California Institute of the Arts

ChucK Racks:
Coding Music in the Digital Audio Workstation

by

Rodrigo Miguel Sena Aguirre

A thesis submitted in partial fulfillment for
the degree of Master of Fine Arts

Herb Alpert School of Music
Music Technology: Interaction, Intelligence & Design
2016

Supervisory Committee

Jordan Hochenbaum, Ph.D.

Mentor

Spencer Salazar, Ph.D.

Committee Member

Ajay Kapur, Ph.D.

Committee Member

Abstract

This thesis examines the relationship between programming-based computer music making and Digital Audio Workstation (DAW)-based computer music making. Is there a way to make creators seamlessly flow between those two domains? Can we make audio software development and prototyping more immediate and fun?

The scope of the research presented here is the application of sound creation and manipulation through text programming incorporated in a DAW. The union of these two components presents new possible tools for artists, including new ways for music production and live performance. It also presents a new workflow that attempts to be more efficient than alternative methodologies.

This thesis presents a software audio plugin that can be loaded into any common DAW. This system, called *ChucK Racks*, is based on the ChucK music programming language and provides a very straightforward way of creating a link between an instance of ChucK and an audio channel in the DAW. ChucK includes many unit generators that can be used to process and generate audio. Many ChucK extensions were written especially for ChucK Racks to facilitate the flow of information between the ChucK instance and the host, including audio, MIDI, automation, transport, and tempo information.

This thesis also proposes using ChucK Racks as an educational tool that introduces programming to electronic musicians. Examples are provided which users can use in their compositional process without modifying the code, exposing them to the potential of custom tool creation.

The primary contributions of this thesis are: 1) An audio plugin that allows a seamless combination of coding, music composition and sound design, 2) Examples demonstrating those possibilities 3) Pre-made scripts for effects and instruments that users can use without modifying the code.

.

Acknowledgments

First of all, I would like to thank Jordan Hochenbaum and Spencer Salazar. Without their contributions, guidance, and assistance this project would have never been possible.

I would also like to thank my other professors in MTIID, Ajay Kapur and Owen Vallis, whose teachings for the past four years helped me become who I am today.

I'd also like to thank all my friends in Music Tech, whose conversations often distracted me from working on this thesis, but usually presented me with invaluable inspiring ideas; Thank you Eric Heep, Daniel McNamara, Cordey Lopez, Carl Burgin, James Meason Wiley, Martín Vélez, JP Yépez, Daniel Reyes, Nick Suda, Jake Penn, Nathan Shaw, Ness Morris, Zach Crumrine, Dexter Shepherd, Shaurjya Banerjee and everyone else who I had the pleasure to study with during my years in the program. I also want to especially thank Eric and Jake for their contributions to the ChucK Racks source code.

I would also like to thank the people that made me feel like I have a home in the United States of America, my roommates and friends Melanie Carrol-Dolci and Emmett Thatcher, and especially my very good friend Bruce Lott, who I knew I could always count on.

I would like to thank my mom, Vivian Aguirre, and my sister, Raquel Sena, for their always present unconditional love.

Most of all I would like to thank my dad, Miguel Sena, whose constant support made so many opportunities in my life possible, including my studies here. I dedicate this thesis to him.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

1.1 On Creation and Computer Music

The appearance of the personal computer has deeply and permanently changed the way people create music. Electronic music is a genre that has particularly benefited from this. In the past musicians dealing with technology needed different hardware units for each different type of sound creation and sound processing. An electronic musician from thirty years ago needed an expensive studio full of machines to fulfill their needs for sound design, production, and recording. Personal computers today are powerful enough that all of those can be simulated in software. Musicians can make complete tracks from start to finish using the computer as their only piece of hardware.

The graphical environment of the personal computer has also made them highly usable. People without technical backgrounds can learn how to use them very quickly. With them came the Digital Audio Workstation, which is a type of program that offers an integrated environment for audio mixing, recording, editing, sequencing, and sound synthesis and processing. They're also inexpensive compared to hardware. Thanks to the internet, users can have free access to all kinds of materials to learn the technical skills to operate these environments. Because of all these reasons DAWs have been adopted by millions of users. The problem lies in the missing potential of mixing these high-level music-making tools with the incredibly deep potential of computer programming.

For many decades, computer music was a field mainly pursued by academia, both because of the unavailability of computers for the general public, and the required technical knowledge to operate them. Many programming languages were created to explore the possibilities of using a computer to generate both sound and note sequences. These systems were profoundly abstracted ways of making music, especially compared to traditional instruments, but they opened a new world of possibilities for sound. However, most of these systems were not adopted by the vast

majority of musicians when personal computers became available. More conventional and usable tools, which eventually evolved into the modern DAW, became by far the most common way of interacting musically with computers. This left the lower level and more abstract music programming languages to be used only in academic environments and by experimental musicians, far from the popular musical revolution they helped pioneer.

Music programming languages are still being actively developed, but they're only used by a very small percentage of computer musicians compared to the DAW. Even musicians who are familiar with both types of environments usually see these as two isolated domains. All the possible current methods of combining these are cumbersome, unintuitive, and restricted. These two very capable kinds of tools continue to be used for completely different things in different contexts.

This thesis presents an alternative: Creating a tool that bridges both worlds. What if any electronic musician could realize the power of creating their own tools, and be motivated to learn computer programming? What if we could give users the possibility to seamlessly flow between these two realms as part of the same creative process? Can we make prototyping possible professional tools more immediate and fun? A tool like that would have to be integrated into a DAW in a familiar way, encapsulating the complexity of programming in a manageable way.

This thesis attempts to answer these questions, through the creation of a new tool in which to apply and test these ideas: ChucK Racks.

1.2 ChucK Racks

ChucK Racks is an audio plugin that can be loaded into any DAW and presents the opportunity of writing code in real time while interacting with other elements inside the DAW. The user can add scripts to manipulate or generate sound.

ChucK Racks provides an opportunity for electronic musicians who are familiar with the workflow of a DAW to explore the deep possibilities of music coding, using the ChucK music programming language, created by Ge Wang and Perry Cook (Wang 2008). ChucK Racks was developed using the JUCE C++ framework, and supports compilation into Steinberg's Virtual Studio Technology (VST) format and Apple's Audio Unit (AU).

ChucK Racks allows for making, loading, running, and editing ChucK programs inside any of the host channels. The DAW can interface with Chuck Racks in multiple ways: internal

parameters can be automated; timing information can be shared and MIDI messages can be both sent and received. MIDI control information can be generated or processed to create sequencers and MIDI effects. Audio can be generated inside ChucK Racks to make synthesizers and other instruments. It can also process the incoming audio from the host DAW to make audio effects.

1.3 Thesis Overview

This document presents ChucK Racks, shows what similar endeavors came before it, why it is necessary for it to exist, how it was made, and what it can do. Chapter 2 discusses some of the history of computer music that has led to the current state in the field. A focus is placed on the history of music programming and the technologies that culminated into the modern Digital Audio Workstation. Chapter 3 defines two ways of making computer music. DAW-based music making and programming based music making. The strengths and weaknesses of each of those methods is examined, as well as existing ways of combining both. ChucK Racks is presented as a solution for integrating both workflows in a very efficient manner, by demonstrating ChucK Racks' uses and its advantages. Chapter 4 briefly details how ChucK Racks was implemented technically, but focus on its features and how to use them. Chapter 5 demonstrates the unique possibilities of ChucK Racks by showcasing some of the instruments and effects that have already been made with and will come as ready-to-use examples in the downloadable version. Chapter 6 discusses the future possibilities of ChucK Racks.

Chapter 2

Related Work

A comprehensive history of computer music is beyond the scope of this thesis, but this chapter will briefly explore some important events and pieces of software that culminated in the modern state of computer music making.

2.1 Computer Programming and Music

When early computers were designed, their potential for sound processing and generation was immediately explored. The first way of interacting with these machines was, of course, computer programming or "coding". Even though making computer music by coding was the only possibility in the early days, coding still has some advantages that keep music programming languages alive today.

2.1.1 Early Computer Music Programming

The pioneers of computer music had very different tools than the ones we have today. In the 1940s and 1950s computer musicians only had access to mainframe machines. These were very expensive and large self-contained machines that were designed for scientific and commercial data processing. Only government agencies, academic institutions or similarly big organizations had the resources to develop or own this type of machine, so very few people had access to them. The software at that time was very limited, and the users had to engage directly with the internal processes and components of the computer. This required users to have a very strong technical background in computer science and electronic engineering. (Manning 2004)

Before the late 1960s, the only way of writing for these computers was writing the program on punch cards, which that computer would decode, process and print out the result. There was no

way of interacting with the program while it was executing, and having to send out the program to be processed meant that the programmers wouldn't know the result until days later.

Figure 1. IBM 704 Electronic Data Processing System (1954 ca.)

The first attempts to use the computer for synthesizing sound date from the mid-1950s. (Manning 2004) The research division of Bell Telephone Laboratories was working with computers to convert audio signals into digital data, as a way of facilitating multiple phone conversations through a single line. It was Max Mathews, an engineer at Bell Labs, who first started using this technology to experiment with sound synthesis. Figure 1 shows the type of computer Bell Labs owned, which was the one used by Mathews for his first developments. He developed his first computer language for creating sound in 1957, and its follow up in 1958. These were called MUSIC I and MUSIC II respectively and consisted of generation and manipulation of different audio waveforms. After several iterations, the language eventually developed into MUSIC 11, which was translated into C by Barry Vercoe at MIT in 1986. This is known as the CSOUND language, which is still widely used today (Manning 2004). One of the features introduced in MUSIC 3 was the idea of unit generators. These were virtual modules with specific functionality that could be interconnected. (Collins, Schedel, and Wilson 2013). Another characteristic of the language is the separation of the sound generation elements (orchestra) and the instructions for those (score). These are separated into different files.

```
<CsoundSynthesizer>
<CsOptions>
csound -d -R -W -Z -f -o xanadu-high-resolution.wav
</CsOptions>
<CsInstruments>
sr          =           88200
ksmps       =           1
nchnls      =           2
;-------------------------------------------------------------
;Instrument 1 : plucked strings chorused left/right and
;       pitch-shifted and delayed taps thru exponential
;       functions, and delayed.
;-------------------------------------------------------------

        instr       1
ishift      =           .00666667               ;shift it 8/1200.
ipch        =           cpspch(p5)              ;convert parameter
ioct        =           octpch(p5)              ;convert parameter
kvib        poscil      1/120, ipch/50, 1       ;vibrato
ag          pluck       2000, cpsoct(ioct+kvib), 1000, 1, 1
agleft      pluck       2000, cpsoct(ioct+ishift), 1000, 1, 1
agright     pluck       2000, cpsoct(ioct-ishift), 1000, 1, 1
adamping    linsegr     0.0, 0.006, 1.0, p3 - 0.066, 1.0, 0.06, 0.0
ag          =           adamping * ag
agleft      =           adamping * agleft
agright     =           adamping * agright
af1         expon       .1, p3, 1.0             ;exponential from 0
af2         expon       1.0, p3, .1             ;exponential from 1
adump       delayr      2.0                     ;set delay line of
atap1       deltap3     af1                     ;tap delay line wit
atap2       deltap3     af2                     ;tap delay line wit
ad1         deltap3     2.0                     ;delay 2 sec.
ad2         deltap3     1.1                     ;delay 1.1 sec.
            delayw      ag                      ;put ag signal into
            outs        agleft+atap1+ad1, agright+atap2+ad2
            endin
;-------------------------------------------------------------
;Instrument 2 : plucked strings chorused left/right and
;       pitch-shifted with fixed delayed taps.
;-------------------------------------------------------------

        instr       2
ishift      =           .00666667               ;shift it 8/1200.
ipch        =           cpspch(p5)              ;convert parameter
ioct        =           octpch(p5)              ;convert parameter
kvib        poscil      1/120, ipch/50, 1       ;vibrato
ag          pluck       1000, cpsoct(ioct+kvib), 1000, 1, 1
```

