cS ICSC2017

Proceedings of the 4th International Csound Conference

ICSC2017

Montevideo, Uruguay
Sep. 29 – Oct. 1, 2017
http://csound.github.io/icsc2017/

Proceedings of the Fourth International Csound Conference

Edited by:

Luis Jure
`lj@eum.edu.uy`

Organizers

eMe estudio de música electroacústica

In collaboration with

GPA Grupo de Procesamiento de Audio

Support

UNIVERSIDAD DE LA REPÚBLICA URUGUAY

escuela universitaria de música

FACULTAD DE INGENIERIA

Espacio Interdisciplinario
Universidad de la República

CSIC
COMISIÓN SECTORIAL DE INVESTIGACIÓN CIENTÍFICA

TEATRO SOLIS

PRESIDENCIA
República Oriental del Uruguay

Conference Chairs

Luis Jure (Chair)

Martín Rocamora (Co-Chair)

Organization Team

Jimena Arruti

Pablo Cancela

Guillermo Carter

Guzmán Calzada

Ignacio Irigaray

Lucía Chamorro

Felipe Lamolle

Juan Martín López

Gustavo Sansone

Sofía Scheps

Sessions Chairs

Pablo Cancela

Pablo Di Liscia

Michael Gogins

Joachim Heintz

Luis Jure

Iain McCurdy

Martín Rocamora

Steven Yi

Music Curator

Luis Jure

Paper Review Committee

Øyvind Brandtsegg

Pablo Di Liscia

John ffitch

Michael Gogins

Joachim Heintz

Alex Hofmann

Tarmo Johannes

Victor Lazzarini

Iain McCurdy

Rory Walsh

Music Review Committee

Pablo Cetta

Joel Chadabe

Ricardo Dal Farra

Pablo Di Liscia

Folkmar Hein

Joachim Heintz

Clara Maïda

Iain McCurdy

Flo Menezes

Daniel Oppenheim

Juan Pampin

Carmelo Saitta

Rodrigo Sigal

Clemens von Reusner

Index

Preface

The International Csound Conference (ICSC) is the principal biennial meeting for members of the Csound community and typically attracts worldwide attendance. After previous successful conferences in Hanover, Germany (2011), Boston, USA (2013), and Saint Petersburg, Russia (2015), in 2017 we had the great honour and huge responsibility of hosting the first ICSC to take place in the Southern Hemisphere. From September 29th to October 1st, Csound users and developers from around the world met in Montevideo for a series of activities that included concerts, paper sessions, keynote talks, and round tables.

The ICSC2017 was possible thanks to the financial and institutional support of the university in Uruguay, Universidad de la República, mainly through its agencies Espacio Interdisciplinario and Comisión Sectorial de Investigación Científica as well as through the School of Music and Faculty of Engineering. We also have to thank Teatro Solís and Presidencia de la República for kindly providing venues for several of our activities. Last but not least, I must express my deep gratitude to the whole team of the Estudio de Música Electroacústica–eMe and the GPA– Audio Processing Group, without whose constant support, hard work, and dedication, the Csound conference in Montevideo would not have been possible.

This document presents all of the papers that were selected for the conference by a committee of prestigious reviewers through a double–blind procedure. The program also featured five distinguished keynote speakers who were especially invited for the conference; the abstracts of their talks are also included in these proceedings. We hope that this volume will contribute to the ever–growing esteemed body of Csound literature.

Luis Jure
ICSC2017 Conference Chair

Keynote talks

Keynote Talk 1

The 60 years leading to Csound 6.09

Victor Lazzarini
Maynooth University, Ireland

Abstract

Today's Csound is but the latest link in an uninterrupted chain of development that stretches back 60 years to 1957, when Max Mathews wrote the first digital synthesis program, MUSIC I. The seeds for what we have today were sown in the early sixties with MUSIC III and IV. The basic shape for Csound was put in place later in the decade with MUSIC 360, followed by MUSIC 11 in the seventies. This talk explores the history of this software, with a close look at the main developments leading to the latest version of Csound.

Biography

Prof. Lazzarini is a graduate of the Universidade Estadual de Campinas (UNICAMP) in Brazil, where he was awarded a BMus in Composition. He completed his doctorate at the University of Nottingham, UK, where he was received the Heyman scholarship for research progress and the Hallward composition prize for one of his works, Magnificat. His interests include musical signal processing and sound synthesis; computer music languages; electroacoustic and instrumental composition.

Dr Lazzarini received the NUI New Researcher Award in 2002 and the Ireland Canada University Foundation scholarship in 2006. He currently leads the Sound and Digital Music Research Group at the NUIM and has authored over one hundred articles in peer-reviewed publications in his various specialist research areas. He is the author of Aulib, an object-oriented library for audio signal processing, and is one of the project leaders for Csound. Prof. Lazzarini has also forged links with Industry, providing consultancy and research support to Irish companies in the area of computer music.

In addition to these activities, he is active as a composer of computer and instrumental music, having won the AIC/IMRO International Composition prize in 2006. His music is regularly performed in Ireland and abroad, and has been released on CD by FarPoint Recordings.

Recent publications include "Ecologically Grounded Creative Practices in Ubiquitous Music" (*Organised Sound*, 22, 2017, with D. Keller), *Csound: A Sound and Music Computing System* (Springer, 2016, with J. ffitch, S. Yi, J. Heintz, O. Brandtsegg, and I. McCurdy), and the forthcoming *Computer Music Instruments: Foundations, Design and Applications* (Springer, 2017).

Keynote Talk 2

Don Quijote, the Island and the Golden Age
Some Experiences and Dimensions of Working "Open Source" and "Free"

Joachim Heintz

Hochschule für Musik, Theater und Medien Hannover, Germany

Abstract

I am using Csound since twenty years, being more active in its community since more than ten years. I will talk about different experiences with this collaborative work in this time, and I will try to reflect some of these aspects in a more general context: What is the situation in which we develop a software like Csound? Why do we do it? What are restrictions, where is our freedom?

Biography

Joachim Heintz studied first literature, than composition with korean composer Younghi Pagh-Paan in Bremen, Germany. Since 2004 he is head of the Electronic Studio FMSBW in the Institute for New Music Incontri at Hanover University for Music Drama and Media, responsible for teaching electronic composition. In 2016/17 he was also invited as guest professor to ICEM at Folkwang University of the Arts in Essen. Although in a way specialized in working with electronic media, his compositions are not purely electroacoustic works. He also works for instruments alone, and in particular for instruments with live electronics (e.g. "S'io non miro non moro" for soprano and electronics 2013, or "Wege" for string quartet and electronics 2017). Except for concerts, he also works for installations and performances (theatre and readings).

Since 2005 he is part of the Open Source Software movement, in particular the well-known audio programming language Csound. He hosted the first International Csound Conference 2011 in Hanover and founded the Csound FLOSS Manual which is now the standard textbook to learn Csound. He is one of the authors of the new Csound Book in Springer Publishing.

He held classes in many countries, recently in Tehran (Iran), Montevideo (Uruguay), Buenos Aires (Argentina) and Seoul (Korea). He tries to teach not only programming but to discuss questions of composition and art in general and in the field of electronic music in particular.

A list of his compositions and texts can be found at www.joachimheintz.de.

Keynote Talk 3

The ATS technique in Csound: theoretical background, present state and prospective

Oscar Pablo Di Liscia
Universidad Nacional de Quilmes, Argentina

Abstract

The ATS technique (Analysis-Transformation-Synthesis) was developed by the composer and researcher Juan Pampin (DXARTS, UW, USA). Essentially, it represents two aspects of the analyzed signal: the deterministic part and the stochastic part. This model was initially conceived by Julius Orion Smith and Xavier Serra, but ATS refines certain aspects of it, such as the inclusion of psycho-acoustic data. The deterministic part consists of sinusoidal trajectories with varying amplitude, frequency and phase. It is obtained performing high-level analysis on the spectral data obtained using Short-Time Fourier Transform analysis. The stochastic part is also termed residual, because it is achieved by subtracting the deterministic signal from the original signal. Since approximately 2001, several applications were developed by a team of academics from UNQ (Argentina) and UW Seattle (USA). These applications included stand alone programs as well as unit generators for environments like Pure Data, SuperCollider and Csound. The talk will address the theoretical background of the ATS technique, the present state of the opcodes and analysis units developed in Csound, and their future improvements.

Biography

Oscar Pablo Di Liscia is a composer and academic born in Santa Rosa (La Pampa, Argentina). Doctor in Humanities and Arts at Universidad Nacional de Rosario. Was Director of the Program in Electronic Composition of Universidad Nacional de Quilmes (UNQ, Argentina) and Research Secretary at Universidad Nacional de las Artes (UNA, Argentina). Presently, he is Professor of Computer Music and Composition in the School of Arts (EUdA) at UNQ and in the Multimedia Arts Department at UNA. He also is Director of the Research Program "Temporal Systems and Spatial Synthesis in Sonic Art" and of the Editorial collection "Music and Science" at UNQ.

He has published papers and books on aesthetics and techniques of new music and technologies, as well as developed software for Digital Signal Processing, Musical Analysis and Composition. His main areas of research are: Digital Signal Processing (specially Sound Spatialisation and Spectral Analysis of Digital Sound), Electronic Composition and Computer Music. His compositions, both electronic and instrumental were awarded by national and international societies, recorded and edited, and performed in several countries (Argentina, Chile, Uruguay, Cuba, USA, France, Spain, Chile, and Holland).

Keynote Talk 4

Csound – the Swiss Army Synthesiser

Iain McCurdy

Maynooth University, Ireland

Abstract

In the 1980s and 90s, Csound cut a pioneering path, enabling a wider range of composers and researchers to access the explore computer music techniques established by the Music N family of programs. During these early stages of its existence, Csound sat relatively unchallenged as the serious tool for computer-based sound synthesis. This talk will explore how, within out current burgeoning toolbox of computer music options, Csound has continued to redefine itself, integrate and innovate. The speaker will also describe his own journey with Csound from the late 1990s and his ongoing pursuit to exhaust its possibilities.

Biography

Iain McCurdy is a composer and lecturer originally from Belfast and currently based at Maynooth University Ireland. He has been working with, and contributing to, the Csound project since 1999. His contributions to Csound have included working on the FLOSS manual project, writing parts of the new Springer Csound Book. He has also written a catalogue of over 600 interactive examples for Csound covering many of its capabilities. This resource has proved valuable for many learning the program. Since 2015 Iain has also taken on the role of co–editor of the Csound Journal with Jim Hearon. More recent work has focussed on the popular Csound front–end, Cabbage.

Since January 2016 Iain has held the position of Lecturer in Music at the University of Maynooth (National University of Ireland) where his teaching duties cover composition, electronic music, programming and research supervision.

As a composer his work has covered the areas of acousmatic, electroacoustic, instrumental, sound installation and cross–disciplinary works involving all four. His work with sound installations and alternative controller design has drawn in exploration of electronics, sensors and instrument building.

More information about Iain is available at his website: www.iainmccurdy.org.

Keynote Talk 5

How and Why I Use Csound Today

Steven Yi

Abstract

The technology of computers and computer music has changed greatly since I first encountered Csound in 1999. Now, in 2017, 31 years after Csound was first released, I find I enjoy using Csound more than ever. In this talk, I will discuss how and why I use Csound today. I will demonstrate and share ways in which my own personal practice and approach to using Csound has evolved over time. I will then look at ways Csound could evolve and consider how those changes may impact how we see and use Csound in the future.

Biography

Steven Yi is a composer and programer. He is the author of the Blue music composition environment, author of the Pink and Score music libraries, and core developer of Csound. He has contributed work on Csound's parser and compiler, helped to develop Csound's language design, developed opcodes for Csound, and worked on moving Csound to mobile platforms (Android, iOS) and the web. He also served as the co-editor of the Csound Journal from 2005–2015, and is co-author of the book Csound. A Sound and Music Computing System, published by Springer International.

