Theater Academy of Cagli

Institution for the Cagli Municipal Theater

Cagli Municipality

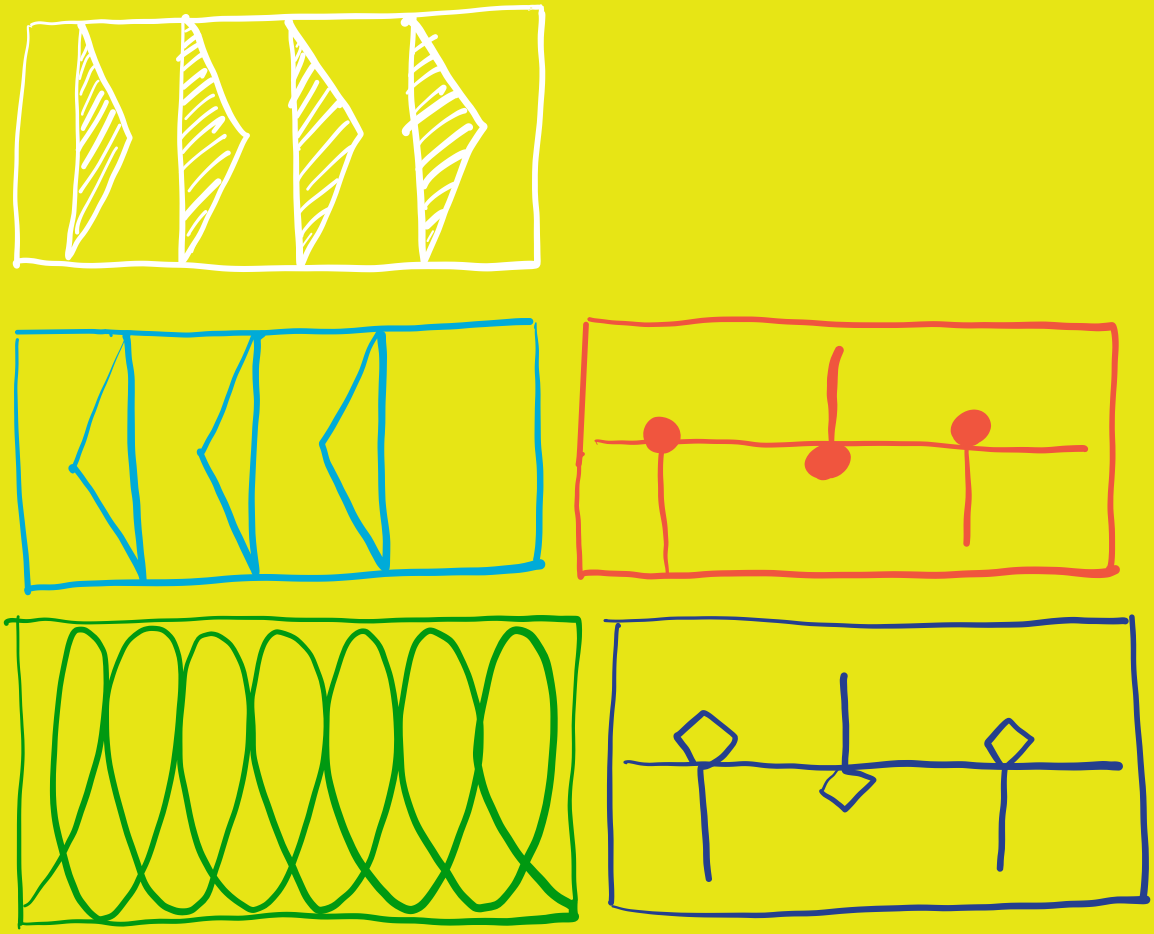
The Fifth International Csound Conference

ICSC2019

27-28-29 September, 2019

PROCEEDINGS

Municipal Theater
Cagli (Pesaro-Urbino), Italy



Proceedings of the Fifth International Csound Conference

Edited by:

Anthony Di Furia and Alessandro Petrolati

icsc2019@apesoft.it

Published by:

Theater Academy of Cagli, Italy

ISSN 2393-7580

https://csound.com/icsc2019

The Fifth International Csound Conference ICSC2019 Cagli, Italy

Organizing Committee
Theater Academy of Cagli
Institution for the Cagli Municipal Theater
Cagli Municipality

Organization Team
Anthony Di Furia
Enrico Francioni
Eugenio Giordani
Laura Muncaciu
Alessandro Petrolati

Institutions
Alberto Alessandri, Mayor of Cagli Municipality
Ombretta Michelini, President of the Institution of Theater
Sandro Pascucci, Director of the Institution for the Cagli Municipal Theater
Benilde Marini, Assessor of Culture
Simonetta Paolucci, President of the Theater Academy of Cagli

Paper Review Committee
Øyvind Brandtsegg
John Fitch
Eugenio Giordani
Michael Gogins
Tarmo Johannes
Luis Jure
Victor Lazzarini
Steven Yi

Music Review Committee
Enrico Francioni
Joachim Heintz
Alex Hofmann
Iain McCurdy
Laura Muncaciu
Alessandro Petrolati
Andrea Petrolati

Conference Chairs
Gianni Della Vittoria
Leonardo Gabrielli
Eugenio Giordani
Alex Hofmann
Massimiliano Tonelli

Special thanks
Bruno Marcucci, for donating ten original Art works *Iceberg,* 2019
Gianpaolo Antongirolami, for the Mencherini's tribute concert
Vines Gelso, for donating five Art pieces from his private collection

Guest musicians
Gianpaolo Antongirolami
Carlo Fatigoni
Nigel Thean

Sebastian Schmutzhard
Jakob Krupp
James Edward Cosby
Pierfrancesco Ceregioli
Alessandro Guerri
Elena Alessandra Petrolati
Andrea Petrolati

Credits
Alberto Alessandri and Gianluca Cespuglio, Unione Montana Catria e Nerone
Federica Pierotti and Maurizio Tagliatesta
Denis Rebiscini, Vodafone

MATME synth exhibition
Paolo Bragaglia
Leonardo Gabrielli

Photographer
Maurizio Tagliatesta

Web, design, music review coordinator
Anthony Di Furia

Audio Service
SwanSound by Enzo Geminiani

Technical service
Andrea Balducci
Anthony Di Furia

Partners
Unione Montana Catria and Nerone
MATME, Marche
Conservatory of Music Rossini in Pesaro
LEMS, Conservatory of Music Rossini in Pesaro
SPACE, Conservatory of Music Rossini in Pesaro

Hostess
Elena Luzietti
Elena Alessandra Petrolati

Internet provider
Vodafone 4G

Sponsors
apeSoft
Vodafone
La Gioconda
SwanSound
Caffè del Teatro
Out Of Range
Associazione Bello Sguardo
BCC

The Fifth International Csound Conference ICSC2019 will take place in Cagli (PU), Italy

The ICSC is an artistic and academic event where members of the international Csound community meet for three days of concerts, paper presentations, keynote talks and round tables.

Organizing Committee
Theater Academy of Cagli
Institution for the Cagli Municipal Theater
Cagli Municipality

Organization Team
Anthony Di Furia
Enrico Francioni
Eugenio Giordani
Laura Muncaciu
Alessandro Petrolati

Contact

ICSC 2019
csound.com/icsc2019
icsc2019@apesoft.it

Municipal Theater
www.teatrodicagli.it
+ 39 0721 780731

The Institution for the Cagli Municipal Theater is the public body that oversees the activities taking place on the historic stage of the city.
It is with pleasure that it welcomes and supports the work of “The 5th International Csound Conference ICSC 2019”, following in the tradition of hosting important events in the field of music, prose and dance.
The Cagli Theater seeks to be a “sanctuary for the arts” or, as the ancient Greeks described it, a “place without the right of capture”, a space open to the live arts for artists and public alike.

Sandro Pascucci, Director of the Institution for the Cagli Municipal Theater

The Theater Academy is an artistic and research action. It is virtual until the moment of the concretization in action. It recognizes music as one of the most extraordinary forms of individual and collective evolution. Being musician is a challenge, but music brings knowledge, expands consciousness and elevates the spirit. Changing realities is a difficult act, but changing ourselves, our lives, our sphere of human interaction is certainly achievable.

Laura Muncaciu, Artistic Director
Simonetta Paolucci, President
Alessandro Petrolati, Executive

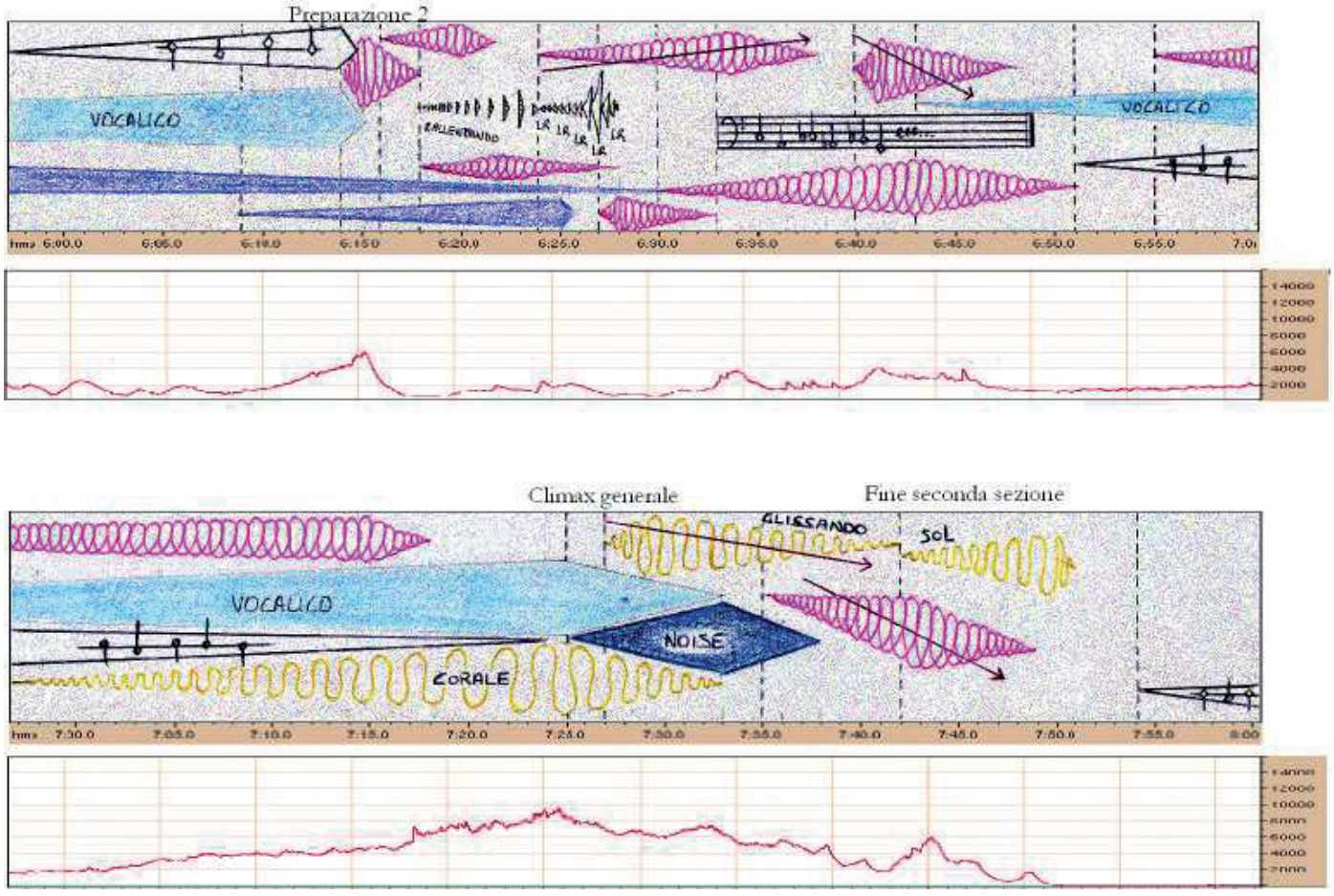
We like to present the fifth ICSC (International Csound Conference), eight years after its first edition, which included a celebratory edition which was held in Maynooth.
The Conference will take place in Cagli a small nevertheless historically important Italian town, being part of the Byzantine “Pentapoli Montana” in the Marche Region, culturally rich, plenty of monuments, historical vestiges and natural beauties.
If we compare Cagli with the greater centers that have hosted all the previous editions from Hannover to Boston, from St. Petersburg to Montevideo we can’t help seeing the difference.
However, the sense of belonging to this extraordinary international community challenged the organizers who nevertheless wanted to measure themselves, still having in mind some logistical difficulties that the project could present. But this, we believe, makes even more significant the active participation that we have received from all those who have with great enthusiasm and generosity cooperated in its realization, starting from the keynotes to the lecturers, from the composers, to the performers and the many volunteers we want warmly thank.
We are also grateful for the great work done by the reviewers of the papers and the compositions and to all the chairmen who have accepted our invitation.
But nothing arises by chance: the Marche Region holds some musical and technical-musical treasures appreciated all over the world such as the prestigious Conservatory of Music Rossini in Pesaro as well as a great construction electronic and acoustic instruments tradition, well known in every part of the world as witnessed by the Museo del Synth Marchigiano.
Moreover, it is not a coincidence that at the Rossini Conservatory is present an almost fifty-year Electronic Music school established in 1971 which still boasts a true vintage electronic music studio (LEMS) and a cutting-edge Ambisonic room (SPACE). And it is from this school that all the organizing group members came out and of which I am honored to be part. With a certain pride I can testify the great passion and dedication these three former students of mine (Alessandro Petrolati, Enrico Francioni and Anthony Di Furia together with Laura Muncaciu) have studied with enthusiasm and dedication to work over the years to cultivate their passion, competence and musical activity through this precious our travel companion which is Csound, now so determined and decisive in the realization of this extraordinary opportunity.

We wish sincerely to acknowledge the local institutions that have concretely supported us: first the Municipality-Culture Department and in particular Sandro Pascucci director of the Institution for the Cagli Municipal Theater and the Theater Academy of Cagli.

We want also to dedicate this event to the composer and dear friend **Fernando Mencherini**, who has honored with his art and his humanity, the city of Cagli in the world.

Eugenio Giordani, Cagli, September 2019

Keynotes


Preparazione 2
VOCALICO
VOCALICO
Climax generale
Fine seconda sezione
GLISSANDO
SOL
VOCALICO
NOISE
CORALE

Øyvind Brandtsegg
Norwegian University of Technology and Science
“21 years of live performance and installations with Csound”
- How Csound has always been there for me

ABSTRACT
I started using Csound in the late 1990’s, when it was just getting possible to use it in realtime. I had some musical desires that prompted me to look into it, even if the learning curve was pretty steep at the time. In the beginning I would combine Csound with hardware synthesizers and samplers, and also used Max to interface with external sensors and overall control. Over the years, as the available processing power increased and Csound developed, it was possible to build a live setup based exclusively on Csound. With the advent of the Csound API it was easier to interface to other languages and technologies. This opened possibilities for writing a realtime algorithmic composition system for use in live performance and sound art installations. Building an audio system for installations running several years required some extra attention to issues of stability and maintenance. As these systems became more complex, the need for modularization grew stronger. Some of the tasks involved could also be identified as being of a more general nature, allowing integration with off-the-shelf tools. This again enhancing Csound’s strong points as a development tool for customized audio processing. Isolating the components that actually need to be new, implementing them as opcodes or instruments.
The talk will be illustrated with projects done in Csound over the last 21 years, including recent efforts into crossadaptive processing and live convolution.

Steven Yi
Assistant Professor, Interactive Games and Media
Rochester Institute of Technology, USA
Tomorrow’s Csound

ABSTRACT
What might Csound look like in the future and how do we get there? In this talk, I will assess the state of Csound today, both the good and the bad, and propose a roadmap to guide us through the next generations of Csound.

Victor Lazzarini
Dean of Arts, Celtic Studies, and Philosophy, Maynooth University, Ireland

Csound + _: Notes on an Ecosystem

ABSTRACT
This talk discusses Csound as a sound and music computing system at the centre of an ecosystem of applications. For about fifteen years now, the software has developed a formidable array of connections to other programs, at various levels of user interaction, from high to low. Since its first release, Csound has provided an ideal studio platform for research and production, providing means for extensions and connections to other systems. In time, this ecosystem was widened as part of a calculated development strategy that placed Csound at the centre of a variety of applications. In this talk, we will explore the Csound ecosystem, with some illustrated examples. As part of this, we will also evaluate critically these developments, proposing some thoughts for the road ahead towards Csound 7.

Richard Boulanger
Professor of Electronic Production and Design Berklee College of Music Boston Massachusetts, USA

Dedicating My Musical Life to the Mastery of a Virtual Instrument – Csound
A Keynote Speech and Presentation to The 5th International Csound

ABSTRACT
I am truly honored to have been invited to present one of the keynote addresses at The 5th International Csound Conference - ICSC 2019 in Cagli (Pesaro-Urbino) Italy. Thank you so very much for this wonderful invitation to share some of my more recent thoughts, to perform some of my newest music, and most importantly, to publicly express my gratitude to so many, here at this conference, and in the international Csound community, whose instruments, code, research, and music have been constant sources of inspiration. And it is just these many “sources of inspiration” that are brought to mind in today’s keynote, especially the beautiful instruments and music of my student Shengzheng Zhang (*a.k.a. John Towse*), in whose memory this keynote is humbly dedicated.

Joachim Heintz
HMTM Hannover,
Germany
Head of Electronic Studio
FMSBW
**Learning Csound,
Learning Music**

ABSTRACT
Whatever music be, it is based on listening. Composing can be considered as listening to sounds and investigating their tendencies. Can learning Csound be considered as learning music by learning to listen? And how well is Csound suitet to materialize the composer's ideas about sounds and structures? The keynote will float around these questions - certainly not with a final answer at its end, but hopefully with some inspirations for the listeners.

Leonardo Gabrielli
Università Politecnica
delle Marche, Italy
From Le Marche to MARS: a journey through accordions, synthesizers and computer music

ABSTRACT
The region where ICSC 2019 takes place, Le Marche, is known worldwide for its long tradition of musical instruments manufacturing, which dates back to 1863, when - according to the tradition - Paolo Soprani built his first accordion. Since then, however, electronic pioneers and DSP developers have joined traditional accordion craftsmen to cyclically renew the industry, in the effort to keep up with global standards.
In 1988, the Bontempi-Farfisa group founded the IRIS lab, led by Giuseppe di Giugno and run by several outstanding developers and computer music researchers. The MARS workstation was one of its most prominent outcomes, and it was employed for several computer music works of the 1900s. It was programmed using ARES, a rich computer music platform based on graphical patching.
All this material and history is now coming back to light after the accidental discovery of machines and documents long forgotten in an abandoned factory. After reactivating and restoring computers and their software, thanks to the effort of the Acusmatiq-MATME association, we are now able to run ARES and its patches. This software will be described and linked to other existing computer music languages, including CSound and Max. The talk includes footages and documents produced by the Acusmatiq-MATME association.

Conference

```
if      i(gk_cicle)
i_Ccl   =           p8      ==      0 then
elseif i(gk_cicle)
i_Ccl   =           p9      ==      1 then
elseif i(gk_cicle)
i_Ccl   =           p10     ==      2 then
elseif i(gk_cicle)
i_Ccl   =           p11     ==      3 then
elseif i(gk_cicle)
i_Ccl   =           p12     ==      4 then
elseif i(gk_cicle)
i_Ccl   =           p13     ==      5 then
endif

ia      =
ib      =           p4
ifn1    =           p5
ifn2    =           p6
                    p7
indx
ilen    init
        =           - 1
reset:              ftlen(ifn1)
```

Processing Nature: Recordings, Random Number Generators and Real-Intrinsic-Extrinsic Perceptual Threads

Mark Ferguson,

University of Birmingham
mgf864@student.bham.ac.uk

Abstract. In this paper, I propose the concept of the *real-intrinsic-extrinsic perceptual thread* in acousmatic composition, which has become deeply intertwined with my wildlife sound recording practice and non-realtime use of Csound as a processing tool in the studio. The concept draws heavily from Denis Smalley's spectromorphological discourse regarding intrinsic-extrinsic threads and source-bonding (referenced throughout). Following a brief introduction (and in an attempt to articulate my thoughts from a practical perspective), I discuss processing approaches for two, recently-completed acousmatic works, in which Csound's random number generating opcodes were employed to break apart natural source recordings and create complex, secondary source materials. I then proceed to break down and describe the real, intrinsic, and extrinsic thread components separately. The paper concludes with a brief summary of the proposed concept and the role of Csound in its development, followed by a consideration of its apparent linear aspect and recent influence on my technical recording methodologies.

Keywords: wildlife sound recording, nature, random number generator, non-realtime processing, acousmatic composition, methodology, spectromorphology.

1 Introduction

My ongoing PhD research fuses two practices: wildlife sound recording and acousmatic composition.

Drawing exclusively from an ever-growing, personal sound library of species, soundscapes and abiotic phenomena, I use Csound (in conjunction with a range of other studio plug-ins and effects units) to extract hidden sonic detail out of my own wildlife recordings, generating complex, secondary source materials for fixed media electroacoustic works. These new sources are often combined and juxtaposed with original, unaltered (primary) source recordings to create unique sound worlds of my own imagining.

A fundamental component of this compositional workflow involves the non-realtime use of Csound instruments constructed around random number generating, control-rate

modulation blocks. These instruments, which often feature opcodes such as *randomi* and *jitter*, are repeatedly applied to wildlife recordings to generate new sonic content.

As my research has progressed, I have been particularly struck by the inherent power of such randomly-driven instruments to 'grow' or give birth to species and natural phenomena; to slowly tease sounds with 'imagined, extrinsic connections'[1] out of processed material (which was itself derived from unaltered, real-world recordings). I call this compositional workflow, which traces a perceptual path from real to abstract to suggestive, the *real-intrinsic-extrinsic perceptual thread:* a concept I attempt to formalise in this paper, and which draws heavily from Denis Smalley's spectromorphological discourse regarding intrinsic-extrinsic threads and source-bonding.

In order to articulate the concept, I first outline some random-number-based processing approaches for two, recently-completed acousmatic works. I then discuss the real, intrinsic and extrinsic thread components individually. The paper concludes with a brief summary of the concept and the role of Csound in its development, followed by a consideration of its apparent linear aspect and recent influence on my technical recording methodologies.

2 A Practical Overview

In the following sections, I provide brief, code-based overviews of my use of random number generating opcodes in Csound. The examples are taken from instruments used during the recent composition of two acousmatic works: *Deadwood* and *Shorelines*.

My intention at this stage is to outline the real-intrinsic-extrinsic perceptual thread concept from a practice-based perspective.

2.1 Deadwood

The first acousmatic composition as part of my doctoral research,[1] my main objectives with *Deadwood* were twofold: to lay an aesthetic foundation for future work, and establish the effectiveness of using wildlife source materials exclusively as compositional building blocks in the studio.

A seven-minute, octophonic piece, *Deadwood* takes the listener on an imaginary journey through the internal and surface sound worlds of a rotten branch. Whilst a lengthy source recording of internal branch vibrations serves as a consistent structural/sonic foundation, the main 'subjects' are actually tiny, invertebrate creatures, whose detailed sounds were crafted from highly-processed, layered recordings of wind, water and birdsong.

Pivotal in the generation of this so-called invertebrate material was the use of a relatively simple, eight-channel random panning instrument in Csound, whose unpredictable fluctuations were built around the following, *jitter* and *randomi* control block:

```
krndcps   jitter   kamp, kcpsMin, kcpsMax
krndcps1  =  krndcps+kamp
```

1 Generously supported by the Midlands4Cities Doctoral Training Partnership and AHRC.

```
krnd1     randomi  0, 1, krndcps1*iscale
krnd2     randomi  0, 1, krndcps1*iscale
krnd3     randomi  0, 1, krndcps1*iscale
krnd4     randomi  0, 1, krndcps1*iscale
```

The four, k-rate *randomi* outputs control the input panning argument values of four corresponding *pan2* opcodes, which output the original source sound as follows:[2]

```
asig      diskin2  ifile, ipitch*krndpitch
amix      = asig*iamp

a1, a2    pan2     amix, krnd1
a3, a4    pan2     amix, krnd1
a5, a6    pan2     amix, krnd1
a7, a8    pan2     amix, krnd1

; French 8 pair routing.
outo      a1, a3, a2, a4, a5, a7, a6, a8
```

This amounts to random, eight-channel control of four loudspeaker pairs, which can be configured to 'French' eight, 'American' double-diamond and other octophonic routing standards via *outo*.

Pushed to the extremes, *jitter* control was used to impose rapidly fluctuating panning values on loudspeaker pairs. After a certain threshold, source materials began to fragment and take on the textural character of pointillistic sounds generated through granular synthesis; at this point, rapid loudspeaker panning (which can be regarded as a form of spatial amplitude modulation) entered the domain of microsound.[3]

With repeated processing in Csound, these ripped, torn, intrinsically-detailed textures took on imagined, extrinsic characteristics: to my ears, strongly source-bonded to a microscopic sound world inhabited by invertebrates such as beetles, centipedes and millipedes.[4] These features were subsequently isolated and shaped to envelop the audience in the intimate, internal spaces of a piece of dead wood.

2.2 Shorelines

Building on the successful UK premiere of *Deadwood*, my processing approach with *Shorelines* focused on relatively simple bandpass filter designs. My intention once again was to generate abstract base textures from real-world source recordings, then repeatedly apply specific Csound instruments to draw out additional sounds with imagined, extrinsic connections.

2 After some experimentation, *pan2* was favoured sonically over the higher-level *pan*.

3 A fascinating approach to processing, and one I hope to explore further in the future.

4 Source-bonding and other spectromorphological terms are addressed in subsequent sections.

A series of reflections and re-imaginings from Talisker Bay Beach on the Isle of Skye, this ten-minute, octophonic work is also concerned with microscopic sonic detail, exploring seaweed textures, the imagined feeding processes of limpets, crabs and snapping shrimp, and jellyfish propulsion mechanisms. Themes of ancient volcanic activity, subterranean tectonic shifts and my own ancestral connections between Scotland and Northern Ireland are also explored.

In the early stages of composition, a random bandpass filter instrument was created in Csound to begin intensively processing and re-processing selected source recordings. The vast majority of materials used in the finished work were generated from soundscape and subterranean beach recordings, with excerpted river, stream and invertebrate sounds used for additional layering.

Three *randomi* opcodes were used in a control block for random modulation of filter frequency and bandwidth as follows:

```
kbwmod    randomi  imodmin, imodmax, imodcps
krandfr   randomi  irfrqmin, irfrqmax, irfrqcps
krandbw   randomi  ibwmin, ibwmax, kbwmod
```

Notable here is the use of an extra *randomi* opcode to modulate input arguments within the control block itself: in this case, the rate of random bandwidth values written to the variable *krandbw*.