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>

<CsInstruments>
sr     = 44100
ksmps  = 300
nchnls = 1

instr  1
  kamp = 20000
  kcps = p4
  icps = 100
  ifn = 0
  imeth = 1

  a1 pluck kamp, kcps, icps, ifn, imeth
  out a1
endin

instr 2
  kamp = 10000
  kcps = p4
  ifn = 1

  a1 oscil kamp, kcps, ifn
  out a1
endin
</CsInstruments>

<CsScore>
#define BAR #2#
#define FOO #[$BAR + 1]#
f 1 0 16384 10 1
{ 3 $FOO
i 1 0 0.1 440
i 2 0 0.1 220
i 1 0 0.1 110
i 2 0 0.1 330
}


e
</CsScore>
</CsoundSynthesizer>
```

Figure 2. CSOUND code example

During this era, musicians had to deal with lots of delays and difficulties to make computer music. The fact that computers eventually became powerful enough to do real time audio processing had affected how subsequent programming languages worked: "as computers increased in computational power and storage, programming languages tended to become increasingly high-level, abstracting more details of the underlying system." (Wang 2008)

2.1.2 Modern Textual Music Programming Languages

A programming language can be considered an instrument. Lack of computing power is not an issue anymore, and current languages have the choice of focusing on high level abstraction and the user experience without worrying too much about efficiency. They can be generators of different ideas (If efficiency is a valid concern, using the general-purpose language C++ is the best option. This is discussed in 3.4.2). The remainder of this section looks at a few common modern textual music programming languages.

2.1.2.1 SuperCollider

```
(
SynthDef(\snare909,{ |out=0,mul=1,velocity=1|
    var excitation, membrane;

    excitation = LPF.ar(WhiteNoise.ar(1), 7040, 1) * (0.1 + velocity);
    membrane = (
        /* Two simple enveloped oscillators represent the loudest resonances of the drum membranes */
        (LFTri.ar(330,0,1) * EnvGen.ar(Env.perc(0.0005,0.055),doneAction:0) * 0.25)
        +(LFTri.ar(185,0,1) * EnvGen.ar(Env.perc(0.0005,0.075),doneAction:0) * 0.25)

        /* Filtered white noise represents the snare */
        +(excitation * EnvGen.ar(Env.perc(0.0005,0.4),doneAction:2) * 0.2)
        +(HPF.ar(excitation, 523, 1) * EnvGen.ar(Env.perc(0.0005,0.283),doneAction:0) * 0.2)

    ) * mul;
    Out.ar(out, membrane!2)
}).add
)

Synth(\snare909,[\mul,0.5,\velocity, rrand(0.5, 1.0)]);

```

Figure 3. SuperCollider code example

SuperCollider is a commonly used musical programming language. The main feature the language is based around is the separation of sound generation and control into a server and a client. ("Client vs. Server" 2017) The client has an interpreter that turns SuperCollider code (or even other programming languages) into Open Sound Control[1](OSC) messages. These messages are then sent to the server. The server receives only OSC messages, which are used to control the sound generation. The server also sends out OSC messages when certain events happen that the client can use to adapt and respond. This separation allows for interesting interactions, including several clients interacting with the same server, including different users on different computers. It also favors toward basing compositions on events, rather than time. The server's sound generation is based around Unit Generators first explored in the CSOUND family of languages. These Unit Generators can be created and rearranged also using OSC messages.

[1] Open SoundControl is a musical information transmission protocol. It is a more modern alternative to MIDI, although it is not as widely used. "Open SoundControl is a new protocol for communication among computers, sound synthesizers, and other multimedia devices that is optimized for modern networking technology." (M Wright 1997)

2.1.2.2 ChucK

```
ModalBar imp;

Gain g => NRev r => Echo e => Echo e2 => dac;
imp => g;
g => dac;
e => dac;

1500::ms => e.max => e.delay;
3000::ms => e2.max => e2.delay;
1 => g.gain;
.5 => e.gain;
.25 => e2.gain;
.1 => r.mix;

[ 0, 2, 4, 7, 9, 11 ] @=> int hi[];