Steven is a long-time supporter of free and open source software for music. He has presented at the International Computer Music Conference, Linux Audio Conference, and Csound Conferences. In 2016, Steven received his PhD from Maynooth University for his thesis work on "Extensible Computer Music Systems."

More information about Steven is available at his website: www.kunstmusik.com.

Conference papers

Working with pch2csd – Clavia NM G2 to Csound Converter

Gleb Rogozinsky[1], Eugeny Cherny[2] and Michael Chesnokov[3],

[1] The Bonch-Bruevich St.Petersburg University of Telecommunications, Russia
[2] Åbo Akademi University, Finland
[3] JSC SEC „Nuclear Physics Research", Russia
`gleb.rogozinsky@gmail.com`

Abstract. The paper presents a detailed review on the *pch2csd* application, developed for conversion of popular Clavia Nord Modular G2 synthesizer patch format pch2 into a Csound-based metalanguage. The Nord Modular G2 was one of the most remarkable synthesizers of late 90s. A considerable number of different patches makes Nord Modular G2 to be a desirable target for software emulation. In this paper we describe the pch2csd work flow, including modeling approach, so the developer may use the paper as a starting point for further experiments. Each model of Nord Modular's unit is implemented as an User-defined Opcode. The paper gives an approach for modeling, including description of ancillary files needed for the correct work. First presented at the International Csound Conference 2015 in St. Petersburg, the pch2csd project continues to develop. Some directions for future developments and strategic plans are suggested. The example of transformation of Nord Modular G2 patch into the Csound code concludes the paper.

Keywords: Nord Modular G2, converter, metalanguage

1 Introduction

In this paper we give the report on the current state of the pch2csd project and also describe the aspects of using our application. The pch2csd was first introduced to the Csound community at the International Csound Conference 2015, St. Petersburg Russia. At that time, the authors presented the results of their first experiments on converting Clavia Nord Modular G2 (NM2) patches into Csound code and demonstrated simple Csound code, automatically created as an output of the converter [1].

The NM2 was a quite remarkable software-hardware modular synthesizer from the late 90s. Its core is built on 4 DSPs and it is programmed via GUI while being connected to the computer. Then programmed, it can run standalone [2].

The NM2 patch consists of two main parts called voice part (VA) and global effects (FX). The VA is a subject of polyphony, comparing to FX part. The

greater the number of voices played simultaneously, the lesser number of modules can be used before overrun. NM2 uses fixed sampling rate of 96 kHz for audio signals and 24 kHz for control rate signals. The overall number of modules is around 200, including numerous generators, filters, mixers, switches, MIDI units, logics, sequencers and some effects.

From the beginning the pch2csd code was written in C. At the present time we are working at translating it to Python. Rewriting the converter in Python made the code much simpler and easier to maintain, improved cross-platform support, as well as provided better distribution through PIP. Prior to our work, the NM2 patch format (pch2) had been already described by [3]. There were also several projects related to alternative graphical interfaces for NM2 [4], [5]. Their repositories also include some useful data.

2   The pch2csd Work Flow

After the program starts the user is prompted to enter the valid path to pch2 file. The program checks the consistency of the following components: Csound opcodes, mapping tables, and input-output tables. The program outputs check results as a table to the terminal window. The output csd file is then created in the program's working directory.

2.1   Modeling Approach

Each NM2 unit should have a corresponding Csound model, placed in the */pch2csd/resources/templates/modules* as a txt file with the module ID, i.e. *12.txt*. The list of module IDs can be found in *mod_type_name.json* file. The NM2 units are modeled as Csound UDOs. Csound compiler reads the code lines from top to the bottom, which is completely different approach to graphical system of patching. Fortunately, Csound has a *zak-patching* system. It provides a given number of *a-rate* and *k-rate* buses to intercommunicate between different instruments. Those two spaces are independent from each other, comparing to NM2 patching system, where all cables are sequentially numbered. Comparing to typical behavior of Csound opcodes, where each opcode typically accepts some input data as some parameters and outputs the result(s), our zak-based UDOs do not have any outputs. After parameter fields our UDOs have an IO field, in which the numbers of buses in zak-space are listed.

Thus in our system we typically have

```
SomeOpcode aP1, kP2 [, ...], kIn1 [, ...], kOut1[, ...]
```

where *aP1* and *kP2* are some module parameters, *kIn1* is a number of k- or a-rate bus to read an input data from, and *kOut1* is a number of some k- or a-rate bus to send data to. Using described approach we are able to place the opcodes in csd file in arbitrary order, just like NM2 user can.

2.2   Values Mapping

Another important aspect is the values mapping. The NM2 modules have several different controller types of a number of ranges, i.e. audio frequency range, amplitude range, normalized values in the range from 0 to 1, delay time ranges, envelope stage durations, etc. The actual values of the controllers can be seen only when using the software editor. The NM2 patch file stores 7-bit MIDI CC values without any reference to the appropriate range. Thus we had to manually copy all data types, ranges and values. The actual values can be found in *value_maps.json* file. We use specially commented mapping lines, which start with an @ symbol and list the tables to be used for each parameter of the module. I.e *112.txt* (*LevAdd* module) contains following lines:

```
;@ map s 2 LVLpos LVLlev
;@ map d BUT002
```

This means that the mapping table for the first parameter of *LevAdd* (the constant value to add to the input signal) is dependent on a second (*s 2*) parameter, which is the two-state button (table *BUT002*). The button switches *LevAdd* from unipolar to bipolar range of values. According to the choice, one of two possible mapping tables is selected (*LVLpos* or *LVLlev*). The order of NM2 parameters are fixed. Each line of a mapping file relates to the corresponding parameter of NM2 unit. The first symbol after keyword *map* marks whether the parameter needs direct (*d*) mapping, i.e. its 7-bit MIDI CC value points to the actual value in given table (i.e. *BUT002* table), or the actual parameter value depends on some selection (*s*). For the last option, there should be several possible mapping tables listed (*LVLpos LVLlev*) after the pointer to selector (*2*). Typically each module starts from one or several mapping lines. The mapping tables are stored in */pch2csd/resources/value_maps.json*.

2.3   Polymorphism Issue

It should also be noted that several NM2 modules are polymorphous, i.e. k-rate filter can turn into a-rate filter depending on a type of an input signal. Unfortunately there is no direct indication of current module type in a pch2 file, so the actual module type can be found only through analyzing of incoming cable connections. Our algorithm checks the module type, and its input connections. In case of non-default type of the input, the corresponding module twin is used instead of the default one. As it mention above, the default module UDOs are placed in Modules directory. The twins are placed in *seludoM*[1] directory under the same name.

---

[1] Reverse of the word 'Modules'

3   The Current State

We currently have 100^2 of 182 modules implemented as Csound user-defined op-codes and 36 mapping tables to map the values from the 7-bit ints to non-linear parameter range. Storing the data as the text allows anyone to change the behavior of the modules without touching the code.

Most of implemented modules still need careful checking and improvements. From the NM2 simulation side, most attention should be paid to generators. Almost every sound begins with the oscillator section, and the corresponding units should be modeled first. The NM2 oscillators produce aliased waveforms at 96 kHz. The Figure 1 shows the amplitude spectra of Clavia's sawtooth. We can clearly see the aliasing part of the spectra, mirrored from the Nyquist frequency (red line, 48 kHz). This feature of NM2 distinguishes it from the popular family of analog-modeling synthesizers, which typically produce alias-free waveforms, and makes it possible to simulate the corresponding waves by simple generation of ideal piece-wise functions using GEN7. The authors have already implement-ed most of the straight-forward modules, i.e. mixing operations, level, delay, logics and switching. Though some of them still need some careful approach for being as close as possible to their NM2 copies, i.e. *NoiseGate* and *EnvFollower*.
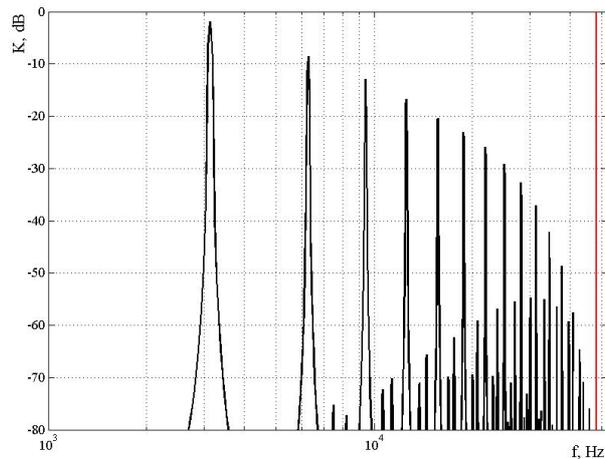
Fig. 1. Spectra of NM2 sawtooth wave.

The main problem regarding envelopes is the NM2's ability to change the envelope parameters during the run. This makes no use of Csound's typical so-lutions based on *linseg*, *expseg* or *transeg* units. Filters are rather the question of time, since their behavior should be carefully copied, i.e. through analysis of white noise filtering.

---

[2] Of which up to 20 modules so far have almost exact behavior as original NM2

Another important goal we started working on recently is to make the project hackable, so users would be able to easily modify module implementations to contribute to the project or to modify sound for their own tastes. Our next possible to-do after providing a completely working solution is an integration with some existing Clavia patch editor. It would actually establish the new Clavia-based software modular system running on a Csound core. Also, the native Csound developments, i.e. Cabbage Studio, seem very promising in the context of further integration. Current pch2csd sources can be found on the GitHub [6].

Below we give an example of code for the NM2 patch on Fig.2. The patch consists of three slightly detuned sawtooth oscillators, mixer, envelope unit, LP filter and a simple delay in FX section.
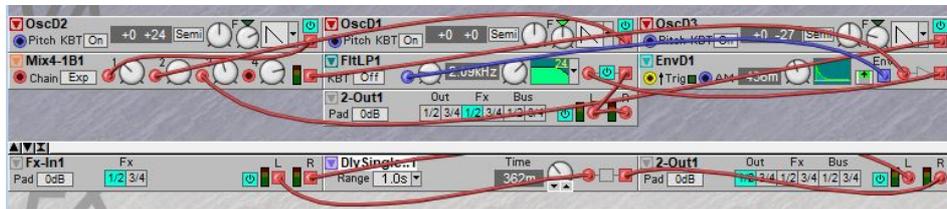
Fig. 2. NM2 patch used for testing.