Various *butterbp* filters were subsequently added in combination with *diskin2*, which could be programmed to skip as needed to interesting portions of the relevant source recording. Using the simple control architecture outlined previously, the frequency and bandwith input arguments of each *butterbp* filter were modulated directly, with a base filter frequency added to focus processing on relevant portions of the audible spectrum:

```
asig      diskin2  ifile, ipitch, iskip
abpfilt   butterbp asig, ibasefr+krandfr, krandbw
```

Given the tendency for random bandpass filtering to 'blow-up' and produce uncontrollable amplitude values, *clip* and *limit* opcodes were used at the output stage.

After employing this relatively simple, random bandpass filter design to generate base textures rich in intrinsic detail, repeated processing led to the emergence of highly complex gestures and textures, many of which were suggestive of marine species and their various behaviours.

3 Perceptual Thread Components

I now attempt to break down and explain the real, intrinsic and extrinsic components of the perceptual thread concept.

3.1 Real

Perception/exploration of the real as a wildlife sound recordist is the first stage in all of my compositional work. This essentially boils down to specialised field recording, using parabolic reflectors, tripods, highly-sensitive microphones and a range of other, custom-built, application-specific solutions.

Recording subjects range from birds, mammals and amphibians to plants, insects and atmospheres. These are what I call primary compositional sources: real-world recordings, entirely unprocessed (save for basic level adjustments and the removal of the very lowest frequencies, where applicable).

All of these primary sources are meticulously documented and catalogued as part of a growing, personal sound library.

3.2 Intrinsic

As Smalley notes, intrinsic features essentially refer to the raw, sonic characteristics of an electroacoustic work: 'sound events and their relationships as they exist within a piece of music'.[2] In other words, the intrinsic focuses on raw, internal spectromorphological detail, and is strongly bound up with Pierre Schaeffer's concept of reduced listening.[5]

In my own work, through the application of random processing methodologies using Csound in conjunction with other software, abstract, highly-complex textures and gestures are generated from original source recordings. It is at this stage that I begin to engage in the process of reduced listening and focus on perceiving intrinsic detail.

3.3 Extrinsic

As processing intensifies, external, extrinsic connections are often made: for example, throaty gestures may suggest complex, internal species vocalisations, or a rough, uneven texture may imply dragged motion across a pebble-strewn surface.

Smalley notes how the 'wide-open sonic world of electroacoustic music encourages imaginative and imagined extrinsic connections because of the variety and ambiguity of its materials'.[4] This ultimately equates to *source-bonding:* a concept also created by Smalley to represent the intrinsic-extrinsic link, and defined by him as 'the natural tendency to relate sounds to supposed sources and causes, and to relate sounds to each other because they appear to have shared or associated origins'.[5] He also notes how, for the listener, source bondings can be actual or imagined, and may never have been envisaged by the composer in the first place.[6]

These perceived extrinsic connections often have a powerful influence on how a work progresses. In *Shorelines*, vivid extrinsic connections were made to perceived, aquatic species snaps, pops and propulsion mechanisms when working with highly-processed sand and water material. These details were subsequently extracted and worked into the final version of the piece.

---

5 Described by Smalley as, 'an abstract, relatively objective process, a microscopic, intrinsic listening'.[3]

4 Conclusion: Threading Everything Together

In proposing the concept of the real-intrinsic-extrinsic perceptual thread, what I have ultimately attempted to do is formalise various aspects of my own compositional process, as opposed to articulate a fully-fledged theory or body of terms. At this early stage of my doctoral research, I find the concept particularly useful as an aid to creative thinking, and to instil an awareness of workflow: of how to trace a path from initial recording to intensive processing and, finally, the imaginative construction of sound worlds through perceived extrinsic connections.

The fact that Csound has played a key role in the development and refinement of my compositional process and the notion of perceptual threads is no accident. I find the use of random number generating opcodes complementary to many of the unpredictable and exciting behaviours encountered whilst recording wildlife. I also see workflow parallels between the use of Csound and the practice of wildlife recording. Planning for the best recording opportunities essentially boils down to planning against the highly unpredictable elements of nature, narrowing-down and quickly adapting once a particular target species emerges; similarly, the unexpected sonic material thrown out by randomly-driven Csound instruments must be adapted to through further input argument refinements within the control block, in order to continually focus processing and chase sonically engaging material.[6]

Although I have highlighted the linear aspect of the perceptual thread concept, this has largely been for the benefit of clear presentation and outlining of compositional process. In my own experience, sonic perception does not function so linearly, and each component (real, intrinsic, extrinsic) is deeply intertwined; it is not a simple matter of proceeding in a single direction. For example, whilst recording the real has certainly informed my intrinsically-focused studio work and subsequent connections with the extrinsic, the extrinsic (what I have imagined and constructed compositionally) has also re-informed my perception of and approach to the real.[7]

Perhaps the most striking influence in this regard has been the modification of technical recording methodologies. In my recent search for bumblebee sounds, my approach has centred largely on using a small, handheld microphone to follow the bees and capture their flight buzz. This has now broadened to include highly-proximate mandible and body sounds as the bees gather nectar and pollen inside flowers, using miniature microphones placed directly on petals and other plant structures. I believe the search for these hidden sounds has been directly influenced by extrinsic connections to invertebrate activity made during intensive processing for *Deadwood*.

It will be interesting to see where all of this leads over the course of my doctoral research, as I explore various Csound instrument designs in my continued exploration of real-intrinsic-extrinsic perceptual threads.

---

6 In addition, I find my continued, preferred non-realtime use of Csound complementary to a wildlife recording approach centred on patience, and the requirement to listen back intently to many hours of recorded material.

7 The possibility for perceptual thread truncation should also be noted: in the future, I may sever the link between intrinsic and extrinsic components, instead developing a non-source-bonded, abstract sonic palette from wildlife source recordings.

5 References

1. Smalley, D.: Spectromorphology: Explaining Sound Shapes. Organised Sound (2)2, 110 (1997).
2-6. Ibid.

A Musical Score for Csound with Abjad

Gianni Della Vittoria,

Liceo Musicale “Canova” di Forlì

giannidellavittoria.audio@gmail.com

Abstract. This paper presents the advantages of using a traditional musical score to make music with Csound. After illustrating some alternative approaches, examines Abjad, a Python library for printing music through Lilypond, and explains the technique for linking Abjad to Csound. Since in order to compose a musical score of synthesizers it is necessary to manage complex envelopes, particular attention is paid to how to represent the envelope profiles of the various parameters on a score and how to interpret them in Csound. As the system is open to several possibilities, the choice is proposed that seems simpler and allows the user to see the envelope as a musical element to be set on a staff, providing a better overview of the whole composition.

Keywords: Score, Abjad, Python, Algorithmic composition, Lilypond

1 Introduction

When writing music in Csound, the issue of realizing the score, as it is well known, can be solved in a vastness of ways. There is a variety of approaches ranging from direct compilation of the score to the use of third-party software. Each of them has its own peculiarities that can best fit the sometimes opposite preferences of those preparing to compose music with Csound.

One of the necessities that arise during the creation of an average complex score is to organize musical events at multiple levels, and not as a simple succession of instances. Today this is certainly possible with the sole use of Csound, without having to resort to external tools. One typical arrangement, in this sense, is to write the score to instantiate instruments which in turn call other instruments according to a specific algorithmic plan. The score, thus, hosts not so much the "notes", but the musical "phrase". By extending the principle, the orchestra can organize calls at multiple levels, leaving the score the task to coordinate the highest level.

There are also a whole series of external programs that deal with the problem of managing music organization levels in very interesting ways, such as Blue by Steven Yi, a rich environment where it is possible to view series of events arranged on a temporal plan with the chance of nested levels, or athenaCL by Christopher Ariza, which instead uses a command line approach to determine series of events according to well-defined parameter masks with many categories of envelopes.

It is useful to be able to represent the complexity of musical thought in some way, so as to be able to observe its unraveling over time. In this article I am going to present an-

other way to organize and visualize the Csound score, that is, through a traditional musical score adapted to Csound's needs.

2 Preparing a Musical Score for Csound

Instantiating Csound with a musical score has been done in various ways. One of them is via MIDI: you create a score with score editing software like MuseScore, extract the MIDI version and import it into Csound. This mode has various appreciable aspects, such as the ease with which Csound can be played by connecting each instrument of the musical score to the desired sound. However, it is not so easy to integrate a mechanism for transferring the parameter envelopes. How to describe them in a musical score? And solved this, how to be able to transfer them in real time to Csound, given that Csound should know in advance where the envelope will end up for at least the duration of the entire note?

Another way is by using visual programming software such as Open Music or PWGL. Open Music, for example, has several libraries specifically built to communicate with Csound, but, alongside the wealth of algorithms for processing various musical parameters, there are constraints such as the difficulty of managing dynamic markings or other musical symbols.

2.1 Lilypond and Abjad

To get a traditional musical score full of details, Lilypond is certainly an excellent choice. This software requires ASCII text notation, which is then compiled into documents such as PDF, PostScript, SVG of acknowledged quality. While remaining faithful to its textual nature, Lilypond has seen the contribution of various external GUI programs to facilitate the introduction of musical content (Frescobaldi, Denemo, Canorus, ...). Furthermore, there are many programming languages that use Lilypond to visualize algorithmic music. Among them Abjad stands out, an extensive Python library that allows the user to work with various Lilypond elements implemented in a sophisticated class structure. The advantage of Abjad lies in the fact that the algorithmic composition becomes simpler than if you were to create a pure Lilypond text file, being practically every element manageable with simple class instances.

Given the richness and the ability to produce very complex graphics, Abjad and Lilypond are particularly suitable for the composition of contemporary music; hence the idea of using these tools in conjunction with Csound.

2.2 From Abjad to Csound

Various ways can be used to connect Abjad to Csound. Here the python ctcsound module is taken into account: it allows you to compile csound through a variable containing the .orc and .sco text.

To show a short example, let's consider a simple orchestra with this beginning of Csound score

```
  sco_text = ['''
  f1 0 8192 10 1 .1 .01 .02 .03 0 0 .01 0 .02 .01 0 .02
  ;      amp midi attack decay pan
  i1 0 1 0.5 60   0.01   0.1   0.5
  ''']
```

After creating the musical score in Abjad, in which each instr has a dedicated staff like in a traditional orchestral score, it is necessary to extract the onset, duration and pitch information relying on the abjad parser, so that it selects the events iteratively, avoiding rests. The iteration must take place over logical_ties, so as to consider tied notes as individual units.

```
for logical_tie in
abj.iterate(score).logical_ties(pitched=True):
  offset = abj.inspect(logical_tie).timespan().start_offset
  offset_seconds = 60*offset/(metronome_mark.reference_du-
ration * metronome_mark.units_per_minute)
  dur = abj.inspect(logical_tie).duration()
  dur_seconds = 60 * dur /
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
scoLine = ['\ni1 ', str(offset_seconds.__float__()),
str(dur_seconds.__float__()), '.5']
scoLine.append(str(60 + abj.NumberedPitch(logical_tie[0])))
```

The onset times (p2) are taken from the Abjad inspector through the timespan().start_offset method, which returns the value in musical figures of duration. The following line makes the conversion in seconds, taking into account the metronome. The same procedure is applied to the duration, while the frequency is drawn from from the NumberedPitch class of Abjad, which is 0 for middle C, 1 for C#, 2 for D and so on. By adding 60 they can be easily intercepted by Csound through the cpsmidinn opcode.

Finally, the other p-fields are added, which obviously could be freely processed in python or derived from the abjad score. After calling csound through ctcsound, what you get is real-time listening and visualization of the score.

2.3 Microtonal tuning

The representation of micro-intonation on a musical score always raises questions that force us to take sides. The choice of the best notation system depends on the compositional needs and can be quite different from one piece to another.

Fortunately, the flexibility of Abjad provides the freedom of choice you prefer. A fairly general approach could be the quarter-tone notation, which is easily readable, with deviations expressed in cents by a small number before the note. This number would therefore range from +25 to -25 and can be float for fractions of a cent.

Once paired with the Note class, all the python code has to do is parse and convert this value to a decimal midi note: Csound will take care of the rest (the cpsmidinn opcode is able to correctly evaluate decimals).

3 Graphical representation of envelopes

Just as in an instrumental musical score it is important to show every detail relating to the execution of each instrument, to put together a score of synthesizers it is useful to define the dynamic developments of the various parameters by displaying the envelopes of the most significant parameters. Defining them only in Csound would require a constant reuse of envelopes with the same number of p-fields, and in any case the fact remains that they cannot be displayed. Not even the Open Music maquette comes to the punctual clarity that only a musical score can allow. Defining the envelopes in Abjad, on the other hand, allows them to be diversified for each event and to show them all in a clear overview.

Also for this there can be various approaches. For example, we could use traditional dynamic markings, such as pp, mf, crescendo, associating them with certain amplitude patterns. Alongside the undoubted advantage of easy readability, however, there is the downside of a reduced number of nuances, compared to the possibilities of a synthesizer. However it would not be bad to employ them in conjunction with other systems.

Another way would be to imitate the envelope profiles of the most popular synthesizer graphics, technically feasible thanks to the powerful PostScript language that Lilypond introduced. Here, however, the problem would consist in deciding how to get to the Csound transcription, which can be solved in various ways, but perhaps a bit cumbersome. For example, PostScript uses elegant cubic Bezier curves that could be converted to Csound, but Csound for now only has the quadratic Beziers (GEN "quadbezier") and these do not have the same flexibility as cubic ones.

3.1 Envelopes on musical staves

Perhaps the fastest method to represent complex and articulated envelopes is to use normal additional staves where for each parameter the profiles will be drawn through pitched notes. Handling notes and durations is the most natural thing in Abjad and this would allow for easy algorithmic manipulations. Moreover, it can be interesting to have an approach of proportional durations definable with rhythmic figures, where more musical parameters of the same instrument can be easily compared and played with subtle synchronizations.

Here we will give an example with a single instrument (FM1) and a single parameter (the IndFM1 modulation index) for the sake of clarity. Let's start with the orchestra loaded in ctcsound.

```
orc_text = '''

  sr     = 48000
  ksmps  = 8
  nchnls = 2
```

```
  0dbfs  = 1

  instr FM
   kamp = .3
   kcps = cpsmidinn(p5)
   kcar = 1
   kmod = .6
   kndx table p4,100; read from Abjad staff "IndFM"
   asig foscili kamp, kcps, kcar, kmod, kndx, 1
   outs asig, asig
  endin

  instr lin
   kval linseg p5,p6,p7,p8,p9,p10,p11,p12,p13,p14,p15,p16,
p17,p18,p19,p20,p21,p22,p23,p24,p25
   tablew kval, p4, 100
  endin
  '''
```

We want to infer the vertices of the envelope for a linseg to handle the modulation index of the FM from the notes on a specific staff. We establish that the range goes from 0 (middle C) to 20 (C 2 octaves above). Since we do not know *a priori* how many vertices the parameter IndFM1 will have for each FM1 instance, we have created an instr "lin" with a very long p-fielded linseg. If less p-fields are used, Csound actually warns, but does not protest and this allows us the flexibility to create very different profiles in the score. This is the starting score.

```
  sco_text = [''']
  f 1 0 16384 10 1 ;sine
  f 100 0 2048 -2 0 ;only for envelopes
  ;How score should look like
  ;            channel midinote
  ;i "FM" 0 5  0       69
  ;            ch linseg_values(no fixed number of pfields)
  ;i "lin" 0 5 0  1 2.5 20 2.5 1
  ''']
```

As you can see from the commented part, each "FM" is activated simultaneously with its "lin" instr and linked by a unique channel so that while "lin" writes, "FM" reads the k data. For each new instance of "FM" the reading channel is updated, provided by ftable 100, so that there is never any interference between any overlapping instances.

The conversion from musical score to Csound happens as follows

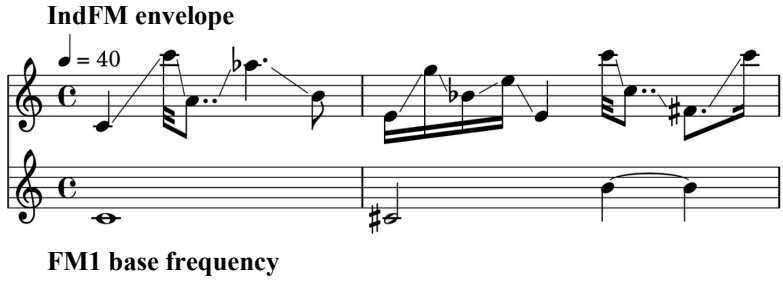
```
channel = 0
instrument_list = ['FM1']
for instrument in instrument_list:
  for logical_tie in abj.iterate(score[instrument]).logi-
cal_ties(pitched=True):
```

```
    offset = abj.inspect(logical_tie).timespan().start_off-
set
    offset_seconds = 60*offset/
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
    dur = abj.inspect(logical_tie).duration()
    dur_seconds = 60 * dur / (metronome_mark.reference_du-
ration * metronome_mark.units_per_minute)
    scoLine = ['\n','i
"FM"',str(offset_seconds.__float__()),
str(dur_seconds.__float__()), str(channel)]
    sco_text.extend(scoLine)
    sco_text.append(str(60 +
abj.NumberedPitch(logical_tie[0])))
    #Envelope for IndFM
    sco_text.extend(['\n','i "lin"',
str(offset_seconds.__float__()),
str(dur_seconds.__float__()), str(channel)])
    for segment in
abj.iterate(score['IndFM1']).logical_ties(pitched=True):
      segmentStart =
abj.inspect(segment[0]).timespan().start_offset
      if offset <= segmentStart < (offset+dur):
        value = abj.NumberedPitch(segment[0]).__float__() /
24 # scaled 0 -> 1 between c' c'''
        segmentDur = abj.inspect(segment).duration()
        segmentDur_seconds = 60 * segmentDur /
(metronome_mark.reference_duration *
metronome_mark.units_per_minute)
        scaledValue = value * 20  # scale relative to the
particular parameter
        sco_text.extend([str(scaledValue),
str(segmentDur_seconds.__float__())])
    sco_text.append(str(scaledValue))# last value repeated
  channel += 1
sco_text = ' '.join(sco_text)
```

As can be seen, the conversions are quite similar to those shown above, but this time the iteration that inspects the notes-vertices of the IndFM parameter is filtered by the time window corresponding to the duration of the relative FM1 note. Finally, the rescaling operated on 24 semitones is spread over a range from 0 to 20.

Notes and final image.

```
IndFM1staff = abj.Staff("c'4 c'''32 a'8.. a''4. b'8 e'16
g'' b' e'' e'4 c'''32 c''8.. fs'8. c'''16 ", name='IndFM1')
FM1staff = abj.Staff("c'1 cs'2 b'4~ b'4 ", name='FM1')
```



References

1. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
2. Lazzarini, V.: Computer Music Instruments. Springer (2017)
3. Baca L., Oberholtzer J. W., Trevino J., Adan V.: "Abjad: An Open-Source Software System for Formalized Score Control" in Proceedings of the International Conference of Technologies for Music Notation and Representation, 2015
4. Oberholtzer J. W.: A Computational Model of Music Composition. Doctoral dissertation, Harvard University, 2015
5. Trevino, J. R.: Compositional and Analytic Applications of Automated Music Notation via Object-oriented Programming. Doctoral dissertation, University of California, 2013

Modeling of Yamaha TX81Z FM Synthesizer in Csound

Gleb G. Rogozinsky and Nickolay Goryachev

The Bonch-Bruevich St.Petersburg State University of Telecommunications
gleb.rogozinsky@gmail.com

Abstract. The paper presents authors' original method of hardware synthesizers and sound processing devices modeling, focusing on the Yamaha TX81Z FM synthesizer as an example. The Csound 6 is used for the software simulation of original TX81Z, which is 4-operator FM synthesizer from 1987, well-known for its peculiar C15 preset called *Lately Bass*, and total amount of 8 waveforms. The paper gives a review of the most prominent FM synthesizers, considering both hardware and software implementations, brief description of Yamaha TX81Z features, the review of modeling method used by authors, and analysis of modeling results. During the modeling, we measured and modeled DAC unit of TX81Z to achieve the same waveforms. It was done using MATLAB *Filter Design Tool*, prior to code the corresponding pair of LP and HP filters in Csound. After that step, we modeled oscillators and envelopes. The given figures show the comparison between original TX81Z recorded sound samples and ours Csound-based model.

Keywords: sound synthesis systems, modeling, Csound, FM synthesis

1 Introduction

The advanced development of sound synthesis and processing software greatly contributed to the reduction of corresponding hardware. The noticeable progress in the field of computer sound has provided software implementations which surpass their hardware counterparts in both sound quality and functionality. At the same time, the field of electronic and computer music directly affects a number of creative and aesthetic factors. In the cultural plane, the timbre of the epoch of electronic music is essential. Thus, the preserving the corresponding electromusical instruments and / or their accurate modeling using computer-aided synthesis algorithms are both relevant and topical. The 80s and 90s of the 20th century were of rather important for the modern timbral landscape of electronic music. It was during these two decades that a significant number of synthesizers were released, and the sound of those synthesizers determined the timbral thesaurus of existing musical styles of electronic music. With detailed study of some (Yamaha DX7, Roland TB303), the modeling of many other synthesizers is still on its way to mature. In particular, among the hardware implementations of the frequency modulation synthesis (FM synthesis) that dominated in the 80s, in

addition to the well-known Yamaha DX7, later models, such as the TX81Z and FS1R, to be outlined. The first had defined the bass sound for the *Eurodance* style popular in the mid-90s. The existing software models are not widely used and are being questioned by experts, and besides, the closeness of commercial software implementations excludes the study of algorithms by experts in the field of computer music and sound processing.

Thus, one of the topical issues of computer music is the exact (perceptual and algorithmically identical) modeling of various sound synthesis and processing devices. The article proposes a solution to this issue based on the method of software modeling of sound synthesis devices, which takes into account the hardware implementation features (the effect of the synthesizer DAC on the generated waveform) and examines the generated sound objects in different time-frequency sound space planes. The considered method can be applied to various devices, allowing the possibility of obtaining the frequency response and other characteristics of the DAC.

2 Prominant hardware and software implementations of FM synthesis

The digital FM synthesis was first proposed by John Chowning at Stanford University in 1967-68, licensed by the Japanese company Yamaha in 1973 [1, 2]. The most known hardware implementation of FM synthesis is the Yamaha DX7, released in 1983. Yamaha stopped production of hardware FM synthesizers in the early 90s with the transition to the production of multi-functional digital audio workstations. Currently, FM synthesis is mainly implemented in software synthesizers, such as Native Instruments FM7 / FM8, Image-Line Sytrus, etc. At the same time, almost any modern synthesizer features frequency modulation between (at least) a one pair of generators. Table 1 contains a summary of the main FM synthesizers in chronological order. Budget analogs of the DX7, i.e. DX9, DX21, DX27, DX100, FB-01, as well as the corresponding rack versions like TX7, TX802 were intentionally omitted.

The table shows that the most of FM implementations recreate the features of the original Yamaha DX7, which confirms the thesis about the relevance of developing models of other hardware. Much of the existing software models use closed code. Among the existing open-source DX7 software models, the most widely known is the Russell Pinkstone's model from Csound Book [3]. Along with good algorithmic accuracy, the model meanwhile does not take into account the DAC influence on signal. In addition, no correspondence was made between the parameters of the simulated Yamaha synthesizer (which are set in the range from 0 to 99) and real values of one or another quantity.

3 Description of Yamaha TX81Z

The Yamaha TX81Z synthesizer (1987) is a four-operator FM synthesizer in rack-mounted version. The main difference between TX81Z and others from DX-

Table 1. Basic hardware and software implementations of FM synthesis

Name (Year)	Form	OPs/ALGs	Waveforms	Compability
Yamaha DX7 (1983) Yamaha DX7-II (1987)	Hardware, 61 keys	6 OPs /32 ALGs	Sine	-
Yamaha TX81Z (1987)	Rack module, 1U	4 OPs /8 ALGs	8 waves	-
Yamaha FS1R (1998)	Rack module, 1U	8 OPs /88 ALGs	8 waves	-
NI FM8 (2006)	Software plugin	6 OPs /-	32 waves	DX7, DX11 TX81Z
Image-Line Sytrus (2008)	FL Studio plugin	6 OPs /-	Any	-
asb2m10 Dexed (2016)	Software plugin	6 OPs /32 ALGs	Sine	DX7
Hexter (2004)	Software plugin	6 OPs /32 ALGs	Sine	DX7
Arturia DX7 V	Software plugin	6 OPs /32 ALGs	25 waves	DX7
LoftSoft FMHeaven (2004)	Software plugin	6 OPs /-	16 waves	DX7, TX81Z
Oxe FM Synth (2004)	Software plugin	6 OPs /-	6 waves	-
DXi FM (2011)	iPad app	4 OPs / 8 ALGs	12 waves	-
KQ Dixie (2018)	iPad app	6 OPs / 32 ALGs	Sine	DX7
Primal Audio FM4 (2014)	iPad app	4 OPs / 8 ALGs	8 waves	-
Yamaha reface DX (2015)	Hardware, 37 keys	4 OPs / 12 ALGs	Sine	-
Korg Volca FM (2015)	Hardware, 15 keys	6 OPs / 32 ALGs	Sine	DX7

series is the ability to use various waveforms (8 waveforms are used). The most featured and well-known sound of this synthesizer is the C15 *Lately Bass* preset, which can be heard in many compositions of the dance scene of the early 90s. At the beginning it is necessary to recreate the TX81Z waveform array as accurately as possible. Figure 1 demonstrates the discrepancy between the ideal waveform 5 and its real form, recorded through a sound card with a sampling frequency of 96 kHz. This discrepancy is due to the non-linearity of the frequency response of the original synthesizer's DAC. This leads to modeling the DAC used in the TX81Z. The measurement of magnitude response was carried for the 1-OP sine wave mode (Fig. 2) and modeled using Matlab's Filter Design Tool as a system of two serially connected low-pass and high-pass filters.

Csound 6 was chosen as the software for real-time sound synthesis. Below we give an example of the implementation of the corresponding filters in the form of User-Defined Csound opcodes. The next step to building a model is to measure the parameters and characteristics of the synthesizer, for example, detune parameters. Most of the TX81Z parameters were manually measured using time, frequency and phase measuring tools in *Cockos Reaper*. For cases where it was possible, some parameter tables were approximated with corresponding math functions i.e. power of N, others are written directly in ftables as values.

