways that real-time signal analysis and music information retrieval can be used in creative applications.

This project is intended for providing simpler access to near real-time MIR data as it is captured. The author's interest in this project relates to music visualization, composition, and the creation of audio-visual performances, installations, and experiences. However, the author believes the technique has vast applications across disciplines.

SoundPrism is made of three components: a real-time database, a graphical dashboard, and a domain specific language for composing data flows out of signal analysis and music infortmation retrieval algorithms.

SoundPrism was built using OpenFrameworks, a creative coding toolkit for C++, and the Essentia Project, a collection of signal analysis and music infortmation retrieval algorithms. Both software have cross-platform compatibility. SoundPrism was developed using Linux using the Linux-rt kernel and JACK.

1.1. Domain Specific Language

SoundPrism offers a domain specific language for composing analysis chains out of Essentia's algorithms. These chains are then available to be visualized, stored, or broadcasted by the graphical dashboard. This allows new analysis chains to be designed without having to delve into all the related scaffolding.

1.2. Graphical Dashboard

As a graphical dashboard, SoundPrism serves as a *tool for thought* by providing a simultaneous view of up to 16 different data streams. This format amplifies the user's ability to quickly discern the usefulness of a certain analysis in a certain context.

The dashboard offers multiple visual styles for rendering the data, for example: line graphs, histograms, and 3d cascading waterfalls. The 3D visualizations use historical data to offer context of time. The dashboard allows the user to select the most useful algorithms to either store their data, broadcast it to the network, or both.

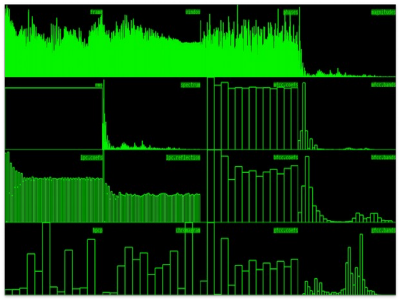


Figure 1: SoundPrism Showing a 4x4 Dashboard

1.3. Data Server

SoundPrism broadcast selected streams so external applications are able to use them. This allows existing applications to take full advantage of the real-time signal analysis and music information retrieval capabilities offered by SoundPrism. Furthermore, the data can be accessed remotely which allows for decoupling of applications from the audio's physical location.

2. IMPLEMENTATION

SoundPrism was built in C++ using OpenFrameworks, for graphical visualization and i/o, and the Essentia Project is leveraged for its signal processing and music information retrieval capabilities. For broadcasting of data, SoundPrism uses OpenSoundControl (OSC) which makes the data available for use by any other OSC-enabled application.

SoundPrism's dashboard uses OpenGL and provides each cell of the matrix with its own framebuffer object containing a unique camera and solitary planar mesh. This allows the cameras for each cell to be controlled manually or snapped to common orientations. Data is streamed into these mesh through the use of textures which are then sent into a vertex shader responsible for deforming the subsequent meshes. This allows for fast simultaneous rendering of up to 16 different graphs.

[1] https://mgs.nyc/projects/2018/SoundPrism

The Domain Specific Language is implemented as a series of macros which parse contents of the user's DSL coded files and transpiles them into the syntax used by Essentia.

2.1. Algorithms Provided By Essentia[2]

FFT, DCT, frame cutter, windowing, envelope, smoothing, low/high/band pass, band reject, DC removal, equal loudness, median, mean, variance, power means, raw and central moments, spread, kurtosis, skewness, flatness, duration, loudness, LARM, Leq, Vickers' loudness, zero-crossing-rate, log attack time and other signal envelope descriptors, Bark/Mel/ERB bands, MFCC, GFCC, LPC, spectral peaks, complexity, rolloff, contrast, HFC, inharmonicity and dissonance, Pitch salience function, predominant melody and pitch, HPCP (chroma) related features, chords, key and scale, tuning frequency, beat detection, BPM, onset detection, rhythm transform, beat loudness, danceability, dynamic complexity, audio segmentation, SVM classifier

2.2. DSL Example: MFCC

Mel-frequency cepstral coefficients (MFCCs) are particularly useful. Here's how an MFCC data flow would look using SoundPrism's DSL.

```
// File: mfcc.mdf

audio{source: PCM}
fc{algo: FrameCutter
   size: 2048}
win(algo: Window}
spec{algo: Spectrum}
mfcc(algo: MFCC)

audio → fc → win → spec → data:mfcc
```

The code above sets up a data flow that begins from the PCM, is cut into a frame of data, windowed, spectrum analyzed, and then MFCC is calculated. The output of the MFCC calculation is stored in a thread-safe database under the id of "mfcc". By adding these files, custom data flows become available to be placed within the dashboard and the data outputs are available for broadcasting via OSC.

3. CONCLUSIONS

SoundPrism demonstrates the usefulness of a robust platform for collecting, analyzing, and visualizing signal analysis and music information retrieval data. By making the data available to external applications, a tremendous number of potential applications are enabled but the following two are particularly interesting:

3.1. Music Visualization

By allowing for more intelligent understanding of the audio signal, more meaningful ways of relating audio and visual outputs can be created. This can be used as a basis for a more sophisticated platform for music visualizations.

3.2. Machine Listening

Tools like SoundPrism may prove useful as a component or prototyping tool for the design and implementation of Machine Listening systems. Machine Listening takes several techniques and technologies, like signal analysis, music information retrieval, and machine learning, and then fuses them with contemporary knowledge of cognitive science in efforts to simulate the human ear-brain system. A software like SoundPrism could prove to be a useful tool for generating datasets for the training of neural networks. It would also be useful as a platform for quickly experimenting with chains of signal analysis.

3.3. Sound Events

By providing the user with simplified access to the multiplicity of sound they are able to begin using sound in more intelligent ways. This opens the possibility for radical new forms of creativity and composition where sound features could be used as hooks caught by an event handler.

4. ACKNOWLEDGMENTS

Many thanks to my family, Yeseul Song, Roy MacDonald, Luke DuBois, Juan Bello, Zach Lieberman, Kyle McDonald, Dan Shiffman, Danny Rozin, Nick Montfort, Ramsey Nasser, Allison Parrish, Taeyoon Choi, Nancy Hechinger, Kathy Wilson, Dan O'Sullivan, Katherine Dillon, Gene Kogan, and all the other wonderful people who offered support and guidance along the way.

5. REFERENCES

[1] Bogdanov, D., Wack N., Gómez E., Gulati S., Herrera P., Mayor O., et al. (2013). ESSENTIA: an Audio Analysis Library for Music Information Retrieval. International Society for Music Information Retrieval Conference (ISMIR'13). 493-498.

[2] https://essentia.upf.edu/documentation/documentation.html