Orchestra Example[3]:

```
sr = 96000
kr = 24000
nchnls = 2
0dbfs = 1

zakinit 12, 4  ; establish z-space of 12 a- and 4 k-rate buses

opcode OscD, 0, iKi  ; a simplified NM2 oscillator OscD model
    iPitch, kFine, iOut xin
    kfine = cent(kFine)
    icps = cpsmidinn(iPitch)
    aout oscili 1, icps*kfine, 1
    zaw aout, iOut
endop

opcode Mix41B, 0, kkkkiiiii  ; a NM2 4-1 mixer model
kLev1, kLev2, kLev3, kLev4, in1, in2, in3, in4, iout xin
    a1 zar in1
    a2 zar in2
    a3 zar in3
    a4 zar in4
    aout = a1*kLev1 + a2*kLev2 + a3*kLev3 + a4*kLev4
    zaw aout, iout
endop
```

---

3 Some ancillary code lines are intentionally omitted for the reasons of size

```
opcode EnvDSimple, 0, kiii ; a simplified NM2 envelope generator
   kH, iIn,iEnv,iOut xin
   aIn zar iIn
   kEnv expseg .00001,.001,1,i(kH),.00001
   zkw kEnv, iEnv
   zaw aIn*kEnv, iOut
endop

opcode FltLP, 0, ikkiii ; a simplified NM2 LP filter model
   iOrder, kMod, kCF, in1, imod, iout xin
   ain zar in1
   kmod zkr imod
   aout tonex ain, kCF+kmod*kMod, iOrder
   zaw aout, iout
endop

opcode Out2, 0, ii ; a simplified NM2 Out2 model
   iL, iR xin
   aL zar iL
   aR zar iR
   outs aL, aR
endop

opcode DlySingleA, 0, kii ; a NM2 delay unit model
   kTime, iIn, iOut xin
   ain zar iIn
   abuf delayr 1
   aout deltap kTime
   delayw ain
   zaw aout, iOut
endop

instr 1 ; this is a VA part of NM2 patch
   OscD 64, 24, 2
   OscD 64, 0, 3
   OscD 64, -27, 4
   Mix41B 0.336, 0.781, 0.430, 0.781, 2, 3, 4, 0, 5
   EnvDSimple 0.456, 5, 3, 6
   FltLP 4, 2050,2090,6,3,9
endin
instr 2 ; this is a FX part of NM2 patch
   DlySingleA 0.362, 9, 10
   Out2 9,10
endin
```

Score Example:

```
f1 0 16384 7 0 8192 1 0 -1 8192 0
f2 0 16384 10 1
i1 0 [60*60*24*7]
i2 0 [60*60*24*7]
```

References

1. Rogozinsky G., Cherny E., and Osipenko I.: Making mainstream synthesizers with csound. In: Proceedings of the 3rd International Csound Conference, pp. 132–140. St.Petersburg (2016)
2. Sound On Sound article on Clavia Nord Modular G2, http://soundonsound.com/reviews/clavia-nord-modular-g2
3. Michael Dewberry Home Page, http://www.dewb.org/g2/pch2format.html
4. NMG2 Open source editor, https://sourceforge.net/projects/nmg2editor/
5. G2Dev page, http://bverhue.nl/g2dev/?page_id=17
6. pch2csd GitHub page, https://github.com/gleb812/pch2csd

Daria: A New Framework for Composing, Rehearsing and Performing Mixed Media Music

Guillermo Senna[1] and Juan Nava Aroza[2] *

[1] Universidad Nacional de Rosario
[2] Universidad Nacional de Rosario

Abstract. In this paper we present a new modular software framework for composing, rehearsing and performing mixed media music. By combining and extending existing open-source software we were able to synchronize the playback of the free Musescore music notation editor with three VST audio effects exported using the Csound frontend Cabbage. The JACK Audio Connection Kit sound server was used to provide a common clock and a shared virtual timeline to which each component could adhere to and follow. Moreover, data contained on the musical score was used to control the relative position of specific Csound events within the aforementioned timeline. We will explain the nature of the plugins that were built and briefly identify the five new Csound opcodes that the development process required. We will also comment on a generic programming pattern that could be used to create new compatible VST audio effects and instruments. Finally, we will conclude by mentioning what other related software exists that can interact out-of-the-box with our framework, how instrument players and computer performers can simulate the performance experience while practicing their corresponding parts at home and what our future plans for this software ecosystem are.

Keywords: Mixed media music, Musescore, Csound, Cabbage, JACK.

1 Introduction

The aim of this paper is to present a new modular framework for composing, rehearsing and performing mixed media music. The pedagogical method presented by Lluán et al. [1] inspired us to build a new open-source alternative to their proposed system. By modifying and extending the free Musescore music notation editor [2], as well as by designing a series of VST effects with the Csound frontend Cabbage [3,4], we were able to build a more open and flexible substitute. The JACK Audio Connection Kit sound server [5] was used for providing a common clock and a shared virtual timeline and the software Carla [6] was chosen as our preferred VST host. As a result every component of this project is not only open-source, but also multiplatform.

---

* We would like to gratefully thank Professor Claudio Lluán for his continued support and guidance.

A short musical fragment and three new VST audio effects were created for demonstrating the capabilities of our system. Although not yet available for its first stable release, all the code that pertains to this project, including forked and modified third-party repositories, is now publicly available [7].

The synchronization between the music notation editor and Csound was achieved by developing a new Musescore plugin and a single VST (the *Conductor*) for controlling the transport part of the audio server. The two other VST audio effects mentioned earlier will serve us to showcase a common programming pattern for triggering Csound events that are required to be kept in sync with the shared timeline.

2   A Brief Overview of the Plugins

2.1   The Musescore Plugin

A new Musescore plugin was developed using the QML markup language. When called, this plugin first inspects the musical score and stores the time cues where each new measure starts. The resulting collection of values is sent to the *Conductor* using the OSC protocol.

For testing purposes, we used the musical score represented in Figure 1. The synchronization between the music notation editor and the *Conductor* plugin is considered essential for this system to work. Consequently, this communication should not be omitted while working on new musical pieces.

Fig. 1. A short musical fragment used to test our system.

A Javascript function is also included in the QML plugin for the composer and/or computer performer to parse, extract and send specific data to other

listening VSTs. In our particular scenario the QML plugin was programmed to transmit two additional arrays, both of which were constructed from information contained inside the musical score. While the first array was sent to the *Random Electronic Sounds (RES)* VST, the second one was directed towards the *Looper* VST.

As can be seen in Figure 1, we decided to include this data inside two distinct Musescore instruments. By following this approach we were able to represent the start and end time, as well as a few other parameters required by the Csound instruments, simply by using traditional music notation.

2.2   The *Conductor* VST

The *Conductor* is a VST audio effect plugin exported from Cabbage. It allows the electronic performer (or the acoustic instrument player in case of practice sessions and rehearsals) to rewind, stop, start, pause and advance the playback cursor; modify and visually keep track of a countdown timeline; be aware of the transport position of JACK in relation to the measures and beats contained in the musical score; and check the total duration of the piece as well as the time of the current playback position.


Fig. 2. The *Conductor* VST.

The construction of this VST demanded the development of two new Csound opcodes: `jackquery`, for controlling the JACK transport; and `floorarray`, for conducting a binary search among the array containing the time cues. The Cabbage source code also had to be modified in order to dynamically display bar lines and numbers inside its *image* widget [8].

2.3 The *Random Electronic Sounds (RES)* VST

The *RES* is a VST audio effect plugin that plays back randomly selected audio files while keeping them in sync with the shared clock provided by JACK. After receiving an array of numeric values from the Musescore plugin, the aforementioned files are laid out on a virtual timeline that corresponds to the rhythmic figures written on the musical score. Moreover, when the user repositions the transport through the *Conductor* (or by means of an external application) the *RES* updates its internal state to reflect the newly assumed position.

We believe the programming pattern used for achieving this kind of functionality to be relevant to composers and/or computer performers while they try to design new VSTs that are meant to be used with this framework. In consequence, we will further explain this technique in 2.4.

The Graphical User Interface (GUI) of this plugin is composed of a button to scan (or later, rescan) the folder containing the audio files; a virtual LED for indicating the status of the plugin; a bypass button for disabling the effect; a rotary knob for adjusting the pan position; a vertical slider for controlling the output level; and a virtual VU Meter to check the levels present at the output.
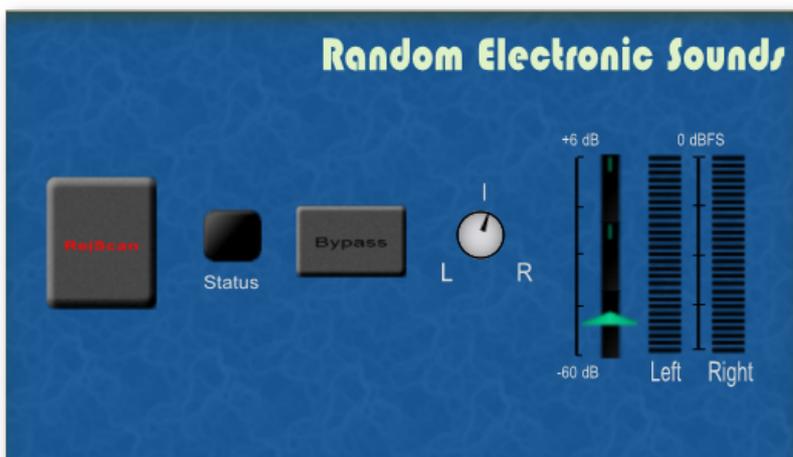
Fig. 3. The *Random Electronic Sounds (RES)* VST.

The *RES* VST required the development of three new Csound opcodes: `filewalker`, a recursive file iterator; `flnarray`, for querying the lengths of a collection of audio files; and `erandom`, for randomly choosing files depending on a list of parameters received by the plugin.

2.4   The *Looper* VST

The *Looper* is conceptually the simplest of the VSTs presented in this paper. After initialization, it expects to receive an array of time cues through an UDP port. The first pair of values triggers a recording instrument, while the remaining subsets of the array activate different instances of a same instrument that plays back what has been previously recorded.

Its GUI consists of two vertical sliders for controlling the gain of the input present at channel 1 and channel 2; a mono VU Meter for each input; two rotary sliders for modifying the pan position of each channel; a color-coded LED for checking the status of the plugin; a bypass button; a vertical slider for adjusting the level of the output; and a stereo VU Meter for controlling the levels present at the output.

Fig. 4. The *Looper* VST.

As previously stated, a particular programming pattern was used in the *Looper* -as well as it was in the *RES*- that can be further utilized while developing new plugins. By using this technique we can control the triggering process of any Csound instrument that is thought to be kept in sync with the transport.

Conceptually, the design pattern can be explained in terms of a chain of instruments that are sequentially called upon. In this chain, the first instrument -from here on the *listener*- is prepared to receive an array of values through the OSC protocol. Upon success, the *listener* calls a new instance of another instrument (the *watchdog*) for every note written on the relevant instrumental part of the musical score.

Each *watchdog* will be in charge of keeping track of the external time in terms of samples and seconds, updating the corresponding Csound control vari-

ables and channels throughout the performance. Later, whenever the instantaneous time position of the transport falls in between the correct time interval, a *performer* instrument will be triggered. It is important to note that an automatically re-triggering process will also occur whenever a repositioning of the transport has taken place.

The *performer* is the last component of our chain and also the Csound instrument in charge of handling the audio processing task we are ultimately trying to accomplish. This means that the sole purpose of the other two instruments is to relay the information coming from the notation editor and also to enforce a synchronism between the performer and the external transport, but by encapsulating each task in a different Csound instrument we ensure a design pattern that can be generically applied to any new VST effect.

3   Conclusions

The framework presented in this paper provides a novel alternative for producing mixed media music. Being modular by nature, all Cabbage effects and instruments already created by the community can be (asynchronously) used and new ones could also be developed to perform in sync with the musical score.

By using MIDI and/or audio automation tracks -both functions provided natively by Carla- the acoustic instrument player and/or the electronic performer can both individually study the musical piece, transforming a practice session at home into a full rehearsal. Moreover, software like Xjadeo [9] could be used to perform live mixed media music perfectly in sync with the playback of a video.

In the near future we expect to test this new framework in an educational environment. We hope that this kind of live experience will enable us to gather feedback and this, in turn, will subsequently help us improve the system to meet the demands of future mixed media music composers.

References

1. Lluán, C. et al.: A theoretical and aesthetics approach to the study and practice of mixed Electroacoustic Music: a pedagogical proposal. In: Electroacoustic Music Studies Network EMS 09: Heritage and Future. Buenos Aires (2009)
2. Musescore website, `https://musescore.org`
3. Walsh, R.: Developing Csound Plugins with Cabbage. In: Ways Ahead: Proceedings of the First International Csound Conference, pp. 64–82. Cambridge (2013)
4. Cabbage website, `http://cabbageaudio.com`
5. JACK Audio Connection Kit website, `http://jackaudio.org`
6. Carla Github repository, `https://github.com/falkTX/Carla`
7. Daria Github repository, `https://github.com/gsenna/Daria`
8. Cabbage Docs: the image widget, `http://cabbageaudio.com/docs/image`
9. Xjadeo website, `http://xjadeo.sourceforge.net`

Interactive Csound coding with Emacs

Hlöðver Sigurðsson

Abstract. This paper will cover the features of the Emacs package *csound-mode*, a new major-mode for Csound coding. The package is in most part a typical emacs major mode where indentation rules, completions, docstrings and syntax highlighting are provided.
With an extra feature of a REPL[1], that is based on running csound instance through the csound-api. Similar to csound-repl.vim[1] csound-mode strives to enable the Csound user a faster feedback loop by offering a REPL instance inside of the text editor. Making the gap between development and the final output reachable within a real-time interaction.