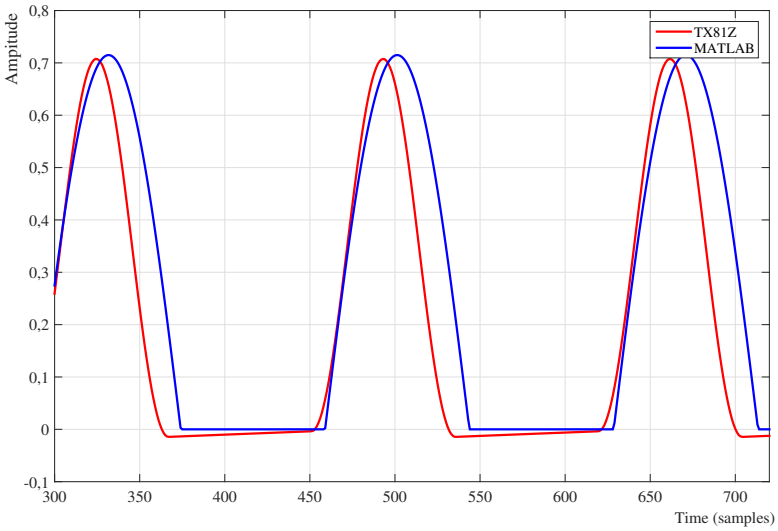


Fig. 1. Comparison of waveforms of type 3 (positive half-period of sine)

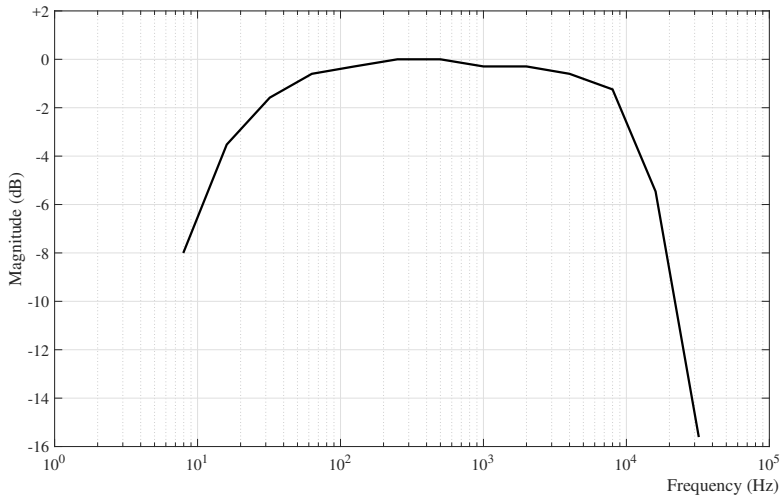


Fig. 2. Measured magnitude response of Yamaha TX81Z DAC

```
;LP filter
opcode TX_LP, a, a
setksmps 1
aL xin
aD0 init 0
aD1 init 0

iA1 = -0.5100490981424427
iB0 = 1
iB1 = 1

aD2=aD1
aD1=aD0
aD0=aL-aD1*iA1
aout=aD0*iB0+aD1*iB1
xout aout*0.24497545092877862
endop
```

```
; HP filter
opcode TX_HP, a, a
setksmps 1
aL xin
aD0 init 0
aD1 init 0

iA1 = -0.99869495948492626
iB0 = 1
iB1 = -1

aD2=aD1
aD1=aD0
aD0=aL-aD1*iA1
aout=aD0*iB0+aD1*iB1
xout aout*0.99934747974246307
endop
```

UDO opcodes for TX81Z DAC modeling filters.

At the present state, the model implements the generation of all 8 types of original waveforms as a table oscillators, envelope generators of AR-D1R-D1L-D2R-RR type and operator connection algorithms (overall to 8 algs). Figures 3 and 4 show the results of the comparison of the original waveforms in the frequency domain. Figures 3 gives the comparison for only first pair of operators (OP2 - OP1). Figures 4 gives the comparison for the complete C15 preset, using 4 operators with a feedback on a OP4. Obviously, the operator pair is easy to model, thus the Csound FM instrument spectrum looks close to the original. The C15 sound still needs some improvements though hearing tests give promising results. During the estimation of modeling results we try to compare signals both in time and frequency domains. Also hearing tests results are considered.

4 Conclusion

The resulting method can be applied to simulate various hardware devices for the synthesis and processing of sound. The lack of accurate software models of hardware synthesizers determines the feasibility of continuing such studies. The method can be especially popular for modeling digital synthesizers - both early samplers and FM synthesizers of the 80s, and virtual analog devices of the end of the 90s. At the next stage, we plan to implement the reading of the original TX81Z presets in MIDI SysEx-format to develop a user interface and conduct a subjective assessment of the accuracy of modeling. The resulting code will be compiled as a VSTi / AU plug-in using Csound Cabbage. The project link is `https://github.com/gleb812/cs81z`

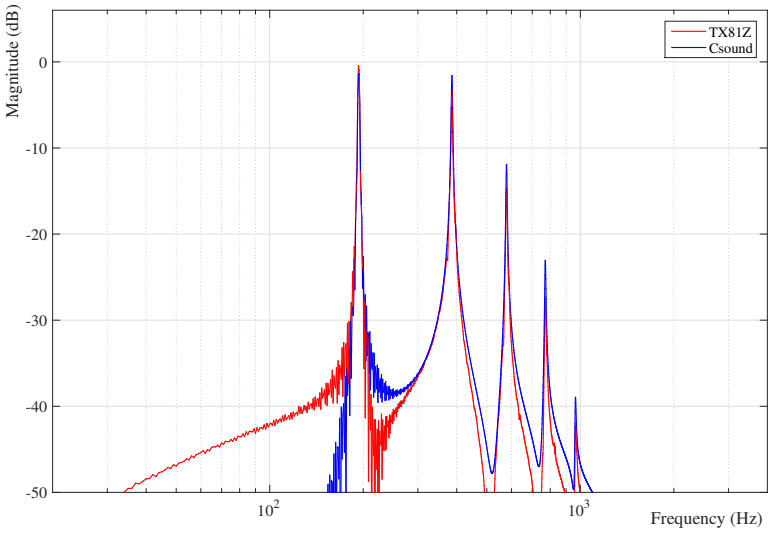


Fig. 3. Yamaha TX81Z vs Csound spectra for two operators only

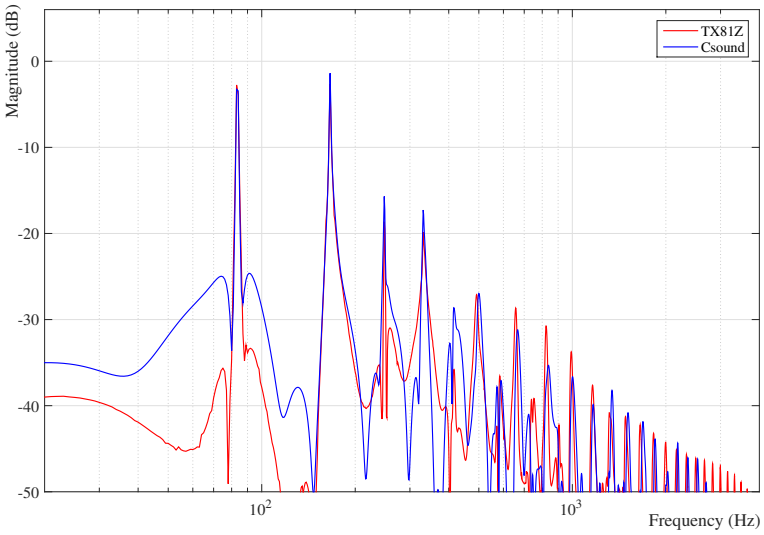


Fig. 4. Yamaha TX81Z vs Csound spectra for the C15 Lately Bass preset

References

1. Chowning, J.: The synthesis of complex audio spectra by means of frequency modulation. The Journal of the Audio Engineering Society 27(7), 526–534 (1973)
2. Chowning, J., Bristow, D.: FM Theory and Applications by Musicians for Musicians. Yamaha Music Foundation, Tokyo (1986)
3. Pinkston, R.: A Guide to FM Implementation in Csound. In: R. Boulanger (ed.) The Csound Book, pp. 261–280. MIT Press, Cambridge (2000)

MIUP Portable User Interface for Music

Example of jo_tracker - a tracker interface for Csound

Johann PHILIPPE*

No Institute Given

Abstract. This article presents graphical tools designed to work with Csound. First section introduces the context in which those tools were built. Then, second part presents MIUP, an open source graphical library designed to build audio softwares. Finally, last part describes jo_tracker, a tracker software for Csound built with MIUP.

Keywords: MIUP, IUP, jo_tracker, Csound, User Interface, Lua, C++

1 Introduction

In it's relatively recent relashionship with softwares and computing tools, contemporary music has been experiencing many difficulties inherent to preservation of tools. This is particularly true for mixt and electroacoustic music, which, at the same time, benefits a lot of those technologies. However, it is harder today to play a music from the last thirty years than to play some complex musics from the early 20th century. Most of the time, this difficulty appears when the electronic part of a music is based on an old program that is no longer working, or an old Max MSP patcher. In this kind of cases, it is almost an archeological work to find how was supposed to work this old software, and it becomes longer to update this program than it was to first create. As it is an important preservation problem, it must be a concern for composers who uses this technologies. Csound likely stands as the best alternative to this preoccupations, for many reasons. First, Csound is open-source, so a program could be reconstructed from scratch. Also, Csound developpers take care of retro-compatibility, which allows an old program to be played on a recent distribution. Moreover, Csound has an important community sharing knowledge about sound and music computing. It is why tools presented in this article are mostly designed to work with Csound, yet it could work with other audio programming languages.

2 MIUP - a C++ user interface library for music

MIUP stands as portable user interface for music. This is a cross platform toolkit designed to easily write musical softwares using Csound. It uses IUP [1], a cross platform C library working with system native interface elements. It

* Thanks to François Roux.

is fully distributed as source code under MIT license (except a modified version of CsoundThreaded). MIUP has a few dependencies : all of them are licensed under MIT. Some of them are included as source code inside the project repository. Though, a few ones need to be linked to the program in order to work : the IUP library (all of it, including canvas draw and im), sqlite3, sndfile. Of course, Csound needs to be linked to the program if it is used as the application's audio engine. MIUP project contains a set of C++ class which describes useful tools for musical softwares : sliders, levelmeters, spinboxes (...), but also more complex class like a curve editor, a waveform visualization plot, a code editor for Csound... MIUP provides a connection mechanism that can be used to fire callbacks between several objects. It also provides a callback system to retrieve Csound output channels and display their values inside the interface. MIUP allows user to take full benefit from Csound with a flexible user interface. It allows user to write its own widgets, following a simple design diagram. This library was first written in Lua as a toolkit to write a particular software : the jo_tracker. It has been fully rewritten in C++ to extend its potential to other softwares.

2.1 MIUP basic functionnalities

IUP C library provides one data type that is used for every interface element :

```
Ihandle *
```

The interface element can be initialized like this :

```
Ihandle *IupButton(const char *action)
```

It also provides a set of functions that can be used to modify attributes of interface elements.

```
void IupSetAttribute(Ihandle *element, const char *name, const char *value)
const char * IupGetAttribute(Ihandle *element, const char *name)
```

Since every IUP element is of type *Ihandle **, users can easily construct some complex interface architecture, imbricating boxes (layouts) inside other boxes. MIUP base class `MiupObj` is a simple C++ wrapper to this C mechanism. It is recommanded that every MIUP interface element inherits from this class. The class contains a private `Ihandle *` that holds a reference to the interface element, and implements some public methods to modify attributes of the object :

```
void setAttr(const char *name, const char *value)
const char *getAttr(const char *name)
Ihandle *getObj()
```

Every MIUP widget inherits from this base class called MiupObj, and constructs the interface element with a different IUP initializer function. The `getObj()`

method returns the `Ihandle *` pointer. It is a necessary method for every interface element, and it allows a compatibility with the standard IUP syntax. The `Ihandle *` element returned by the `getObj()` method will be the one displayed in the interface.

2.2 Main Features

Here is a quick list of the available widgets and classes :
-Widgets : button, toggle, sliders, levelmeter, gainmeter, spinbox, matrix
-Plots : curve editor, waveform visualizer (realtime and soundfile)
-Containers : boxes (vertical, horizontal, scrollable), radio...
-Audio : Threaded callbacks, AudioFileReader, CsoundThreaded (slightly modified)
-Text : Text widget (one line, or multiline), Scintilla editor
-Utilities : filesystem, string conversion, templated print facilities, some data types, signaled value...
It also contains a set of features, including JSON [5], Signal and Slot [4] used to connect objects, and a thread safe callback mechanism for Csound control channels.

2.3 Code Examples

Any MIUP program must contain at least a call to `Miup::Init()` at the beginning, and `Miup::MainLoop()`, `Miup::Close()` at the end. Creating a widget can be as simple as :

```
Slider<double> sl(-90,-90,6,0.01,"HORIZONTAL");
LevelMeter<double> lv(-90,-90,6);
Button but("PLAY"); // creates a button displaying "PLAY"
```

Widgets can be pushed inside containers like this `Vbox vbx(&sl,&lv,&but);`.Then, the final container must be pushed in a new dialog.

```
Dialog dlg(&vbx);
dlg.show();
```

When their internal callback is triggered (like button click), Widgets emits signals that can be connected to any function or method with the same signature. For example, we could do :

```
sl.valueChangedSig.connect(&lv,&LevelMeter<double>::setLevelMeter);
```

This would update level meter value to slider value. If the signature is different, the connect method can also be called with a lambda as argument. It allows to connect one signal to multiple actions in one statement.

```
CsoundThreaded cs;
sl.valueChangedSig.connect([&cs](double val){cs.setChannel("gain",val);});
```

This would send the slider value to Csound as a control channel when it changes. Interface can also be refreshed with Csound control output control channels. This functionnality works by passing a std::function as argument to CsoundThreaded pushMethodCallback method. It can be done using lambdas, or std::bind.

```
cs.pushMethodCallback("level",[&lv](double val){lv.setLevelMeter(val);});
```

Internally, Csound will look the "level" channel value at k-rate, and push the lambda and the value in a queue. The queue is then processed in the `Miup::MainLoop()` function. This part of the work benefits from the great work of Michael Gogins on *CsoundThreaded* [2], which has been slightly modified to perform those callbacks.
Example MIUP Simple program playing a sine

3 J_tracker - a tracker interface for Csound

Tracker softwares, also called soundtracker, are musical sequencers which tracks are based on a grid. Users can write values in the grid, corresponding to synthesizers instructions. Those are often MIDI instructions (note, velocity), that can be added to some basic controls on the note (pan,delay...). Sequencer starts at the top of the grid, and iterates over each line, before reaching the last one. Soundtrackers can be thought as step sequencers with an improved writing precision. This kind of software was very popular in the 1990's. Today, only a few of them are still under active development, including Renoise and OpenMPT.

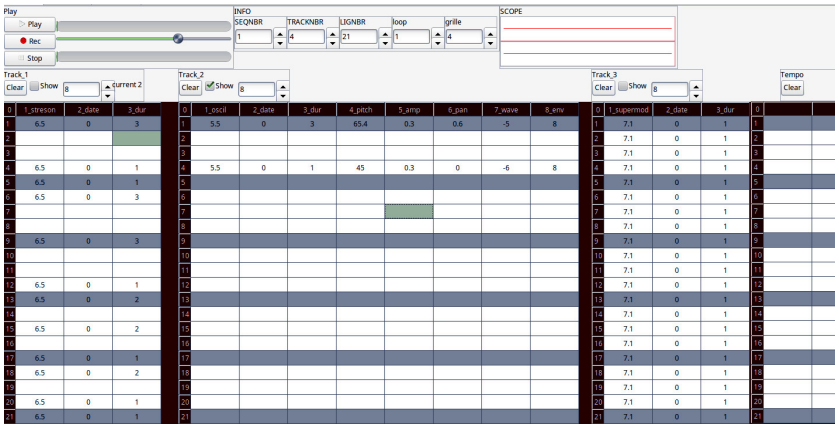


Fig. 1. jo_tracker version 2 (JPG).

Jo_tracker is a tracker interface for Csound. Inspired by Renoise, it's intention is to mix editing ease of tracker softwares with synthesis precision of Csound.

It can be used to generate some precise sound sequences and to write electronic music. Its first version was a Max MSP patcher. Though, the software span quickly required to be thought as an independant software, so it needed to be based on a real programming language. The second version was written with Lua (using IUP for user interface and Terra as a low level programming language for C libraries). This version is far more efficient, really quicker than the first one. Though, the codebase wasn't easy to read, and so, was hard to maintain. In order to distribute a clean version of jo_tracker and MIUP, third version of jo_tracker and second version of MIUP are both fully rewriten in C++, with improvements and some new features. It is a necessary work for further developments.

3.1 Base and principle

First requirement of jo_tracker was to provide a track system, with tracks able to manage an infinite number of parameters. With this feature, one line on a track is equivalent to one csound score "i" statement. This allows to combine or choose between very descriptive scores and algorithmic orchestras. Each track is shown as a spread sheet (See Fig. 1), where each column index corresponds to its Csound equivalent p-field. Eeach track can contain a different number of columns according to the needs of the instrument. Tracks number of lines and columns can vary between two sequences. Though, the number of lines in a sequence is the same for every track. In order to allow users to write some sequences, the tracker implements the notion of sequence : a sequence is equivalent to a time section in Csound. It's an abstract concept that can be thought as a time item containing score data. The main tab also contains one particular track : the tempo track. It can be used to do some tempo interpolation, or to instantaneously jump to another tempo. Obviously, jo_tracker provides some facilities such as copying any sequence's data to another sequence, cleare data, save a project (in a human readable text file), export a CSD file.

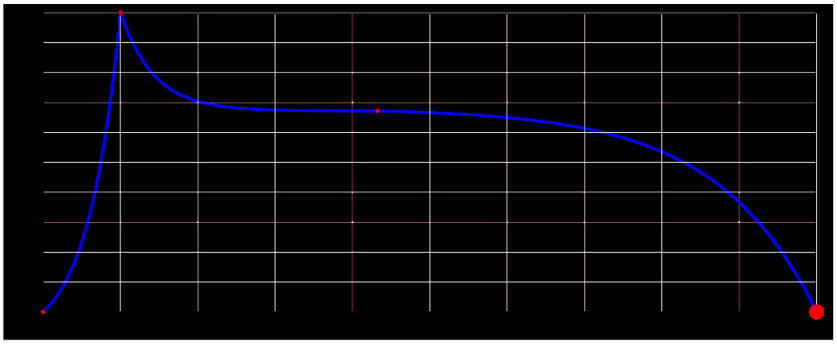


Fig. 2. Curve editor (JPG.

Second tab (See Fig. 2) is dedicated to GEN routines editing. It contains three major elements :
- A curve editor, allowing user to draw curves that will be translated in the GEN16 syntax.
- A waveform plot, mostly used to display waveforms from samples used as GEN01 data.
- Another spread sheet which function is to describe some other GEN routines data (simple arrays in GEN02, synthesis waveforms in GEN10).

There are three modes for starting Csound inside jo_tracker. Each mode generates a csound score, composed with one t statement (tempo track), multiple (many) i statements, and some f statements corresponding to GEN editors data. The main and first mode is activated by clicking the "Play" button. It triggers a call to Csound C API which starts a performance in realtime mode. The second mode (Record button) triggers a realtime recording of the current project. It is mostly useful if the orchestra is used with input channels or any realtime external device. It records the project into a stereo WAV file. The last mode calling Csound API acts as a non-realtime renderer, which also creates a stereo WAV file. It can be activated by clicking the "Render Stereo" item in "Files" menu.

3.2 New features, and upcoming improvements

Jo_tracker's third version brings a set of new features, allowing for a more flexible use. Though, since third version is still under development, some of these features are not ready yet.
-An embedded code editor for Csound orchestras. It allows user to write orchestas directly in the software. It provides syntax highlighting, autocompletion. It also calls the Csound API function `EvalCode` to check whether current instrument uses valid Csound syntax. On success, it registers the instrument in a database.
-Curve editor now supports spline mode (GEN08) and bezier quadratic curves (GENquadbezier)
-A new editor allowing user to manage sequences order and number of loops. As such, it can be considered as a meta editor allowing to manage the general shape of a composition.
-Record and render modes are now available for multichannel audio files
-Tracks are connected to a macro system, based on Csound powerful macro system.
-An audio device list is already implemented and will be added to the parameters menu

As future improvements, both MIUP and jo_tracker could benefit from Eric Wing's work [3] to port IUP on new platforms : MacOSX, Android, iOS, and Web browser. Since a lot of musicians use MacOSX as their composing environment, this target is considered as the major one.

4 The References Section

References

1. IUP Tecgraf Puc site, https://www.tecgraf.puc-rio.br/iup/
2. Csound Github site, http://csound.github.io
3. Eric Wing Github site, https://github.com/ewmailing?tab=repositories
4. cpp11nullptr Github site, https://github.com/cpp11nullptr/lsignal
5. Niels Lohmann Json project Github site, https://github.com/nlohmann/json

Red-Tratos. Visual Art and Sound Art for the Web

Emiliano del Cerro[1]

[1] Universidad Alfonso X el Sabio
ecerresc@gmail.com

Abstract. "RED-TRATOS" is a work made for the web and is hosted by the CVC (Cervantes Virtual Center) belonging to the Cervantes Institute, an institution dependent on the Spanish Government. RED-TRATOS was designed as a mix of visual poetry and as Sound Art. The central part is dedicated to Cervantes and has audio files attached to the visual poem and plays with the name of Cervantes and with phonemes and syllables derived from his name.The work was a pioneer in the field of interactive sound art and visual art and was a key piece in the combination of both worlds for the net (net art).This paper will explain how the project was developed with information on the technology used in the digital signal process as well as the software needed to carry out the work. The main audio application used for audio was Csound, as well VRML, CORTONA, and Softimage for the visual aspects of the work.

Keywords: shynthesis, random distribution, net art, sampling,..

1 Introduction

Red-tratos is a visual and sound work made with visual poems by Eduardo Scala, about authors from the world of universal culture of different nationalities and from different periods of time.

The central part is dedicated to Cervantes, and has several visual poems made by Scala about the author of Don Quixote.

Within this section dedicated to Miguel de Cervantes, there is a hypermedia work, made with poems, and with music composed, recorded and made by the author of this paper.

The work has had the collaboration of Miguel Martin who has made the technical part derived from the completion of the work for placement in the network.

Red-tratos is immersed in the website of the Cervantes CVC Virtual Center, for its vision and listening throughout the world network through the World Wide Web.

The two authors have extensive experience in visual arts and sound art and their works have been presented at international institutions around the world.

2 Music, Sound, Poetry

This central part dedicated to Cervantes has a sound attached to the visual poem and plays with the name of Cervantes and with phonemes and syllables from its name. The piece is divided in three parts.

The musical part operates as an hypermedia exposition and has 3 parts in which visual poetry and sound art are mixed:

- Cube. Cervantesmirror
- Cervantesphera
- Finicio

2.1 "Cube. Cervantesmirror"

This section called "Cervantesmirror", "Cube", consists of a new verbal game where the signifier provides several multidirectional variations of the name of Cervantes, located on the six faces of a cube in motion.

This movement plays with aspects in 3D, and has interactive possibilities: the music includes new elements that change with the movement of the mouse.

2.2 "Cervantesfera"

The central part is the fundamental sound part in this work. This section has three parts:

- Black Tone on White,
- White Time on Black, and
- Gray Folia

2.2.1. Black tone on white

The first part of "Cervantesfera" is linear and the music follows the visual exposition of the poem.

Figure 1 is an example of this first movement of Cervantes sphere.

Fig. 1. Examples of the section Cervantes sphere.

2.2.2. White time on black,

It constitutes the second part of "Cervanstesfera"and introduces visual aspects in 3D, and the cube and sphere form appear.

Figure 2 is an example of the visual part of Cervantesfera's second movement.

Fig. 2. Example of the section Cervantes sphere

2.2.3. "Gray Folia",

The third part of the movements of this musical work, "Gray Folía" is an invitation to walk through the "RED-TRATO" and allows an internment in the most hidden angles of the fascinating typography that composes the picture.

It is an interactive game that has visual aspects in 3D and musical aspects that follow the movement of the viewer depending on the movement of the cursor on the computer screen.

In this third part, the name of Cervantes is written with fonts of the same historical period of the author of Don Quixote: Paull Renner. The sound files change according to the movement of the text and the place where the viewer places the computer mouse

The computer has several sensors on the computer screen allowing the user, to move between syllables, phrases, and even enter inside the fonts with which the name of Cervantes is written.

Figure 3 is an example of Gray Folia.

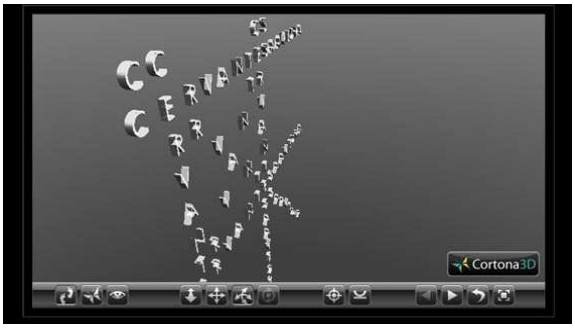


Fig. 3 Example of section Cervantes sphere.

2.3 "Finicio"."A – Z"

The sound part ends with a section called Finicio. A-Z

It is a metaphor for the beginning and the ending, that allows a kind of timeless meditation.

Figure 4 is an instantaneous snapshot of this section..

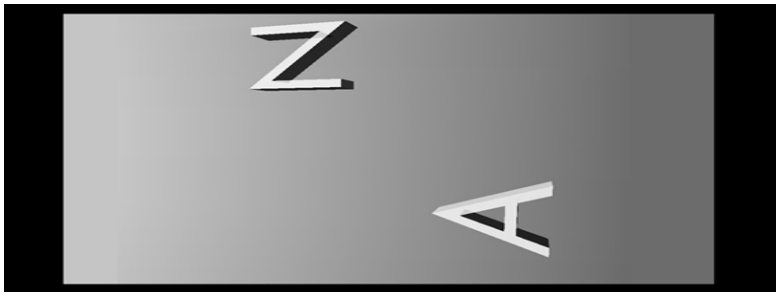


Fig.4 Example of the section FINIZIO

3 Realization

The technical part of this work can be addressed as visual art, sound art, and Interactivity.

3.1 Visual Art

The visual part has been made with Photoshop, Maya, 3d MAX, Softimage Autodesk.

This type of software has allowed the visual process of static image and it gives the possibility of animation and movement in different sections of this work.

3.2 Sound Art

The audio part of this web page, has had several phases that have consisted in Recording, Edition, Sound synthesis.
pure sine waves, filtered white noise,Synthesized guitar and recorded files are processed using DSP and subsequently mixed and compressed for distribution on the network.

3.3 Interactivity section. Gray Folia.

The last part of Cervantesfera, Gray Folia, has been made with a special program for Virtual Reality, called VR Cortona.