while( true )
{
    45 + Math.random2(0,3) * 12 +
         hi[Math.random2(0,hi.cap()-1)] => int note;
    Std.mtof( note )=>imp.freq;
         imp.noteOn(1);
    195::ms => now;
    imp.noteOff(0);
    5::ms => now;
}

```

Figure 4. ChucK code example

The ChucK programming language, on which ChucK Racks is based on, is founded on the idea of creating a language that is "expressive and easy to write and read with respect to time and parallelism." (Wang 2008) Besides being based around the concept of Unit Generators first explored in the CSOUND family of languages, it provides easy but deep ways of exploring a concurrent programming model and precise control of time during code execution. Different fragments of code can be "sporked" (launched) as independent "shreds" (threads) that will run in parallel. Also, execution of code can be paused for multiples or divisions of precise and familiar time durations, including seconds, milliseconds, samples or even weeks. These features create a powerful and intuitive tool for musical purposes.

2.1.3 Visual Music Programming Languages

Many of the earliest analog synthesizers allowed the individual to create and manipulate sound by physically "patching" sound generator and processor modules together in any order. This enabled a level of sophistication, experimentation, and play, which has seen a resurgence in recent years and analog and digital modular synthesis has become more affordable. Visual music programming languages make the abstraction of unit generators and processors as visual representations of

modules in software, with inputs and outputs that can be interconnected. The data flow is shown directly, which it makes programming music much more approachable. The remainder of this section looks at a few common visual music programming languages.
(Reference Figure 5 and Figure 6 for examples of how these visual programming languages look.)

2.1.3.1 Max/MSP and PureData

Max/MSP is currently one of the most popular audio programming languages. The original version of Max was written by Miller Puckette at IRCAM in the mid-1980s. Originally it had no real-time audio processing/generation. When real-time audio manipulation was added it was renamed Max/MSP. Video and image manipulation was eventually added in a new internal library called Jitter. Since Max became a commercial product owned by a company called Cycling '74, Puckette developed a language called PureData (Pd), which works similarly to Max/MSP, but is open source. Both Max and Pd are widely used.

These languages are used in almost all computer music academic institutions to some degree. Their visual approach and ease of use made them popular with composers and other kinds of artists without technical backgrounds. With the release of Max for Live, as an add-on for the very popular DAW Ableton Live, it reached more electronic musicians. Max for Live accomplishes many of the objectives that ChucK Racks aims for, but its visual programming approach limits both the transferable skills from other programming languages and the ability to define complex logical behavior.

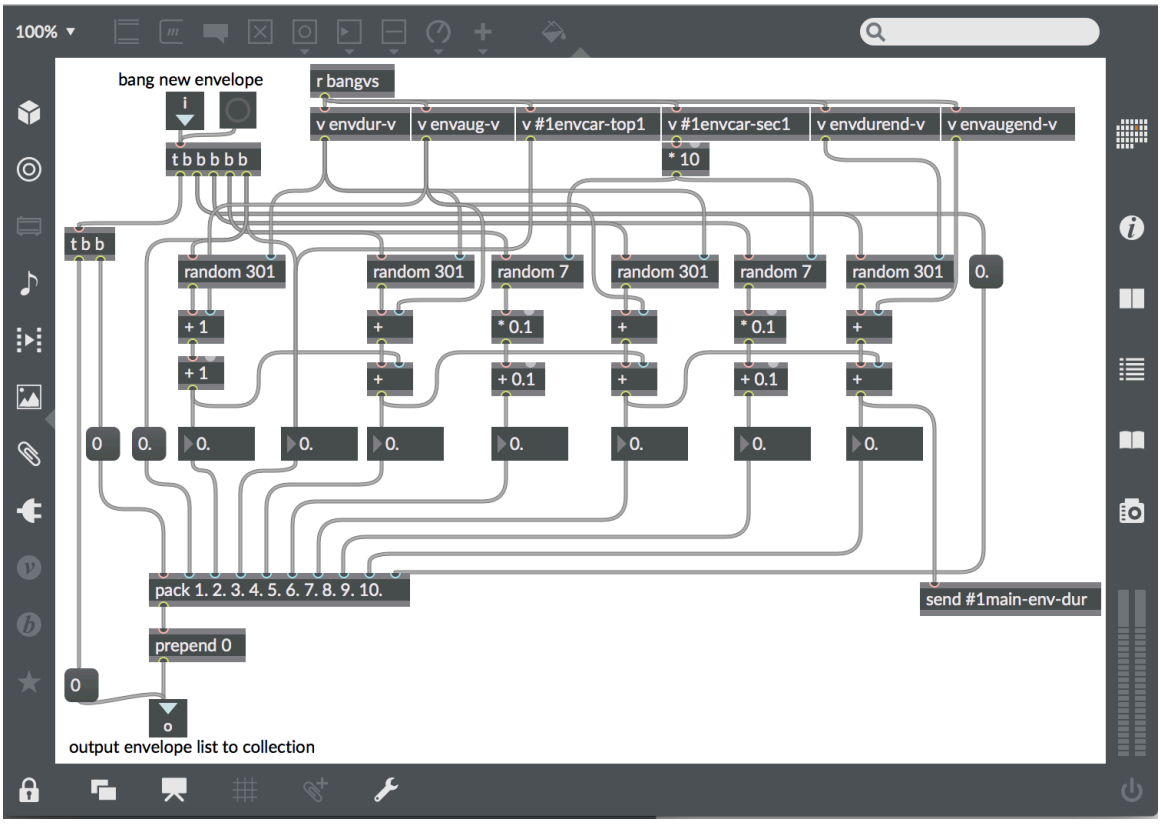

Figure 5. Max/MSP patch

2.1.3.2 Reaktor

Reaktor was created by the German company Native Instruments. It also uses modules that have specific functions, like oscillators and filters. But unlike Max, Reaktor was primarily made for sound synthesis and processing and has an extremely good sound quality. Reaktor can be used both as a standalone program or as a plugin inside a DAW.

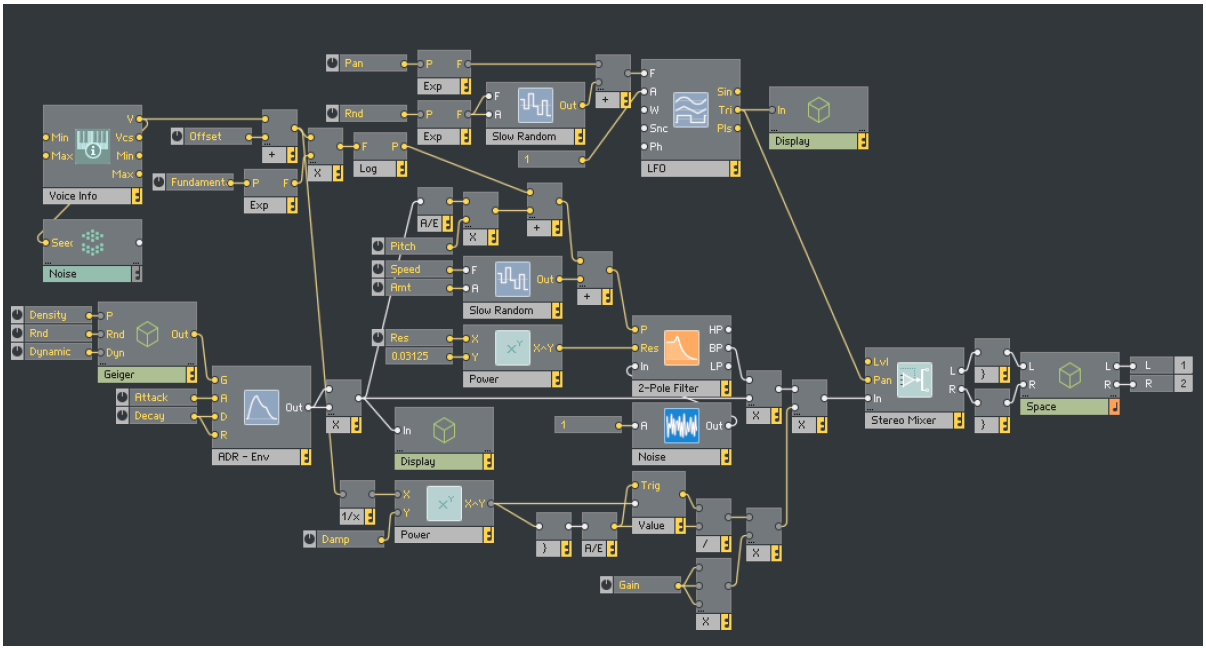

Figure 6. Reaktor patch for preset instrument "Space Drone"

2.2 History of DAWs

Modern DAWs are a culmination of various technologies developed for different sound related purposes. They are led by industry more than academia. Film and music producers wanted more reliable ways of recording, editing and archiving audio than tape, which led to the development of digital recording. Electronic instrument manufacturers wanted a standard way of having their instruments communicate with each other, which led to the invention of MIDI.

2.2.1 MIDI and Digital Sequencing

A significant development that defined the path of DAW was the creation of MIDI. MIDI (which stands for Musical Instrument Digital Interface) is a protocol for transmitting musical information between musical devices, like synthesizers, samplers and drum machines. It was designed by Dave Smith, of Sequential Circuits, in the early 1980s. By the end of 1983, most synthesizer manufacturers included MIDI interfaces as a standard communications feature in their products. (Manning 2004) This protocol is still almost universally used today, both in hardware and software. It's usually the core of sequencing in a DAW.

It was because of the MIDI protocol that proto-DAW started existing on some of the first home computers. Computers like the Yamaha CX5 Music Computer (1984) and the Atari 520ST (1985) included built-in MIDI ports. (Leider 2004) These computers had either very limited sound

cards or none at all, but they were capable of sequencing and controlling external MIDI hardware instruments. Musicians could use their computer as the brain of their equipment, which could be mixed by an external hardware mixer and recorded using a tape recorder, for example. Some of the important software from this era are C-Lab/Emagic Creator/Notator, which eventually evolved into Logic and Cubase. Both were released for the Atari ST and on Apple Macintosh shortly after.

2.2.1.1 Logic

Emagic Notator was a MIDI sequencer for several early home computers. Its sequences were based on "patterns," which were short sequences that could be looped over on a timeline. There were two ways of inputting the information for each sequenced pattern. One was based on a vertical "tracker" style input field (discussed in 2.2.1.3), and the other one was based on traditional western musical notation. This software was eventually renamed Logic. It is now owned and supported by Apple and it evolved to have all the features of any modern DAW.

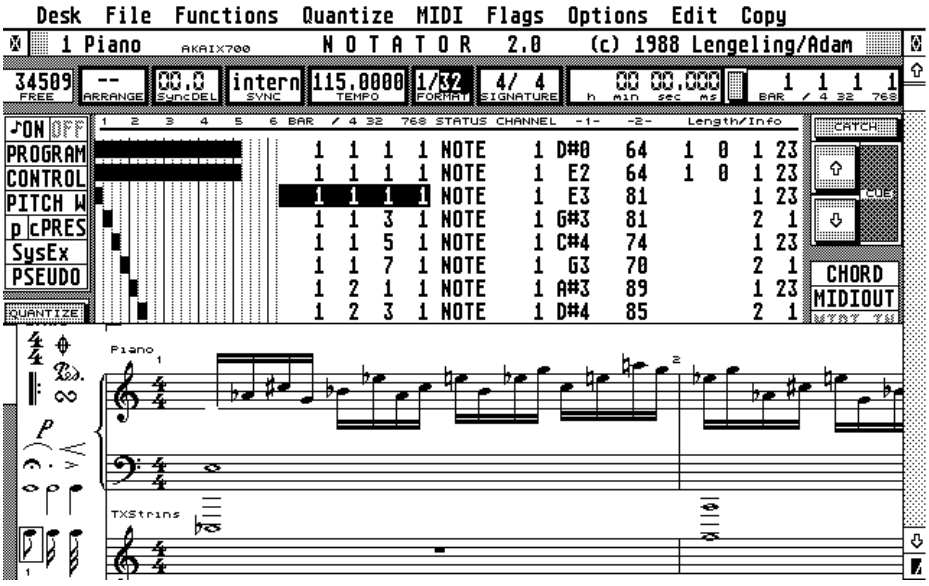

Figure 7. Early version of Notator (Logic)

2.2.1.2 Cubase

The first version of Cubase, created by the German company Steinberg, was released in 1989 on the Atari ST and in 1990 for the Macintosh. It was a MIDI sequencer, with the innovative way of seeing MIDI note events in a vertical timeline using a piano roll (shown in Figure 8), which was

much more user-friendly to non-traditional musicians compared to Logic's musical notation style (shown in Figure 7). This way of displaying MIDI notes became so popular that every major DAW ended up implementing it.

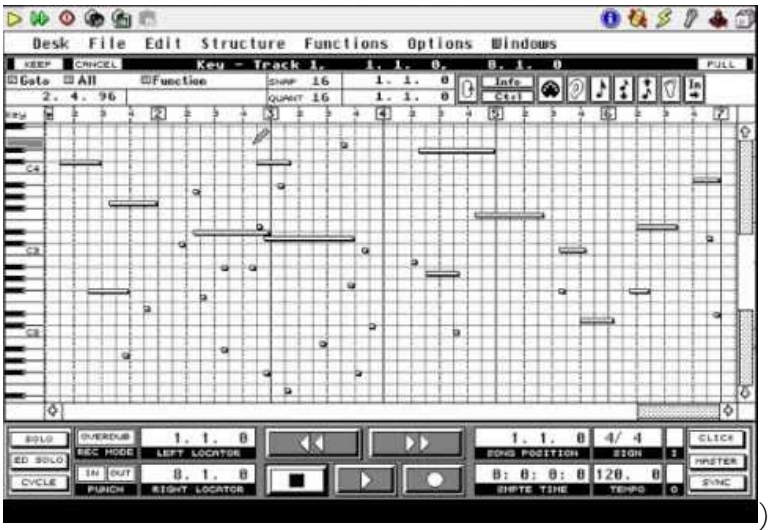

Figure 8. First piano roll: Early Cubase version running on the original Macintosh

2.2.1.3 Trackers

Some of the first interfaces for sequencing were what we know today as "trackers." The name is derived from the software "Ultimate Soundtracker," released for the Commodore Amiga in 1987. The Amiga had the ability to play low-quality short sound samples. Soundtracker presented several channels where audio note information could be sequenced, to play and control parameters of these short samples. The sequencer consisted of a spread sheet with spaces for note and parameter information. Each row represents a different step in time, and each column group represents a different channel (As shown in Figure 1). This way of sequencing was widely used in video-game music and early electronic dance music, but trackers mostly disappeared when more intuitive graphical interfaces such as the one found in Cubase became available. Still, some users prefer the deep sequencing and audio manipulation possibilities of trackers. A screenshot a modern tracker, Renoise, is presented in Figure 9.

Figure 9. Renoise: a modern tracker

2.2.2 Recording and Audio Editing

The advantages of using digital audio over tape and other analog techniques were recognized since the 1960s. Recording, manipulation, and reproduction of high-quality digital audio were a recognized need, so many companies started manufacturing stand-alone audio interfaces for some of these early home computers. Some examples are the Commodore Amiga 0+ and the Apple II. (Leider 2004) In 1991, Digidesign released its first version of Pro Tools as a suite of software and hardware for the Apple Macintosh computer. Pro Tools became the industry standard for digital audio recording. Even though this version of Pro Tools had mixing capabilities in it, most audio production professionals still preferred to work in a hybrid environment. Normally these hybrid environments used Pro Tools for recording and editing, and hardware mixers and processors for mixing and mastering. Pro Tools pave the way for modern DAWs where everything can be done using only a computer. "The basis of the modern digital audio workstation's very existence is the idea that sounds – where recorded or synthesized can be edited and mixed entirely in the digital domain. " (Leider 2004)

2.2.3 The Modern DAW

Some of the most used modern DAWs include Ableton Live, Logic Pro, FL Studio, Pro Tools and Steinberg Cubase. While each of them started as a quite distinct product, they ended up influencing each other to the point where they all have very similar features. (A comparison of how similar their interfaces are is shown in Figure 10.) All of them have multitrack recording an audio editing, which was Pro Tools original stronghold. All of them have MIDI sequencing using a piano roll, which was first seen in Cubase. All of them have extensive sample manipulation like time-stretching and re-pitching, which was a big part of Ableton Live's original appeal. All of them support external plugins. Any of these DAWs has all the tools for a musician to make an entire song in an intuitive way, and that's why they are so popular.

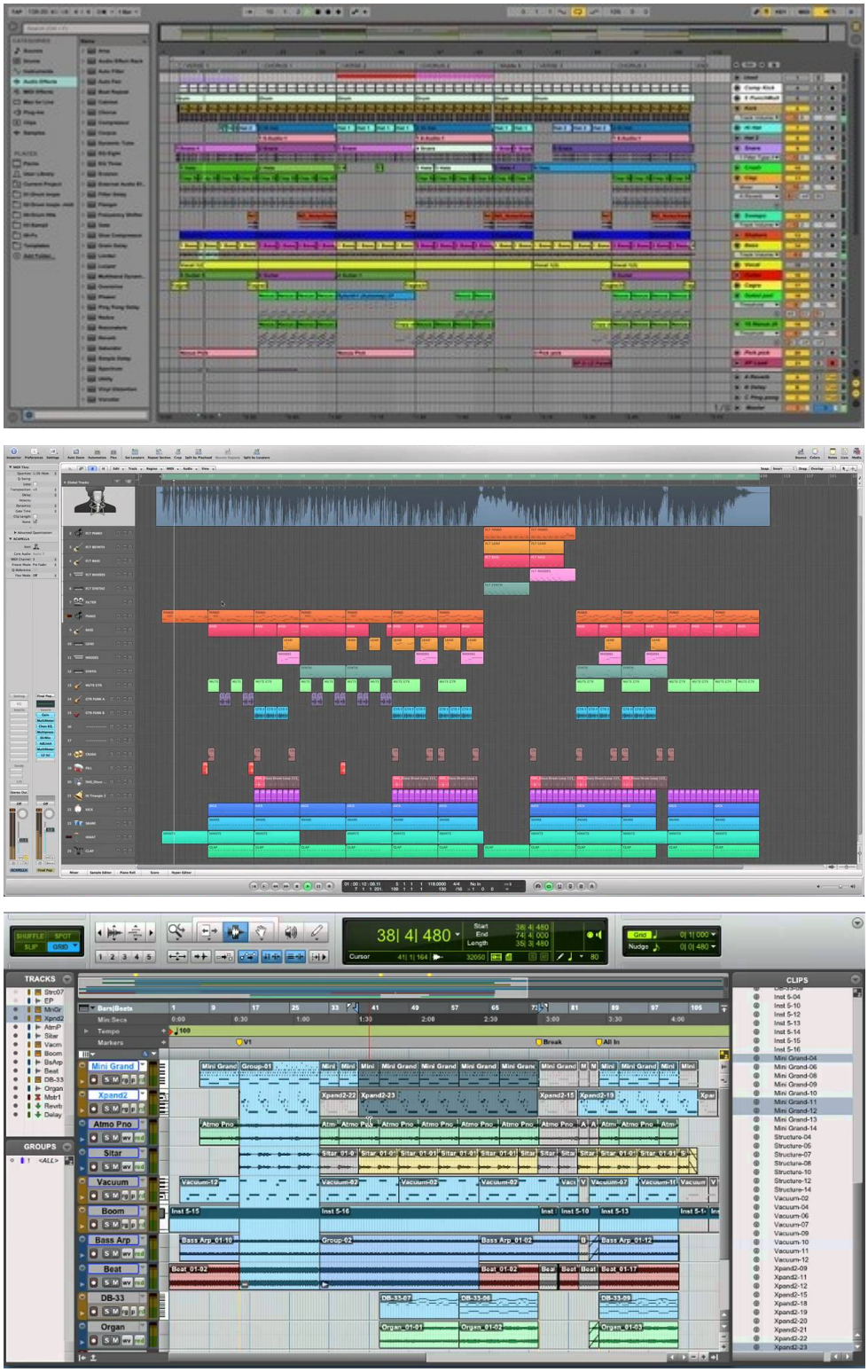

Figure 10. Comparison of the "arrangement" view of Ableton Live, Pro Tools and Logic

2.3 Plugins

During the early 1990s, software companies started developing packages that could be sold as self-contained "plug-ins" for audio software. (Dean 2009). The format created by Steinberg, VST (Which stands for Virtual Studio Technology) was very influential in this development. It was open, which meant that anybody could use the format without having to pay royalties. The appearance of the VST format helped create a vibrant market place for all sorts of audio plugins, including simulations of classic and expensive hardware synthesizers, compressors and equalizers, to new complex digital processes impossible in the analog domain. The availability of this new technology contributed to producers moving away from using hardware to switch completely to an in-the-box production workflow.

2.4 Summary

In this chapter, some of the history of the various musical technologies that culminated in the current computer music making environments was presented. This included early computer music programming languages as well as current musical programming languages. Next, important steps in the development of music technology and personal computers and their software counterparts that came together to form the modern DAW was discussed. Lastly, a brief look at audio plug-in technology which helped produce the ecosystem for a tool like ChucK Racks to exist was also discussed.

Chapter 3

ChucK Racks

Since the beginning of Computer Music, Max Mathews realized that on principle computers could generate any sound (Dean 2009). How do we tell the computer what sounds to create then?

There are two broad approaches that we will discuss in this chapter: Using computer programming and using a DAW. Each of them has advantages and disadvantages. ChucK Racks attempts to combine both of these approaches into a single workflow, providing the advantages of both.