1   Introduction

After reading the changelog of Emacs 25.1[2] I discovered a new Emacs feature of dynamic modules, enabling the possibility of *Foreign Function Interface(FFI)*. Being insired by Gogins's recent Common Lisp FFI for the CsoundAPI[3], I decided to use this new feature and develop an FFI for Csound. I made the dynamic module which ports the greater part of the C-lang's csound-api and wrote some of Steven Yi's csound-api examples for Elisp, which can be found on the Gihub page for CsoundAPI-emacsLisp[4].

This sparked my idea of creating a new REPL based Csound major mode for Emacs. As a composer using Csound, I feel the need to be close to that which I'm composing at any given moment. From previous Csound front-end tools I've used, the time between writing a Csound statement and hearing its output has been for me a too long process of mouseclicking and/or changing windows. I needed something to enhance my Csound composition experience and I decided to develop a new Csound major mode for Emacs, called simply *csound-mode*.

Other Emacs packages already exist for Csound, one called *csound-x*[5] and an older one from John ffitch[6]. Both of them are based on dual mode between orchestra and score, *csound-x* takes the dual mode setup presented in John's Emacs packages to a more complete level. That which differentiates *csound-mode* from *csound-x* in terms of user experience is that *csound-mode* does not attempt to separate a csound document(csd) in two modes and seeks to keep the user in one buffer[2]. *csound-x* also makes Csound very configurable where paths and options can be configured separately from global Csound options, *csound-mode* tries to keep all Emacs configuration at minimum and respects all system

---

[1] REPL stands for read-eval-print-loop and is a term that is used in many programming languages that offer a language interpreter in a shell, like which can be found in Python, Node and Clojure.

[2] A buffer is an Emacs lingo for a window, as you can have multiple tabs open in many text editors you can have multiple buffers running inside Emacs.

paths and global Csound configuration should one exist. Furthermore *csound-x* or its dependencies are not compatible as of today with Emacs version 25 or newer whereas *csound-mode* is not compatible with Emacs version 24.5 or older.

I hope this tool will be as useful for other composers as it is for me. In the following sections I will go further into the mechanics of *csound-mode*.

2   Csound coding style

No official community driven Csound style guide exists, which makes Csound coding style today rather unstandardized. When deciding on indentation rules for *csound-mode* I had to make few opinionated decisions, based on common best practices. Through informal observation of Csound code from various Csound users throughout the years, it can be noticed that some coding tendencies are fading away. For example indented **instr** statements are very rare in modern Csound code.

```
          instr 2
a1        oscil     p4, p5, 1       ; p4=amp
          out       a1              ; p5=freq
          endin
```

Fig. 1. Richard Boulanger's toot2.orc

Rather in most cases, **instr** and **endin** statements are set to the beginning of line with its body indented to right.

```
instr   1
  aenv      linseg    1,p3-.05,1,.05,0,.01,0
  a1        oscili    p4, 333, 1
            outs      a1*aenv,a1*aenv
endin
```

Fig. 2. John ffitch's beast1.orc

What makes these two figures clear and easy to read, is the fact that a visual distinction is given between the return value, operators/opcodes, input parameters and optional comments at the end. While this holds true in most cases, with the introduction of indented boolean blocks, this can quickly get messy.

```
   opcode envelope, a, iiii
 iatt, idec, isus, irel  xin
   xtratim   irel
 krel     release
if (krel == 1) kgoto rel
 aenv1     linseg   0, iatt, 1, idec, isus
 aenv    =  aenv1
   kgoto    done
rel:
 aenv2     linseg   1, irel, 0
 aenv    =  aenv1 * aenv2
done:
   xout      aenv
   endop
```

Fig. 3. Jonathan Murphy's UDO envelope.udo

Despite its old fashioned indentation in **Fig 3**, it can be seen that a code with boolean blocks does not align well with code that is otherwise trying to form vertical blocks based on outputs and operators. Therefore, as a matter of opinionated taste, code that forms boolean blocks should be visually aligned to one another, and for each new depth of nesting, an equal width of indentation should be added. The following figure displays indentation pattern which follows the default *csound-mode* indentation.

```
opcode gatesig, a, ak
  atrig, khold xin
  kcount init 0
  asig init 0
  kndx = 0
  kholdsamps = khold * sr
  while (kndx < ksmps) do
    if(atrig[kndx] == 1) then
      kcount = 0
    endif
    asig[kndx] = (kcount < kholdsamps) ? 1 : 0
    kndx += 1
    kcount += 1
  od
  xout asig
endop
```

Fig. 4. Steven Yi's UDO gatesig.udo

In *csound-mode* the indentation width defaults to 2 spaces, this width is adjustable with a customizeable global variable in Emacs under *csound-indentation-spaces*. Single line indentation is in Emacs by default bound to the `<TAB>` key, like with most in Emacs, this is highly customizeable. *csound-mode* happens to work very well with the minor-mode *aggressive-indent-mode*[7], which automates indentation based on buffer's current active major-mode indentation rules and can in turn saves many keystrokes.

The score section is simpler and more straightforward in terms of indentation as a score statement is separated with newlines. If we look at spreadsheet applications like *Microsoft Excel* we see why they are useful for composing Csound scores. They provide resizable vertical and horizontal blocks, giving a clear distinction between each parameter field. *csound-mode* comes with the function *csound-score-align-block* that indents sequence of score events, separated by newline. As will be mentioned later, *csound-mode* treats series of score events seperated by newline as a score-block unit. The function *csound-score-align-block* is as of current version bound to `<C-c C-s>`, by applying this function with the cursor located anywhere within the score-block, all common score parameters will adjusted to the same width.

```
i 1 0 100 440 0.000001
i 1 100 100 850 10
i 1 2000 1 60 1|

;; C-c C-s
i 1 0     100 440 0.000001
i 1 100   100 850 10
i 1 2000 1   60  1|
```

Fig. 5. Score-block transformation in csound-mode where the pipe character represents cursor location.

By setting the variable *csound-indentation-aggressive-score* to true, the function *csound-score-align-block* will be called on every indentation command. This function was designed function with *aggressive-indentation-mode*, but for very long score blocks it may cost a lot of CPU.

In conclusion, the way which *csound-mode* indents Csound code is based on the depth of nesting of boolean statements within the orchestra part or file of given Csound code. And for the score part or file, no statement will be indented to right but a possibility of aligning blocks of score statements is available to the user.

3   Syntax highlighting

csound-mode utilizes the built in *font-lock-mode* to provide syntax highlighting. Most colors can be modified via `M-x customize-face` if wished. For example if

the user wishes to change the color of i-rate variables, it would be found under *csound-font-lock-i-rate* and global i-rate variables under *csound-font-lock-global-i-rate* etc.

By default *csound-mode* comes with enabled rainbow delimited parameter fields, meaning each parameter within a score statement will get a unique color, up to 8 different colors before they repeat. This can potentially give visual aid to scores statements that have many parameters. If wanted, this feature can be turned off with the customizeable variable *csound-rainbow-score-parameters-p*.

4   Completions and documentation

Like most Emacs major-mode packages, *csound-mode* will also provide *eldoc-mode* functionality and autocomplete, where autocomplete can be *ac-mode* or the more modern version of it, *company-mode*. All the data for completions and *eldoc-mode* is based on crude xml parsing of the Csound Manual, meaning that some opcodes and symbols could be missing. The documentation data is stored in a hash-map and is static, meaning that global symbols from UDOs (user defined opcodes) evaluated into the Csound instance, do not enjoy the benefits of *font-lock* syntax highlighting or *company-mode* completions.

The echo-buffer is where the docstrings from the autocomplete suggestions and argument names are printed into. This is where *eldoc-mode* plays a big role in *csound-mode*. While user is typing in values for a given opcode's parameter, *eldoc* will highlight the current argument at the point of the cursor. This works on multiple lines for opcodes and operators, as well as for nested (functional-style) opcodes calls, which are given argument values between parenthesis.

5   Csound interaction

Before talking about the Csound REPL, it's worth mentioning the two "offline" possibilites, namely the functions *csound-play* bound to `<C-c C-p>` and *csound-render* bound to `<C-c C-r>`. Running those functions pops up compilation buffer that gives user all logs printed to stdout and stderror from the command. Calling *csound-play* is the same as if `csound -odac file.csd` would be typed into the command line while *csound-render* would equal to `csound -o filename.wav filename.csd` where *csound-mode* will prompt the user for a filename before rendering.

When starting Csound REPL via *csound-start-repl* a new buffer called ***Csound REPL*** will open. This buffer is running on *CsoundInteractive* for major-mode and *comint-mode* as minor mode. *comint-mode* provides a prompt functionality, enabling evaluation of the user input into the prompt, as well as storing history of the commands given to the prompt. As of current version, score statements are the only commands that the prompt understands, this is expected to be extended in future versions.

With an open REPL buffer, a Csound instance is running in the background with

indefinite performance time. Through this Csound instance, live-coding and/or live-interaction can take place, trough two different functions bounded to keys. Which are *csound-evaluate-region* bound to `<C-M-x>` and *csound-evaluate-line* bound to `<C-x C-e>`.

csound-evaluate-region is more dynamic of the two, it tries to recognize a Csound score or orchestra statement at the point of cursor, which may or may not cover multiple lines. After a Csound statement is recognized, the string is sent to forementioned Csound instance to be evaluated, followed by a "special effect" where the code that was recognized gets highlighted for a sub-second. The result of this operation should get printed immediately into the REPL buffer, where Csound may print syntax errors if one were found. *csound-evaluate-line* does the same, but will only evaluate the current line of the cursor. *csound-evaluate-line* could be more convenient to run within score blocks, as *csound-evaluate-region* will send all the score statements separated by newline into the Csound instance. As a use-case example, a composer has with these functions the possibility to compose short phrases/bars at a time, separated by newlines and have Csound play immediately the evaluated phrase, as well as evaluating one single score line to hear how one note sounds within a block of notes.

Note that *csound-mode* has a built-in transformation of the score-blocks that are being evaluated. Trough this transformation the lowest value of p2 in the score-block is subtracted from all p2 and p3 fields. This eliminates all waiting time for scores-blocks that have long starting time (high p2 value), but could in some cases be unwanted, in which case playing the file "offline" via *csound-play* could be more suitable option.

6   Conclusion

csound-mode is new and growing package that provides set of functionalities aimed to enhance the flow of the composer. *csound-mode* is developed by a seasoned Emacs user for Emacs users, which may set a barrier for potential new users of *csound-mode* without prior knowledge of the text editor Emacs. Unlike CsoundQt, *csound-mode* does not come with Csound Manual lookup functionality and provides only short documentation snippets via autocomplete and *eldoc-mode*. *csound-mode* is also the only Emacs package for Csound that is now available on MELPA, which is the largest package manager for Emacs packages. Being a new package, it's not battle tested and has potentially bugs, which is why I think it's important that *csound-mode* has a github page where future users can report bugs, suggest improvements and submit pull-requests. Something that other Emacs packages for Csound have lacked up to this point.

References

1. Steven Yi. csound-repl.vim. `https://github.com/kunstmusik/csound-repl`, 2016.
2. Mickey Petersen. What's new in emacs 25.1. `https://www.masteringemacs.org/article/whats-new-in-emacs-25-1`, 2016.

3. Michael Gogins. Steel bank common lisp ffi interface to csound.h. `https://github.com/csound/csound/blob/7b4cebf8cf0a3f49232123ee3d752db2116c4c6c/interfaces/sb-csound.lisp`, 2017.