VR Cortona has a free viewer, which must be loaded on the computer, to be able to make use of all the possibilities that this Folia offers Virtual Reality with Screen Sensors.

4 Csound

RED-TRATOS uses Csound as the only tool for the synthesis and DSP of the audio part in this piece.
The piece is based on two techiniques Recording and fragmentation, and synthesis.

4.1 Recording and Reproduction of Audio Files

Red-tratos uses a fragmentation of sound and image. This process is generated for a sentence of cutting phrases and words into phonemes and syllables.

The voice sound came from a recording of the voice of Eduardo Scala reading the poems by Miguel de Cervantes.

These audio sources are processed as sampling sources, by cutting and distributing the samples into the piece.

```
a1, a2 bbcuts  asource1, asource2, ibps, isubdiv,
         ibarlength, iphrasebars, \
     [, istutterspeed] [, istutterchance] [, ienvchoice]
```

After the cutting and fragmentation of sonorities, the sonority is produced by means of stochastic probabilities, (Gauss, …)

```
a1 randh  xamp, xcps [, iseed] [, iuse31]
a1 gauss 1
```

the sound recorded is distributed with somne techniques of cut and shuffle with forms to play forward backward

```
a1 Soundin "speech1.aif", 0
a1 Loscil
```

the reading process came from the reading from a table

```
a1 tableshuffle ktablenum
a2 tableshufflei itablenum

giBuffer ftgen  0, 0, 2^17, 7, 0; table for audio data
storage
```

4.2 Shynthesis of Sound Files

The synthesis process try to imitate a possible sonority derived from the time in which Cervantes lived.

The basic material is derived from recorded voice, sinusoidal sound, guitar like synthesis, and percusive sound from white noise source.

The main ideas came from pure sinusoidal sound with some modulation.

The guitar that remembers the vihuela sonority from the renaissance period in the Spanish tradition of instrumental music. The synthesis came from a mix of Karplus Stong Algoritm and Waveguide shinthesis.

Karplus strong

```
    ares pluck kamp, kcps, icps, ifn, imeth [, iparm1] [,
iparm2]
```

The percussion sonority produced with noise source with some filtering for change the timbre properties of the sound, to imitate different instuments.

4.3 Interactivity

There are also a part of the piece that have an important interactivity property.

The user can move the mouse and listen and see different parts of audio file.

The image is designed as a 3D picture and the user can move around and inside the image in order to have the illusion of a travel inside the visual poem.

In the last section, the interactivity produced with cortona is associated with csound. The user can control, with the mouse, the space (stereo) distribution of the samples and the production of the sound into different sound planes. To have a similarity with filmic planes (plane, general, …)

```
    a1, a2  space asig, ifn, ktime, kreverbsend, kx, ky
    ktime
    line  0, 5, 5
    a1, a2  space asig, 1, ktime, ...
```

5 Conclusion

The work presented in this article involved an effort of collaboration between a plastic artist, father of the original idea, together with a composer and a technical team for the realization and representation of images and sound in a hypermedia and interactive page.

Its place of presentation, and of vision and listening is a primordial and a preferential form, in a browser for the web. This work has been hosted in the CVC network.
This fact, period of time, together with a high number of visitors, gives validity and relevance to these RED-TRATOS.

The work could be seen and represented in a real space, such as a space belonging to a museum, art gallery, or physical space where a technologically complex installation such as RED-TRATOS is allowed.

A problem related to this type of presentation, is associated with the inexorable passage of time and possible obsolescence of technology, so it is necessary to update software and hardware everytime this possibility becomes a reality.

As in other disciplines, the passage of time has given a validity to this presentation that has gained over the years. In the same way, this passing of time offers an extra patina to the work.

References

Boulanger, Richard, ed.. The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing, and Programming. MIT Press. (2000)
Dodge, C., Jerse, C.: Computer Music: Synthesis, Composition and Performance, 2nd edn. Schirmer, New York (1997)
Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
Lorrain, D.: A panoply of stochastic 'cannons'. Computer Music Journal 4(1), 53–81 (1980)
Moore, F. Richard Elements of Computer Music Prentice Hall. Englewood Cliffs 1990
Risset, J. C "An Introductory Catalogue of Computer Synthesized Sound . MIT pPess 1971

http://csound.github.io
http://cvc.cervantes.es/actcult/redtratos/webflash/papeles.pdf

Implementing Arcade by Günter Steinke in Csound

Daria Cheikh-Sarraf, Marijana Janevska, Shadi Kassaee, and Philipp Henke *

[1] Incontri - Institut for contemporary music at the HMTM Hannover
[2] FMSBW
incontri@hmtm-hannover.de

Abstract. This paper is about the process of implementing the live-electronics of the solo cello and electronics piece "Arcade" by Günter Steinke. We will discuss problems that occurred during the process of implementation and how we approached the transfer of the electronic procedures that were originally on big hardware machines to the Csound programming environment. The main focus of this paper also discusses the possibilities of the Csound FrontEnd *CsoundQt*, that we mainly used for the performance with its GUI capabilities.

Keywords: CsoundQt, Live-electronic, Instrument, Hannover, Incontri, FMSBW, Günter Steinke, Arcade

1 Introduction

Back in the time of 1992, the German composer Günter Steinke begun work on a piece which was to become his cello and live-electronics piece. He couldn't have known that much of the equipment he was using in the Freiburger Experimentalstudio would soon become obsolete. As time passed, the computer became increasingly accessible, convenient, and powerful. A piece which would have required truckloads of equipment before could now be realized in a small machine, namely the *notebook*. Powerful programming languages like Csound made it possible to realize the needs of a piece like Arcade and to create a sophisticated version which is purely software-based. In this paper, we will describe the process it took to realize a complex piece like Arcade in Csound and how we dealt with other implementations of the piece in other music-programming languages like Pure Data and Max/MSP.

2 Speakers and Microfon

In Günter Steinke's Arcade, the cello, played live, should be amplified in addition to the electronics. To achieve and maintain a good balance in the overall volume, we used two microphones for the performance at the Sprengel Museum

* Without the help of Joachim Heintz this would not have been possible.

in Hannover. Firstly, the Shure microphone: the advantages of this microphone is its consistent cardioid characteristic, which should reduce feedback, and an optimum transmission range for drums, percussion and instruments, which is good for the pizzicati in the cello part. However, due to the weight of the coil, the Shure microphone sometimes sounds sluggish, especially the high frequency response, which may sound a bit covered. And secondly, the DPA microphone: the DPA microphone is well suited for instruments. It is clear, clean, and has a high-resolution. In this piece, the cello has a wide dynamic range. For example, there are very quiet passages (see min. 09:00) of sul ponticello or pianississimo, but also loud, dominant pizzicato parts (see min. 05:02), which sound very percussive and present a strong contrast. It was often necessary to emphasize the above-mentioned passages manually by amplifying the input of the cello, i.e. how much came into the microphone or the two microphones. For the pizzicato parts, we had to amplify especially the Shure microphone precisely for the percussive parts. However, this could have been automated in order to avoid supposedly minor errors.

3 Electronics and Problems of Implementation

Steinke's Arcade uses a wide array of different electronic procedures. He uses different modules like, pitch-shifting, Halafon, delay-lines, noch weitere hinzufügen!!! Since 1992, Arcade has been translated for the computer. The first computer realization was made with the programming language Max/MSP in 2000. Another significant implementation has been made for the Pure Data programming environment by Orm Finnendahl, which was realistically, a translation of the Max/MSP implementation. Analyzing all the implementations from the past, there is no denial that each of the implementations had to deal with problems of translation from hardware to software. One of the first tasks was to analyze the possibilities of the Csound programming language, in order to avoid creating a PD version in Csound, but rather a native Csound implementation using the power of the Csound programming language.

3.1 Analysis

One of the first pitfalls to avoid when confronting an implementation is to resist direct translation e.g. between PD-Objects and their Csound equivalents. Oftentimes, one finds a similar opcode of the same name in Csound. However, it should be noted that one has to first look at the specific functionalities of the opcode, like the quality of the filter, and the order of the bandpass filter used. While analyzing the Max/MSP and PD versions, one notices that both patches do not actually use a filterbank like in the original realisation of *Arcade*, they used stations of spectral masks done with fft, to create a similar sound to the original filterbaks. However, this would contradict our approach and goal of a Csound native implementation which is true to the orginal realisation, proposed in Steinke's score. Consequently, we sought solutions dealing with the wide array

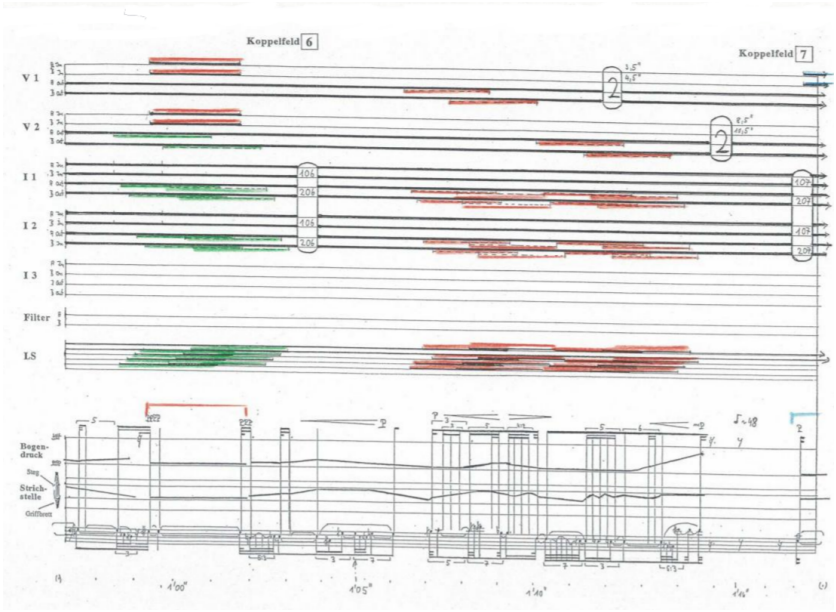


Fig. 1. Example of the notation in Steinke's *Arcade*

of filters that the Csound programming environment has to offer. Visually, the biggest difference one finds when working on an implementation is that Steinke used an analog matrix (*Koppelfeld* during the premiere of the piece. Because the matrix is so essential to the functionality of the piece, Max/MSP and PD[1] come with their respective matrix applications, whereas in Csound, a text-based programming environment, one has to build the matrix to work while also using the GUI possibilities of CsoundQt to make it more useable in the performance situation.

Example of the Matrix in Csound

```
/*** MATRIX SETTINGS ***/

instr Mtx_1

 puts "Mtx_1", 1
 chnset 1, "show_mtx"

 ga_Harm_in = ga_Del_out
 ga_Chn1_in = ga_Harm1A_out
 ga_Chn2_in = ga_Harm1B_out
 ga_Chn3_in = ga_Harm2A_out
 ga_Chn4_in = ga_Harm2B_out
```

```
 ga_Chn5_in = 0
 ga_Chn6_in = 0
 ga_Filt_in = 0
 ga_Rev_in = 0
 ga_HalaA_in = 0
 ga_HalaB_in = 0
 ga_HalaC_in = 0
 TurOffOtherMtxs gS_Mtxs, "Mtx_1"

endin

instr Mtx_2

 puts "Mtx_2", 1
 chnset 2, "show_mtx"

 ga_Harm_in = ga_Del_out
 ga_Chn1_in = ga_Harm1A_out
 ga_Chn2_in = 0
 ga_Chn3_in = 0
 ga_Chn4_in = ga_Harm2B_out
 ga_Chn5_in = ga_Harm2A_out
 ga_Chn6_in = ga_Harm1B_out
 ga_Filt_in = 0
 ga_Rev_in = 0
 ga_HalaA_in = 0
 ga_HalaB_in = 0
 ga_HalaC_in = 0
 TurOffOtherMtxs gS_Mtxs, "Mtx_2"

endin
```

Example from the Csound implementation of *Arcade* by Günter Steinke. In creating a hybrid gui application handling the matrix fucntion for us, we found a convinient way to solve the problems concerning the realisation of Steinke's analog *Koppelfeld*.

3.2 Filters

An important aspect of our implementation is that we did not use fft to recreate a sound emulation sounds of the premiere, but instead implemented Csound native filters to patch the piece. It has been the first realization since the premiere that uses true filter processing instead of spectral masks done in former implementations. In the process of programming the filters, we made a long process testing out the different filter opcodes in Csound. The filters are a crucial part of

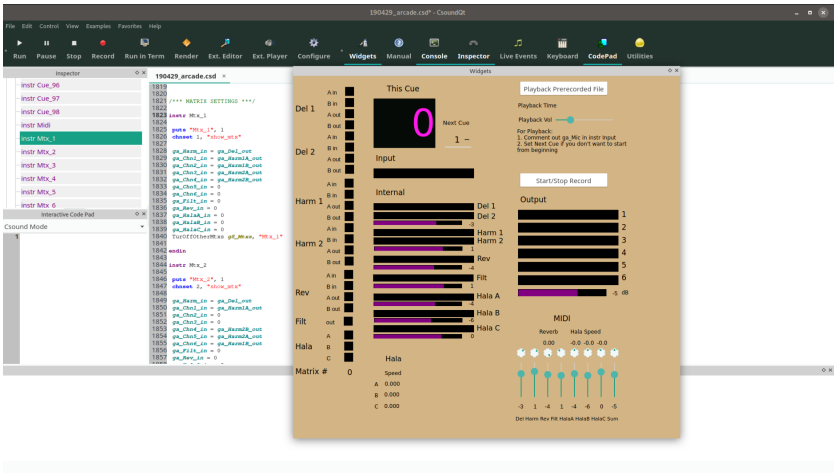


Fig. 2. Example of the Widget view of our implementation (PNG).

the piece, in particular the way the piece sounds, for that we had to understand after what sound the composer is after. We decided to choose the *mode* filter opcode, because it could produce a very transparent and resonant sound combined with the cello. However there was an argument concerning the stability of the opcode and the advantages of using the *reson* filter over the *mode* filter.

Implementations of the original filter modules

```
/*** FILTER ***/

instr Filt_Seq_1

 kndx init 0
 kTime init 0
 kFiltSeq[] = gk_Filt_Seq_1
 iFirstProg = 1
 if kTime <= 0 then
  event "i", "ReadFiltProg", 0, 0, iFirstProg+kndx
  kTime = kFiltSeq[kndx]
  kndx += 1
  if kndx == lenarray(kFiltSeq) then
   printks "  Filt_Seq_1 turned off\n", 0
   turnoff
  endif
 endif
 kTime -= 1/kr
endin
```

```
instr Filt_A

 iBand = p4
 S_chnl sprintf "Filt_A_%d", iBand

 //midi pitch one tone below the first band
 iBasPch = 34
 iQ = 1

 iFreq mtof iBasPch + iBand*2
 kDb chnget S_chnl
 kDb port kDb, gi_Filt_FadeTim
 aFilt mode ga_Filt_in*ampdb(kDb), iFreq, iQ

 chnmix aFilt, "filt_A_collect"

endin
```

Example from the Csound implementation of *Arcade* by Günter Steinke.

4 Performenace Situation

In the case of Steinke's Arcade, not only the electronics and amplification played an important role. In the original score, the "programs" indicate which effects are triggered and which cello parts are recorded and edited with filters, delays etc. In our Csound version, the so-called "cues" always have sections that have been recorded through the microphone, which can be activated and stopped, effects being played on them and previously recorded patterns repeated. The difficulty was to activate the cues at the right moment. In certain places, for example where a delayline of the cello should be played back through the speakers which then would occur at the same time with the live cello, one has to be as precise as possible. No sounds or noises appearing too early or too late should be allowed into the triggered cue, since they could partly pull through the delays and through the whole piece, which would be a major disruptive factor. It is even more important not to leave everything to technology and to operate the cues and mixers by ourselves, as any performance of the live cello could vary in speed. It is a great help at particularly critical points in the play to agree with the player on assignments, so as to adapt the cues to the live cello as precisely as possible. Since this piece, and this is what makes it so special, even one wrongly timed cue can be heard as an error in the process of the piece. That is why one has to be precise with the triggering of the programs/cues.

4.1 Summary

Csound together with it's frontend CsoundQt provide sophisticated means to implement complex sounds and structurs into a simple and easy to use per-

formance enviroment. As a text-based programming enviroment, csound is also easy on the CPU and can handle difficult calculation tasks, like multiple filter layers and harmonizer layers as well as complex spatialisation. However where Csound shines the most is it's tonal flexibilities and wide array of opcodes that help to shape the sound in many different ways. The csound frontend *CsoundQt* proved to be very useful in the performance situation, concerning the capability to use the widget to control the parameters of the electronic in realtime in a convenient way.

References

1. Heintz, J. et McCurdy, I.: Csound Floss Manual. Creative Commons Attribution 2.5 (2015)
2. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
3. Steinke, G.: Arcade für Solo Cello und Live-Elektronik. Boosey and Hawkes (1992)
4. Csound Github site, `http://csound.github.io`

Improving Csound's Ambisonics decoders

Pablo Zinemanas[1], Martín Rocamora[1] and Luis Jure[2] *

[1] Facultad de Ingeniería
[2] Escuela Universitaria de Música
Universidad de la República
lj@eumus.edu.uy

Abstract. This paper describes the efforts we devoted to improve Ambisonics decoders in Csound. Current version of the existing opcode, namely `bformdec1`, has some limitations that should be surpassed in order that the decoders better fulfill the Ambisonics criteria. In particular, the implemented decoders have no near–field compensation and do not use a different decoding matrix for low and high frequencies. These issues are addresses in a new implementation of the opcode, namely `bformdec2`, that also adds some features, such as additional loudspeaker array configurations (rectangle, hexagon) and a binaural output for headphones.

Keywords: Ambisonics decoder, HOA, Csound

1 Introduction

Ambisonics is a spatial sound metatheory (a theory of theories) for audio recording, coding and reproduction, developed by Michael Gerzon in the 1970s [4]. It provides a method of codifying physical properties of a sound field (the pressure and velocity components), that captures directional information of the sound sources, and enables its accurate reconstruction in a point in space.

Unlike traditional multichannel audio for spatialization in which each channel corresponds to a given loudspeaker, the Ambisonics sound format—called B-format—contains a speaker-independent representation of a sound field. By means of an appropriate decoder, i.e. matched to the geometry of the loudspeaker array, this sound file can be played back in different speaker layouts [2].

The channels of the B-format file can be regarded, from a theoretical perspective, as the coefficients of a series expansion of the sound field around the origin, in terms of spherical harmonics [2,9]. The spherical harmonics are a complete set of orthogonal functions on the sphere, and thus can be used to represent functions defined on the surface of a sphere.

The order of the series expansion determines the number of channels involved. Thus, zeroth order Ambisonics represents the omni–directional component of the sound field and corresponds to the the sound pressure, which consist of one single

* This research was partially funded by Comisión Sectorial de Investigación Científica, Universidad de la República, Uruguay. We thank Aaron Heller for his advise.

channel (the W channel). First order Ambisonics adds three directional components (channels X, Y and Z), corresponding to the pressure gradient and representing the acoustic velocity. Higher order Ambisonics (HOA) add additional coefficients to the series expansion, corresponding to higher order derivatives of the sound field [9]. Increasing the order of the series expansion provides better approximation of the sound field, which leads to increased spatial resolution.

2 Ambisonics decoding

The decoder has to provide suitable linear combinations of the B-format signals for each loudspeaker in the array, so that the pressure and particle velocity is reproduced correctly at the listening position, i.e. the centre of the array. The set of coefficients needed to produce that linear combination is called the decoding matrix. The number of loudspeakers must be at least the number of the B-format signals [8]. There are essentially two different approaches that can be adopted: the basic (or physical)[3] decoding and the energy (or psychoacoustic) decoding [9]. It turns out that the basic decoding achieves accurate perception of spatial localization only at low frequencies, where as the energy decoding provides optimal localization only for high frequencies. For this reason, a better approach consist in using different solutions for low and high frequencies.

2.1 Physical decoding

The basic decoding seeks the reconstruction of the sound field, up to a given Ambisonics order, from the superposition of the sound waves emitted by the loudspeakers, assuming phase coherence among the signals [9]. In essence, the solution of the decoding equations corresponds to the projection of the spherical harmonics to each of the directions of the speakers. In most cases the number of speakers is greater than the number of Ambsionics channels, which yields an under-determined system of equations whose solution can be obtained with algebraic methods (pseudo-inverse). For regular speaker arrays the problem is well–conditioned and the method will result in a correct solution. For irregular arrays the solution could be still obtained with algebraic methods but the problem is often ill–conditioned, making the obtained solution inappropriate.[4] The basic decoding succeeds at reproducing the impression of sound source locations only at low frequencies (approximately below 500 Hz), and close to the center of the loudspeaker array [9,2]. For higher frequencies or a large listening area it is better to use a psychoacoustic decoder.

2.2 Psychoacoustic decoding

The psychoacoustic decoding aims at reproducing the original energy and acoustic intensity of the sound field, assuming an incoherent sum of the speakers

[3] Other names are used to refer to this decoding solution, such as exact or velocity.

[4] Although there are several proposals to deal with irregular arrays, this still remains as an area of open research [9,5].

signals [9,2]. By incoherently summing the signals of several loudspeakers it is physically impossible to exactly reconstruct the acoustic intensity, so the decoder will instead try to maximize a statistical estimator of the signal energy. In the case of regular (or semi-regular) speaker arrays it is possible to obtain the energy decoder by altering the coefficients of the basic decoder matrix.

The *in-phase* decoders additionally impose the restriction that no loudspeaker emits in opposite phase [9]. This provides more robust localization for listeners who are far from the center of the loudspeakers array.

2.3 Dual–band decoding

Given that no decoding approach is adequate for both high and low frequencies, many Ambisonics decoders split the B-format signals into (at least) two bands and use independent solutions for low and high frequencies [2]. Then the output of each band is recombined to produce the audio signals for the loudspeakers. It is important to note that the band–splitting filters must be carefully designed to preserve the magnitude and phase response of the signal [8]. Besides, when combining the output of each band, different criteria can be used to deal with the signal's level difference between the low and high frequencies, such as preserving the amplitude, the root–mean–square RMS level or the total energy [8].

2.4 Near–field compensation

Another important aspect of an Ambisonics decoder is to provide near–field compensation [3]. The recreation of the sound field at the central position holds under the hypothesis that the wavefronts are planar. Given the finite distance to the loudspeakers, the sound wavefronts at the listening position present instead a curvature, which produces a bass–boosting effect that has to be compensated. The compensation is essentially a high–pass filter, which depends on the order of reproduction and on the distance of the loudspeakers to the center of the array.

2.5 Criteria for correct Ambisonics decoding

In summary, as suggested in [8], apart from having a decoding matrix matched to the geometry of the loudspeaker array, we focus on the following key aspects for correct Ambisonics decoding:

- dual–band decoding (high and low frequencies) using phase–matched filters
- near–field compensation, implemented as a high–pass filter.

These features are not provided in the Ambisonics decoders currently available in Csound, so they are addressed in a new opcode implementation.

3 Current Ambisonics decoders implementation

Ambisonics decoders in Csound are implemented in the `bformdec1` opcode.[5] There are five loudspeaker layouts available: stereo, quad (2D square), 5.0, octagon and cube. Given the constrain on the number of loudspeakers for a given Ambisonics order,[6] all decoders are first–order, except for the octagon layout, which provides first–, second– and third–order decoders, and the 5.0 which has first and second order. It is important to note that the decoders are of the in–phase type—the same decoding matrix used in low and high frequencies—and without near field compensation.

4 New Ambisonics decoders implementation

The new implementation of the Ambisonics decoders, namely `bformdec2`, focus on providing dual–band decoding and near–field compensation. It is designed with backward compatibility in mind, so it offers the same loudspeaker layouts available in `bformdec1`. Besides, some additional loudspeaker array configurations are provided, including a binaural output for headphones.

4.1 Dual–band decoding

The decoders implemented are dual–band, providing a different decoding matrix for low and high frequencies. The band splitting filters are designed to be phase–matched, as described in [8] and explained below.

Phase–matched dual–band splitting filters The dual–band splitting is obtained by combining two second order–filters, a low–pass and a high–pass filter, that are phased matched. The phase match is achieved by reversing the phase response of the high-pass filter in order to match that of the low-pass filter. The two filters acting together give a first–order all–pass filter [8,6].

The filters are implemented as infinite–impulse response (IIR) filters, as,

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$$

Coefficients a_i are calculated by:

$$a_1 = \frac{2(k^2 - 1)}{k^2 + 2k + 1}, \quad a_2 = \frac{k^2 - 2k + 1}{k^2 + 2k + 1}$$

for both filters, whereas b_i coefficients are:

$$b_0 = \frac{k^2}{k^2 + 2k + 1}, \quad b_1 = 2b_0, \quad b_2 = b_0,$$

[5] There is a deprecated opcode, namely `bformdec`, which was tested in [8].

[6] For an Ambsiconic order l, $(l+1)^2$ loudspeakers are needed for full–sphere systems, and $2l + 1$ for horizontal–only reproduction.

for the low-pass filter and

$$b_0 = \frac{1}{k^2 + 2k + 1}, \quad b_1 = -2b_0, \quad b_2 = b_0,$$

for the high-pass filter; and $k = \tan\left(\pi \frac{f_c}{fs}\right)$, where f_c is the splitting frequency and fs is the sampling rate [8].

The filters of the `bformdec2` opcode are implemented by the Direct-Form II, using a similar code to that of the `filter2` Csound's opcode algorithm. The splitting frequency can be selected by the user, and its default value is 400 Hz.

Low– and high–frequency balance Starting from the basic decoding matrix, which is used for the low frequencies, the decoding matrix for the high frequencies is obtained by applying a set of coefficients, as explained in [7]. Different criteria can be used to balance the gain between low and high frequencies. By default, `bformdec` uses conservation of total energy, but via an optional parameter, any of two additional methods can be selected: preservation of the amplitude, and preservation of the root–mean–square (RMS) level. The coefficients are computed using the Ambisonics Decoder Toolbox[7] (ADT) [7,5].

4.2 Near–field compensation

The near–field compensation is achieved through the high–pass filters proposed by [3], following the implementation described in [1] and the code provided by ADT [5,6]. The order of the compensation filters corresponds to the Ambisonics order. The current implementation of `bformdec2` allows for near–field compensation of decoders up to order five. The user can set the distance to the speakers as an input parameter, and can also disable the near–field compensation.

4.3 Loudspeaker layouts and binaural output

The decoders implemented in `bformdec1` are hard–coded, that is, each decoder's data is fully embedded into the source code. In contrast, the implementation of `bformdec2` was designed to be modular, so that, from a decoding matrix for the basic solution and some input parameters, a set of functions compute the decoding solution. This offers some flexibility for the addition of loudspeaker layouts, even from a decoding matrix supplied by the user.