3.1 Programming-based Music Making

The first approach to making music with a computer is using text-based programming languages. Because of the processing power and low-resolution interfaces of early computers, this was essentially the only approach. A musician would use code to manipulate elements of the sound on a very low level, including sample by sample of a digital audio stream, or somewhat higher-level modules (e.g. unit generators). For input, the user would normally use a typewriter-style computer keyboard, and they would input and manipulate numerical values representing parameters such level, frequency or time. Modern textual programming languages such as ChucK and SuperCollider allow other types of physical inputs such as MIDI controllers and keyboards, or microcontrollers with sensors, which can be used to control programs developed through coding. In modern visual programming languages for music, a user can use a mouse to connect virtual representations of the modules using virtual cables on a screen.

Using programming to create music offers advanced capabilities for encoding musical processes. It can allow very complex developments in rhythm, pitch, scales, tuning, and timbre. This way of making computer music has always been more linked to academia, requiring more training and abstraction. While this method provides near limitless control over sound, it is very different from playing traditional instruments. Since the input and feedback cycle is virtually instantaneous when using a traditional instrument (e.g. Pushing a key on a piano and immediately

hearing a sound), music coding requires a completely different approach from a creative process perspective, and traditional music knowledge is not very transferrable to the programming domain. One of these programming environments can be overwhelming to a new user, and will, therefore, be approached with reluctance or even immediate rejection. Completely alien environments might lead to experimentation, but it will take a lot of learning and practice from the artist to develop that experimentation into a finished piece.

3.2 DAW-based Music Making

As explored in Chapter 2, Digital Audio Workstations are the culmination of multiple technologies for different purposes in the music technology world. It offers very direct and visual ways of doing multiple necessary processes in the production of a musical piece. From recording and sequencing to manipulating synthesizers and mixers, all these functions have intuitive visual interfaces.

Regarding sequencing, DAWs give precise control and visualization of sequential events with respect to time, such as laying out a polyphonic musical progression or rhythm. Looping sequences or audio clips is extremely easy. So is drawing precise automation curves to through composing parameter changes over the duration of a piece.

These tools, however, are aimed to speed up the workflow of a certain type of music within certain constraints. They're aimed toward traditional musical compositional systems, with few options for randomness or complex logical evolving variations. For example, a DAW's piano roll (and MIDI in its case) can't draw notes for frequencies outside the standardized defined twelve notes in equal temperament. Sequencing probabilities of notes happening instead of a precise score is not possible. So is defining complex logical behavior, such as the ones used for algorithmic composition.

3.3 The Best of Both Worlds: Combining Approaches

Many musicians throughout history have created their own instruments and processors, and they can greatly influence the final sound of a composition, giving it a more unique and personal sound. A musician limiting themselves to only using a DAW and tools made by other people could be considered a huge missed opportunity, considering the possibilities a computer can offer. While modern DAWs offer options to create custom macros and presets for existing instruments, they're still limited compared to computer music programming. Computer music programming on the

other hand, offers a more affordable and efficient way of creating custom tools and processes, compared to the difficulties of working with hardware engineering.

Another factor to have into account when creating ChucK Racks is the facilitation of *flow*. Flow is a mental state first described by psychologist Mihály Csíkszentmihályi, which is characterized by a deep concentration in the activity one is doing. (Csikszentmihalyi 2008) Flow can be extremely important for the creative process since it's in that state where artists often describe themselves doing their best work. When somebody is working on a creative endeavor, there's an *iteration loop* (Figure 11) happening between the artist and the medium, and if the time between each step in the iteration loop is short enough, flow is more likely to occur. When designing tools for artists, it's important to consider what could stop the immersion of their creative processes. Having to step out of their current state to solve a technical issue or solve a problem, especially multiple times, will pause the iteration loop and most definitely interrupt flow.

For users who are familiar with music programming languages, there are some current alternatives of integrating both, but they lack a unified workflow or have limited possibilities. The remainder of this section will look at a few approaches.

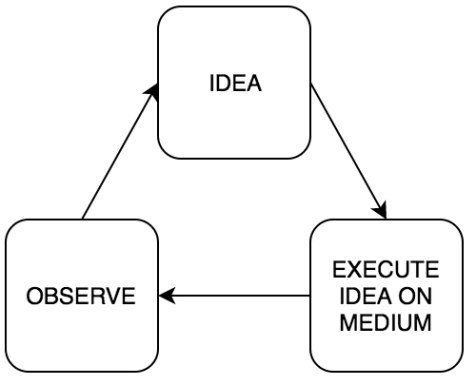

Figure 11. Art Creation Iteration Loop

3.3.1 Combining as Independent Processes

The first and easier way of integrating these two environments is by combining them as two independent processes. A music programming environment like ChucK or SuperCollider can be used to generate content that can later be imported into a DAW. For example, several oscillators in ChucK with various processes manipulating their parameters can generate sound that can be

recorded as an audio file. It can then be imported into a DAW and used inside an audio channel or inside a sampler, just how other samples are normally used. An audio file could also be generated from some sound source inside a DAW and then imported into ChucK to be processed and manipulated. A programming environment could also be used to generate note information by writing MIDI note messages to a MIDI text file and then importing it into a DAW's MIDI channel. This would be a suitable way of creating algorithmic compositions.

This approach, while valid, limits the possibilities of interaction between the two environments. It separates the two into two distinct processes, which interrupts flow. It also makes it impossible for live performance.

3.3.2 Routing audio signals and control messages

This method uses external applications to send and receive information from one environment into another. For example, Mac OS includes the application Audio MIDI Setup which can be used to create virtual MIDI ports. Virtual inputs and outputs from ChucK and Ableton Live can be used to share note information and control messages. OSC (Open Sound Control) can also be used to share other control information, though currently, most DAWs only support MIDI and not OSC.

There are applications such as Soundflower that can be used to route audio between applications. Using this the output of ChucK could be sent to Soundflower, and Soundflower could be selected as an input inside of a DAW.

There are however many shortcomings to these approaches. Firstly, set up is cumbersome and takes time, which will likely have to be repeated every time a new project is started, or when the computer is turned on, or even for every audio track added if the user wants different processes on different tracks. If the user is in a creative mood, dealing with all these technical issues before being able to make anything creative often becomes be a hindrance (this workflow problem is discussed in detail in 3.4.1). In the case of sharing MIDI for algorithmic composition, only limited synchronization is shared between the applications, and it is often impossible to seek to a position within the composition in the DAW and have the other application seek to the same location or compositional state. Furthermore, when sharing inter-app audio via Soundflower (or similar software) audio latency (an audible delay between when sound is generated, processed and reproduced) can be a huge problem, and audio delay composition which is typically used by DAWs to keep all of the tracks in sync with one another is also not possible.

3.3.3 Native Integrations: Max For Live and Reaktor

Max For Live has bridged the gap between programmatic and DAW-based music making to a significant degree. It brings visual programming to the masses by integrating Max/MSP directly within Ableton Live. However, though Ableton Live is a popular DAW, there are many people out there who use other popular DAWs like Logic, Pro Tools, Digital Performer, and countless others. On the other hand, Native Instruments' Reaktor, which is also a visual programming language can be used as a VST/Audio Unit plugin in any DAW.

While many praise visual programming languages for ease and accessibility, there are also some drawbacks of visual programming languages compared to text-based programming languages. One of them is transferable skills. There is little to no correlation between visual programming languages and textual languages used professionally for software development. Someone with some experience with C-like languages (C, C++ C#, Java) will be able to jump into ChucK quite easily, and vice-versa. Some ChucK code can be translated into C++ relatively easily, in case someone wants to prototype tools in ChucK Racks. Doing it in a visual programming language would mean a complete re-write.

In theory, most synthesis and musical tasks can be implemented in any text-based or visual programming language, but the languages lend themselves to different tasks. Making very complex logical behaviors and dynamically copying and iterating on objects is more easily achieved with text-based languages. It also comes down to personal preference. Many musicians prefer developing with text-based code, and the Reaktor and Max For Live options don't allow for that.

3.3.4 Writing your own plugins in C++

Audio plugins and most audio applications are written in C++. If a musician wants the flexibility of programming, writing their own tools in C++ seems like a reasonable solution, especially with common frameworks for this such RackAFX[2] and JUCE. Still, the learning curve for C++ is very steep compared to ChucK.

Even if the user is familiar with C++, using it means having to spend a lot of time dealing with things that have nothing to do with their creative vision, for example, managing memory. Making mistakes in C++ could lead to serious bugs that will crash the host application.

[2] RackAFX is a popular and easy to use framework to make audio plug-ins. It is available for free at: http://www.willpirkle.com/rackafx/

Additionally, there are serious workflow issues when working in pure C++, primarily due to the fact that it is a compiled language. This means that every time the user wants to make a change to the code, they have to disrupt the creative process, recompile the plugin, reopen the DAW, reinitialize the plugin, and so forth. As discussed further in 3.4.2, ChucK Racks attempts to solve this issue by ChucK's inherent on-the-fly programmability.

3.4 Creating a new workflow with ChucK Racks