4. Hlöðver Sigurðsson. Emacslisp link to csound's api via emacs modules. `https://github.com/hlolli/csoundAPI_emacsLisp`, 2017.

5. Stéphane Rollandin. Csound-x for emacs. `http://www.zogotounga.net/comp/csoundx.html`, 2015.

6. John ffitch. Emacs-macros. `http://www.cs.bath.ac.uk/pub/dream/utilities/Emacs-macros/`, 2003.

7. Artur Malabarbara et al:. aggressive-indent-mode. `https://github.com/Malabarba/aggressive-indent-mode`, 2017.

Chunking: A new Approach to Algorithmic Composition of Rhythm and Metre for Csound

Georg Boenn

University of Lethbridge, Faculty of Fine Arts, Music Department
`georg.boenn@uleth.ca`

Abstract. A new concept for generating non-isochronous musical metres is introduced, which produces complete rhythmic sequences on the basis of integer partitions and combinatorics. It was realized as a command-line tool called *chunking*, written in C++ and published under the GPL licence. *Chunking*[1] produces scores for Csound[2] and standard notation output using Lilypond[3]. A new shorthand notation for rhythm is presented as intermediate data that can be sent to different backends. The algorithm uses a musical hierarchy of sentences, phrases, patterns and rhythmic chunks. The design of the algorithms was influenced by recent studies in music phenomenology, and makes references to psychology and cognition as well.

Keywords: Rhythm, NI-Metre, Musical Sentence, Algorithmic Composition, Symmetry, Csound Score Generators.

1 Introduction

There are a large number of powerful tools that enable algorithmic composition with *Csound*: *CsoundAC*[8], *blue*[11], or *Common Music*[9], for example. For a good overview about the subject, the reader is also referred to the book by Nierhaus[7]. This paper focuses on a specific algorithm written in C++ to produce musical sentences and to generate input files for *Csound* and *lilypond*.

Non-isochronous metres (NI metres) are metres that have different beat lengths, which alternate in very specific patterns. They have been analyzed by London[5] who defines them with a series of well-formedness rules. With the software presented in this paper (from now on called *chunking*) it is possible to generate NI metres. It is even possible to construct sequences of changing NI metres, which are held together by an over-arching musical concept, namely tension and release. Thakar[10] coined the terms 'impact and resolution' for he is concerned with the phenomenological dynamics of musical energy. In his theory, several factors can contribute to the build-up and decline of musical energy; contrast between rhythmic values is one of them. *Chunking* was designed specifically to implement the directed impact and release of musical energy in rhythm,

---

[1] `https://github.com/gboenn/chunking`
[2] `www.csounds.com`
[3] `www.lilypond.org`

thereby enabling the composition of musically interesting rhythmic sentences. Further inspiration was drawn from the analysis of Eastern-European rhythmics, notably the Aksak[4] rhythms that were studied by Arom[1] and Brăiloiu[2], for example. The theory of Aksak offers prime examples of NI metres. An Aksak metre can be described as a multi-set of beats that are either 2 or 3 pulses long, for example $\{2, 2, 2, 3\}$.

Miller[6] made an important discovery in the field of cognitive psychology by finding a 'magic number' that reappeared again and again in various experiments to test the capacity of human cognition and memory. The number 7 ± 2 seems to accurately describe how many items can be held in short-term memory. This number can be increased by a technique called 'chunking'. If a long list of items is cognitively separated into groups (chunks) it will be easier for the mind to grasp, memorize and to reproduce the entire list correctly. Gobet[3] and others have expanded Miller's theory. Their model includes a template mechanism, which allows for slots with variable content in addition to a core of chunks. A chess position, for example, is memorized by building chunks of pieces in the mind, which group together via proximity. There is a certain number of fixed chunks (groups of chess pieces), but also slots which may contain a different variation of a chunk. In this case only a pointer is needed that will lead to the right chunk in long term memory. Chunks and templates facilitate information processing in our minds. The algorithm we describe in this paper makes use of both concepts for the creative process of composing rhythmic phrases.

2   The Sentence Algorithm

Chunking uses a top-down approach for generating rhythmically interesting musical sentences. The software is working on the assumption that musical rhythms are often based on a small underlying pulsation. There are two main input-arguments for chunking: First, the number of pulses n ranging between 1 and 120. Secondly, the number of distinct parts k can range between 1 and 5. The output of chunking consists of a series of recursive partitions of n into k parts. A partition of an integer n is simply any sum of integers $< n$ resulting in n, for example $16 = 7 + 5 + 4$, which is a partition into three distinct parts ($k = 3$). For the reminder of the paper, partitions will be written as (multi-)sets of integers with subscript n.[5] Chunking creates a single musical sentence of length n by partitioning n into k phrases. Phrases are further partitioned into at most k patterns, which finally partition into chunks of 2s and 3s. The proposed hierarchy of all musical components is: $Sentence, Phrases, Patterns, Chunks$. The algorithm is designed to make sure that the lengths of the components on each level are always co-prime. For example, $Sentence\{19\}, Phrases\{10, 9\}_{19}, Patterns\{\{7, 3\}_{10}, \{4, 5\}_9\}_{19}$, $Chunks\{\{\{3, 2, 2\}_7, \{3\}_3\}_{10}, \{\{2, 2\}_4, \{3, 2\}_5\}_9\}_{19}$. After experimentation it was found that the co-primality between individual parts of a rhythmic sentence is a condition, which yields the best musical results amongst partitions because of

---

[4] The term Aksak is Turkish and means 'limping'

[5] For example, $\{7, 5, 4\}_{16}$ is equivalent to writing $16 = 7 + 5 + 4$.

the high amount of rhythmic impulse and resolution (tension and release) that can be generated. With the same reasoning, the algorithm searches for the partition whose distribution of distinct parts is as close as possible. This is achieved by making sure that the standard deviation, σ, is as small as possible. If all distinct parts are in an arithmetic progression with $\Delta = 1$, for example: $\{7, 6, 5\}_{18}$, then $\sigma \approx 0.816497$ reaches its smallest value amongst all partitions of 18 with 3 distinct parts (of course, for other n, Δ and σ can be very different). In this little example, all conditions for searching a partition have been met: Co-primality of distinct parts, and the lengths of the parts are closest together. Partitions like these are the basis for the unique phrase and pattern lengths within the sentences.

Because the parts in all partitions are ordered from high to low integers, *chunking* reorders them into a musical triangle. A musical triangle is a special order of values where the largest one is in a central position, and the smaller values are leading up towards the central peak, and descending thereafter. For example, $\{1, 2, 4, 3\}$, or $\{2, 3, 4, 6, 5, 1\}$. There are fourteen musical triangles of this kind for the reordering of all partitions with three to seven distinct parts. Partitions with only two parts are being swapped randomly.

The next iteration uses these re-ordered partitions and constructs the lower structural level: patterns *within* each of the phrases. A new partition with distinct parts is generated, one that divides up each phrase length, thus generating a pattern. Subsequently, for each of the pattern, the re-ordering process applies again one of the musical triangles.

The final step towards the complete sentence is to take each of the patterns and to partition them into chunks of two or three pulses in length. 2 and 3 are the only integers allowed on this level of partitioning. If, for example, a pattern has the length 7, then the only partition into 2s and 3s is $\{3, 3, 2\}_7$.[6] Larger n can produce many different partitions of this kind. In order to distinguish these partitions from the ones we have discussed so far, we call them *templates*, where 2s and 3s are the only *chunks* accepted. In the next step, one can *rotate* the templates in a cyclic manner, i.e. $\{3, 3, 3, 2\}_{11}$, $\{3, 3, 2, 3\}_{11}$, $\{3, 2, 3, 3\}_{11}$, and $\{2, 3, 3, 3\}_{11}$ are all rotations of the template $\{3, 3, 3, 2\}_{11}$. A template produces a unique set of combinations. Each one of the combinations has its consequences on the perception of the rhythm in a musical context, and it falls therefore into a special musical category.

We have reached the lowest level of the hierarchy, from a sentence to phrases, to patterns, to chunks. Patterns and phrases are organized according to the sets of musical triangles, like smaller waves leading to a larger one. With regard to the *templates*, i.e. partitions into 2s and 3s, *chunking* uses several categories of combinations that are useful for music composition. It is an interesting fact that the *templates* are equivalent to a special mathematical structure called *bracelets*[4].[7] A bracelet is a cyclic pattern with a special order, and although it

---

[6] For a partition, the order of the terms is irrelevant.

[7] To form a bracelet, the template itself has to be in its lexicographically lowest rotation.

can be rotated and also reversed, it would never be equal to any other bracelet (or template) that has the same sum of its elements, including under rotations and reverse directions. In general, every partition of n into 2s and 3s, and after rotation into its lowest lexicographic position, generates a bracelet with a fixed content.[8] In order to compose music using the large amount of possible rhythmic patterns that emerge from templates, they had to be sorted into seven different musical categories. The names of these categories reflect the musical forces of impulse and resolution that govern them and that are a direct consequence of the specific order of 2s and 3s.

2.1   Seven categories of rhythmic patterns

The seven categories of rhythmic patterns are *resistor, release, arch, catenary, growth, decline,* and *alternating.* The *resistor* pattern consists of one or more 2s followed by one or more 3s. For example, here is a *resistor* based upon a partition of 12: $\{2, 2, 2, 3, 3\}_{12}$. The name reflects the phenomenon that the transition from a pulsation of 2s into a pulsation of 3s forms a resistance against the flow of the binary pulsation.[9] We experience a fundamental change from binary to ternary pulsation. This phenomenon creates musical impulse, a force that is balanced at a later stage by a corresponding resolution.

The *release* pattern is the opposite form of the above: One or more 3s are followed by one or more 2s, for example $\{3, 2, 2, 2\}_9$, which is derived from a partition of 9. If one combines the *resistor* patterns with a *release* pattern, you obtain the *arch* pattern. This is a natural consequence of the fact that a musical impulse is counter-balanced by a force of release, similar to the situation where the force of pulling a spring out of its equilibrium position is opposed by the spring-force, which will return the system back to its original state as soon as the spring has been released. We call this configuration *arch*, because one or more 2s are surrounding one or more 3s. Example: $\{2, 2, 3, 3, 3, 2, 2, 2\}_{19}$ As it is the case with all the other patterns, the length of an *arch* can vary.

The shape opposite to the *arch* is the *catenary*[10], which is a combination of the *release* pattern followed by the *resistor* pattern. It has one or more 3s surrounding one or more 2s, for example: $\{3, 3, 2, 2, 2, 3\}_{15}$.

A *growth* pattern has, for example, this structure: $\{2, 3, 2, 2, 3, 2, 2, 2, 3\}_{21}$. Here, an initially small resistor pattern, $\{2, 3\}_5$, grows by inserting more and more 2s in front of the 3 at each repetition. There are many possible variations of this pattern, for example, one could let only the 3s grow: $\{2, 3, 2, 3, 3, 2, 3, 3, 3\}_{24}$. Or, one could develop both at the same time: $\{2, 3, 2, 2, 3, 3\}_{15}$. The maximum length for a growth pattern in the top-down approach is twenty-five pulses with

---

[8] An efficient algorithm for generating bracelets has been published by Karim et al.[4].

[9] Inspiration for the name came from the field of Electronics with regard to the resistance against the flow of electric charges.

[10] "In physics and geometry, a catenary is the curve that an idealized hanging chain or cable assumes under its own weight when supported only at its ends." (see fx-solver.com)

a maximum of nine elements. This is a consequence of applying Miller's 'magic number'. It is easy to see that one could chain resistor patterns together in order to form a phrase or an entire sentence as one large *growth* structure.

The *decline* pattern is like the *growth* pattern, only the direction is reversed. We start with a relatively long pattern, and recursively subtract elements from it. Example: $\{2, 3, 3, 3, 2, 3, 3, 2, 3\}_{24}$. Finally, the *alternating* pattern consists of repeating a smaller pattern of 2s and 3s, for example $\{3, 2, 2, 3, 2, 2\}_{14}$.