Current implementation of `bformdec2` provides the same layouts available in `bformdec1`, for backward compatibility. The additional loudspeaker arrays implemented so far are horizontal–only, namely hexagon and rectangle (for different length–to–width ratios). A binaural output for headphones is also provided.

The binaural output, up to third–order Ambisonics, is obtained through a two–step process. First, the input signal is decoded to a virtual loudspeakers

[7] `https://bitbucket.org/ambidecodertoolbox/adt/`

array (octagon for horizontal–only, and dodecahedron for full–sphere). Then, we compute the convolution of the signal of the loudspeakers and head–related transfer functions (HRTFs) corresponding to the direction of the virtual loudspeakers. The sum of the obtained signals yields the binaural output for headphones. The implementation of the HRTF convolution is based on the code of the `hrtfstat` opcode, and the set of HRTFs used is already available in Csound.

5 Discussion and conclusions

The implementation of a new opcode for Ambisonics decoding is released[8] that aims to remedy the lack of dual–band decoding and near–field compensation of the previous implementation. The opcode offers some backward compatibility and the option to try out the best decoder for one's particular needs, through parameters set by the user (e.g. band splitting frequency, disable near–field compensation). Ultimately, the most appropriate decoder may depend on the sound source material and the intended use, for instance, the size of the loudspeaker array and the listening area. In the future, the opcode will include more loudspeaker layouts and the option to use a decoding matrix specified by the user.

References

1. Fons Adriaensen. Near Field filters for Higher Order Ambisonics. online, accessed 29-Oct-2018, `https://kokkinizita.linuxaudio.org/papers/hoafilt.pdf`.
2. Jérôme Daniel. *Représentation de champs acoustiques, application à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimédia*. PhD thesis, Université Paris 6, 2001.
3. Jérôme Daniel. Spatial sound encoding including near field effect: Introducing distance coding filters and a viable, new Ambisonic format. In *Proceedings of the Audio Engineering Society (AES) 23rd International Conference: Signal Processing in Audio Recording and Reproduction*, May. 2003.
4. Michael A. Gerzon. General Metatheory of Auditory Localisation. In *Proceedings of the Audio Engineering Society (AES) 92th Convention*, Mar. 1992.
5. Aaron Heller and Eric Benjamin. The Ambisonic Decoder Toolbox: Extensions for partial-coverage loudspeaker arrays. In *Linux Audio Conference (LAC)*, May. 2014.
6. Aaron Heller and Eric Benjamin. Design and implementation of filters for Ambisonic decoders. In *Proceedings of the 1st International Faust Conference (IFC)*, Jul. 2018.
7. Aaron Heller, Eric Benjamin, and Richard Lee. A toolkit for the design of Ambisonic decoders. In *Linux Audio Conference (LAC)*, Apr. 2012.
8. Aaron Heller, Richard Lee, and Eric Benjamin. Is my decoder Ambisonic? In *Proceedings of the Audio Engineering Society (AES) 125th Convention*, Oct. 2008.
9. Davide Scaini and Daniel Arteaga. Decoding of higher order ambisonics to irregular periphonic loudspeaker arrays. In *Proceedings of the Audio Engineering Society (AES)55th International Conference: Spatial Audio*, Aug. 2014.

[8] Available at `https://github.com/pzinemanas/bformdec2`.

Preliminary study for a *chorus* opcode

Daniele Cucchi and Stefano Cucchi

I.T.B. Project Studio
d_cucchi_1976@yahoo.it
s.cucchi@itbprojectstudio.com

Abstract. In this paper we submit the hypothesis of a new "chorus"opcode in *Csound*. We wrote a simple program in Octave language witch generate random values read by Csound in a further step. These values are used to modify the original playback speed of a audio file. Good choice is to use white sequence of uniform distribuited samples filtered by 1-pole system to obtain low-pass behaviour. Some considerations about amplitude distribution of results and proposals to manage it will be done.

Keywords: Chorus, New opcode, Asynchronous playback, Variable delay, Random speed, Random, Noise.

1 Introduction

One of the main features of computer music is the "perfection"of synthetized sound and music in terms of intonation, timing, waveforms, etc... Avoiding any kind of aesthetic judgement, there are many cases in which we want add some imperfection to the sound in order to make it more "real"and, maybe, pleasant to the ears. There are many ways to obtain the same results, tipically using through noise. The idea of this paper is to analyze some aspects of the noise "opcodes" and of introduce an alternative form of it.

2 The "noise"opcode in *Csound*

In *Csound* there is an effective "noise"opcode with a IIR lowpass filter.

$$Y_n = \sqrt{(1-\beta^2)} * X_n + \beta Y_{(n-1)} \tag{1}$$

The β (*kbeta* value in the score) determines the filter's cutoff frequency. According to the differents values of *kbeta* the behavior of the noise can vary from oscillation around a value to a kind of "drift"similar to the loss of intonation typical of analogic instruments. The *a-rate* noise can be used in association with the couple *phasor - tablei* in order to have subtle changes in speed, or downsampled to a *k-rate* signal to achieve little fluctuation of volume or intonation.

Asimptoticaly, the values of Y are distribuited like a gaussian, so it means that not all possible values are extracted with the same probability. Moreover the maximum values reached depends by the length of the sequence: the longer

the sequence then more possibility you have to extract higher values. This is not a real problem with the most part of applications but sometimes could be useful to have a major control of amplitude distrubution.

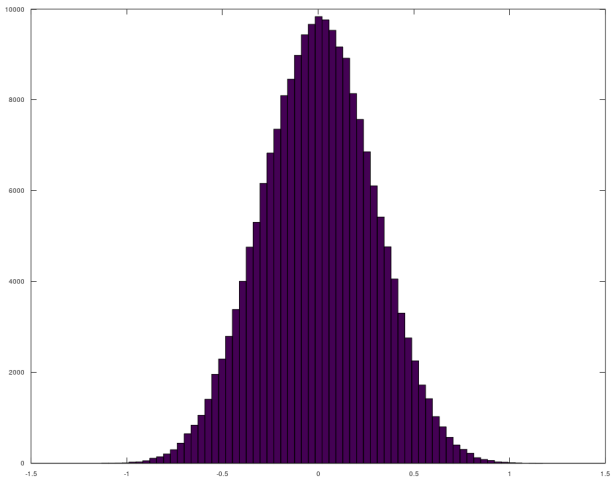


Fig. 1. original opcode

The figure 1 is the histogram distribution obtained by invoking noise opcode with parameter β of value 0.9.

3 The first modified noise generator

We define T (threshold) as the maximum admitted value for the sequence. So we modify the original equation as below:

$$W = \sqrt{(1-\beta^2)} * X_n + \beta Y_{(n-1)} \tag{2}$$

$$Y_n = min(|W|, T) * sign(W) \tag{3}$$

With this form of recursive formula we have the assurance that no value has module greater than T.

The figure 2 is the o obtained histogram distribution with T=0.7. It's clear that this simple algorithm is not the optimal one cause it create artificial and unwanted excess of extracted samples with value T.

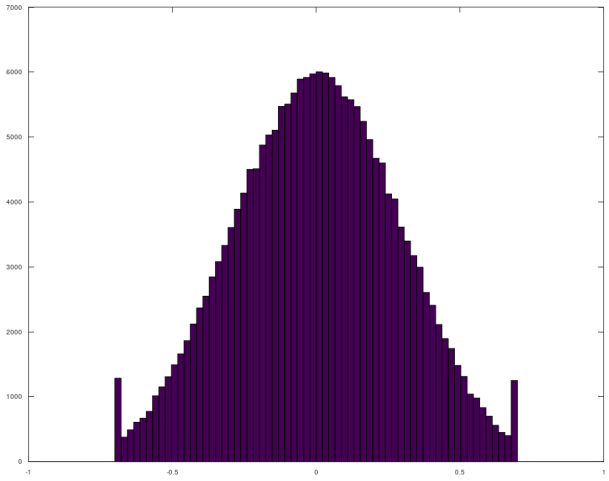


Fig. 2. 0.7 threshold

4 The second modified noise generator

The kernel of evolution is the well kown equation.

$$W = \sqrt{(1-\beta^2)} * X_n + \beta Y_{(n-1)} \tag{4}$$

If abs(W) is smaller than T nothing happens, otherwise some bounce back from threshold is implemented. The Octave code describe one of possible implementation of this bounce.

As expected there are no peaks of distribution around T visible in figure 3.

4.1 Octave Code

Here Octave code used to generate random sequences

Octave code

```
clear
close all

rand("seed",32);

standard_T = 0;
wrapped_T  = 1;
T = 0.7;
```

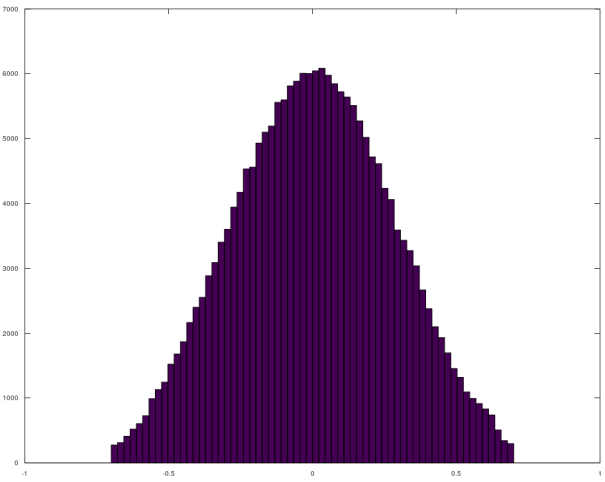


Fig. 3. 0.7 advanced threshold

```
nomefile="random_y";
L = 200000;
beta = 0.9;

x = rand(1,L);
x = x-0.5;

base = 0;

for k =1:L
  passo = x(k)*sqrt(1-beta*beta);
  old_base = base;
  base = base*beta+passo;

     if wrapped_T == 1
        if base > T
           passo = passo - (T-old_base);
           base = T - passo;
        end;
        if base < -T
           passo = passo - (-T-old_base);
           base = -T - passo;
        end;
     end;
```

```
     if standard_T == 1
        if base > T
           base = T;
        end;
        if base < -T
           base = -T;
        end;
     end;


  y(k) = base;
endfor;
```

4.2 *Csound* code

Here the *Csound* instrument used to test some sequences. This code use the input sequence to vary the playback speed of an audio file, but it's only one of the possible utilizations.

Csound code

```
<CsOptions>
</CsOptions>
<CsInstruments>
sr =  44100
kr =  4410
ksmps   =  0
nchnls =  2
0dbfs = 1
strset 1, "sinusoide.aif"
instr 1
iformat1 = 7
iprd1 = 0.08
kveldev1 readk "random_y", iformat1, iprd1
Sfile strget p4
aoriginal diskin2 Sfile, 1
achorus1 diskin2 Sfile, 1 + (kveldev1*p5)
outch 1, aoriginal
outch 2, achorus1
endin
</CsInstruments>
<CsScore>
i1 0 10 1 0.99
e
</CsScore>
```

```
</CsoundSynthesizer>
```

5 Conclusions

We described two variation of original noise opcode which give us an adjoint parameter controlling the dynamic of the system and can be useful when it's important to limit the maximum module of the sequence. Probably to really arrive to define a new opcode would be necessary a more depth study about control of amplitude distribution. It should be interesting investigate also the combination of different effects varying β and T parameters.

References

1. ffitch, J.: A look at Random Numbers, Noise, and Chaos with Csound. In: R. Boulanger (ed.) The Csound Book, pp. 321–338. MIT Press, Cambridge (2000)
2. Csound Github site, `http://csound.github.io`

Digital Signal Processing Techniques Used to Model the Ibanez Tube Screamer Guitar Pedal

Rory Walsh[1] and Conor Walsh[2]

Dundalk Institute of Technology

Abstract. This paper aims to provide a basic overview of methods used in replicating analogue distortion units, using the Csound audio programming language. A key focus will be on the TS- 9 Tube Screamer (TS) analog overdrive guitar pedal by Ibanez. Although lacking in the complex theoretical analysis seen in typical digital audio effects papers, it is hoped that enough information is provided for beginners who wish to begin their own journey into the world of digital emulations of hardware devices.

Keywords: Csound, Analogous Distortion, Emulation

1 Introduction

Distortion effects can be traced back to the mid 1960s. For decades they have played an important role in the sound of electric guitars within popular music. Such effects are based on non-linear distortion of the signal path and can be thought of as wave-shaping circuits. Digital imitations of these analog processors have appeared in many different forms. Certain emulations aim to directly copy a particular analog circuit to replicate its exact behaviour [1]. However, the likelihood of exactly replicating the device chosen for this research is quite low considering the complexity of the device itself. This paper merely looks at methods that could potentially be used to emulate the unit using simplified models.

2 Ibanez Tube Screamer

The TS overdrive effects pedal is produced by Ibanez. It is known for its light distortion, similar to the sound produced from overdriven tube amplifiers. The TS pedal differs from other overdrive pedals on the market due to its unique compression of the waveform, resulting in very little loss of the original signal and capable of creating a full sounding blues tone. Unlike many overdrive pedals, the TS generates symmetrical soft clipping of the incoming signal [2]. Other overdrive pedals available on the market, such as the Boss SD-1, contain similar asymmetrical waveform clippings, which in turn result in a tube-like overdrive [3].What the TS actually does is overload the amplifiers preamp circuit with artificial gain. When the preamp gain is turned up on an amplifier, the TS saturates the signal, creating a full, overdriven tone.

Fig. 1. TS-9 Tube Screamer [4]

2.1 Schematic Breakdown

There are various schematics of the TS available online. ElectroSmash provides the most accurate layout, therefore, it will be the primary reference [5]. This analysis is split into three sections: Input/output buffers, the clipping stage and tone/volume stage. A full schematic is provided below in Figure 2.

Along with the symmetric clipping, the TS also provides a unique mid frequency boost. A characteristic associated with the low pass filtering stages of the circuit.

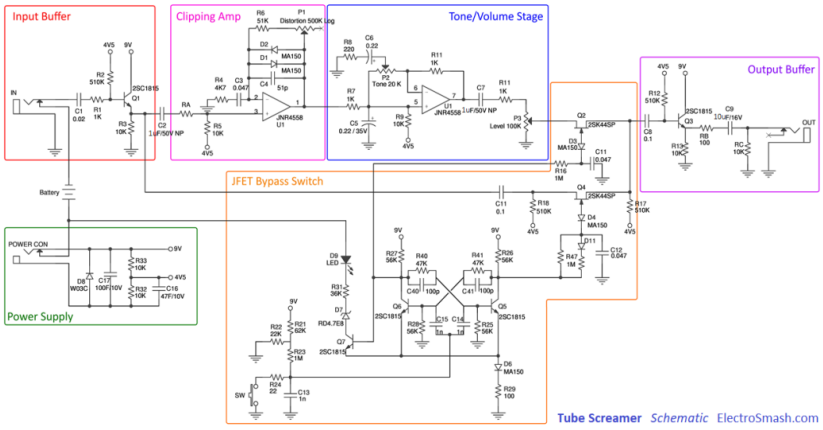


Fig. 2. Tube Screamer schematic [5]

2.2 Csound Implementation

The main features to focus on regarding the TSs unique characteristics are its overdrive and tone controls. Various types of wave-shaping transfer functions were researched during the implementation stages of this research, along with low pass filtering opcodes such as `moogladder`, `butterlp` and `tone`. Ultimately, the use of a `tanh` transfer function [6], along with basic tone controls seemed to be the most suitable combination of processes needed to approximate the sound of the TS.

A simplified implementation using the `tanh` transfer function and `tone` low pass filter opcode has been provided below. This example uses a GEN04 table to analyse the transfer function and create a complimentary normalised table which is used to scale the output.

```
<CsoundSynthesizer>
<CsOptions>
-odac
</CsOptions>
<CsInstruments>
; Initialize the global variables.
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1
gifn1 ftgen 1, 0, 4097, "tanh", -180, 180
gifn2 ftgen 2, 0, 1024, 4, 1, 1
instr 1
a1, a2 diskin2 "samples/02_C_Note_DI.wav", 1, 0, 0
;high pass filter all frequencies above 720 mid-hump

aHp butterhp a1, 1220
;low pass all frequencies below mid hump frequency
aLp butterhp a1, 1220
;apply distortion to frequencies above mid-hump only
aDist tablei (aHp+1)/2, gifn1, 1
kScl tablei 1, gifn2, 1
aDist = aDist*kScl
;apply low pass filter to distorted signal
aDist tone aDist, 3090
;sum distorted signal with original low pass filtered signal
aDist = aDist+aLp
outs aDist, aDist
; fout "FinalImp.wav", 4, aDist*.5, aDist*.5
endin
</CsInstruments>
<CsScore>
```

```
;starts instrument 1 and runs it for a 3 seconds
i1 0 3
</CsScore>
</CsoundSynthesizer>
```

The high pass and low pass filters in this code are used to split the signal chain in the same way the TS does. The higher frequencies are passed through the distortion chain, whilst the lower frequencies remain unchanged. Figure 3 displays the similarities between the original TS signal and the Csound emulation. The lower frequencies in the spectrum show a good likeness, whilst the higher frequencies seem to tail off a lot faster in the Csound emulation than the TS. Modifying the filters cut off frequency will help to alleviate this, but it involves a little tweaking.

To gain easier control of the output signal, the final implementation features a set of graphical controls that users can use to tweak aspects of the instruments output. Users can also choose between tanh clipping, or extreme hard clipping. It is possible, with some basic experimentation, to dial in a tone that sounds quite close in timbre to the original TS guitar pedal.

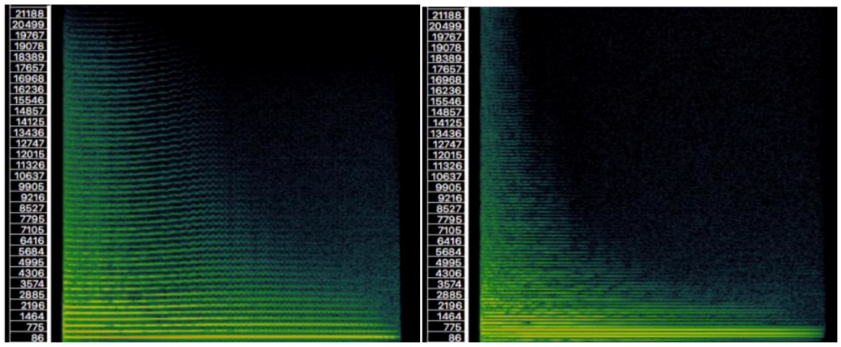


Fig. 3. Spectral analysis of the Tube Screamer (left), *tanh* function and *tone* low pass

3 Conclusion

The focus of this paper is on simple methods that can be used in emulating the output of the Tube Screamer distortion pedal. In many ways it represents quite a naive approach, considering little or no consideration was given to the possibility of aliasing or other complex side-effects of wave-shaping with complex sound sources. Regardless, a combination of a `tanh` transfer function and basic tone controls seem to work quite well as basic tools for emulating the Tube

Screamer. This can be largely attributed to the symmetric form of clipping they produce.

The use of analog electronic circuit simulators such as the SPICE programme [7] were investigated early on during this research. While the ability to emulate schematic circuity provides an interesting pedagogic insight, a full exploration of this tool was far beyond the scope of this research.

The combination of ctcsound [8] and Scipy [9] also proved to be very useful in the explorations of both the original and emulated signals. The ability to forensically plot pressure and frequency graphs proved invaluable. That being said, it is important to keep in mind that the human ear also plays an significant role in DSP emulation, and is summed up nicely by George Massenburg, who states that A DSP engineer with very good ears generally does better than the guy staring at MATLAB emulations. [10]

References

1. Timoney, J., Lazzarini, V., Gibney, A. and Pekonen, P. (2010). Digital Emulation of Distortion Effects by Wave and Phase Shaping Methods. [ebook] Maynooth, Ireland: Maynooth University, pp.1,2,3. Available at: `http://eprints.maynoothuniversity.ie/4116/1/dafx-distortion-final.pdf` [Accessed Feb. 2019].
2. Keen, R. (1998). The Technology of the Tube Screamer. [online] Geofex.com. Available at: `http://www.geofex.com/article_folders/tstech/tsxtech.htm` [Accessed 16 Jan. 2019].
3. Piera, M. (n.d). Boss OD-1 Overdrive Mods. [online] Analogman.com. Available at: `http://www.analogman.com/od1.htm` [Accessed 16 Apr. 2019].
4. Gig Gear (n.d.). TS-9 Tube Screamer. [image] Available at: `https://giggear.co.uk/buy/ibanez-ts9-tube-screamer-pedal` [Accessed 16 Apr. 2019].
5. Rodriguez, J. (2012). ElectroSmash - Tube Screamer Circuit Analysis. [online] Electrosmash.com. Available at: `https://www.electrosmash.com/tube-screamer-analysis` [Accessed 13 Jan. 2019].
6. Lazzarini, V. (2009).Distortion Synthesis. [online] Csoundjournal.com. Available at: `http://csoundjournal.com/issue11/distortionSynthesis.html` [Accessed 7 Feb. 2019].
7. Koren, N. (2003). Improved Vacuum Tube Models for SPICE, Part 1. [online] Normankoren.com. Available at: `http://www.normankoren.com/Audio/Tubemodspice_article.html` [Accessed 24 Apr. 2019].
8. `https://github.com/conorwalsh182/ConorWalshCsound/blob/master/README.md`
9. Scipy.org. (2019). SciPy.org SciPy.org. [online] Available at: `https://www.scipy.org` [Accessed 16 Aug. 2019].
10. Lambert, M. (2010). *Plug-in Modelling* —. [online] Soundonsound.com. Available at: `https://www.soundonsound.com/techniques/plug-modelling` [Accessed 25 Apr. 2019].

Synthesis by Parametric Design

Simone Scarazza

Abstract. As a composer I started my research aiming at developing a relationship between the graphic elements and the sound ones. Particularly I focused on the possibility to employ in the sound synthesis, processes and concepts belonging to Parametric Design used in computer graphics. Thanks to this study I built up a kind of library consisting of several models useful for the composition and source of stimulus to master the research toward a graphic approach. In this paper it will shown an example trough Csound.

1 Introduction

In the digital environment the beginning of the Parametric Design dates back to 1963 when Ivan Sutherland, conceiving the Sketchpad the forerunner of Graphical User Interface, introduced new functions in order to create in the patterns, variable and scalable geometries.

Compared to Computer-Aided Design (CAD), the software supporting Parametric Design allows to represent and produce models that, thanks to the synchronism of the parameters, they may grow and can be modified as organisms.

One of the best known software for parametric design is Grasshopper, a visual programming language and environment developed by David Rutten at Robert McNeel & Associates, that runs within the Rhinoceros 3D computer-aided design (CAD) application.

Today the use of graphical digital interfaces and the manipulation of the codex have deeply penetrated the designer expressive language and creative concept. It cooperates in finding solutions in which assisted design doesn't merely consist in supporting its production but it permits to bring new possibilities of useful interacting for the customization of the project and helps to overcome its limits.

2 Sound Synthesis and Graphic Sign

Sound synthesis and graphics have had many points of contact since the beginning of their history. One of the first was "Graphic 1" developed by William Nike in the Bell laboratories in 1961. It is a hardware and software system started for engineering purposes, which was employed by Max Mathews, combined with "Music IV", in order to be able to define sound parameters graphically.

Another important informatics tool to be mentioned is UPIC (Unité Polygogique Informatique CEMAMu) realized by Iannis Xenakis in 1977. UPIC allows to translate drawings, realized with an electromagnetic pen on an electromagnetic board, into music.

Once the drawing has been digitized it is interpreted by computers which combine the graphic form with specific frequency and duration parameters. For example the drawing of an ascending trait corresponds to an ascending glissando.

3 General Description

In order to use Parametric Design as a technique of sound synthesis it has been taken the same model implemented by Xenakis with the UPIC, where in a graphic drawing, definite in a Cartesian coordinate system, the abscissa indicates the time and the ordinate the frequency.

To convert graphic elements, placed in a Cartesian coordinate system into the parameters to generate sound synthesis, the same will be realized in an unit square, (Fig.1) that is a square consisting of the points where both x and y lie in a closed unit from 0 to 1.

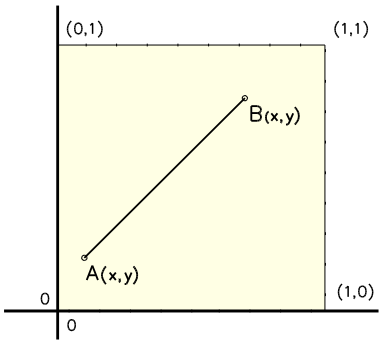


Fig. 1 - Unit Square

In this way, once calculated starting, duration and frequency of the lines, these values can be sized according to the compositional needs of duration, minimum frequency and band-width indicated in the *note-statment* of the score.

Example

```
istart    = Ax
idur      = Bx - Ax
ifreqA    = Ay
ifreqB    = By

------- note statement ------------------------
p1        p2   p3   p4         p5
                  Min.Freq   Band-width
I1        2    10   100        300
```

```
------- sizing --------------------------------
inotestart = p2 + (p3*istart)
inotedur   = p3 * idur
inotefreqA = p4 + (p5*ifreqA)
inotefreqB = p4 + (p5*ifreqB)
```

In Parametric Design one of the most interesting topic to deal with using generative algorithms are geometric patterns. Basically their construction in digital domain results very simple, but the combination between different patterns allows the realization of very interesting shapes.

In the Csound's example that follows, the graphic shape used is composed by the construction lines of Bézier curves.

In this case the shape has been used to create additive synthesis, but in other cases it can be used to control any other parameter of a synthesis algorithm, such as the frequencies of a filter bank, FM parameters, or in granular synthesis, speed, volume, and frequency of grains.

4 Implementation

For the realization of the Csound program we need to start from a graphic model to understand how to automate the generative process. (Fig.2)

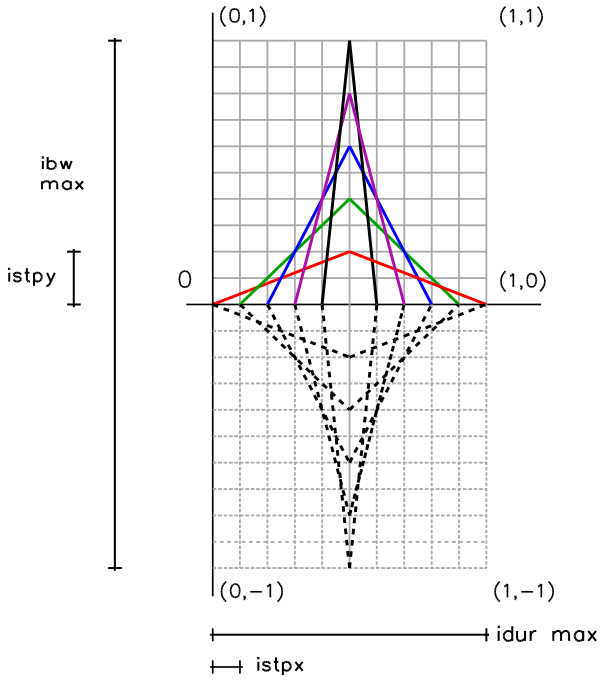


Fig. 2 - graphic model in a Cartesian coordinate system

As shown the drawing is mirrored on the x-axis so, in the first step, it is possible to consider only the positive plane to implement the algorithm. The graph shows a *n* number of poly-lines[1] that have an ascending direction to the middle of the diagram and a descending direction until their end. Each poly-line respect to the previous one has an increase in glissando frequency of a certain step, a delayed start of a certain step and a proportionally decreasing duration.