As mentioned earlier, flow is an important part of the creative process. When creating with any medium, there's an *iteration loop*. This process is visualized in Figure 11, and it can be this described in this way: Something is created or modified using the medium, it is observed, and then it is modified again. This cycle is repeated until the artist decides that the work produced is ready, or an external force, like a deadline, requires the piece to be finished.

The fewer the number of steps between the create-observe-modify-observe cycle, the easier it is to keep creativity flowing and stay in a state of flow. In some forms of art, like painting on a canvas, that feedback is virtually instantaneous. The painter will apply the paint on the canvas and will see the result immediately. Then they will respond to that result by applying more paint, starting from the beginning of the iteration loop again.

In creative coding, immediate feedback is usually impossible almost by definition. A series of abstract instructions expressed in the language by the programmer or artist must be translated by the machine into a set of instructions that it can understand. This process is usually called compilation and it can take anywhere from a few seconds to minutes or longer. In the context of creative coding, it is desirable to accelerate the iteration process by reducing the steps and time required between writing code, compiling the code, observation the result, and going back to tweak the code.

3.4.1 Making ChucK feel like just another part of a DAW

Millions of musicians are familiar with the workflow of their favorite DAW. What are the options for users who want to integrate the capabilities of ChucK with their DAW of choice? Imagine that someone wanted to create a filter in ChucK that processed one of the instruments in their composition, and was also modulated in sync with the tempo of their song. The DAW would have to route the audio from the desired channel(s) into ChucK. The audio would be

processed by the filter, and the resulting audio would be sent back to the DAW, to be processed further or mixed into the master channel.

To do this without ChucK Racks, one would have to use an external communications protocol like MIDI or OSC, and external audio routing application, like Soundflower. The setup alone would require a significant amount of configuration and time and interrupt the creative flow. Figure 13. C++ vs. ChucK Racks Plugin Prototyping shows the difference in number of steps required to route audio into ChucK using this external audio application approach vs. ChucK Racks. Notice how many more steps are required to reach the iteration loop using external audio.

By making ChucK work inside a native audio plugin, setting it up is just one click away. Like any other audio plugin, all that is required is dragging-and-dropping it on to a track/channel. Additionally, there is the added benefit of ChucK Racks benefitting from the familiarity of the plugin workflow that the DAW user may already be familiar with. Going inside the plugin and making changes feels natural, and makes it straightforward to integrate into the user's existing workflow. After some practice with ChucK, the user will be able to integrate coding into one of the activities that they do while working in a DAW. They will transition effortlessly between mixing, arranging, sequencing, *coding*, and synthesizing in a unified experience or creative process.

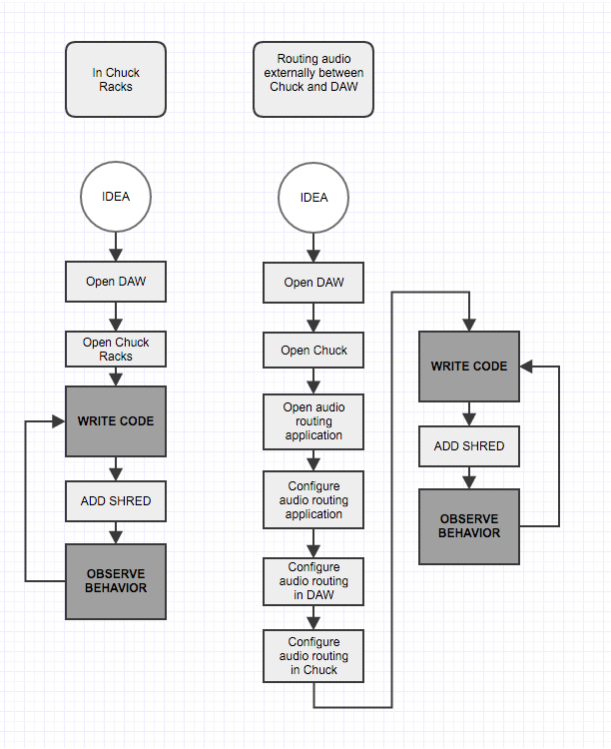

Figure 12. Using ChucK Racks vs. Manual Audio Routing

3.4.2 Accelerating rapid prototyping for Audio Plugins

Most audio plugins are written in C++, and the standard plugin formats such as VST and Audio Unit provide SDKs to develop your own plugins. The biggest advantage of using C++ is it's speed. Though the language is over 35 years old, C++ is still widely used today because of its efficiency and flexibility. This efficiency is key for audio applications which require tens of thousands of audio samples to be generated or manipulated every second, and fine control over memory access and management.

On the other hand, compared to other recent programming languages, programming in C++ is full of disadvantages, especially for artistic applications. While recent updates to core C++ specification has allowed many recent programming paradigms to be expressed natively in the language, C++ is still a relatively verbose language compared to other more specialized languages or frameworks designed for more specific tasks, or ways of working.

While the compiled nature of C++ adds many benefits to the language in the general use case, it presents significant disadvantages to the creative process and flow, getting in the way of the iterative creative loop.

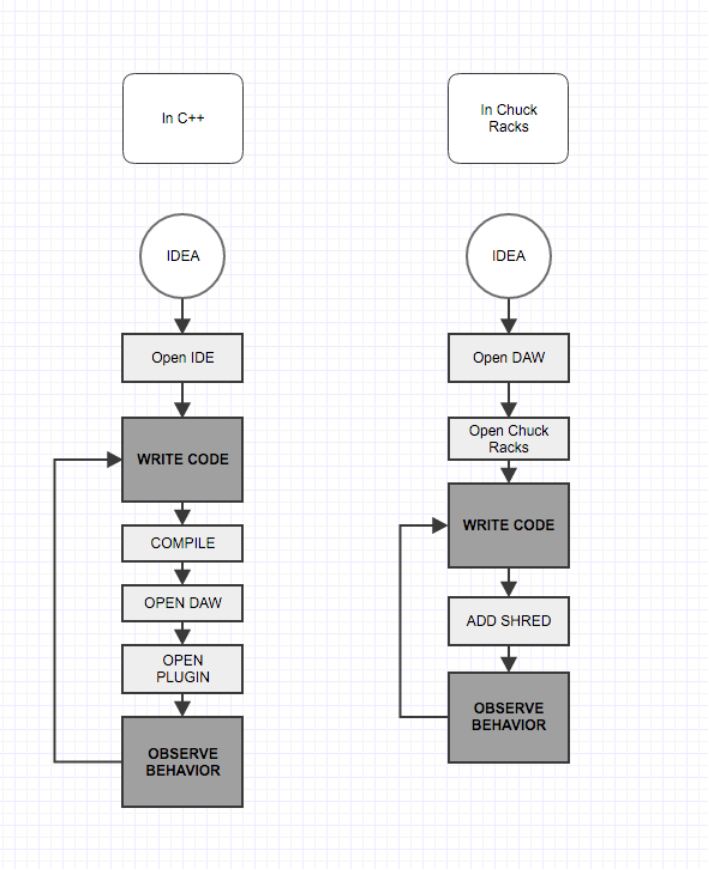

Figure 13. C++ vs. ChucK Racks Plugin Prototyping

From a setup standpoint, working in C++ means having to code in a separate IDE. After the code is written it needs to be compiled, then the DAW needs to be opened, and then the plugin has to be loaded. And these three steps have to happen every time any change is made to the code in order to observe changes. Figure 13 shows how using ChucK Racks removes these steps from the iteration loop. All the code can be written and run instantly from the ChucK Racks plugin window. This makes it ideal for fast experimentation that could eventually develop into an actual audio plugin, to be implemented in C++.

The disadvantages of working directly in C++ for creative applications is widely recognized. In the video-game industry, it is extremely common to use a scripting language, such as Lua, in conjunction with C++, to edit code without having to recompile. This served as a model for this thesis in thinking about how to integrate ChucK in the creative process of musicians and composers by integrating it with the workflow of the DAW.

Another disadvantage of using C++ is that it is a general-purpose language, not a language specifically geared toward sound. To make audio applications the user will depend on using third

party libraries or writing their own sound capabilities from scratch. On the other hand, other languages like ChucK and SuperCollider are made specifically with audio and musical concepts as first-class citizens.

Besides experimenting with high-level concepts using ChucK's unit generators, it is also useful to test low-level DSP. Using ChucK Racks and creating a Chugen object (*Chugen* is a native ChucK class that permits overloading the sample by sample processing method) makes testing DSP algorithms extremely quick and easy, without the need to recompile or take buffer sizes into account. This makes the creative iteration loop much shorter than using any other method to experiment with DSP inside of a DAW, making the process faster and more inspiring.

3.4.3 Introducing programming to DAW users

Today, the ability for a musician to work with a DAW is ubiquitous, but many still don't see or know about the possibilities of musical programming. For those that do, the learning curve often seems too steep, or it's difficult to envision how to integrate it within their compositional process. By integrating ChucK into a workflow that the user is familiar with, the barrier to entry is much significantly lowered. Even if the user doesn't have any programming skills, the downloadable version of ChucK Racks comes with several instruments, effects and other examples. The user can use these immediately and tinker with the code for an immediate and intuitive learning experience. Because of the similarity of the syntax of ChucK to other C-like languages, the acquired knowledge of working with ChucK is transferable to develop applications in widely used languages like Java, C# or C++.

3.5 Summary

In this chapter, two broad categories of contemporary computer-based music making were defined: Programming-based music making, and DAW-based music. The pros and cons of each were discussed, and existing methods to combine both were presented and shown to be unsatisfactory. ChucK Racks attempts to solve this issue by integrating these two approaches to music making with the computer. The importance of *flow* when creating tools for artists, a crucial design philosophy that is part of ChucK Racks was discussed. Finally, possible usages that emerge from the workflow created by ChucK Racks were laid out on a high level.

Chapter 4

Implementation and Design of ChucK Racks

4.1 Architecture and Implementation

ChucK Racks builds on top of two important development for the ChucK and audio programming in C++, respectively. The first is *libchuck*, a C++ library version of ChucK created by Spencer Salazar, that allows ChucK to be embedded into larger applications. The other one is JUCE (Jules' Utility Class Extensions), an open source application framework in C++ created by Julian Storer and now maintained by ROLI Ltd.

4.1.1 libchuck

ChucK Racks integrates the ChucK compiler and virtual machine using libchuck. This library provides functionality for compiling ChucK code, reporting code errors, and running or removing code from the virtual machine. Through libchuck, ChucK's virtual machine can be executed in conjunction with an existing real-time audio engine, such as that of a DAW, by requesting the desired number of samples from libchuck. Libchuck also allows a host application to load customized *chugins* (Salazar and Wang 2012*)* to extend the default functionality of ChucK. In this way, the custom language extensions *PluginHost* and *PluginParameters* were created to interact with the host (see section 4.3).

4.1.2 JUCE

JUCE was chosen over several other audio libraries for a number of reasons. The first reason was that the ChucK source code is in C++, so choosing a framework in the same language makes integrating both more straightforward. Audio plugins are also normally written in C++, for

reasons described in 3.4.2. JUCE is also cross-platform, and it was desired to eventually maintain ChucK's cross-platform support. The same application can be compiled for Mac OS and Windows, and audio plugins can be compiled into multiple formats like VST, AU, and AAX. JUCE also has an open source license for non-commercial applications, which makes it straightforward to integrate into an open source project such as ChucK Racks. JUCE also provides an extensive library for many tasks outside of audio programming, such as designing user interfaces with a vast collection of graphical and interactive components.

4.2 Plugin Panel UI Implementation

The Plugin Panel for ChucK Racks presents an interface to interact with the ChucK virtual machine, an editor to write and edit code, and the audio plugin parameter mapping system.

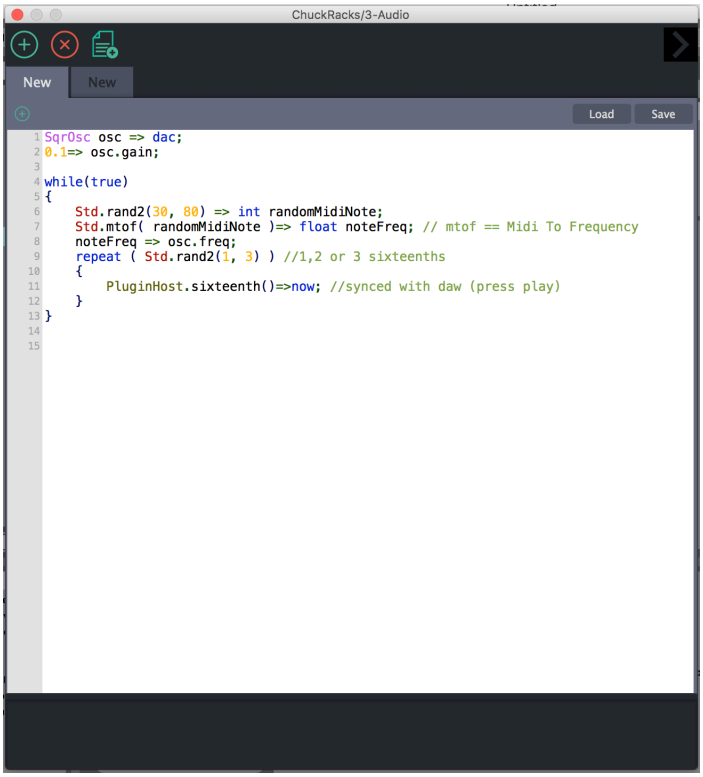

Figure 14. ChucK Racks Plugin Panel

4.2.1 Top Panel

The top panel in the plugin editor window includes the basic buttons to interact with ChucK Racks. The "add code file" button, represented by a text file icon and a plus sign, adds a ChucK

code tab to this instance of ChucK Racks, allowing the user to write new code from scratch, or load a previous saved file.

The "add all shreds" button, represented by a green plus icon, simultaneously adds the code present in each tab. The "remove all shreds" button, represented by a red X icon, removes all current shreds (code) running in the ChucK virtual machine.

Figure 15. Top panel screenshot

4.2.2 ChucK Code Tabs

Just like ChucK's popular development environment called the miniAudicle, ChucK racks can have multiple tabs where ChucK code and files can be created, opened, and edited. In this way, multiple files can be added independently or all at the same time to the ChucK virtual machine. The code editor also provides syntax highlighting that is similar to miniAudicle's including support for the newly added ChucK Racks language extensions (note the difference of coloring in *PluginHost* and *PluginParameters* in Figure 16).