2.2 Shorthand notation for rhythm as an intermediate output format

On the basis of the resulting 2-3 patterns discussed above, a technique is now presented for breaking down the binary and ternary chunks into specific rhythmic chunks. *Chunking* uses all possibilities of beats within a grid of either 2 and 3 pulses. The complete list of binary and ternary rhythm chunks is shown in table 1 along with binary and shorthand representations. For each 2 and 3 in a pattern, a weighted random choice is being made from the list of binary or ternary chunks. The probability weighting of the chunks is a matter of personal choice. After experimentation it was found that the probability of chunks that start with a rest had to be reduced, otherwise there would have been too much dissociation between the rhythmic patterns.

Table 1. Set of symbols for rhythmic chunks.

Symbol	Transcription	Binary Form	Category
.	♪	1	unary
I	♩	10	binary
:	♪♪	11	
v	𝄾♪	01	
X	♪♩	110	ternary
>	♩ ♪	101	
<	𝄾♩	010	
w	𝄾𝄾♪	001	
+	𝄾♪♪	011	
i	♪♪♪	111	
—	♩.	100	
~	e.g. I ~ I = ♩	1000	tie
( )	e.g. (X) I = 𝄾.♩	00010	silence

The shorthand notation is a useful intermediate output of *chunking*. The notation is human readable and can be sent as a C++ string to an appropriate translator object that decodes the notation and generates automatically code

for csound score events listed under the score section of a csd file. The csound #include directive is very useful here. The translator is also capable of generating a lilypond file, for which we can give a practical example in figure 1. It has the following structure: $Sentence\{37\}, Phrases\{19, 18\}, Patterns\{\{10, 9\}, \{7, 11\}\},$ $Chunks\{\{\{3, 3, 2, 2\}, \{2, 2, 2, 3\}\}, \{\{2, 2, 3\}, \{2, 2, 3, 2, 2\}\}\}\}$

Fig. 1. A sentence with 37 pulses generated with *chunking* and rendered with *lilypond*. The conductor signs show the metric grouping in beats that are 2 or 3 pulses long.

3   The Csound score output

In order to add a musical performance quality to the synthesis, *chunking* adds phrase envelopes and an overarching sentence envelope. Phrase envelopes modify the overall amplitude of a phrase, and are calculated to reach a climax at the beginning of the second half of the phrase. The *expseg* envelope is used in the orchestra code.[11] The sentence envelope modifies the overall amplitude of the sentence and uses the *cosseg* opcode. Its climax is reached at the point when the first half of the phrases has just ended (rounded up for odd numbers of phrases). For example, if the sentence has five phrases, the envelope reaches its climax at the beginning of the fourth phrase. The time parameters for the envelopes are: total length, ascending, and descending phrase lengths in seconds. In order to control independent phrase and sentence envelopes, their effect is programmed into separate instruments, away from the sound generating source. This modular approach uses *Csound*'s global audio variables.

4   Conclusion

Chunking's algorithm creates rhythmic sentences that subdivide into phrases, patterns of binary and ternary chunks, and it generates the specific composition of rhythms based on these chunks. The algorithm uses a top-down approach based on partitions, bracelets and seven musical categories of patterns. It adheres to Miller's magic number on multiple levels and it uses the concept of chunks and templates for musical variation. It further generates musical phrasing envelopes for Csound synthesis. The output in a general notation format helps to navigate choices during the composition process. That process does not stop with the output of *chunking*, but uses its material to compose entire pieces of music.

---

[11] Example *csd* files are provided with the source code.

References

1. Arom, S.: L'aksak: Principes et typologie. Cahiers de Musiques Traditionnelles 17 (Formes musicales): 11–48 (2004)
2. Brăiloiu, C.: Le rythme Aksak. Revue de Musicologie 33(99 and 100) (December), 71–108 (1951)
3. Gobet, F. and Simon, H. A.: Templates in Chess Memory: A Mechanism for Recalling Several Boards. Cognitive Psychology 31(1), 1–40 (1996)
4. Karim, S. et al.: Generating bracelets with fixed content. Theoretical Computer Science, 475, 103-112 (2013)
5. London, J.: Hearing in Time. Psychological Aspects of Musical Meter, 2nd edn. Oxford University Press, Oxford (2012)
6. Miller, G. H.: The Magical Number Seven, Plus Or Minus Two: Some Limits on Our Capacity for Processing Information. Psychological Review, 63, 81–97 (1956)
7. Nierhaus, G.: Algorithmic Composition: Paradigms of Automated Music Generation, Springer, Wien (2010)
8. Phillips, D.: Introducing CsoundAC: Algorithmic Composition With Csound And Python. Linux Journal [online] (2010), `http://www.linuxjournal.com/content/introducing-csoundac-algorithmic-composition-csound-and-python`
9. Taube, H. K.: Notes from the Metalevel, Taylor & Francis, London and New York (2004)
10. Thakar, M.: Looking for the "Harp" Quartet: An Investigation into Musical Beauty, University of Rochester Press (2011)
11. Yi, S.: blue: a music composition environment for Csound `http://blue.kunstmusik.com`

Interactive Visual Music with Csound and HTML5

Michael Gogins[1]

Irreducible Productions
michael.gogins@gmail.com

Abstract. This paper discusses aspects of writing and performing interactive visual music, where the artist controls, in real time, a computerized process that simultaneously generates both visuals and music. An example piece based on Csound and HTML5 is presented.

Keywords: Visual music, generative art, algorithmic composition, computer music, Csound, HTML5

1   Introduction

This paper presents an approach to writing interactive visual music using Csound [1] [2] [3] with HTML5. Artistic and technical problems are discussed, and some solutions are presented in the context of an example piece [4] that runs on csound.node [5] [6], currently the most stable and highest-peforming HTML5 environment for Csound. These techniques also work in Csound for Android [7] [8], PNaCl [9] [10], Emscripten [11] [12], and WebAssembly [14] [6]. In all these, Csound appears in the JavaScript context as a `csound` object that exposes the Csound API [15]. *Note*: By 2018 or so browsers are expected to have deprecated all other "native" execution environments in favor of WebAssembly.

HTML5 is the programming environment of current Web browsers. It is a world standard [16] with vast capabilities [17] that are programmable in JavaScript [18] [19]. In addition to HTML and JavaScript for defining user interfaces, features relevant to visual music include three-dimensional, animated computer graphics (WebGL) and high-resolution audio (WebAudio).

Visual music can mean (a) purely visual displays having a music-like evolution in time; (b) visualizations of music; or (c) a hybrid art in which the author, perhaps using software or other automatic processes, generates both visual and musical forms. This last type is the subject of this paper — in particular, where one process is performed interactively, in an improvisational way, to generate both visuals and sounds. In any case, visual music tends to abstraction, otherwise it would be no different from music videos.

The pioneers of visual music were the typical lot of visionary outsiders [20]. Walt Disney's **Fantasia** [21] showed stellar examples. Later, experimental filmmakers created some visual music [22], still later *light shows* [23] became standard at concerts of psychedelic music, and from the late 1970s the *demoscene*

[24] [25] showed programmed animations with musical accompaniment, some of high quality. Visual music became notable in computer music starting perhaps with **Circles and Rounds** by Dennis Miller at the 2006 ICMC [26].

2   Technical Problems (and Some Solutions)

Artistic issues are more important than technical ones, but technology must be discussed first as it is the foundation for the art. Historically the technical issues with visual music have been expensive tools, expensive labor, and incompatible standards. As computer power has doubled and redoubled every year or so, the problem of expensive tools has been mooted, the problem of expensive labor has not changed, and the problem of incompatible standards has perhaps worsened.

Visual music started with painting both images and sounds by hand on movie film. Then of course hand-drawn animation became highly developed, followed by all kinds of tricks used in experimental film. Eventually these were built into hardware for editing film or video with "effects," and then into software for computer animation. Today these technologies are collected in commercial applications such as 3ds Studio Max [27], open source software such as Blender [28] and Processing [29], and game engines such as Unreal [30] and Unity [31]. But this profusion of tools has not solved the problem of labor – see the amazing list of credits for any blockbuster animated film – and also has contributed to the problem of incompatible standards. As in other computer-based arts, there is now a Babel of applications and languages that tends to fragment the field as different artists choose different software for different reasons, and thus lose the ability to understand each other on a *technical* level.

All the above-mentioned software packages are just *different containers* for the *same algorithms*: time lines, scene graphs, texture mappings, shaders, convolvers and filters, software synthesizers, etc. And these are *all* present in the JavaScript context of standard Web browsers where they run at high speed by virtue of SIMD, GLSL, and expertly written C++. In short, HTML5 offers a viable solution to the problems of expensive tools and incompatible standards.

Blender, Processing, and the Unity engine offer similar solutions but, as Csound is arguably the most powerful software synthesizer and can be used directly in HTML5 via csound.node or Csound for WebAssembly, Csound in HTML5 offers a standards-based, high-performance platform for visual music.

Technical issues arise not only from the platform, but also from the artistic objectives. These are considered in the next section.

But first, here is an overview of the architecture of the example piece [4] running in csound.node (similar designs would work in any other Csound/HTML5 environment). The piece itself is one Web page with all code, including Csound itself and the Csound orchestra [4, lines 87-239], either embedded in the page or loaded locally. The user interface consists of sliders and key bindings with JavaScript event handlers, defined using the dat.gui library [32] [4, lines 1172-1219]. An embedded style sheet formats the elements [4, lines 7-52]. The generating process is a real-time GLSL [33] animation, a "shader toy" [34] adapted

from the work of lomateron [35], which computes an animated fractal at exceptionally high speed on the massively parallel, dedicated GPU of the computer [4, lines 240-297]. JavaScript code samples pixels from the drawing buffer in real time [4, lines 994-1030] and translates them into musical notes [4, lines 974-992] that are sent to a running instance of Csound, which is called by the WebAudio driver to play the sound.

3 Artistic Problems (and Some Solutions)

In visual music, usually the music comes first and then the visuals ("light shows"), or the visuals come first and then the music (experimental films or demos with derivative music). Either way, one half of the visual music equation usually suffers by comparison with the other half. In theory, this imbalance can be righted by generating both visuals and music from the same process:

1. Both visuals and music are generated by the same underlying, more abstract process. This is rare, as usually the processes have already been designed for one purpose or the other.
2. The music generator is sampled or processed to also generate the visuals. This is quite common.
3. The visual generator is sampled or processed to also generate the music. This is not so common, but not unknown.

Understanding the tradeoffs of the second and third options requires analysis.

3.1 Bandwidth and Format Disparities

Both visuals and music can be digitally processed at different levels of abstraction. For visuals, the highest level of abstraction consists of scenes of geometric objects or meshes covered with textures, illuminated by lights, and viewed by a virtual camera; the lowest level of abstraction is a screen of pixels, a thousand or so wide and high, presented at up to 60 or so frames per second.

A perspective rendering of three dimensions is very common, and virtual realities that immerse the viewer in a stereoscopic perspective view are becoming more common. But for the purposes of visual music, the perspective rendering and the stereoscopic rendering are the same: a three-dimensional scene.

For music, the highest level of abstraction is the score, which consists of notes assigned to instruments, which produce actual streams of audio that are further processed and mixed. There are usually a few to a few dozen discrete notes per second. (There can be an intermediate level of abstraction not considered here, grains of sound that stream at a rate of hundreds or thousands per second.) The lowest level of abstraction is 44,100 to 96,000 frames per second of stereo (or, increasingly, multi-channel) audio samples.

There are obvious disparities of data formats and rates between visuals and music. At the highest level of abstraction, dozens to thousands of visual objects

are moving, but no more than a dozen or so musical notes are moving. At the lowest level of abstraction, for uncompressed raw data, the bandwidth of high-definition video is on the order of 3,732,480,000 bits per second, whereas the bandwidth of uncompressed high-definition stereo audio is on the order of 4,608,000 bits per second. In reality visual data is more redundant than audio data; a compressed stream of video runs at about 30,000,000 bits per second, whereas a compressed stream of audio runs at about 500,000 bits per second. Hence visual bandwidth runs about 60 times audio bandwidth.