Thus, it is possible to implement the algorithm creating an iterative structure.

The program is based on two instruments that work together: instrument 1 generates the control parameters for instrument 2 that contains synthesis algorithm.

Instrument 1 will be implemented by a loop (`loop_it` opcode), which will be iterated according to *n* number of required poly-lines. The Loop will iterate *n* times mathematical calculation between the initialization variables and a step value in order to generate a note-list for instrument 2.

In the note-statement of the score file, that calls instrument 1, `P4` will indicates the global amplitude, `P5` the central frequency, `P6` the band-width and `P7` the number of the poly-lines.

Inside the algorithm, `P7` plays a key role: it will indicate the number of called instances of loop, it will divide the global amplitude to share it on each poly-line and it will define the steps values (`istpx,istpy`) that are inversely proportional to `P7`.

Instrument 2 contains the control variables for the glissando, an anti-foldover logic filter, an amplitude envelope, two sinusoidal oscillators and a DC filter.

The control variables for glissando are two: one for the positive plane, that *sums* the central frequency with the band-width and one for the negative plane, that *subtracts* the central frequency with band-width. In this way the drawing will be mirrored.

If the note-list, generated by instrument 1 has negative frequency values, the anti-foldover logic filter avoids this problem bringing them to 0Hz. Consequently to eliminate the 0Hz, at the end of the audio process, another filter has been implemented with the `dcblock2` opcode.

Example of Program Code

```
<CsoundSynthesizer>
<CsInstruments>

sr        = 44100
ksmps     = 32
nchnls    = 1
0dbfs     = 1

instr     1;----------------------------------------

istart    = 0
idur      = 1          ; duration
ibw       = 0
```

1 poly-line in computer graphics is a continuous line composed of one or more line segment

```
inop      = p7          ; number of poly-line
istpx     =.5/ inop     ;(1/p7)*0.5
istpy     = 1/ inop     ; 1/p7
;--- Event note ------------------------------------
ieamp     = p4 / p7     ; Amp / number of  poly-line
iefreq    = p5
iestart   = 0
iedur     = p3
iebw      = 0           ; band-width

event_i   "i",2,iestart,iedur,ieamp,iefreq,iebw

;--------- loop parameters ---------------------------
indx      = 0  ;
incr      = 1  ;
inumber   = inop  ; number of iterations
;-------------------------------------------------
sequence:    ;loop start process

ibw       = ibw+istpy       ; band-width
iebw      = ibw*(p6*.5)     ; sizing

event_i   "i",2,iestart,iedur,ieamp,iefreq,iebw

istart    = istart+istpx    ; starting point
iestart   = istart*p3       ; sizing
idur      = idur-(istpx*2)  ; duration
iedur     = idur*p3         ; sizing

loop_lt indx,incr,inumber,sequence      ;loop
endin

instr 2   ;-----------------------------------------

idur      = p3
kglis     linseg p5, idur*.5, p5+p6, idur*.5, p5
kglisneg  linseg p5, idur*.5, p5-p6, idur*.5, p5

;-------  Anti-foldover filter  ----------------------
if (kglisneg <= 0) then
    kglisneg = 0
endif
;-------------------------------------------------------
kenv      linseg 0,idur*.5,p4,idur*.5,0

a1        poscil kenv, kglis, 1
a1n       poscil kenv, kglisneg, 1
aout sum a1,a1n
```

```
adcb dcblock2 aout ;------ DC filter -----------------
out       adcb
endin
</CsInstruments>
<CsScore>
f1        0  4096  10   1

; P1     P2   P3   P4    P5      P6      p7
i 1      3    30   0.5   2000    3000    10

</CsScore>
</CsoundSynthesizer>
```

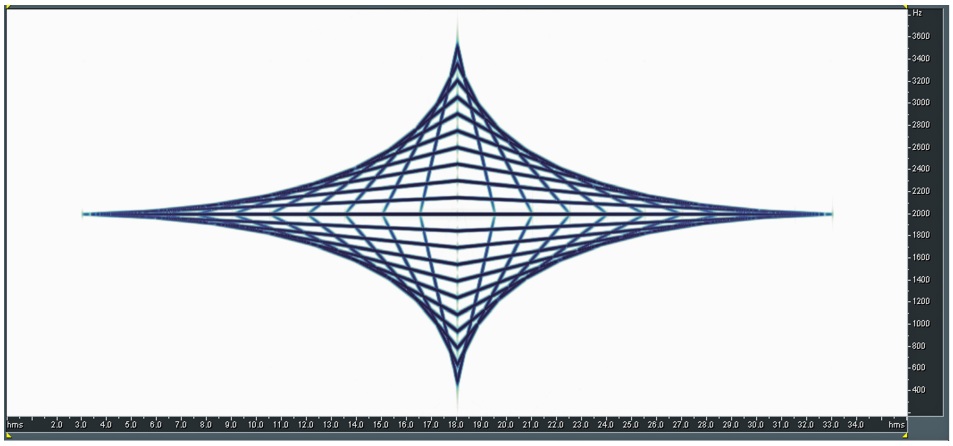


Fig. 3 - Spectrogram of the sound file written by the program

5 Conclusion

In this example an essential program has been realized to clarify as much as possible its working logic but it is possible to obtain substantial transformations adding p-fields, that will intervene on loop parameters; or applying in instrument 2 whatever synthesis technique on the oscillators used for the sound generation.

Working with this kind of program it's possible to create morphologies which possess their own consistency because parametric software permits their adaptive management and to control, through tiny parameters, a great amount of data.

The use of Parametric Design to generate control parameters, provides an interesting alternative to the most used technique by generation of data trough random distributions.

My next step will consist in developing a collection of User Defined Opcodes in order to be able to easily recall each algorithm and interface it with the others.

References

1. Gérard, Ma., et al.: The UPIC System: Origins and Innovations. Prospectives of New Music Vol.31 No.1, 258-269 (1993)
2. Jabi, W.: Parametric Design for Architecture. Laurence King Pub (2013)
3. Woodbury, R.:Elements of Parametric Design. Routledge (2010)
4. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)

iVCS3 Programming & The Repurposing of Audio Files To Carry Control Voltage Levels.

Author: James Edward Cosby

jamesedwardcosby@jecdesign.com

Abstract…

1. To Show how CV Signals can be encoded using Audio Samples and stored within a standard ".wav" File to be used to expand the sonic possibilities of iVCS3 and equivalent hardware…

2. To show Examples of iVCS3 Programming including the transmission of CV Signals over Audio busses within iOS showing the Control of another iSO Synth using APEMatrix for the audio bus connection. Show the use of CV Audio Files to add highly programmable enveloping to iVCS3 patches…

Keywords: iVCS3, Control Voltages, Audio Files, Sound Design, Modular Synthesis.

The Author…

The author has over forty years of experience in hardware analogue synthesis and around thirty years of experience with analogue/digital hybrids and FM. His other credits include 1980's 8-bit game design primarily on Z80 platforms at an Assembler Level and including Graphic and Sound Design. In addition to iVCS3 the author has recently contributed presets to several iOS Software Synths including Audio Kit Synth One and D1.

1 CV Changes over Time, Encoded using Audio Samples.

Consider the similarity between Audio Signals and Control Voltage Signals... Both are Bi-Polar Voltages which change over time… Thus it follows that both can be represented digitally and stored for subsequent replay at will.

In the case of an LFO CV Signal, this can be **simply** represented by an audio wave with a cycle frequency in the sub audio range..

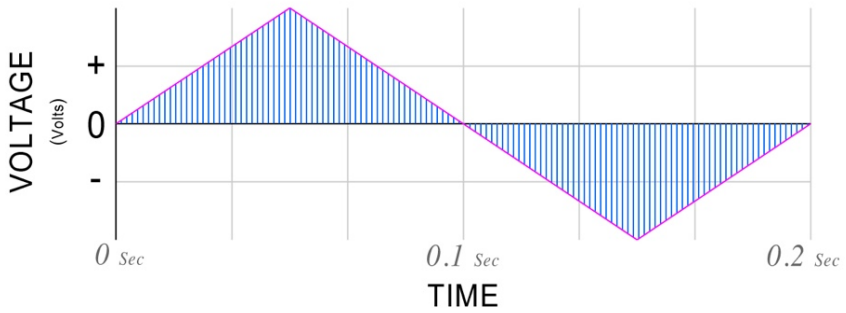


Fig1.

In the case of an Envelope, the perceived change in amplitude over time of any given sound wave can also be represented by samples in the same way…

First, consider Fig 2. a simple Sine Wave upon which an Amplitude envelope is applied resulting in a Percussive Transient…

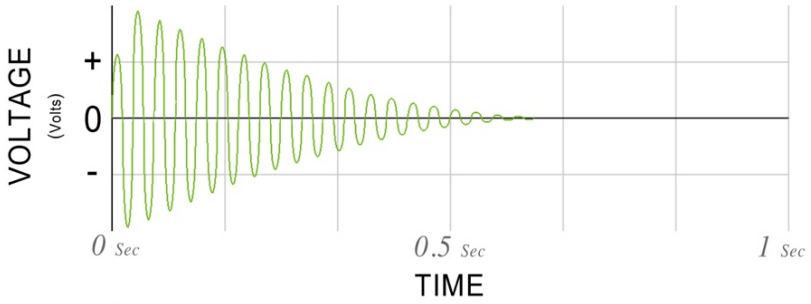


Fig 2: Wave with percussive transient

The Perceived Rise and Fall of Amplitude over Time (Envelope), shown in Fig 3. can be shown by considering the "Absolute Values" of the Sampled Waveform…

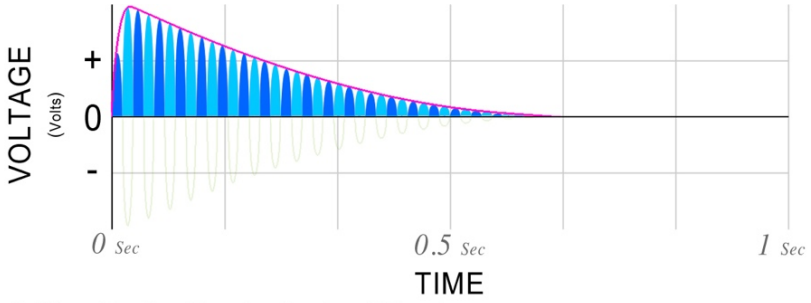


Fig 3: Wave Absolute Showing Percieved Transient

Thus, as shown below in Fig 4. The Perceived Envelope can be represented by digital samples which can therefore be stored in a standard ".wav" or other format "Audio" file. This "Audio" file can then be processed using existing subroutines and used to Modulate any desired parameter…

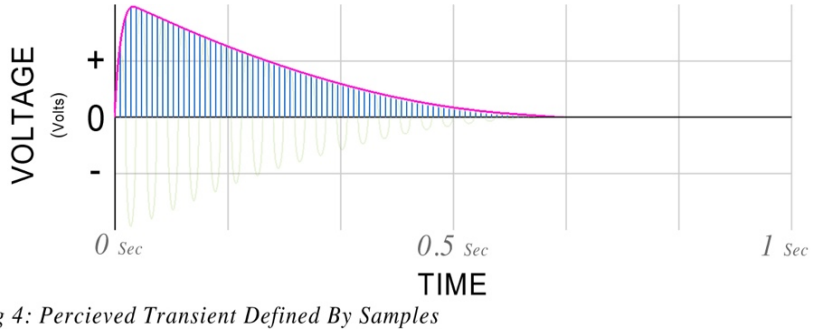


Fig 4: Percieved Transient Defined By Samples

Loading into iVCS3 to Modulate Parameters…

Once loaded into iVCS3's Sampler Module using the Folder Icon, in this case, Fig 5, the file is loaded into Channel 2, various parameters can be set to refine the modulation effect…
"Rate" will adjust the duration of the modulation, (the rates of both sampler channels can also be synchronised to each other). The "Mix" parameter will adjust level balance between the sample file and the incoming iPad audio bus. For a stereo file, the "L/Mix/R" switch assigns which stereo channel(s) of the file are used. The "Off/DK/Seq" switch selects the Retrigger Mode, i.e. simple continuous loop or Dynamic Keyboard/Midi Note Triggering or Internal Sequencer Note Triggering. A portion of the file can also be selected using the Crop Icon.

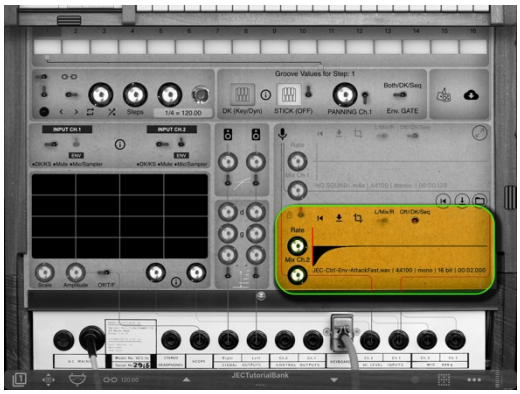


Fig 5. CV Envelope Loaded into iVCS3

Once the CV File is loaded to the desired Sampler Channel, the signal will be present at the Matrix… Row 8 for Input Channel 1 and Row 9 for Input Channel 2. This Signal can be attenuated using the Input Channel Level Parameter Knobs and thus setting the "Amount" of modulation applied to the destination parameter. The example in Fig 6. shows a basic playable patch with Osc 1&2 patched through the Filter and Trapezoid to the Outputs. Both Osc 1&2's Frequencies are modulated by the DK which is routed through Input Channel 1. The loaded CV Envelope File is patched from Input Channel 2 to modulate the Filter Cutoff Frequency and synchronised to trigger with a DK/Midi Note.

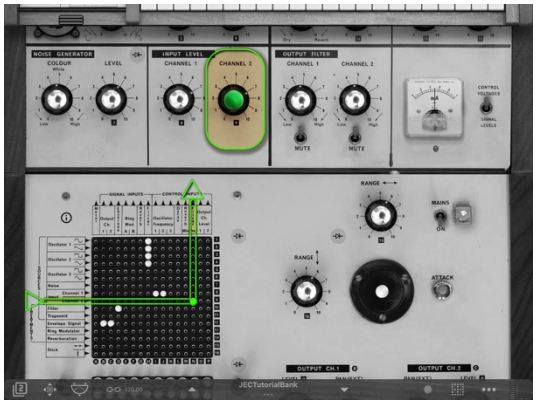


Fig 6. Example CV File Filter Modulation

This results in much greater dynamic possibilities than are otherwise achievable, indeed, the idea has been tested and found to work well on the original hardware using a synchronised sample player and transmitting the CV Files over an audio bus. The idea has also been tested using the iPad internal audio busses allowing successful modulation control of one modular synthesiser from another.

2 Video Presentation using iVCS3 as the prevalent sound source

Show a video of the re-created 1960's Dr Who television theme using iVCS3 as the main sound generator synchronised to the original black & white recursive graphics…

3 iVCS3 Programming Examples and using Audio CV Files.

This section of the presentation to include a short description of iVCS3's modules and their peculiarities and will be an open Q/A discussion on iVCS3 / VCS3 programming, including a breakdown of some patches used in the aforementioned theme…

Available Topics to Include…

The Matrix, direction of signal flow, touching on Pin values and Meter.

Oscillators 1, 2 & 3… and their waveforms, briefly describing their hardware voltage control specifications and how this relates to Pin & DK settings for equal temperament.

Ring Modulator… functionality, "Differential" output and how this can be used to alter the amplitude of signals including use of the Joystick to generate a "bowing" effect.

Filter… Parameters including Ladder & ZDF_Diode selection, "Slew" and "Saturation"

AHDR Trapezoid Envelope Shaper… covering Decay Modulation, Looping, CV and Signal Paths

Spring Reverb… Level, Mix Modulation and Settings.

Noise Generator… Colour and Level Parameters and S&H / Glide Modes

Output Filters… Twin Dual Low/High Pass Filters

Joystick and Range Parameters… Covering Modulation possibilities and demonstration of special values / calculations for Octave and Note transposition and OSC3/Filter DK Tracking.

Output Channels 1 & 2… Level Modulation including CV File Amp Envelope Modulation and Panning.

References

EMS: EMS VCS3 “The Putney” User’s Manual…
https://www.manualslib.com/products/Ems-Vcs3-Putney-3970479.html

MVerb: A Modified Waveguide Mesh Reverb Plugin

Jon Christopher Nelson

University of North Texas College of Music,
Center for Experimental Music and Intermedia (CEMI)

Jon.nelson@unt.edu

Abstract. MVerb is a plugin that is based on a modified five-by-five 2D waveguide mesh developed in Csound within the Cabbage framework. MVerb is highly flexible and can generate compelling and unique reverberation effects ranging from traditional spaces to infinite morphing spaces or the simulation of metallic plates or cymbals. The plugin incorporates a 10-band parametric EQ for timbrel control and delay randomization to create more unusual effects.

Keywords: Reverberation, Effects Plugin, Physical Modeling, Waveguide, Scattering Junction, Csound, Cabbage.

1 Introduction

Artificial reverberation can provide a compelling sense of acoustic space that factors significantly in the creation and production of digital audio. Digital reverberation models have changed dramatically both in quality and complexity as computational capabilities have evolved. These models include tapped recirculating delays comprised of comb and allpass filters coupled with multitap delay lines [1], physical models based on the projection of source vectors [2], impulse response convolution models [3], temporal smearing via asynchronous granular synthesis models [4], feedback delay networks [5] [6], closed waveguide mesh networks [3], and a wide variety of hybrid models.

Of these models, waveguide meshes provide interesting creative possibilities for great diversity of reverberation effects. The model for a classical waveguide mesh consists of a network of 4-port scattering junctions connected with waveguides that exhibit diverse delay times that can simulate different echo times and prominent resonances within an acoustic space [7]. With a sufficient number of waveguides, this physical model can emulate a wide variety of room colors and sizes

as well as large plate reverberators and any number of other metallic (or non-metallic) percussion instruments. Waveguide mesh boundaries reflect the signal back into the mesh, inverting the signal with some signal loss and often filtering the reflection to emulate the absorptive qualities of a given physical space. Since it is a closed waveguide network, infinite reverbs will result unless a reflection coefficient of less than 1.0 is utilized. MVerb is based on a modified waveguide mesh that capitalizes on some of the unique possibilities of this physical model.

2 MVerb Waveguide Mesh Design

The classical 2D waveguide mesh consists of a network of 4-port scattering junctions configured with intermediary waveguides (see figure 1).

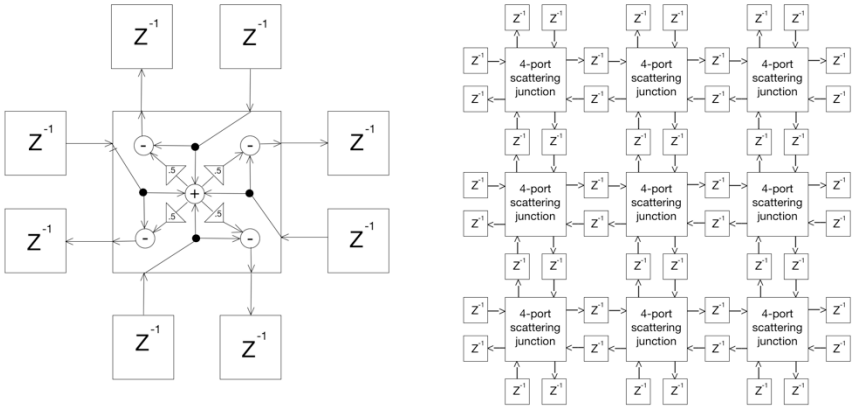


Fig. 1. 4-Port Scattering Junction and 2D Waveguide Mesh

While MVerb is fundamentally based on this model, it includes minor modifications that were incorporated to facilitate both slightly more efficient coding using Csound [8] within the Cabbage framework [9] and greater control and flexibility. MVerb consists of a 5 x 5 mesh with parametric EQ embedded within each scattering junction. A traditional 5 x 5 waveguide mesh includes a waveguide between every horizontal and vertical adjacency as well as each boundary, resulting in 120 delays organized in discrete pairs. Within Csound, the classical 2D waveguide mesh could be coded using UDOs to define each scattering junction

and each connecting waveguide. However, the audio signal routing code connecting the scattering junctions and delay lines is extensive. In an effort to simplify this signal routing, the MVerb mesh model incorporates the waveguides within the scattering junction UDOs. Specifically, each scattering junction includes only the outgoing delays (half of each waveguide) with each of the 4 delays assigned a single delay time. While this still requires the use of 100 delays (25 scattering junctions with 4 outputs each), this modification provides greater efficiency and ease in structuring the waveguide mesh while retaining a rich set of prominent resonant frequencies.

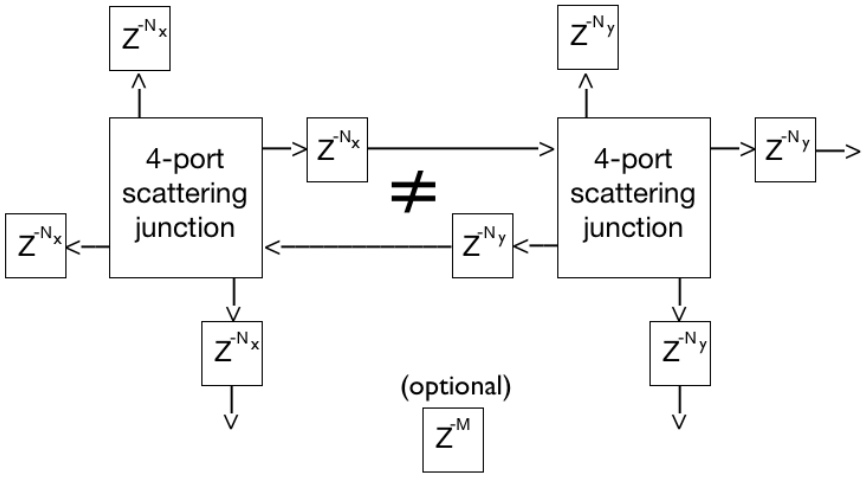


Fig. 2. MVerb Mesh Design

In this model, the waveguides between adjacent scattering junctions will consist of unequal delay values. Each unequal delay value will consequently have a unique resonant frequency. In addition, the MVerb mesh model also allows for variable delay lines.

The MVerb mesh modifications facilitate coding efficiencies through the incorporation of several primary UDOs. The *EQ* UDO creates a 10-band parametric equalizer that is embedded within each scattering junction. The *meshEQ* UDO defines a scattering junction and its associated output delay lines with identical delay times as follows:

```
opcode  meshEQ,aaaa,aaaaak
aUin,aRin,aDin,aLin,adel,kFB xin
afactor=(aUin+aRin+aDin+aLin)*-.5      ;calculate raw value
```

```
aUout   vdelay  aUin+afactor,adel,1000 ;calculate outputs
aRout   vdelay  aRin+afactor,adel,1000
aDout   vdelay  aDin+afactor,adel,1000
aLout   vdelay  aLin+afactor,adel,1000
aUout   EQ   aUout  ;apply EQ UDO to each output
aRout   EQ   aRout
aDout   EQ   aDout
aLout   EQ   aLout
xout    aUout,aRout,aDout,aLout
endop
```

After initializing the necessary delay lines, the code simply builds the mesh one scattering junction at a time. The naming conventions for this 5 x 5 model assign each scattering junction a letter (A through Y) with scattering junction inputs and outputs designated as up (U), right (R), down (D), or left (L). Thus, the audio signal *aGD* is the downward output from scattering junction G. The following code represents the top two rows of five scattering junctions using the *meshEQ* UDO:

```
aAU,aAR,aAD,aAL   meshEQ        aAU,aBL,aFU,aAL,adel1,kFB
aBU,aBR,aBD,aBL   meshEQ        aBU,aCL,aGU,aAR,adel2,kFB
aCU,aCR,aCD,aCL   meshEQ        aCU,aDL,aHU,aBR,adel3,kFB
aDU,aDR,aDD,aDL   meshEQ        aDU,aEL,aIU,aCR,adel4,kFB
aEU,aER,aED,aEL   meshEQ        aEU,aER,aJU,aDR,adel5,kFB
aFU,aFR,aFD,aFL   meshEQ        aAD,aGL,aKU,aFL,adel6,kFB
aGU,aGR,aGD,aGL   meshEQ        aBD,aHL,aLU,aFR,adel7,kFB
aHU,aHR,aHD,aHL   meshEQ        aCD,aIL,aMU,aGR,adel8,kFB
aIU,aIR,aID,aIL   meshEQ        aDD,aJL,aNU,aHR,adel9,kFB
aJU,aJR,aJD,aJL   meshEQ        aED,aJR,aOU,aIR,adel10,kFB
```

MVerb code also includes a master feedback coefficient, *kFB*, that is applied to each scattering junction as well as code to clear all delay lines. The plugin code contains ample gain and DC offset control to minimize potential signal problems prevalent in a closed waveguide mesh, thus providing convincing and controlled infinite reverberation when using a reflection coefficient of 1.0.

3 MVerb Features

Working within the Cabbage framework, MVerb incorporates a user interface (see figure 4) that includes numerous end-user controls that modify and shape the sonic result.