```
//use this as an audio effect on a channel
adc => LPF filter => dac;
ADSR env => blackhole;
env.set(10::ms, 100::ms, 0.0, 1::ms);
100.0 => float minFreq;
3000.0 => float maxFreq; //max frequency of the filter
[1.0, 0.0, 0.1, 0.5] @=> float freqSeq[]; //intensity of
PluginHost.sixteenth()=>now;
PluginParameters.getValue( "cutoff" ) => filter.freq;
```

```
//use this as an audio effect on a channel
adc => LPF filter => dac;
ADSR env => blackhole;
env.set(10::ms, 100::ms, 0.0, 1::ms);
100.0 => float minFreq;
3000.0 => float maxFreq; //max frequency of the filter
[1.0, 0.0, 0.1, 0.5] @=> float freqSeq[]; //intensity of
PluginHost.sixteenth()=>now;
PluginParameters.getValue( "cutoff" ) => filter.freq;
```

Figure 16 . Syntax highlighting in ChucK Racks (left) vs. miniAudicle (right)

4.2.3 Debug Console

The debug console which can be dragged up from the bottom of the window shows useful messages coming from ChucK or ChucK Racks. The most useful feature is showing the error messages coming from ChucK when code fails or throws an error. It will also print events such as when shreds are added or removed (i.e. when code is added or removed from the virtual

machine). The user can also use this console to print messages that are useful for debugging, using ChucK's regular print statements "<<< >>>".

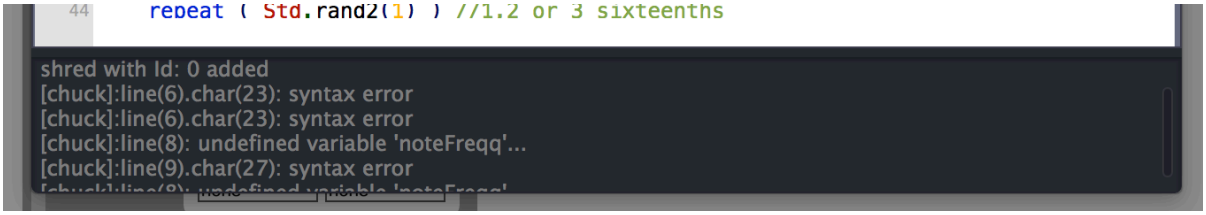

Figure 17. Debug console screenshot

4.2.4 Plugin Parameter Creator Tab

All audio plugins have the option to expose certain parameters to its host DAW. These parameters can be controlled using the DAW's automation drawing tools to orchestrate changes over time along the length of the composition. For example, opening the cutoff frequency of a filter gradually can be used to generate tension or expectation for a musical event, like the introduction of a different section of the piece. ChucK Racks has the possibility to create custom Plugin Parameters that can be accessed in ChucK code to control any value in the script's code from the DAW. How these custom plugin parameters can be used will be discussed further in section 4.3.4.

The panel window has a tab just for creating Plugin Parameters. It can be opened and closed using the arrow on the top right of the plugin window. In here there is a button (represented with a green plus icon) to add a new parameter. Under that, there is a list of all the current existing parameters. Double clicking any of them allows the user to rename it. Right clicking on any of them will show the option "Map," which allows the parameter to be mapped in current DAW's plugin window (Figure 19).

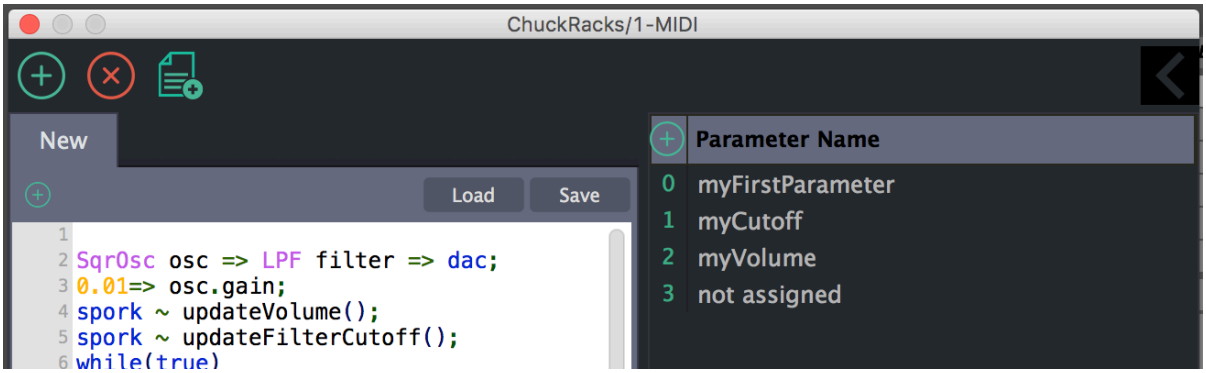

Figure 18. Parameter Tab open with a few parameters added.

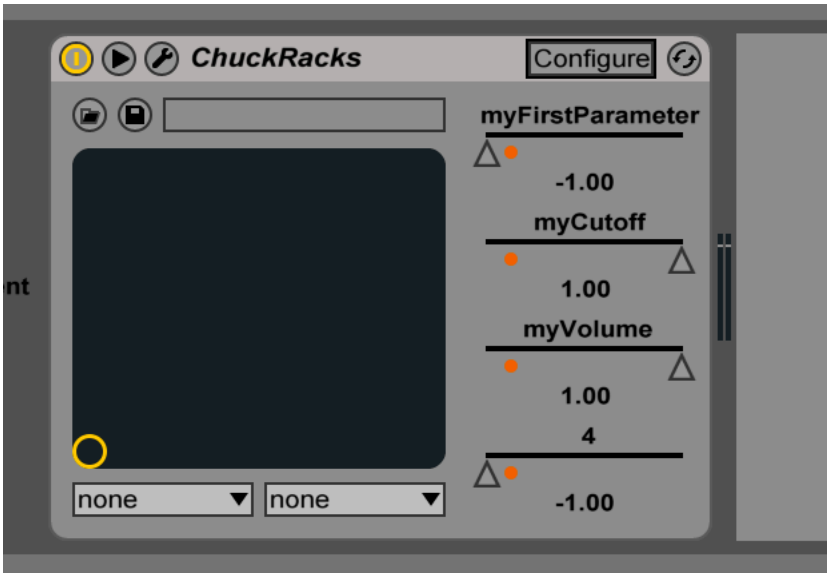

Figure 19. Plugin window with mapped parameters

4.3 Extending the ChucK Language

In order to access the information from our DAW and the plugin from ChucK code, ChucK was extended in a number of ways. New classes were created in C++ to handle all this interaction. The following details these new language extensions and how to use the new functionality when writing ChucK code in ChucK Racks to accomplish several things.

4.3.1 Inputting, Outputting, and Processing Audio

Audio can be transmitted to/from ChucK Racks just like regular standalone ChucK, allowing it to be used as an audio effect or as an instrument in the DAW. In ChucK, the "DAC" represents the main output (sound card or speakers), and "ADC" is the main input (typically a microphone). The DAC and ADC are still used in ChucK Racks; however, they represent the input and the output of the plugin instance wherever ChucK Racks is inserted into the signal chain inside a channel in the host. With this in mind, audio effects developed in ChucK can be ported to ChucK Racks without any modification whatsoever.