Finally, the visuals are not always computed as objects in a scene; they may be computed directly at the pixel level. This is attractive, because HTML5 environments can execute runtime-compiled "shader" programs, which operate directly on pixels, on the graphics processing unit (GPU) at *much* higher speeds than on the general purpose central processing unit (CPU).

The much greater data bandwidth of visuals is one reason it makes sense to derive the music from the visuals, instead of the other way round. But then it also becomes necessary not only to map the visual data to musical parameters, but also to filter or reduce the density of data – while still preserving a perceptible relationship between the visuals and the music.

3.2   Mapping, Triggering, and Filtering

"Mapping" actually involves *dimensional mapping*, *filtering* to reduce the data bandwidth, and *triggering* musical events. Triggered events may in addition be post-processed, e.g. to tie overlapping notes, or to fit into a harmony.

Dimensional Mapping Mapping visual *objects* to music is complex, and must be considered case by case. Such a mapping amounts to using the visuals as a sort of score for the music. A minimal set of dimensions for visual objects might be the following. Lower-case letters stand for visual attributes, and upper-case letters for musical attributes.

t Time   Real Seconds from beginning of performance.
x Horizontal Cartesian coordinate Real Arbitrary units
y Vertical Cartesian coordinate   Real Arbitrary units
z Depthwise Cartesian coordinate Real Arbitrary units

For mapping visual objects to musical events, musical attributes can be computed from, or even attached to, the objects in the scene, thus reinforcing its dual role as a score. Mapping visual pixels is more straightforward, as there is the following fixed set of dimensions:

t	Time	Real	Seconds from beginning of performance
x	Horizontal Cartesian coordinate	Integer	0 to image width
y	Vertical Cartesian coordinate	Integer	0 to image heght
h	Hue	Real	0 through 1
s	Saturation	Real	0 through 1
v	Value (brightness)	Real	0 through 1

The dimensions of notes, at a useful minimum, are:

T	Time	Real	Seconds from beginning of performance
I	Instrument	Integer	0 to I_{Max}
K	MIDI key	Real	K_{Min} through K_{Max}
V	MIDI velocity	Real	V_{Min} through V_{Max}
P	Stereo pan	Real	-1 through 1

Given the dimensional units with their minima and maxima, the actual mappings are obvious. For animated visuals [4, lines 922-992]:

$$T = t$$
$$I = \lfloor 1 + x/x_{Max} \rfloor$$
$$K = \lfloor (y/y_{Max})(K_{Max} - K_{Min}) + K_{Min} \rfloor$$
$$V = v(V_{Max} - V_{Min}) + V_{Min}$$
$$P = s2 - 1$$

Such mappings are necessary but not sufficient. Visual bandwidth is still far greater than musical bandwidth, so the number of events must be cut down even further without breaking the perceptible relation between visuals and music.

To accomplish this, musical events can be triggered only from the most salient visual events. Then, the triggered events can be filtered to further cut down the number of events.

Triggering In the retina, a neural network specializes in detecting edges. Here, edges can similarly be used to detect easily perceptible events. (An actual computerized neural network could perhaps be used to identify these features.)

For animated visuals, an edge occurs when, for a single pixel, the color at frame f_t changes at time f_{t+i}. When the value of a pixel changes from a level below a threshold, to a level at or above that threshold, a note on event is triggered [4, lines 1017-1020]; and when the value changes from a level at or above that threshold, to a level below that threshold, a note off event is triggered [4, lines 1021-1024].

This already reduces the visual bandwidth by a considerable amount, as no new musical events are generated at a pixel while its color remains stable from frame to frame, or its value does not cross the threshold.

However, for animated visuals, this still creates too many musical events per second. Additional filtering or sampling must also be used.

Filtering and Sampling The kind of filtering or sampling required obviously differs between objects and pixels. In the case of objects, for example, only the centers of volume could be considered, or even only a certain level of ramification in the tree of objects in the scene graph. In the case of pixels, which is considered here, the actual grid of pixels can be sampled at some modulus of its width and height [4, lines 1003-1007], or along radii from the center, or so on. If only every 1000th pixel is sampled, the artistic intelligibility of the relationship between the visuals and the music may suffer; this must be judged case by case. To restore intelligibility, the sizes of visual features can perhaps be increased.

Experience shows that the correlation between visual and musical events need by no means be constant. If there is some perceptible synchrony every bar or so, that seems to do the job.

4   Improvisational Control

The whole purpose of the approach to visual music discussed here is to put on a show: to improvise a work of visual music involving both visual and audible forms.

For a single performer to do this, the controls must be manageable. The computer mouse and keyboard provide a limited palette of control gestures. But if the process generating the visual music is a fractal or recursive process controlled by a few numerical parameters, a few gestures suffice for playing [4, lines 1312-1346]. A few key combinations can also be dedicated to changing the arrangement or tonality of the music, for example by applying voice-leading transformations that generate chord progressions in the music [4, lines 1279-1309].

References

1. Csound home page, `http://csound.github.io`
2. Lazzarini, V. et al.: Csound: A Sound and Music Computing System. Springer (2016)
3. Boulanger, R. (ed.): The Csound Book. The MIT Press (2000)
4. AuthorA: Csound Visual Music Example, `https://AuthorA.github.io://csound/csound_visual_music.html`. This piece requires NW.js and csound.node to run.
5. NW.js, `https://nwjs.io/`.
6. Gogins, M.: csound.node, `https://github.com/csound/csound/tree/develop/frontends/nwjs`.
7. Yi, S. and Lazzarini, V.: Csound for Android. In: Proceedings of the Linux Audio Conference 2012, pp. 29-34. Stanford (2012).
8. Csound for Android App, `https://play.google.com/store/apps/details?id=com.csounds.Csound6&hl=en`.
9. NaCl and PNaCl, `https://developer.chrome.com/native-client/nacl-and-pnacl`.
10. Csound for PNaCl, `https://github.com/csound/csound/tree/develop/nacl`.
11. emscripten, `http://kripken.github.io/emscripten-site/`.

12. Csound for Emscripten, `https://github.com/csound/csound/tree/develop/emscripten`.
13. WebAssembly, `http://webassembly.org/`.
14. Csound for Emscripten (for WebAssembly see `build-wasm.sh`). `https://github.com/csound/csound/tree/develop/emscripten`.
15. Csound API, `http://csound.github.io/docs/api/index.html`.
16. HTML5, `https://www.w3.org/TR/2016/REC-html51-20161101/`.
17. HTML 5 Test, `https://html5test.com/`.
18. ECMAScript 2016 Language Specification, `http://www.ecma-international.org/ecma-262/7.0/index.html`.
19. Flanagan, D.: JavaScript: The Definitive Guide (6th ed.). O'Reilly (2011).
20. Brougher, K. et al.: Visual Music. Thames & Hudson (2005).
21. Culhane, J. and Walt Disney Productions: Walt Disney's Fantasia. H.N.Abrams (1983).
22. Rees, A.L.: A History of Experimental Film and Video (2nd ed.). British Film Institute (2011).
23. The Joshua Light Show, `http://www.joshualightshow.com/about`.
24. Demoscene Research, `http://www.kameli.net/demoresearch2/`.
25. Demoscene Portal, `http://www.demoscene.info/the-demoscene/`.
26. International Computer Music Association: International Computer Music Conference (2006).
27. 3ds Studio Max, `https://www.autodesk.com/products/3ds-max/overview`.
28. Blender, `https://www.blender.org/`.
29. Processing, `https://www.processing.org/`.
30. Unreal Engine, `https://www.unrealengine.com/`.
31. Unity, `https://unity3d.com/`.
32. dat.gui, `https://github.com/dataarts/dat.gui`.
33. Kessenich, J.: The OpenGL Shading Language: Language Version 4.50. The Khronos Group (2016).
34. Shadertoy, `https://www.shadertoy.com/`.
35. lomateron: Lights pattern generator, `https://www.shadertoy.com/view/Xl3SzB`.
36. Tasajarvi, L: Demoscene: The Art of Real Time. Even Lake Studios (2004).

Spectral and 3D spatial granular synthesis in Csound

Oscar Pablo Di Liscia

[1] Programa de Investigación Sistemas Temporales y Síntesis Espacial de Sonido, PICT-2015-2604, Escuela Universitaria de Artes, UNQ, Argentina.

odiliscia@unq.edu.ar

Abstract. This work presents ongoing research based on the design of an environment for Spatial Synthesis of Sound using *Csound* through granular synthesis, spectral data based synthesis and 3D spatialisation. Spatial Synthesis of Sound may be conceived as a particular mode of sonic production in which the composer generates the sound together with its spatial features. Though this type of conception has long lived in the minds and work of most composers (specially in electroacoustic music), some strategies applied here were inspired by the work of Gary Kendall [13]. Kendall makes specific mention to both Granular Synthesis and Spectral data based synthesis as examples of resources through which the composer may partition the sonic stream in both the time domain and the frequency domain, respectively. These procedures allow a detailed spatial treatment of each one of the obtained parts of a sound which, in turn, may lead to realistic or unusual spatial images. The aim is not to describe in detail granular synthesis, nor spectral data based synthesis or sound spatialisation techniques, but to describe the particular strategies in the design for the aforementioned purposes.

Keywords: Spectral data based synthesis, granular synthesis, sound spatialisation.

1   Introduction and theoretical background

This work presents an environment for Spatial Sound Synthesis (from here on abbreviated as SSS) using *Csound* through granular synthesis, spectral data based synthesis and 3D spatialisation. This is a part of the author's ongoing research project, in which one of the main objectives is the development of software for high level programming environments (such as *Csound*, *Pure Data* and

Super Collider) for SSS. Another of the software developments being pursued was already presented in Di Liscia [5]. The techniques of Granular Synthesis, Spectral Data Based Synthesis and Spatialisation will not be treated in detail, since those are well known subjects in Computer Music and Digital Signal Processing[1].

SSS[2] may be conceived as a particular mode of sonic production in which the composer generates the sound together with its spatial features. Though this type of conception has long lived in the minds and work of most composers (specially in electroacoustic music), some strategies applied here were inspired by the work of Gary Kendall [13]. Analyzing the function of sound spatialisation in electroacoustic music, Kendall places significant emphasis on the interplay between the perceptual grouping and the spatial features of sound. He makes specific mention of both granular and spectral data based synthesis, as examples of resources through which the composer may partition the sonic stream in both the time domain and the frequency domain, respectively. These procedures allow a detailed spatial treatment of each one of the obtained parts of a sound which, in turn, may lead to realistic or unusual spatial images. Since granular synthesis may modify the "normal" time development of a sound it may be used, along with spatialisation, to generate ambiguities related with the number of virtual sources and their spatial projection. On the other hand, spectral data based synthesis in conjunction with spatial processing makes possible the modification of the spatial projection of the spectral components also leading to the aforementioned ambiguities.

Generally speaking, Granular Synthesis (from here on abbreviated as GS) is a technique based on the juxtaposition of small portions (called grains) of a source audio signal. In a typical GS application, the user is allowed to set (and eventually change over time) several parameters of a grain sequence, the most common of which are: duration, temporal gap, source audio signal, amplitude envelope, peak amplitude, and pitch shifting. The use of GS on electroacoustic and audio visual creation is very well known and widely documented (see, among others, Roads [21], Batty et al [2], and Del Campo [4]), as well as electronic works by many composers. There are many excellent GS opcodes in *Csound* some of the most important being *sndwarp, granule* and *partikkel*, [8]).

Sound synthesis using spectral data may compromise several analysis techniques, but some of the most well-known and used are based on the Fast Fourier Transform (FFT) analysis (See Moore [17], Embree & Kimble [6], Moorer [18],

---

[1] Some appropriate references on each one will be provided in the next section.