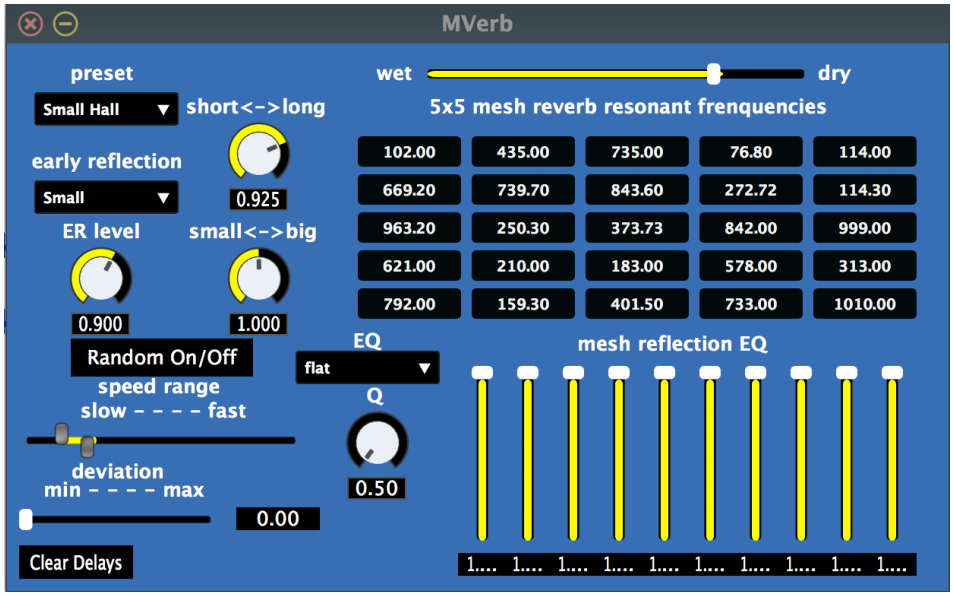


Fig. 3. MVerb User Interface

A variety of preset values can be selected to control an optional multitap delay line that adds early reflections to the incoming audio signal with independent output level control. Similarly, a number of preset delay times have been stored for the user to select. These presets include more traditional concert spaces, very colored or unusual reverbs, and effects with prominent resonances derived from sampled cymbals. MVerb also allows the user to define 25 prominent resonant frequencies, thus tuning each scattering junction output. The user interface also includes parametric equalizer controls, a master reflection coefficient, a button to clear all delays, and a master size control that applies a delay time multiplier to every scattering junction delay line. Finally, in an effort to create some more unusual and interesting effects, this plugin also includes an optional random deviation for each scattering junction delay value. This provides a possible means of creating the sense of a very slowly evolving and morphing room or, conversely, a rapidly changing and very noisy delay effect.

4 Conclusion

MVerb provides flexible and rich reverberation effect possibilities. While its current instantiation is a stereo effect, future plans include the development of MVerb plugins with various channel counts for inputs and outputs and better user preset capabilities. In addition, greater exploration of the sonic properties of meshes with a higher dimensionality (3D, 4D, . . . xD) and diverse configurations may prove to be a fertile area of discovery. In particular, non-planar mesh structures with more arbitrary or random scattering junction connectivity and reflection boundary placement may facilitate the development of more unusual effects.

References

1. Schroeder, M. R.: Improved Quasi-Stereophony and "Colorless" Artificial Reverberation. Journal of the Acoustic Society of America 33(8), 1061-1064 (1961)
2. Moore, F. R.: A General Model for Spatial Processing of Sounds. Computer Music Journal 7(3) 6-15 (1983)
3. Smith, J.: A New Approach to Reverberation Using Closed Waveguide Networks. In: B. Truax (ed.) Proceedings of the 1985 International Computer Music Conference, pp. 47-53. Vancouver (1985)
4. Roads, C: The Computer Music Tutorial. MIT Press, Cambridge, Massachusetts (1996)
5. Stautner, J. and Puckette, M.: Designing Multichannel Reverberation. Computer Music Journal 6(1), 52-65 (1982)
6. Zölzer, U. DAFX Digital Audio Effects. John Wiley & Sons, Ltd, West Sussex, England 2002
7. Smith, J.: Physical Audio Signal Procssing web site, `https://ccrma.stanford.edu/~jos/pasp/pasp.html`
8. Csound site, `https://csound.com`
9. Cabbage site, `http://cabbageaudio.com`

Kairos - a Haskell Library for Live Coding Csound Performances

Leonardo Foletto

Berklee College of Music
fltleonardo@gmail.com

Abstract. Kairos [1] is a library for the Haskell programming language designed to live code patterns of Csound score instructions to be sent to a running UDP server [2] of a pre prepared Csound orchestra.

Keywords: Live Coding, Csound, Haskell

1 Introduction

Within the context of the arts, live coding has seen an increasing adoption. Today, a growing community of artists who, already accustomed to using software as a tool in their artistic practice, strive to find new methods to interact with their machines in more personal and meaningful ways. In the past ten years, many open source software tools have been developed to perform and compose live coded music including SuperCollider[3], TidalCycles [4], Conductive[5], Sonic Pi[6] .

Kairos is a live coding system inspired by some of the operational principles of the above mentioned softwares, but born from the need of the author to create an environment to perform and compose music more closer to his needs that feels intuitive to use and provides a high degree of control with a simple, yet powerful syntax.

2 System Overview

There are two main parts to the system: a Csound file, `kairos.csd`, and an accompaning Haskell library. The focus of the project has been developing an Haskell library to format patterns of data into Csound readable score to be sent over a UDP network to a running instance of a Csound server.

2.1 Csound File

The file `kairos.csd` contains a number of pre defined instruments and global effects. All the instruments have been programmed to have a positive finite value of `p3` and are written to maximize the number of common pfields amongst all the instruments.

Pfields shared amongst every instrument

```
 p4 : amplitude (0 - 1)
 p5 : reverb send (0 - 1)
 p6 : delay send (0 - 1)
 p7 : panning (0 - 1)
```

Example of a simple sampler instrument

```
<CsInstruments>

instr 1 ;Sampler

inchs filenchnls p8

if inchs = 1 then
aLeft diskin2 p8, p9
outs aLeft*p4* sqrt(1-p7), aLeft*p4* sqrt(p7)
garvbL = garvbL + p5 * aLeft  * sqrt(1-p7)
garvbR = garvbR + p5 * aLeft  * sqrt(p7)
gadelL = gadelL + aLeft   * p6 * sqrt(1-p7)
gadelR = gadelR + aLeft  * p6 * sqrt(p7)

else
aLeft, aRight diskin2 p8, p9
outs aLeft*p4* sqrt(1-p7), aRight*p4* sqrt(p7)
garbL = garvbL + p5 * aLeft * sqrt(1-p7)
garvbR = garvbR + p5 * aRight * sqrt(p7)
gadelL = gadelL + aLeft  * p6 * sqrt(1-p7)
gadelR = gadelR + aRight  * p6 * sqrt(p7)
endif

endin

</CsInstruments>
```

Differently from the instruments, global effects are all built to run forever (`p3 = -1`) and use channels to manipulate their parameters, instead of using pfields.

Example of a reverb global effect

```
<CsInstruments>

;Reverb

garvbL, garvbR init 0

gkfbrev init 0.4
gkcfrev init 15000
```

```
gkvolrev init 1

gkfbrev chnexport "fbrev", 1, 2, 0.4, 0, 0.99
gkcfrev chnexport "cfrev", 1, 2, 15000, 0, 20000
gkvolrev chnexport "volrev", 1, 2, 1, 0, 1

instr 550 ; ReverbSC

aoutL, aoutR reverbsc garvbL, garvbR, gkfbrev, gkcfrev
outs aoutL * gkvolrev ,  aoutR * gkvolrev
clear garvbL, garvbR

endin

</CsInstruments>
```

3 Haskell Library

3.1 Data Structures

The ensemble of instruments contained in `kairos.csd` gets triggered with instructions coming from the Kairos Haskell library. This is the part of the software responsible for scheduling score events, sending them to Csound at the appropriate time and changing parameters of global effects.

All of the instruments and effects are represented in Haskell using the `Instr` data type, which not only has information about the instrument number and pfields of an instrument, but also about its status (active or inactive), the current note to be played and the patterns of parameters for every pfield.

All of the instruments are collected in a data structure called `Orchestra` that holds them and associates every instrument with a string that identifies it's name.

The two preceding instruments represented in Haskell with their Orchestra

```
sampler :: String -> IO Instr
sampler path = do
  pfields <- newTVarIO $ M.fromList [(3,Pd 1),(4,Pd 1)
                                    ,(5,Pd 0),(6, Pd 0)
                                    ,(7,Pd 0.5),(8,Ps path)
                                    ,(9,Pd 1)] -- p8 : Sample path, p9 : pitch
  emptyPat <- newTVarIO M.empty
  return $ I { insN   = 1
             , pf     = pfields
             , toPlay = Nothing
             , status = Stopped
             , timeF = ""
```

```
              , pats = emptyPat
              }

reverb :: IO Instr
reverb = do
  pfields <- newTVarIO $ M.fromList [(3,Pd (-1))]
  emptyPat <- newTVarIO M.empty
  return $ I { insN   = 550
             , pf     = pfields
             , toPlay = Nothing
             , status = Stopped
             , timeF = ""
             , pats = emptyPat
             }

defaultOrc :: IO Orchestra
defaultOrc = do
  k <- sampler "/KairosSamples/kicks/Kick909.wav"
  rev  <- reverb
  orc  <- atomically $ newTVar $ M.fromList [("K909",k) ,("rev",rev)]
  return $ orc
```

The `Orchestra` is held in a container named `Performance` that holds the Orchestra and the informations about tempo, time signatures and rhythmic patterns readily available to be used by the instruments.

The library is designed to be used within the `GHCi` [7] environment and can be loaded and started running `:script BootKairos.hs` from within GHCi, launching it from the folder containing the script. This script also sets up a number of convenient functions that help compose and modify patterns of instructions easily.

3.2 Operational principles

To use the library, first start the Csound server running the file `kairos.csd` and then launch an instance of GHCi and run the script `BootKairos.hs`. This script will load the necessary modules of the library, run all of the necessary setup steps to start a new `Performance` and also load many functions designed to reduce the amount of typing necessary to perform and simplify the interaction with the Csound orchestra.

To play an instance of an instrument a rhythmic pattern must be assigned to it and then start the play loop.

Playing a four on the floor pattern with the kick instrument from before

```
Kairos>  cPat "fourFloor" "K909" >> p "K909"
```

Patterns of values can then be assigned to the exposed synthesis parameters of the instrument. Every one of this pattern of values gets assigned an update function that determine in which way the value for that parameter will be picked for the next score event. The function can be picking a value from the list or modify the list itself.

Changing the panning and volume parameters

```
Kairos>  vol "K909" [Pd 1, Pd 0.8, Pd 0] randomize
Kairos>  pan "K909" [Pd 0, Pd 1] nextVal
```

An alternative option is the params function, that allows to declare and assign multiple parameters of pfields at the same time.

An example of the params function

```
Kairos> params "K909" [(keep, vol, [Pd 1]),(randomize, pan, toPfD [0, 1])]
```

4 Future Directions

The author uses the library in performances of live dance music and in a experimental electronic band context in a setup with Eurorack modular synthesizers.

One of the features that will soon be introduced is the ability to have a shared clock between multiple instances of Kairos to allow for ensemble performances.

On the musical side, the current focus of the research is on how to more effectively generate and manipulate streams of pfields in interesting ways. The author is now focusing on fractal-based models, Markov chains and autonomous agents.

References

1. Kairos Github repository, `https://github.com/Leofltt/Kairos`
2. Csound UDP Server, `https://csound.com/docs/manual/udpserver.html`
3. SuperCollider `https://supercollider.github.io/`
4. McLean, A. and Wiggins, G.: Tidal - Pattern Language for the Live Coding of Music. In: Proceedings of the 7th Sound and Music Computing conference (2010)
5. Bell, R.: An Interface for Realtime Music Using Interpreted Haskell (2011)
6. Sonic Pi `https://sonic-pi.net/`
7. GHC/GHCi HaskellWiki, `https://wiki.haskell.org/GHC/GHCi`

The Hex System: a Csound-based Augmentation of Hexaphonic Guitar Signal

Tobias Bercu,

Berklee College of Music
tbercu@berklee.edu

Abstract. The impetus behind the Hex system was a desire to create new guitar effects using Csound processing of hexaphonic guitar audio, and to present these effects to the user in a format that allows playing the instrument to meld with playing the effects. Processing one's guitar signal with a laptop or a desktop opens many doors, but can also be cumbersome. The Hex system is meant to provide guitarists a smaller and more liberated DSP apparatus that feels more like an augmentation of the instrument itself than a separate module. The Hex system processes audio via a Raspberry Pi running Csound. Using the Pi's onboard wifi, the system accepts control from TouchOSC, so that parameters can be adjusted in real-time from a nearby smartphone. It is intended for this smartphone to be attached to the guitar adjacent to the pickup and tone controls. The Raspberry Pi and its audio hat are housed in a small box, and this container is roofed by footswitches used to engage and disengage effects.

Keywords: Csound, real-time guitar effects, DSP, hexaphonic, Raspberry Pi

1 Introduction

The Hex system is a DSP prosthesis of sorts for guitar players, mainly intended to process the 6-channel output of a hexaphonic guitar pickup. Its purpose is to arm its user with a Csound-generated multi-effects suite comprising effects both traditional and innovative. As a smartphone-controlled guitar pedal, it is meant to present these DSP powers to the user as a natural extension of the instrument to which the smartphone is attached.

2 Overview of Hardware and Software

2.1 Hardware

The basic hardware ingredients of the Hex system are a guitar, a hexaphonic pickup, a splitter cable, a Raspberry Pi with an audio hat, a smartphone, and an Arduino. For this build, a Graphtech Ghost hexaphonic pickup, a homemade splitter cable, a Raspberry Pi 3 B+, an Audio Injector Octo Injector soundcard from Flatmax Studios, a Samsung Galaxy S7, and an Arduino Mega were used, respectfully. Figure 1 represents signal flow within the Hex system.

As most hexaphonic pickups send audio through a 13-pin cable, which typically connects to a companion processing unit such as the Roland GR-55, it was necessary to preempt the Octo Injector's RCA female ADC inputs with a splitter cable. The splitter cable feeds each of the six strings' audio into its own RCA head, and sends +9v, -9v, and a ground signal back to the hexaphonic pickup using two 9v batteries wired in series.

The Arduino Mega (a standard Arduino Uno would suffice) monitors voltages from eight on/off footswitches and sends their statuses to Csound via a USB serial connection. The TouchOSC values sent via smartphone are used to control the parameters of each effect. The Pi 3 B+ is used as a wifi source for the transmission of the TouchOSC values.

2.2 Software

Each of the effects in HEX's .csd is comprised of a single instrument or combination of instruments that are inserted into and removed from the global audio streams using the event_i and turnoff2 opcodes. An always-on control instrument sends these on/off commands based on footswitch statuses received from the Arduino via the Serialread opcode. An additional always-on control instrument receives parameter values from a smartphone via TouchOSC and assigns them to global k-rate variables.

This format was inspired by Iain McCurdy's patch "MultiFX.csd", with some of the effects being copied verbatim. For now, they showcase the versatile nature of the device. As one intention of creating this system has been to equip the user with an arsenal of unique effects, these copied effects are partially placeholders to be supplanted upon further optimization of the more CPU-demanding effects that are still in development.

Table 1. Hex's effects - made with Csound

Effect	Description	TouchOSC Parameters
Arpeggiator	Sequentially iterates notes in a chord upon detection of a transient	Trigger threshold Tempo Attack, Release Octave mode
Hex-Wobble	Rhythmic pitch bend	Tempo Magnitude
Pitch Shifter	Uses the delay-line UDO	Ratio Feedback
Pitch-Tracking Mono Synth	Uses pitchamdf analysis	Note Duration Waveform
Lofi	Downsamples using fold opcode	Fold amount
Filter	Bandpass filter - stacked Butterworth filters	Low cut High Cut
Reverb	Uses reverbsc opcode	Size Mix
Ringmod	Uses poscil opcode	Speed Mix

2.3 Figures

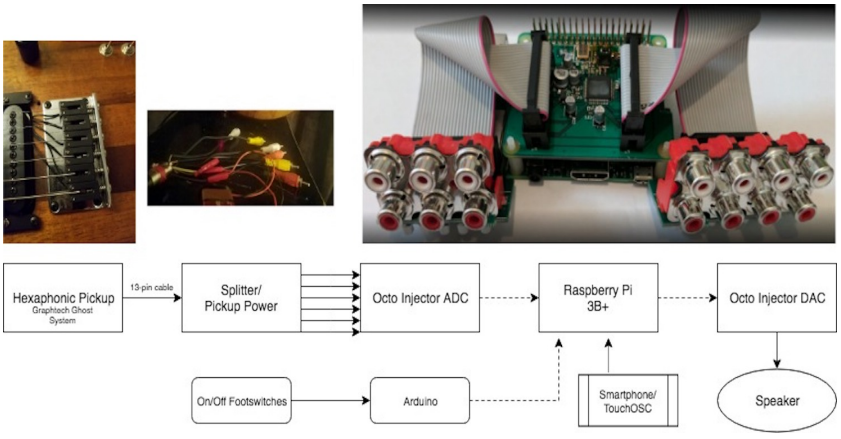


Fig. 1. Hex's components and the flow of audio and control signals..

3 Difficulties and Development

All Csound code for this project was originally written on a MacBook Pro using CsoundQt, and a couple of unanticipated difficulties arose during the process of porting the project to the Pi. There were two main issues to be reckoned with, and dealing with them has given the author a few ideas about how to improve the Hex system moving forward.

3.1 Intermittent Sound Card Detection

One challenging obstacle encountered during the build process was intermittent sound card detection. The Audio Injector Octo sound card by Flatmax was the only soundcard the author saw on the market with six channels of audio-rate input, though others may exist. Though functional, the Octo was not always detected by the Pi on boot. Other Octo users reported the same issue on the Octo's Github page[1] and a script found there resets connection to the Octo That script runs whenever Hex's Pi boots up. One additional command was added to the boot script to run Csound with the necessary RT audio module, input and output device, and buffer size.

It turned out that Portaudio Callback was the only Octo-compatible RT Audio Module; the others would either cause a crash or produce no sound or strange noises.

This is a script called "fix.sh" registered in /home/pi/etc/rc.local to run at boot time.

```
#!/bin/bash

sudo modprobe -r snd_soc_audioinjector_octo_soundcard
sudo modprobe -r snd_soc_cs42xx8_i2c
sudo modprobe -r snd_soc_cs42xx8
sudo modprobe snd_soc_cs42xx8
sudo modprobe snd_soc_cs42xx8_i2c
sudo modprobe snd_soc_audioinjector_octo_soundcard

csound -+rtaudio=pa_cb -iadc0 -odac0 -B512 -b512 /home/pi/hex.csd
```

[1] Audio Injector Octo Github support page, https://github.com/Audio-Injector/Octo/issues

3.2 CPU Limit

Some of the effects originally envisioned and prototyped on the MacBook Pro demand too much CPU to run stably on the Pi 3 B+. Further work is now required if these effects are to be successfully integrated into the Hex system.

Table 2. Effects still in development.

Effect	Description
Polyphonic Synth	Uses pitchamdf opcode to track pitch of each string on which a transient is detected
Glissando Enging	Uses pvsfreeze and pvscale to freeze chords and slide the frozen voices to notes in the next detected chord
Live Granulator	Granulates a short, destructively-recorded buffer of live input
Convolution Engine	Performs real-time convolution with pconvolve and user-loaded IR

3.3 Development

Moving forward, the most seemingly cut and dry course of action is to switch to a more powerful single board computer. The recently released Raspberry Pi 4 B may prove a timely supplicant. It boasts improved specs across the board when compared to the 3 B+, including vastly improved RAM. The Pi 4 is compatible with the Octo and the construction of a second Hex system based around a Pi 4 is well underway here at Hex HQ at the time of writing.

The Asus Tinker Board is another affordable SBC that outperforms the Pi 3 B+ in some areas. Using a Github patch that was shared on the Audio Injector Facebook page[2], and attempts were made to compile a debian kernel for the Tinker Board that could support the Octo. The kernel runs and the card is detected but does not produce sound as of yet.

4 Acknowledgements

Thanks to Dr. Richard Boulanger for sharing so much advice and guidance. Thanks to Bill Bax for sharing his knowledge of breakout cables.

[2]Octo patch for Tinker Board, https://github.com/TinkerBoard/debian_kernel/pull/37

Algorithmic Composition with Open Music and Csound: two examples

Fabio De Sanctis De Benedictis

ISSM "P. Mascagni" – Leghorn (Italy)
fabio.desanctis@consli.it

Abstract. In this paper, after a concise and not exhaustive review about GUI software related to Csound, and brief notes about Algorithmic Composition, two examples of Open Music patches will be illustrated, taken from the pre-compositive work in some author's compositions. These patches are utilized for sound generation and spatialization using Csound as synthesis engine. Very specific and thorough Csound programming examples will be not discussed here, even if automatically generated `.csd` file examples will be showed, nor will it be possible to explain in detail Open Music patches; however we retain that what will be described can stimulate the reader towards further deepening.

Keywords: Csound, Algorithmic Composition, Open Music, Electronic Music, Sound Spatialization, Music Composition

1 Introduction

As well-known, "Csound is a sound and music computing system"([1], p. 35), founded on a text file containing synthesis instructions and the score, that the software reads and transforms into sound. By its very nature Csound can be supported by other software that facilitates the writing of code, or that, provided by a graphic interface, can use Csound itself as synthesis engine.

Dave Phillips has mentioned several in his book ([2]): *hYdraJ* by Malte Steiner; *Cecilia* by Jean Piche and Alexander Burton; *Silence* by Michael Gogins; the family of HPK software, by Didiel Debril and Jean-Pierre Lemoine.[1] The book by Bianchini and Cipriani encloses Windows software for interfacing Csound, and various utilities ([6]). We can also recall score generation software like *Cmask*, *Cscore*, *Pmask*, and other ones like *AthenaCL*, *Ceres3*, *Rosegarden*, *Rain*, only to quote some, that can export their outputs in the form of a `.sco` file.[2]

[1] *Cecilia* software described by Dave Phillips is not the actual *Cecilia 5* (`http://ajaxsoundstudio.com/software/cecilia/`) developed by Olivier Bélanger. About *Cecilia 5* we send back to [3]. HPK Composer software family articulates in: *HPKC22*, *HPKComposer 3*, *HPKComposer AV*, *HPKComposerCsound* (see [4]) and *AVSynthesis* (see [5]).

[2] Some softwares can only be used in a Linux environment, other ones are no more maintained. Anyway this *excursus* accounts the widespread custom of pairing Csound to graphic interface software.

Today on Csound site we find links to *CsoundQt*, *Blue*, *Cabbage*, *WinXound*.

Algorithmic Composition and Csound

Algorithmic Composition, as discussed in other place,[3] can be differentiated into constructive and declarative approach. Constructive when the software is used to develop compositional material, declarative when instructions for a complete musical composition are furnished to the software. These definitions fit to instrumental composition, while in electronic music, particularly according to David Cope's CGS category, *Computer Generated Sound*, the boundaries are less defined, so at the same time we can fall into both constructive and declarative approach.

Also Csound owns Algorithmic Composition possibilities and can be used both in a constructive and declarative manner. Giorgio Zucco offers some interesting examples in his book ([10]).

The code underlying the performance of *Solo* by Stockhausen, developed by Francioni ([11]) can be a good example of declarative approach. However Algorithmic Composition is not the main aim of Csound functions, so the alliance with a software such as Open Music appears particularly suitable for this purpose.[4] In following section two examples drawn from my compositions will be treated.

Open Music and Csound

Csound has been used by the writer both directly programming the code, and recurring to graphic interfaces like *HPKComposerCsound* for using simple Csound instruments, but generating complex scores.

In Figure 1 an example relative to the generation of clouds of bell-like sounds in FM, opcode `fmbell`, used in *Anagrammi*, for Flute and live electronics. Thus the passage to use Open Music as interfacing tool to Csound has been a natural evolutive process. Open Music has been utilized in *Balaenoptera*, for Bass Clarinet and live electronics, and in *Fabula*, for Baritone Sax and live electronics, both quadraphonic works, for realizing electronic materials and spatialization. Below some patches related to Csound will be described.

In *Balaenoptera* Open Music has been mainly used for developing audio material and for its spatialization, by using several libraries, both Ircam and free.[5] Because it is frequent in my compositions to allude to the ambiguity between natural and artificial, Csound has been called into question by Marco Stroppa's

[3] See [7], [8] and [9]

[4] Also PWGL offers interesting examples of use of Csound, thanks to Zucco's PWCsound library, but to respect the limits of this paper it is not possible to show any example in this place.

[5] A more complete description of the use of Open Music for developing audio material in this work is in [14]

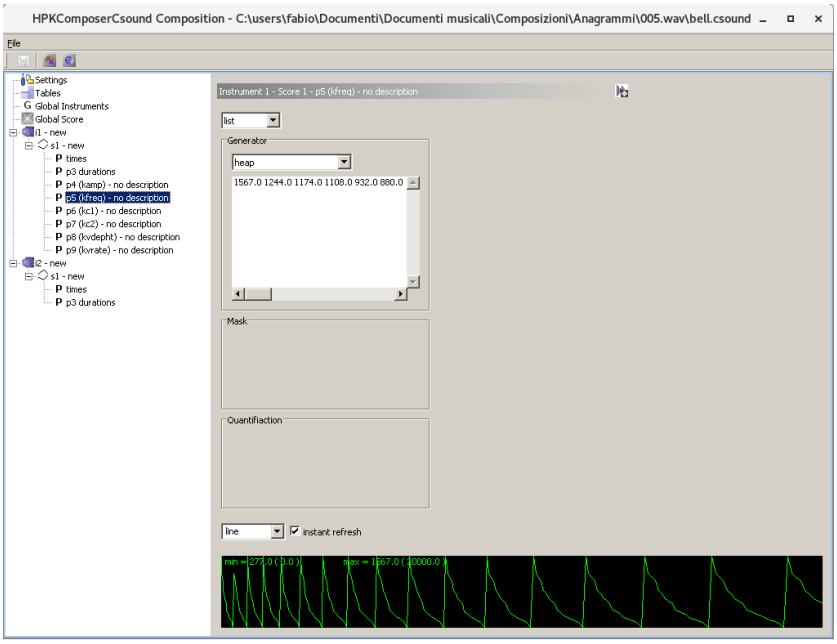


Fig. 1. *HPKComposerCsound*: selections of frequencies in an instrument based on `fmbell` opcode.

OMChroma library, creating two instruments *ad hoc* founded on `STKClarinet` opcode.[6] The used Open Music patch can be seen in Figure 2. Here a chord sequence made in Audiosculpt, created by audio analysis of a Bass Clarinet multiphonic, is used as score for Csound and rendered by both `STKClarinet` instruments.[7] Instruments parameters are chosen in random way, pitch by pitch, by `om-random` functions inside selected boundaries (1-5, 100-50, 1-12, 1-12 e 50-128).

Below the beginning of the score of the second instrument:

```
<CsoundSynthesizer>
<CsOptions>
-W -odac
</CsOptions>
<CsInstruments>
; HEADER
sr = 44100
kr = 44100
```

[6] About using Open Music and Csound for audio generation and transformation, see: [15], [16], [17], [18] and [19].