```
adc => LPF filter => dac;
```

Listing 1. Input and output

Listing 1 shows an example of audio going into the ChucK Racks instance which is then routed into a low-pass filter, and the audio coming out of the filter is routed into the output of the ChucK

Racks instance. This audio processed by ChucK is now available in the DAW to be further processed or mixed.

4.3.2 MIDI[3]

4.3.2.1 Receiving MIDI

For ChucK Racks to be used as an instrument, it is essential that it receives MIDI messages. It can be mainly used to receive note information, but it can also be used to transmit control messages.

By receiving MIDI note information synthesizers, samplers, sequencers, and more can be created in ChucK Racks and used as any other MIDI plugin instrument. Just drag and drop an instance of ChucK Racks on a MIDI channel, and sequence using the DAW's sequencer just how you would in any instrument. To receive MIDI ChucK Racks uses the existing chuck MIDI message class, alongside some additional methods created in PluginHost: *onMidi()* and *recvMidi()*.

```
MidiMsg msg;

while(true)
{
    PluginHost.onMidi() => now;
    while(PluginHost.recvMidi(msg))
    {
        <<< msg.data1, msg.data2, msg.data3 >>>;
    }
}
```

Listing 2. Receiving MIDI messages

In this example (Listing 2), a MIDI message object called *msg* is created. After that, an infinite loop waits for *onMidi()* events. When one of these occur, the MIDI message information is retrieved and saved into the *msg* variable. After that, the content of the MIDI message is printed to the console.

[3] MIDI has been the standard protocol for digital musical information transfer for over three decades. For more information on MIDI look at 2.2.1

4.3.2.2 Sending MIDI

ChucK also has the power of being an extremely versatile sequencer. Because of its precise timing system, very specific timings can be created, and because of its complexity as a programming language extremely intricate logical rules and randomization can be used. MIDI messages can be generated in ChucK Racks, and the output will be transmitted through the selected DAW channel. In this way, other MIDI instruments and third-party synthesizer plugins can be sequenced by ChucK Racks.

```
while(true)
{
    MidiMsg msg;

    0x90 => msg.data1;
    60 => msg.data2;
    127 => msg.data3;
    PluginHost.sendMidi(msg);

    1::second => now;
}
```

Listing 3. Sending MIDI messages

In this example, an infinite loop is creating a MIDI message, populating it with values, and sending it out to the DAW's channel where this instance of Chuck Racks is. This whole process repeats itself every second.

Because MIDI can be sent and received in the same instance of ChucK Racks, it can be used to create MIDI effects, like arpeggiators (playing a group of notes sequentially that were received simultaneously) or scale effects (quantizing notes to the closest note in a scale[4]).

4.3.3 Sharing host information

One of the main challenges that users face when trying to integrate ChucK (or any other music programming language or standalone application) with other music software is synchronizing their clocks and tempo. To address this, information has to be shared between ChucK Racks and the

[4] Most DAWs already have a "scale" MIDI effect, which can be used to convert all received MIDI notes to notes inside a musical key, such as C major. This can be used to make everything sound "in key," even from pressing random notes on a keyboard. Doing this in ChucK Racks provides more flexibility, though. These "scale" modules can be dynamic, changing and adapting depending on several factors. For example, the scales could change according to the notes received, following a predetermined chord progression, or even randomly once in a while.

DAW. A new language extension to ChucK called *PluginHost* was created to facilitate this communication.

PluginHost has several static methods to receive and send information between the DAW and ChucK Racks. There are two types of methods that *PluginHost* contain. The first type is a group of methods that return primitives and are used to query information from the host. For example, to get the host tempo you can just do:

```
PluginHost.getTempo() => float hostTempo;
```

Listing 4. Getting the host's tempo

While this could be useful in a few situations, it is not very practical for tempo syncing. It could be used to set a parallel clock in ChucK, but what if the tempo of the host changes? What if the independent clocks shift over time due to small quantization errors?

Because of this ChucK Racks uses the robust ChucK Event system. When an Event value is used with "=> now" command, execution of code will be halted until that event happens. This is very useful for musical purposes. When certain events happen in the plugin instance in the host, ChucK Events that are triggered can be received by any shred in ChucK Racks waiting for it. For example, a good way of syncing something in ChucK Racks using sixteenth notes would be using PluginHost.nextSixteenth(). This method returns an event every time the next sixteenth note happens according to the host's transport.

```
SqrOsc osc => dac;
[50, 60, 55, 60, 63, 48, 60, 63] @=> int melody[];
while(true)
{
    for(0 => int i; i<8; i++)
    {
        Std.mtof(melody[i]) => osc.freq;
        PluginHost.nextSixteenth() => now;
    }
}
```

Listing 5. Playing a Sequence

The code in Listing 5 shows a square oscillator object being created and routed into the DAC. After that, and array of integers called *melody* is created and filled with MIDI note numbers. The infinite while true loop contains a for loop that steps trough the melody array, converts the current MIDI note into a frequency value, and sets the frequency of the oscillator to that value.

Next the PluginHost.nextSixteenth() function call waits for the next sixteenth note event from the host, which happens four times per beat while the host transport is playing. This a quick example of playing a melody that's synced to the host tempo.

There's also other functions to get events from the host, like PluginHost.onPlay(), PluginHost.onStop() and PluginHost.onBeat(). A full description of the PluginHost API is provided in Table 2.

Another way of syncing ChucK programs to the DAW is using durations instead of events. Functions like PluginHost.sixteenthDur() (Shown in Table 1) will return a duration based on the current tempo of the host at the time the function is called. This method is more consistent on how ChucK uses durations in general, but it allows for the possibility of the ChucK program going out of sync in case the tempo of the host tempo changes. This would be equivalent to the drummer and bassist of a band speeding up, but the guitarist keeping the same tempo. Typically the musicians all adjust and synchronize with one another. An appropriate approach would be using both events and durations, using an event to re-sync periodically. That being said, it is common for the host tempo to remain constant for an entire piece of music, and so it is also possible that synchronization isn't necessary at all, or only needed once at the start. In either case, ChucK Racks provides mechanisms to cover all use cases.

Table 1. Overview of PluginHost

Function	*Info*
float getTempo ()	Returns the current tempo in Beats-Per-Minute
int timeSigUpper()	Returns the number of beats in a bar (3 in 3/4)
int timeSigLower()	Returns the note value of one beat (the beat unit, 4 in 3/4)
int isPlaying ()	Returns 1 is host is currently playing, 0 if it is stopped
float pos()	Returns play position (in quarter notes)
float posInBeat()	Returns play position in current between 0.0 and 0.9999
float posInBar()	Returns play position in current bar between 0 and number of quarter notes in time-signature (e.g 0.0 - 3.999 in 4/4)
float posLastBarStart ()	Returns start position of the last bar in quarter notes
dur barDur()	Returns the length of a bar

dur halfDur()	Returns the length of a half note
dur quarterDur()	Returns the length of a quarter note
dur eighthDur()	Returns the length of an eighth note
dur sixteenthDur()	Returns the length of a sixteenth note
void sendMidi (MidiMsg msg)	Sends a midi message *msg*
int recvMidi(MidiMsg msg)	Used to get the last available MIDI message from the host. If a message is received, it returns 1

4.3.4 Getting Plugin Parameter Information

A big advantage of working with DAWs and compatible plugins is the possibility for the plugin to expose the values of some internal parameters. In this way, the DAW can access those parameters, and they can be automated or controlled with a MIDI controller. A way of interacting with custom parameters was included into ChucK Racks, making it possible to modify the values of parameters inside the code/ChucK program during playback or live. Since automation can be quantized by the DAW's timing grid, getting time synced modulations is also extremely easy. Parameter automation can also be used to modify the program over the length of an entire song, making ChucK easier to integrate into the context of the arrangement, which is very difficult to do in any other music programming scenario. Parameter automation can also be used for live performances since the "add" and "remove" shred buttons can be mapped too. Continuous parameters could also be controlled using knobs or sliders of a MIDI controller.

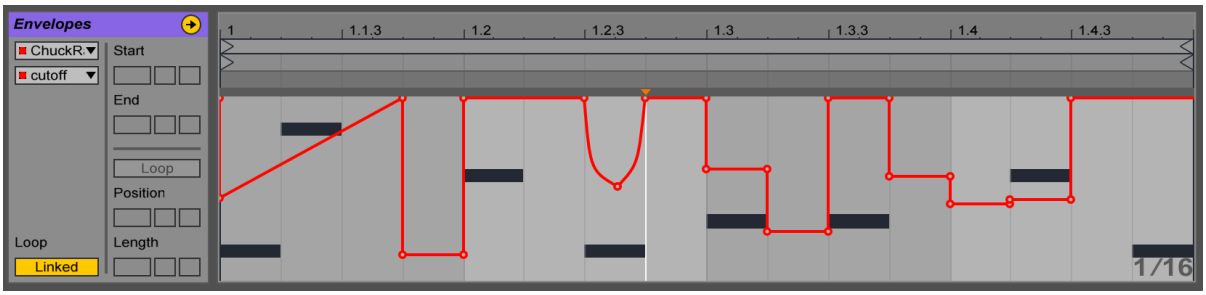

Figure 20. Automating a ChucK Racks parameter called "cutoff" in Ableton Live

4.3.4.1 Getting Parameter Values

Plugin parameters can be created and edited in the Plugin Parameter Tab in the main plugin window (see 4.2.4). In order to access the value inside the ChucK code, the user can use the function *float PluginParameters.getValue(string nameOfParameter)*.

```
PluginParameters.getValue( "volume" ) => osc.gain;
```

Listing 6. Getting the value of a parameter called "volume"

Listing 6 shows an example of how this would be used. In this case, the value of a parameter called "volume" is being used to set the gain of an oscillator. The PluginParameters.getValue() function will always return a float number between 0 and 1. This is useful for a field such as Oscilator.gain, which expects a value between 0 and 1, but to use this on other types of ranges, we need to convert the value using some math on the ChucK side. Listing 7 shows an example of mapping a parameter

called *cutoff* to the cutoff frequency of a lowpass filter called *filter*. The parameter's value is first skewed exponentially, before finally converting its range to be between 50 and 10000.