[2] To the knowledge of the author, the first authors that coined the name of *Spatial Sound Synthesis* were Bresson and Schumacher [3].

and Wessel & Risset [26]). High resolution analysis combined with a model which attempts to represent the attributes of a sound taking into account its deterministic and stochastic parts is also a well known improvement of FFT based analysis techniques (see [7], [20] and [23]). *Csound* provides several opcodes for FFT-based Analysis-Synthesis, some of the most important being the groups of opcodes for *pvoc, pvsanal* [9] and *ATS* [10] [20].

Sound spatialisation by computer means involves a huge group of techniques and resources. In the opinion of the author, these may be classified in three groups: a) the ones related to the virtual sources location, b) the ones related to the sources directivity, and c) the ones related to rooms/environments. *Csound* offers several excellent opcodes for source location and room/environment treatment. For the former, it is possible to use the *intensity panning* [14], [15], the *VBAP* [21] and the *Ambisonics* [16] techniques. The latter may be achieved via several pre-designed reverberators, using networks of IIR filters, *multitaps* and *delay* units connected in series and parallel, or performing fast convolution [11] with impulse responses of rooms / environments. The *spat3d* opcode [25] provides 3D sound spatialisation of both the direct signal and the early echoes.

2   Per grain 3D spatialisation and spectral GS in Csound

Though *Csound* provides several very powerful GS opcodes, none of them makes feasible individual 3D spatialisation and spectral treatment of overlapping grains in a way that the author found suitable for his purposes. Fortunately, *Csound* provides all the opcodes and resources required to design a GS environment with the capacity of individual processing of each grain generated. The environment that will be presented includes an instrument that implements GS (*the_grainer*) by calling recursively another instrument (*the_grain*) that generates each grain with its own spectral and 3D spatial features, and another instrument (*greverb*) that provides multi-channel reverberation for the complete stream of grains. In what follows, the details of each one of the mentioned instruments, their capacities, and their interaction are discussed.

the_grainer instrument is the part of the GS environment that creates a stream of grains computing all its features and calling with the appropriate parameters another instrument (*the_grain*) that will be described in the next section. The parameters involved are the usual ones in GS synthesis, plus the ones related to 3D spatialisation and spectral features. The user may set the grain's duration, audio source sound file, amplitude envelope function, temporal gap, peak amplitude, audio starting read point on the audio source sound file, pitch

transposing, spatial location (azimuth angle, elevation angle and distance) and spectral features (these will be further explained).

For each one of the aforementioned parameters, there is the possibility of setting a base value plus a random deviation value that may change over the grain stream. There are, at present, four ways of setting these two values, which are handled by *ad hoc* macros for the user convenience (details can be found in the source code an its documentation). The audio source for the grains must be, at present, a sound file. The user may, however, have a "pool" of audio source sound files out of which a particular audio source sound file for each one of the grains of a stream may be selected by means of the aforementioned methods.

Since for the spectral processing of the grains, the *pvsanal* / *pvsadsyn* [9] opcodes are presently used, there is the possibility of setting the offset bin value, the bin increment value and the number of bins for the synthesis value which may also be invariant or change over the duration of the stream of grains.

the_grain instrument synthesizes the grains with the parameters computed by its caller instrument (*the_grainer*). The parameters are invariant over the duration of each grain.

As mentioned, the *pvsanal* / *pvsadsyn* opcodes are presently used for the spectral processing of the grains. The time domain signal of the grain generated is sent to *pvsanal* opcode and the frequency domain signal generated by it is sent to *pvsadsyn* opcode to generate the time domain signal of the grain with the spectral modifications requested.

The spatialisation of the grains is achieved mainly using the Ambisonics [16] technique through the *spat3di* [25] and the *bformenc1* [12] opcodes. The *spat3d* opcode provides 3D sound spatialisation computing both the direct signal and the early echoes (using the image method [1]) for a virtual source and listener located into a virtual room whose features must be set by the user in a *Csound* Table. The output of this opcode was set to B-Format First Order Ambisonics (which will produce the four signals usually termed in Ambisonics parlance as W, X, Y and Z) and the echo recursion was set to 2 (the number of early echoes computed will be 24 in this case). After that the first order Ambisonic signals are encoded using *spat3di* and stored in the first four cells of the output array, if requested, the Second or Third Order Ambisonic B-Format of the direct signal only is computed using the *bformenc1* opcode and the remaining signals[3] are added to the corresponding remaining cells of the output array which will -then- contain a *mixed-order* Ambisonic set of signals (MOA). This strategy was al-

---

[3] In the case of Second Order Ambisonic B-Format, these will be the five signals (R, S, T, U and B) and, in the case of Third Order Ambisonic B-Format, seven signals (K, L, M, N, O, P and Q) will be included as well.

ready used by Noisternig et al [19] in order to reduce computational expenses while also minimizing a potential quality loss in the perceptual results. Generally speaking, MOA systems take advantage of the human auditory system's spatial acuity in the horizontal plane, thus using a higher resolution in this plane compared to the resolution used for other directions [24]. Though this is not strictly the case in the present development the aim is similar, since it is commonly assumed that both, the early echoes and the dense reverberation, require less spatial definition than the direct sound.

The output of the instrument is set accordingly to the *nchnls* variable of *Csound*. At present there are four types of possible values: *nchnls*=2 will set stereo (*UHJ* trans-coded) output, whilst *nchnls*=4, *nchnls*=9 or *nchnls*=16, will set respectively First, Second and Third B-Format Order Ambisonics outputs.

The dense reverberation is achieved by a third instrument (*greverb*) by means of fast convolution using the *pconvolve* [11] opcode.

3   Conclusions and future improvements

The environment presented is in continuous assessment and development. However, even in the present state, it proved to be a very powerful, perceptually effective, and versatile tool for SSS applied to electronic composition.[4]

Since there is not enough space for an extensive treatment on the use of the environment for SSS, just two cases -taken from the examples included in the code- will be briefly addressed here. In the first case a broad frequency band sound, synthesized granulating a pitched note of a single sound source, performs a circular movement around the audience. As the movement evolves, the sound "drops" three specific audio bands of its frequency components each remaining steadily in the spatial zones through which their "source" sound has passed. At the same time, the granulation of each one of the partial bands becomes more apparent because the duration of the grains becomes gradually smaller than their gap times. In the second case a sequence of speech sounds is divided into two granulated streams: one containing vowels and the other containing occlusive consonants. This allows a distinctive spatial treatment of each one and, furthermore, the vowel stream is divided into two frequency bands which are segregated in space differently as well.

---

[4] All the referred *Csound* code fully commented (and with several examples that the reader is encouraged to test and analyze) is available at:
https://github.com/odiliscia/the_grainer

Future developments may include, among other, specific Ambisonic decoding stages, the use of higher order Ambisonics orders, the improvements of the distance cues using specially designed filters, the use of other spectral based techniques and the design of a graphic interface to handle the complexity of the environment more comfortably.

4   Acknowledgements

5   References

1. Allen, J. and D. Berkley: Image Method for Efficiently Simulating Small Room Acoustics. Journal of the Acoustical Society of America, 912–915 (1979).
2. Batty, J., et al: Audiovisual granular synthesis: micro relationships between sound and image. In: Proceedings of The 9th Australasian Conference on Interactive Entertainment: Matters of Life and Death, pp. 8, Australia (2013).
3. Bresson J., Schumacher, M.: Compositional Control of Periphonic Sound Spatialization. In: Proceedings of 2nd International Symposium on Ambisonics and Spherical Acoustics, Paris, France, (2010).
4. Del Campo, Alberto: Microsound. In: Scott Wilson, David Cottle, and Nick Collins (Eds.) The SuperCollider Book. The MIT Press, London, UK. pp. 463–504 (2010).
5. Di Liscia, Oscar Pablo: Granular synthesis and spatialisation in the Pure Data environment. In: PDCon 2016 Proceedings. Waverly Labs, NYU, New York, USA. pp.25–29. http://www.nyu-waverlylabs.org/pdcon16/proceedings/ (2016).
6. Embree, P. & Kimble, B.: C languaje algorithms for DSP, Prentice Hall, New Jersey, USA (1991).
7. García, G. and Pampin, J.: Data compression of sinusoidal modeling parameters based on psychoacoustic masking. In Proceedings of the International Computer Music Conference, Beijin (1999).
8. Heintz, Joachim et al: Csound FLOSS Manual.
   http://write.flossmanuals.net/csound/f-granular-synthesis/ (Last access: 05/2017)
9. Heintz, Joachim et al: Csound FLOSS Manual.
   http://write.flossmanuals.net/csound/i-fourier-analysis-spectral-processing/ (Last access: 05/2017)

---

10. Heintz, Joachim et al: *Csound FLOSS Manual.*
    http://write.flossmanuals.net/csound/k-ats-resynthesis/ (Last access: 05/2017)
11. Heintz, Joachim et al: *Csound FLOSS Manual.*
    http://write.flossmanuals.net/csound/h-convolution/ (Last access: 05/2017)
12. Heintz, Joachim et al: *Csound FLOSS Manual.*
    http://write.flossmanuals.net/csound/b-panning-and-spatialization/
    (Last access: 05/2017)
13. Kendall, Gary: La interpretación de la espacialización electroacústica: atributos espaciales y esquemas auditivos. In: Basso, Di Liscia y Pampin (Eds.): Música y espacio: ciencia, tecnología y estética. Editorial de la Universidad Nacional de Quilmes (2010).
14. Karpen, R. (1998), Locsig Space opcode Documentation. In: *The Csound Manual.*
    http://www.csounds.com/manual/html/locsig.html (Last access: 05/2017)
15. Karpen, R. (1998), Space opcode Documentation. In: *The Csound Manual*
    http://www.csounds.com/manual/html/space.html (Last access: 05/2017)
16. Malham, Dave: "El espacio acústico tridimensional y su simulación por medio de Ambisonics". In: Basso, Di Liscia and Pampin (Eds.): Música y espacio: ciencia, tecnología y estética, Editorial de la Universidad Nacional de Quilmes, pp. 161–202 (2010).
17. Moore, F. R.: An introduction to the mathematics of DSP, Part II. CMJ 2(2):38–60, MIT Press, USA (1978).
18. Moorer, J. A.: The use of the Phase Vocoder in Computer Music Applications. JAES, 26(1/2): 42-45 (1978).
19. Noisternig, M., Musil, T., Sontacchi, A., Höldrich, R.: A 3d real time rendering engine for binaural sound reproduction. In Proceedings of the 2003 International Conference on Auditory Display, Boston, MA, USA, 6-9 (2003). (Last access: 05/2017)
    https://www.researchgate.net/publication/228747180_A_3D_real_time_Rendering_Engine_for_binaural_Sound_Reproduction (Last access: 05/2017)
20. Pampin, J., Di Liscia, P., Moss, P., Norman, A.: ATS user Interfaces. In: Proceedings of the International Computer Music Conference, Miami University, USA (2004).
21. Pulkki, V.: Virtual sound source positioning using vector base amplitude panning. JAES, 45(6) pp. 456–466 (1997).
22. Roads, C.: Microsound. The MIT Press, England (2004).
23. Serra, X. and Smith J. O. III: A Sound Analysis/Synthesis System Based on a Deterministic plus Stochastic Decomposition, Computer Music Journal, 14(4), MIT Press, USA (1990).
24. Trevino, J., Koyama, S., Sakamoto, S., Suzuki, Y.: Mixed-order Ambisonics encoding of cylindrical microphone array signals. In Journal of Acoustic Science and Technology, 35, 3, The Acoustical Society of Japan, (2014).
25. Varga, I.: Spat3d opcode Documentation, In: The Csound Manual.
    http://www.csounds.com/manual/html/spat3d.html (Last access: 05/2017)
26. Wessel, D. and Risset, J.: Exploration of Timbre by Analysis and Resynthesis. In: The Psichology of Music, D. Deutsch (Ed.), Academic Press. pp. 26–58 (1985).