[7] About realizing personal instruments in OMChroma we refer to the very good online documentation, described in [20]

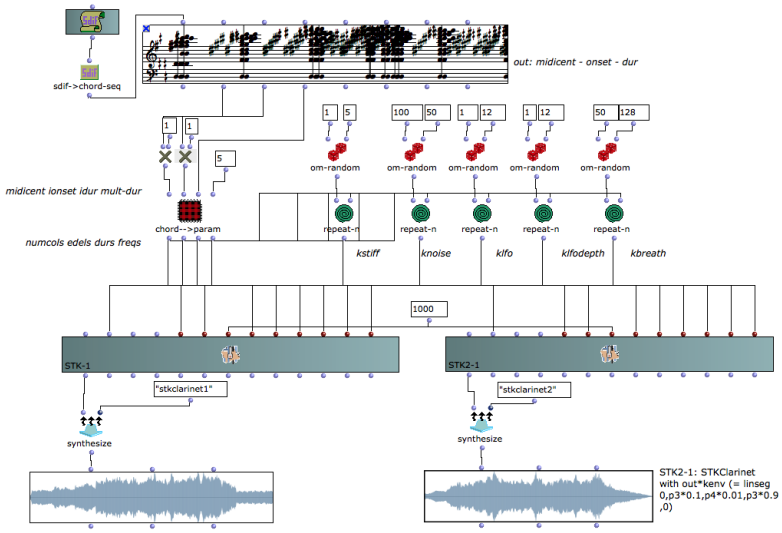


Fig. 2. Opcode STKClarinet as OMChroma class.

```
ksmps = 1
nchnls = 1
;INSTRUMENTS
;==============
instr 1
;==============
;Instrument 1 from file stk2
;asig STKClarinet ifreq, amp, kstiff, kv1, knoise, kv2,
;klfo, kv3, klfodepth, kv4, kbreath, kv5
kenv linseg 0,p3*0.1,p4*0.01,p3*0.9,0
asig STKClarinet p5,p4,2,p6,4,p7,11,p8,1,p9,128,p10
out asig*kenv
endin
</CsInstruments>
<CsScore>
;This synthesis process called my_synt started on 3 26, 2019 - AT 16:10 (24 sec)
; Global Variables: sr = 44100, kr = 44100, ksmps = 1, nchnls = 1
; Defined by chroma classes:
; Loaded tables:
; Generated tables:

;------ Lines for event n. 1 --------
i1 0.000 0.830 1000.000 148.710 5.000 66.000 9.000 1.000 78.000
i1 0.000 0.830 1000.000 299.143 1.000 66.000  12.000 10.000 84.000
i1 0.000 0.830 1000.000 745.136 4.000 100.000 4.000  6.000  91.000
```

```
i1 0.000 0.830 1000.000 886.121 3.000 81.000  8.000  9.000  62.000
i1 0.000 0.830 1000.000 148.710 1.000 58.000  4.000  5.000  77.000
i1 0.000 0.830 1000.000 298.798 2.000 72.000  12.000 11.000 90.000
i1 0.000 0.830 1000.000 745.136 2.000 100.000 7.000  6.000  60.000
i1 0.000 0.830 1000.000 885.098 5.000 73.000  5.000  6.000 119.000
i1 0.166 8.165 1000.000 149.226 3.000 53.000  7.000  5.000 116.000
i1 0.166 8.165 1000.000 297.764 5.000 71.000  3.000  9.000  62.000
```

In *Fabula* we have made a progress towards a more diffuse use of automatic sound spatialization, mainly relying on OMPrisma library by Marlon Schumacher.[8] In Figure 3 an example of a patch, effective with long sounds, that makes a very fast spatialization, acelerating or decelerating, so disintegrating or aggregating the sound.

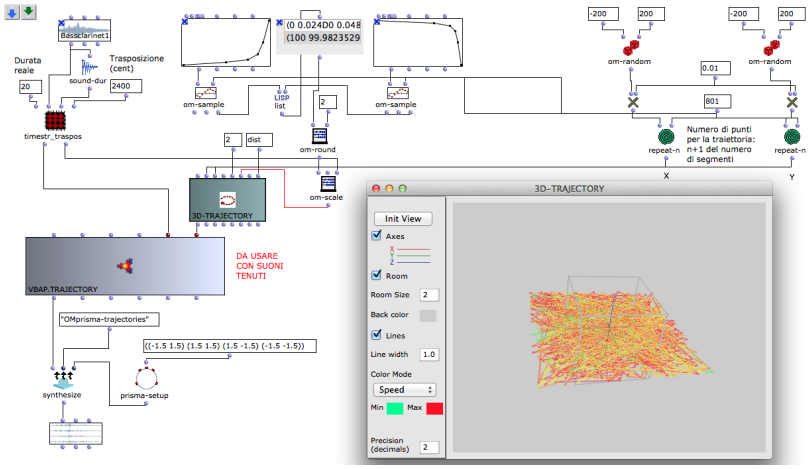


Fig. 3. Quadraphonic spatialization of an audio file acelerating or decelerating, at great velocity, causing sound disintegration or aggregation. Inside 3D-TRAJECTORY object chaotic trajectories and their velocities are visualized.

The curves on the top must be selected for choosing an accelerating or decelerating process, while X-Y coordinates of each trajectory point are random determined by `om-random` functions, in the top on the right of the patch.

Conclusion

We have seen how it is possible to use Open Music together with Csound, mainly using OMChroma and OMPrisma libraries. However that is not the unique way

[8] About using OMPrisma for spatialization and spatialized sound synthesis see: [21], [22] and [23].

to couple Open Music and Csound. Only to quote some: the possibility to export Open Music results in different formats like MIDI and XML permits to utilize very refined musical structures inside Csound; it could be possible to generate score data by algorithmic processes; the possibility of generating personal Csound instruments in OMChroma opens virtual infinite possibilities, not excluding the conversion of historical Csound instruments in Open Music classes, to be used inside the software. This is only the tip of the iceberg.

References

1. Lazzarini, V.: Computer Music Instruments. Foundations, Design and Development. Springer 2017, p. 35.
2. Phillips, D.: Linux musica e suoni. Hops Libri, Milano 2001, pp. 226-304 (or. ed. Linux Music & Sound. No Starch Press, San Francisco 2000).
3. Blanger O.: Cecilia 5, la bote outils du traitement audio-numerique. In: JIM2014 - Journes d'Informatique Musicales, 21-23 mai 2014, Bourges, France.
4. Lemoine J.P.: HPKComposer A Csound 5.0 based Audio Video Composition Tool. Csound Journal, Issue 5, January 1, 2007. Internet: `http://csoundjournal.com/issue5/HPKcomposer.html`.
5. Phillips D.: Composing With Csound In AVSynthesis. Csound Journal, Issue 10, January 19, 2009. Internet: `http://csoundjournal.com/issue10/avs-cs-composition.html`.
6. Bianchini R. and Cipriani A.: Il Suono Virtuale. Sintesi ed elaborazione del suono – Teoria e Pratica con Csound. ConTempo, Roma 2001.
7. De Sanctis De Benedictis, F.: Dall'analisi musicale alla composizione e formalizzazione algoritimica: esempi applicativi con PWGL. In: MUSICHE LIQUIDE, XX Colloquio di Informatica Musicale. Internet: `http://cim.lim.di.unimi.it/2014_CIM_XX_Atti.pdf`
8. Agon C., Assayag G and Bresson J.: The OM Composers Book. Volume One. Editions Delatour France/Ircam, Parigi 2006.
9. Bresson J., Agon C. and Assayag G.: The OM Composers Book. Volume Two. Editions Delatour France/Ircam, Parigi 2008.
10. Zucco, G.: Sintesi digitale del suono. Laboratorio pratico di Csound, Giancarlo Zedde editore, Torino 2012 (English edition: Inside Csound, Giancarlo Zedde editore, Torino 2014).
11. Francioni E.: SOLO_MV_10.1. Solo Multiversion for Stockhausen's Solo [N.19]. Csound Journal, Issue 13, January 19, 2010. Internet: `http://www.csounds.com/journal/issue13/solo_mv_10_1.html`
12. Avantaggiato, M.: Composizione assistita e processi di trasferimento di dati musicali da PWGL a Csound. In: XVIII CIM - Colloquio di Informatica Musicale, Torino-Cuneo, 58 Ottobre 2010. Internet: `http://cim.lim.di.unimi.it/2010_CIM_XVIII_Atti.pdf`.
13. Lanza M., Verlingieri G. and Biagioni N.: LA LIBRERIA OPENMUSIC om4Csound. INTRODUZIONE E PROGETTO DI DOCUMENTAZIONE. In: XVIII CIM - Colloquio di Informatica Musicale, Torino – Cuneo, 58 Ottobre 2010. Internet: `http://cim.lim.di.unimi.it/2010_CIM_XVIII_Atti.pdf`.
14. De Sanctis De Benedictis F.: Electronic sound creation in Balnoptera for bass clarinet, electronic sounds, and live electronics. In: Bresson J., Agon C. and Assayag G. (eds.) THE OM COMPOSERS BOOK. Volume 3, Editions DELATOUR FRANCE/Ircam-Centre Pompidou, Parigi 2016.

15. Bresson J., Stroppa M. and Agon C.: Symbolic Control of Sound Synthesis in Computer-Assisted Composition. In: International Computer Music Conference, 2005, Barcelona, Spain.
16. Bresson J. and Agon C.: Temporal Control over Sound Synthesis Processes. In: Sound and MusicComputing (SMC06), 2006, Marseille, France.
17. Agon C., Bresson J. and Stroppa M.: OMChroma: Compositional Control of Sound Synthesis. Computer Music Journal, 35:2, pp. 6783, Summer 2011.
18. Bresson J. and Nichoin R.: Implémentations et contrle du synthétiseur CHANT dans OpenMusic. In: Actes de Journées d'Informatique Musicale, Saint-Etienne, France, 2011.
19. Stroppa M., Lemouton S. and Agon C.: omChroma: vers une formalisation compositionnelle des processus de synthèse sonore. In: Journées d'Informatique Musicale, 9 e édition, Marseille, 29 - 31 mai 2002.
20. Richelli L.: La Libreria OpenMusic OMChroma - Documentazione online. In: Proceedings of XX CIM, Roma, October 20-22, 2014. Internet: `http://cim.lim.di.unimi.it/2014_CIM_XX_Atti.pdf`.
21. Scumacher M. and Bresson J.: Compositional Control of Periphonic Sound Spatialization: In: 2nd International Symposium on Ambisonics and Spherical Acoustics, IRCAM, Paris, May 6-7, 2010.
22. Bresson J., Agon C. and Schumacher M.: Représentation des données de contrôle pour la spatialisation dans OpenMusic. In: Journées d'Informatique Musicale, 15ème édition, Rennes, 18-20 mai 2010.
23. Schumacher M. and Bresson J.: Spatial Sound Synthesis in Computer-Aided Composition. Organised Sound 15(3): 271-289 Cambridge University Press, 2010.

An opcode implementation of a finite difference viscothermal time-domain model of a tube resonator for wind instrument simulations

Alex Hofmann[1], Sebastian Schmutzhard[2], Montserrat Pàmies-Vilà[1], Gökberk Erdoğan[3], and Vasileios Chatziioannou[1] *

[1] Dept. of Music Acoustics, University of Music and Performing Arts Vienna, Austria
[2] Acoustics Research Institute, Austrian Academy of Sciences, Vienna, Austria
[3] Dept. of Electrical & Electronics Engineering Boğaziçi University, Istanbul, Turkey
corresponding author: hofmann-alex@mdw.ac.at

Abstract. This paper presents an opcode for Csound, that is based on a physical time-domain model of a closed-open tube resonator which is capable of simulating wind instruments like clarinets or saxophones. The tube model hereby considers sound radiation parameters as well as viscothermal losses that occur inside the tube. The model was implemented in C++ using the *Csound Plugin Opcode Framework*. The `resontube` opcode allows users to provide complex geometries for the model construction in k-time together with arguments for sound radiation and pick-up position. The opcode is published together with its source code as a git repository including documentation and examples.

Keywords: csound, opcode, physical model, resonator, tube

1 Introduction

Physical modelling-based sound synthesis is an estabilished technique in the field of computer music [11] and is well supported by a large number of opcodes in Csound [7]. A majority of physical models available for Csound are based on the digital waveguide method[4] (e.g. opcodes `wgclar, wgflute, wgbow` and opcodes from the Synthesis Toolkit (STK) by Cook and Scavone [4,9]).

Digital waveguides are computationally efficient algorithms that use simple delay lines to model a wave travelling in space. When the wave hits a boundary it is reflected. Depending on the boundary condition the sound wave is damped and may change its phase. In digital waveguides, boundary conditions are modelled by inserting filters into the delay line that mimic the effect of the respective boundary. The abstraction of the complex physical phenomena that happen with

* This research was supported by the Austrian Science Fund (FWF) P28655-N32 and the"mdw Call fuer Artistic Research Projekte" by the University of Music and Performing Arts Vienna. The authors like to thank Rory Walsh and Steven Yi for their support via the Csound Slack Chat.

[4] http://www.csounds.com/manual/html/SiggenWavguide.html

musical instruments as simple delay+filter algorithms allows for computationally efficient code that can easily run in real-time on consumer computers. However, this may restrict the model complexity and the quality of the sound [6].

The general approach to physical modelling directly considers the differential equations that describe the oscillating system. These can be solved numerically, using a variety of methods [12]. The prevalent approach in the last decades is to discretise the model equations using the finite difference method [2,1]. This approach might require a large number of computations but is capable of producing more realistic sounds. As of our knowledge, there are currently only three opcodes in Csound that make use of finite differences for physical modelling of musical instruments, namely `barmodel` (model of a metal bar), `prepiano` (model of a prepared piano string), and `platerev` (model of a resonating two dimensional rectangular plate).

This paper presents a new opcode to extend Csound by a mathematical time-domain wave propagation model for a closed-open tube resonator that takes viscothermal losses into account and allows for a varying cross-sectional area, as found in wind instruments such as clarinets or saxophones [10].

2 Tube Resonator Model

This section gives a short summary of the physical tube resonator model that is the basis for the new opcode. A detailed description, including a validation of this model can be found in [10].

A tube of length L is considered, with a cross-sectional area $S(x), 0 \leq x \leq L$. The time-domain model for the dynamics of the pressure p and the particle velocity v is given by

$$\partial_x p + \rho \partial_t v + z_v * v = 0, \qquad \partial_x (Sv) + \frac{S}{\rho c^2} \partial_t p + S y_\theta * p = 0, \tag{1}$$

where * denotes a convolution with respect to time, and the functions z_v and y_θ are the time domain versions of the series impedance Z and shunt addmitance Y. The boundary conditions are given by $v(t,0) = v_{\text{in}}(t)$ at $x = 0$ and for $x = L$

$$p(t,L) = S(L) z_r * v(t,L), \tag{2}$$

where z_r is a stipulated radiation impedance. The convolutions $z_v * v$ and $y_\theta * p$ are related to the viscothermal losses along the tube. For the computational algorithm used in this opcode, we replace equation (1) by the approximation

$$\partial_x p + \rho \partial_t v + R_0 v + \sum_{k=1}^{K} w_k = 0, \qquad \partial_x (Sv) + \frac{S}{\rho c^2} \partial_t p + S \sum_{k=1}^{K} q_k = 0, \tag{3a}$$

$$\text{where} \quad w_k(t) = R_k \int_0^t e^{-L_k (t-\tau)} \partial_t v(\tau) d\tau, \quad k = 1, \ldots, K \tag{3b}$$

$$\text{and} \quad q_k(t) = G_k \int_0^t e^{-C_k (t-\tau)} \partial_t p(\tau) d\tau, \quad k = 1, \ldots, K. \tag{3c}$$

Techniques for the computation of the coefficients R_k, L_k, G_k and C_k are discussed in [10,3]. We set $K = 4$, since we observed that taking K larger than four does not audibly change the result. The boundary condition is approximated by

$$S(L)R_r\partial_t v(t,L) = L_r p(t,L) + \partial_t p(t,L), \tag{4}$$

for certain coefficients R_r and L_r, see [2]. Using finite differences to approximate the derivatives, we compute approximations p_m^n, v_m^n, $w_{k,m}^n$ and $q_{k,m}^n$ to the solutions p, v w and q of (3), respectively, at discrete points (t_n, x_m) in time and space, where $t_n = n\Delta_t$, $n = 0, 1, 2\ldots$ and $x_m = m\Delta_x$ for $m = 0, \ldots, M$ and $L = M\Delta_x$, for fixed Δ_t and Δ_x. p_m^{n+1} and v_m^{n+1} are iteratively computed from results obtained at previous time steps. The derivation of the finite difference scheme is given in [10]. The boundary condition on the left gives for v_0^{n+1}

$$v_0^{n+1} = v_{\text{in}}^{n+1}. \tag{5}$$

Equation (3a) is discretised by finite differences. We compute $v_m^{n+1}, m = 1, \ldots, M$ from

$$\begin{aligned} &\frac{p_m^n - p_{m-1}^n}{\Delta_x} + \rho\frac{v_m^{n+1} - v_m^n}{\Delta_t} + R_{0,m}v_m^{n+1} + \\ &\sum_{k=1}^{K}\left[e^{-L_{k,m}\Delta_t}w_{k,m}^n + R_{k,m}(v_m^{n+1} - v_m^n)e^{-L_{k,m}\frac{\Delta_t}{2}}\right] = 0. \end{aligned} \tag{6}$$

and p_m^{n+1}, $m = 0, \ldots, M-1$ from

$$\begin{aligned} &\frac{S_{m+1}v_{m+1}^{n+1} - S_m v_m^{n+1}}{\Delta_x} + \frac{S_m}{\rho c^2}\frac{p_m^{n+1} - p_m^n}{\Delta_t} + \\ &S_m\sum_{k=1}^{K}\left[e^{-C_{k,m}\Delta_t}q_{k,m}^n + G_{k,m}(p_m^{n+1} - p_m^n)e^{-C_{k,m}\frac{\Delta_t}{2}}\right] = 0. \end{aligned} \tag{7}$$

Taking finite differences in (4) yields for $m = M$

$$S_M R_r\frac{v_M^{n+1} - v_M^n}{\Delta_t} = L_r p_M^{n+1} + \frac{p_M^{n+1} - p_M^n}{\Delta_t}, \tag{8}$$

from which p_M^{n+1} can be computed. Finally for $k = 1, \ldots, K$ and $m = 1, \ldots, M$, $w_{k,m}^{n+1}$ and $q_{k,m}^{n+1}$ are updated by

$$w_{k,m}^{n+1} = e^{-L_{k,m}\Delta_t}w_{k,m}^n + R_{k,m}(v_m^{n+1} - v_m^n)e^{-L_{k,m}\frac{\Delta_t}{2}}, \tag{9}$$

and

$$q_{k,m}^{n+1} = e^{-C_{k,m}\Delta_t}q_{k,m}^n + G_{k,m}(p_m^{n+1} - p_m^n)e^{-C_{k,m}\frac{\Delta_t}{2}}. \tag{10}$$

3 Opcode Implementation

The tube model is implemented in C++ following the *Csound Plugin Opcode Framework* [6]. All code is made available as a public git repository [5]

The implementation of the `resontube` opcode, a tube resonator with viscothermal losses as described in Section 2, can be found in the file `resonators/resontube.cpp`. A library of functions is given in `src/tube.cpp` and constants are in `src/const.cpp`. Following, we give an overview of the implementation of the model as an opcode for Csound.

Initialisation: When the opcode is loaded (`init()`), an equispaced grid in the longitudinal direction of the tube with M grid points is instantiated. Based on the user given geometry, the cross sectional area S is calculated for each grid point (see Figure 1). The grid consists of five csound arrays (`csnd::AuxMem<MYFLT>`) in which the status of the tube is processed. The size of the arrays is initially allocated for Mmax=400 (`src/const.cpp`) so that no additional memory needs to be allocated during runtime, even when the user is changing the length of the tube. The five main arrays are `vnew` for the particle velocity ($v_m^{n+1}, m = 1, \ldots, M$ from eq.6), `pnew` for the air pressure (p_m^{n+1} from eq. 7), `S` for cross sectional area at each grid point (S_m), and `qloss` and `wloss` for the viscothermal loss related variables ($w_{k,m}^{n+1}$ and $q_{k,m}^{n+1}$ given in eq. 9 & 10). `AuxMem` iterators (e.g. `csnd::AuxMem<MYFLT>::iterator iter_pnew`) are created for each array. Computations on the arrays are only done within the range $m = 0, \ldots, M$. As a two-point scheme in time is used, the algorithm requires a memory of the previous tube state. Therefore, a copy of all grid values needs to be preserved in additional arrays (`pold, vold, qlossold, wlossold`) with the respective iterators.

In this model, two types of losses are calculated. A) radiation losses (`rad_alphaS`) at the end of the tube where the sound is radiated out of the tube, depending on the cross sectional area at the last grid point as given in eq.(4). B) viscothermal losses that apply at each grid point along the virtual tube are calculated. Factors for the convolutions in eq.(1) are prepared in the function `compute_loss_arrays()`.

Runtime: During runtime (`aperf()`), it is checked if any of the k-rate input arguments (length, geometry, see Section 4 for details) were changed by the user. If this happens, the grid has to be re-computed. This involves calculating the required number of grid points (M < Mmax), the spacing of the grid points (dx), the cross sectional area (S) at each grid point (x), as well as the losses. To maintain the wave inside the tube (pressure, velocity, losses), all new grid arrays are updated with interpolated values taken from the preserved grid arrays.

[5] `https://github.com/ketchupok/half-physler/tree/visco_pointers`. Currently, the repository holds three slightly different opcodes. Two are tube resonators without viscothermal losses, used in a different project [5], where `halfphysler_bela` is specifically optimised to run on the ultra-low latency embedded computing platform Bela [8].

In the audio-loop (`for (auto & o_sound : out_sound) {...}`), the wave propagation in the tube is computed for each time step (sample), and the audio I/Os are assigned. An input signal is given to the model as a particle velocity to the closed end of the tube (`vnew[0] = in;`). Two different audio outputs are assigned. `Out_Feedback` returns the pressure at the beginning of the tube (`pnew[0]`) prior to computing the next time step, whereas `o_sound` is returning the pressure at a variable grid point `pnew[x]`, $x < M$ after the update function `update_visco()`[6]. The function `update_visco()` updates the pressure and velocity properties according to equations (6) and (7). `Update_losses()` is updating the loss arrays following equations (9) and (10). Finally the status of the grid is copied to `pold & vold` to be preserved for the next call of `aperf()`.

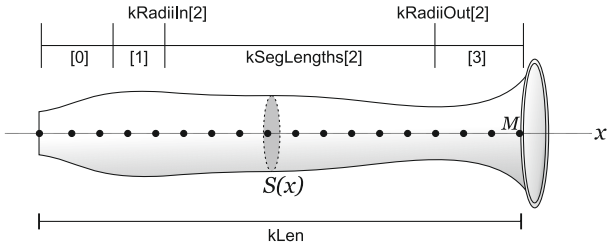


Fig. 1. Schematic of the tube model with a closed end at the left side and an open end at the right side. Geometry is given in segments as Csound arrays. Here an example with four segments from which the cross-sectional area (S) is computed for each grid point.

4 Usage

The `resontube` opcode implements a tube resonator with one closed and one open end, similar to the resonator of a clarinet or a saxophone. An overview of all user parameters of the opcode including a description of the underlying physical parameters is given in Table 1. In Csound the opcode is called by:

```
aFeedb, aSnd resontube aVelocity, kLen, kSegLengths[], kRadiiIn[],
                       kRadiiOut[], kCurveType[], [kEndReflect,
                       kDensity, kPickPos, kComputeVisco]
```

The resonator is driven by an input particle velocity (`aVelocity`), a parameter that describes the speed of the air entering the tube, for example via a single-reed instrument mouthpiece. The second input parameter `kLen` was introduced, to allow to change the resonance frequency of the resonator in a woodwind

[6] To save CPU, computation of all viscothermal losses can be turned off (`computeVisco=0`), and respectively the function `update_vp()` is called

instrument-like style. A given `kLen` in meters cuts the resonator at this point, similar to the function of opening toneholes at acoustic woodwind instruments.

The initial geometry of the entire resonator is given in segments via the Csound arrays `kSegLengths[], kRadiiIn[], kRadiiOut[], kCurveType[]`. Each segment is defined by its length, input radius, output radius and an interpolation curve type. We allow up to 25 segments. The example below gives a good approximation of a Bb-flat clarinet geometry using only 4 segments.

```
kSegLengths[] fillarray 0.0316, 0.051, .3, 0.02
kRadiiIn[] fillarray 0.0055, 0.00635, 0.0075, 0.0075
kRadiiOut[]  fillarray 0.0055, 0.0075, 0.0075, 0.0275
kCurveType[] fillarray 1, 1, 1, 2
```

Additional parameters to shape the sound by modifying the end reflection, the air density and the pick-up position along the tube are provided as k-rate inputs. An option to switch between the computation of viscothermal losses or not was added, which allows real-time playback also for complex geometries and long resonators on consumer PCs or embedded platforms like *Bela* [8].

5 Discussion

The presented opcode extends Csound by a finite difference model of a tube resonator, similar to resonators we find in clarinets or saxophones. Publishing the model for Csound allows live-electronic performers, composers or instrument makers to explore numerical modelling in an environment that handles I/O management and allows to combine physical modelling with other signal processing opcodes. Exciting experiments are possible when using Csound's capability of creating an internal feedback (ksmps=1), as shown in one of the online examples. A future version of the opcode could involve an extension to an open-open tube (flute) resonator, as well as adding details like tonehole geometry or register key modeling.

References

1. S. Bilbao. Direct simulation of reed wind instruments. *Computer Music Journal*, 33(4):43–55, 2009.
2. S. Bilbao. *Numerical sound synthesis*. John Wiley & Sons, 2009.
3. S. Bilbao and R. Harrison. Passive time-domain numerical models of viscothermal wave propagation in acoustic tubes of variable cross section. *JASA*, 140(1):728–740, 2016.
4. P. Cook. *Real Sound Synthesis for Interactive Applications*. AK Peters, 2002.
5. A. Hofmann, V. Chatziioannou, S. Schmutzhard, G. Erdoğan, and A. Mayer. The half-physler. In *Proc. NIME 2019*, page (accepted), Porto Allegre, BR, 2019.
6. V. Lazzarini. The csound plugin opcode framework. In *SMC*, pages 267–274, 2017.
7. V. Lazzarini, S. Yi, J. ffitch, J. Heintz, Ø. Brandtsegg, and I. McCurdy. *Physical Models*, pages 385–405. Springer International Publishing, Cham, 2016.

The Fifth International Csound Conference ICSC2019
Cagli (Pesaro-Urbino), Italy

Theater Academy of Cagli

Institution for the Cagli Municipal Theater

Cagli Municipality

apeSoft.it