```
//values from parameters are between 0 and 1
PluginParameters.getValue( "cutoff" ) => float valueFromParam;

//since it's for frequency we want an exponential response
valueFromParam * valueFromParam => float expValue

//we have to scale it to the desired range of the filter (in Hz)
Std.scalef( expValue, 0.0, 1.0, 50.0, 10000.0 ) => filter.freq;
```

Listing 7. Retrieving and scaling a parameter value

Table 2. Overview of PluginHost Events

Event	*Info*
onPlay ()	Triggered when host transport starts playing
onStop ()	Triggered when host transport stops playing
onMidi ()	Triggered when new midi event is available
nextBar()	Triggered on the start of the next bar
nextHalf()	Triggered on the next half note
nextQuarter ()	Triggered on the next quarter note
nextEighth ()	Triggered on the next eighth note
nextSixteenth ()	Triggered on the next sixteenth note

Chapter 5

ChucK Racks in Practice

The following sections describes various examples of what can be created using ChucK Racks. These examples are provided in the Examples folder that comes with the ChucK Racks installer.

5.1 MIDI Strummer

This example shows the possibilities of MIDI processing in ChucK Racks since it uses both MIDI input and output. This program works similarly to a typical arpeggiator, where a group of played MIDI notes are stored and are played back sequentially at a specific length. In this case, instead of playing the notes sequentially, the patch will play all the stored notes at the same time following the values in the *velocitySequence* array defined in the first line. When stepping through the *velocitySequence* in a time-synced fashion, the current cached array of notes, or "chord," will be played any time the value of in the current index of *velocitySequence* is larger than 0, and that value will be assigned to the velocity of all the played notes.

The point of this patch is to create a repeating "groove" with different accents and rests independently of the note information by editing the values in *velocitySequence* before running the code. While the code is running, the user can play in chords using a MIDI keyboard or draw them in using the host's piano roll. The inputted chords will be played using the same rhythmic groove, just how, for example, a funk rhythm guitar part would work.

Since this entire patch works only by manipulating MIDI, the sound generation is completely independent, and any virtual instrument inside of the DAW can be used.

In the example, there are 16 values in the default sequence to allow for a pattern of 1-bar divided in 16ths, but it can include any number of steps, including odd numbers to create poly-metric musical results.

Listing 8. MIDI Strummer Part 1 shows the sequence filled with the int values that can be modified. The main loop of the steps through the sequence endlessly, and calls a function to play the MIDI notes if the value is not 0. The function called *midiMessageCatchProcess* is a sporked

(parallel) process that listens to the MIDI input (using the *PluginHost* functions *onMidi()* and *recvMidi(MidiMsg msg)* described in Chapter 4) and adds MIDI notes to a buffer when a note-on message is received and removes them when a note-off message is received. Note that the implementation of *addNoteToBuffer()* and *removeNoteFromBuffer()* are not shown in this document.

Listing 9. MIDI Strummer Part 2 shows the implementation of the functions that play the MIDI notes and send the necessary messages to the host. *PlayNoteProcess*() is a parallel process that is sporked in the *playMidiNote()* function. It creates a MIDI message that will contain the information of the note to be played. That message is configured with a note-on event, midi note number, and velocity, and it is sent to the host using the *PluginHost.sendMidi(MidiMsg msg)* function. After a certain amount of time (a sixteenth) the message is populated with the note-off info for that same note, and it's sent again. In this way is it guaranteed that a note-off message will always be sent after a note-on happens, in order to avoid notes hanging.

This patch shows how ChucK Racks can be used to make new MIDI information out of incoming MIDI information, which can be very musical and useful. It's also a good example of how to process MIDI data inside of ChucK that can be used for internal sound generation. Additionally, it demonstrates how tempo-synced MIDI messages can be generated and sent to the host, which is very useful to make MIDI sequencers.

```
/=== edit this to change sequence ( 0-127 )======
[127, 0, 56, 12,
0, 127, 0, 0,
80, 100, 110, 120,
0, 0, 0, 127] @=> int velocitySequence[];

//================================

int midiNoteBuffer[0];
spork ~ midiMessageCatchProcess();

while(true)
{
      for( int i; i < velocitySequence.size(); i++ )
      {
            if( velocitySequence[i] != 0 )
            {
                  playAllNotesInBuffer( velocitySequence[i] );
            }
      PluginHost.nextSixteenth() => now;
      }
}

//-----midi functions-------------------------

fun void midiMessageCatchProcess()
{
      MidiMsg msg;
      while( true )
      {
            PluginHost.onMidi() => now;
            while( PluginHost.recvMidi(msg) )
            {
                  if( msg.data1 == 0x90 ) //note on
                        addNoteToBuffer( msg.data2 );
                  else if( msg.data1 == 0x80 ) //note off
                        removeNoteFromBuffer( msg.data2 );
            }
      }
}
```

Listing 8. MIDI Strummer Part 1

```
fun void playAllNotesInBuffer( int velocity )
{
    for (int i; i < midiNoteBuffer.size(); i++)
    {
        playMidiNote( midiNoteBuffer[i], velocity );
    }
}

fun void playMidiNote(int midiNote, int velocity)
{
    spork ~ playNoteProcess( midiNote, velocity );
}

fun void playNoteProcess( int midiNote, int velocity ) //only used when
sporking
{
    MidiMsg msg;

    0x90 => msg.data1; //note on
    midiNote => msg.data2;
    velocity => msg.data3;
    PluginHost.sendMidi( msg );

    PluginHost.sixteenthDur() => now;

    0x80 => msg.data1; //note off
    midiNote => msg.data2;
    0 => msg.data3;
    PluginHost.sendMidi( msg );
}
```

Listing 9. MIDI Strummer Part 2

5.2 Modulated Filter Random Sequence

This patch is made to be used as an audio effect to be put on an audio channel or after a MIDI instrument to process its audio output. This patch will process the incoming audio from the DAW through ChucK's low-pass filter. This filter is modulated by an envelope, which is being triggered by a looping randomly generated sequence of values between 0.0 and 1.0. The value affects the amount the envelope will affect the filter on that step, making the filter open more or less on different steps.

This patch can be used to give rhythmic qualities to samples or other sound sources that are not highly rhythmic. For example pads, or strings, or sustained vocals. In addition to the generated random sequence, each time the envelope is triggered its decay value is randomized within a range, making every instance of the loop slightly different which makes it sound more organic and interesting. Additionally, three parameters are exposed to be able to be modulated and automated from the DAW using the method described in 4.2.4. These parameters are "cutoffMod," "decay" and "resonance."

```
//---settings----
100.0 => float minFreq; //range of filter
7000.0 => float maxFreq;

50 => float minDecayInMs;
300 => float maxDecayInMs;

16 => int numberOfSteps;
//---settings end-----

adc => Gain volume => LPF filter => dac;
0.4 => volume.gain; //to avoid distortion
ADSR env => blackhole;
env.set(10::ms, 100::ms, 0.0, 1::ms);

float freqSeq[numberOfSteps];
//sequence generation
for (int i; i<numberOfSteps; i++)
{
      if(maybe) //50% chance
            Math.randomf() => freqSeq[i];
}

1.0 => float freqMultiplier;
1.0 => float lastModMultiplier;
0.5 => float lastDecayMultiplier;
spork ~ updateFilterEnv();

fun void updateFilterEnv()
{
    while(true)
    {
       Std.scalef( env.value(), 0.0, 1.0, minFreq, maxFreq ) *
      freqMultiplier    * lastModMultiplier => filter.freq;
        1::ms => now;
    }
}
```

Listing 10. Modulated Filter Sequence Part 1

```
//sequence playing
while(true)
{
    for( int i; i<freqSeq.cap(); i++ )
    {
        if( freqSeq[i] != 0.0 )
        {
            if( PluginParameters.getValue( "cutoffMod" ) > 0.0 )
              PluginParameters.getValue( "cutoffMod" ) => lastModMultiplier;

            if( PluginParameters.getValue( "decay" ) > 0.0 )
               PluginParameters.getValue( "decay" ) => lastDecayMultiplier;

            PluginParameters.getValue( "resonance" ) => float resonanceVal;
            if( resonanceVal > 0.0 )
                  Std.scalef( resonanceVal, 0.0, 1.0, 0.4, 3.0 ) => filter.Q;

            freqSeq[i] => freqMultiplier;
             Std.rand2f( minDecayInMs, maxDecayInMs)
            *(lastDecayMultiplier*2)::ms => env.decayTime; //random variation
            in envelope decay time
            env.keyOn(1);
        }
        PluginHost.nextSixteenth()=>now;
        env.keyOff(1);
    }
}
```

Listing 11. Modulated Filter Sequence Part 2

Chapter 6

Conclusion

6.1 Summary

The appearance of the modern DAW has greatly expanded the number of people that could make computer music, since it has greatly simplified the interaction with computers compared to music programming languages. While it streamlines the music composition process, it also limits musical possibilities to a set of pre-defined interactions. This thesis proposes a way of combining the strengths of the DAW with the flexibility of music computer programming in a simple way, using a tool that encapsulates a programming language inside of a plug-in a DAW. This tool called ChucK Racks, links a channel inside a DAW with code written in ChucK, and lets users use ChucK to generate and manipulate audio and MIDI.

6.2 Primary Contributions

ChucK Racks is an audio plugin that will help bridge the gap between the traditionally academic programming-based music making and the immensely popular DAW-based workflow. This software is free and open-source and comes with ready-to-use examples that will help inspire traditional DAW users to explore the possibilities of music programming. It will also help experienced programmers and musicians to explore a new and fast workflow where programming is just another part of the DAW-based music making progress and will make rapid prototyping of audio tools even easier.

6.3 Future Work

At the time of this writing, there are some important features that are missing in the current version of ChucK Racks. Work still needs to be done by the creators of this project to include several of these.

Right now ChucK Racks only runs on Mac OS, because of how *libchuck* is currently implemented. Since ChucK Racks was made using the multi-platform framework that is JUCE, all of the UI and plugin code and most other code should be able to be directly ported to Windows and Linux platforms, once *libchuck* is modified to run on these.

Also, *libchuck* currently supports a single instance of ChucK running on a computer. At the moment, it is impossible to run more than one instance of the ChucK Racks plugin. Ideally the user should be able to run instances of the plugin on as many channels as wanted, in order to process or generate different audio or MIDI signals that can be processed and mixed independently using the host DAW. Currently an error message is shown if the user tries to add more than one instance. Fixing this will require a major overhaul of *libchuck*, but it is possible.

There's some features present in midiAudicle that are missing too, like printing custom debug messages to console, and removing specific running shreds. These will be added in future updates.

An important factor when creating this tool was making if free and open source, just how the original ChucK programming language is. The entirety of the project source files is available on a public repository[5] hosted on Github. In this way, other programmers outside of the core team who are interested in the project can contribute with new features and bug fixes. This will also hopefully extend the lifetime of this software and will make it easier to keep being updated for years to come.

6.4 Final Thoughts

This thesis explored the possibilities of creating a new workflow where coding and DAW-based music composition are intertwined in a way that hasn't been possible before. ChucK Racks is a tool that not only attempts to integrate these traditionally segregated fields of computer music making, but it does is in a way that is easy, fun, and essentially, inspiring. Hopefully this project

[5] ChucK Racks public online repository: https://github.com/mtiid/chuck-racks/

will help music technologists, music coders and electronic musicians explore new ideas in a faster and a more natural flowing way.

Bibliography

"Client vs Server." n.d. Accessed April 20, 2017. http://doc.sccode.org/Guides/ClientVsServer.html.

Collins, Nick, Margaret Schedel, and Scott Wilson. 2013. *Electronic Music*. Cambridge University Press.

Csikszentmihalyi, Mihaly. 2008. *Flow: The Psychology of Optimal Experience*. 1 edition. New York: Harper Perennial Modern Classics.

Dean, Roger T., ed. 2009. *The Oxford Handbook of Computer Music*. 1 edition. Oxford University Press.

Leider, Colby N. 2004. *Digital Audio Workstation*. 1 edition. McGraw-Hill Education TAB.

Manning, Peter. 2004. *Electronic and Computer Music*. Revised edition. Oxford University Press.

M Wright, A Freed. 1997. "Open SoundControl: A New Protocol for Communicating with Sound Synthesizers." In . http://cnmat.org/ICMC97/papers-html/OpenSoundControl.html.

Salazar, Spencer, and Ge Wang. 2012. "Chugens, Chubgraphs, ChuGins: 3 Tiers for Extending ChucK." In *ICMC*. http://www.cs.princeton.edu/~prc/ChucKU/CalArtsMar2014/ExtendChucK/ChugensChubgraphsChuGins2012.pdf.

Wang, Ge. 2008. *The Chuck Audio Programming Language. a Strongly-Timed and on-the-Fly Environ/Mentality*. Princeton University. http://dl.acm.org/citation.cfm?id=1559215.