FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

FEUP

Real-time Audiovisual and Interactive Applications for Desktop and Mobile Platforms

Inês Vale Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Luís Teixeira (PhD)

February, 2012

Real-time Audiovisual and Interactive Applications for Desktop and Mobile Platforms

Inês Vale Ferreira

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: Jorge Barbosa, PhD (Assistant Professor, FEUP)
External Examiner: Pedro Moreira, PhD (Adjunct Professor, IPVC)
Supervisor: Luís Teixeira, PhD (Assistant Professor, FEUP)

February, 2012

Abstract

The computer, since its first appearance, has been seen by many as a new expressive instrument. The best way to use it, with that purpose is to speak in its own language.

There are many tools that help and simplify the development of creative projects using computer programming. However, these tools neglect the audio component, specially audio analysis capabilities.

The present study was developed with the objective to fill this gap. Starting from the definition of "audiovisual and interactive applications for desktop and mobile platforms", it presents some tools for the development of these applications and discusses a solution for their flaws.

In order to achieve the proposed objective, various audio tools with sound analysis, processing and synthesis capabilities are presented and analyzed.

From the study, we conclude that the Marsyas audio tool arises as the best candidate to complement those creative tools and, that the best way to achieve it is through an Open Sound Control (OSC) interface.

This way, OSC interface for Marsyas is the presented and discussed solution. It gathers a set of features, that makes Marsyas a reliable analysis tool that can be easily adaptable to different number of platforms and applications (all they need is just to support OSC). It also allows Marsyas to be used as an external part of the developed application making it suitable for platforms with limiting memory and processing constraints, such as mobile devices.

Resumo

O computador, desde o seu aparecimento, tem sido encarado, por muitos, como uma nova forma de expressão, e a melhor forma de o utilizar será recorrendo à sua própria linguagem.

Existem várias aplicações que auxiliam de modo simples e acessível a criação de projetos criativos, que recorrem à própria linguagem do computador, isto é, usam linguagens de programação.

Nestas ferramentas a componente de áudio é normalmente negligenciada, em particular as suas funcionalidades de análise. Com o intuito de colmatar esta falha, desenvolveu-se o presente estudo que partindo do que são aplicações audiovisuais e interativas para plataformas fixas e móveis, são analisadas algumas ferramentas para o seu desenvolvimento e apresentadas possíveis soluções.

Para atingir o objetivo proposto identificaram-se e estudaram-se várias ferramentas áudio com funções de análise, processamento e síntese de som.

No final deste estudo pode concluir-se que a ferramenta Marsyas surge como a melhor candidata para complementar as ferramentas criativas existentes, e que o melhor modo será através de uma interface *Open Sound Control* (OSC).

Deste modo, a interface OSC para o Marsyas é a proposta de solução apresentada e discutida que reúne um conjunto de potencialidades para ser uma ferramenta sólida. Esta solução não apresenta restrições ao nível da plataforma utilizada ou aplicações que a utilizam excepto enviar/receber mensagens OSC. Também permite que o Marsyas seja utilizado numa máquina diferente da aplicação, tornando-o uma boa opção para plataformas com restrições a nível de memória e capacidade de processamento, como dispositivos móveis.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Abbreviations

2D	Two dimensions
3D	Three dimensions
API	Application Programming Interface
BSD	Berkeley Software Distribution
CNMAT	Center for New Music and Audio Technologies of the University of California, Berkeley
DBN	Design by numbers
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
GLSL	OpenGL Scripting Language
GPL	Gnu Public License
I/O	Input/Output
IDE	Integrated Development Environment
IEEE	Institute of Electric and Electronic Engineers
IRCAM	Institut de Recherche et Coordination Acoustique et Musique
IP	Internet Protocol
MIDI	Musical Instrument Digital Interface
MIR	Music Information Retrieval
MSL	Marsyas Scripting Language
NIME	New Interfaces for Musical Expression
NTP	Network Time Protocol
oF	openFrameworks
OOP	Object Oriented Programming
OS	Operating System
OSC	Open Sound Control
OSS	Open-source Software
Pd	Pure Data
REST	Representational State Transfer
RDF	Resource Description
RPC	Remote Procedure Call
STK	Synthesis ToolKit
TCP	Transport Control Protocol
UDP	User Datagram Protocol
ZIPI	Zeta Instrument Processor Interface

Chapter 1

Introduction

This chapter describes the orientation, motivation and relevance of the problem in study as well there are identified the main goals.

1.1 Context

Since its origins, mankind has been searching for different ways to express itself. Art is the product (or process) of those creations which influence and affect our senses, emotions and intellect. This way of expression has been grown with the Human and our Society. There have been developed new means from painting and sculpture to photography and cinema, and each one has also evolved itself, e.g. from pre-historic music to baroque and post-rock.

There has not been an attempt to merge the Arts as significant as the Classical Greeks until the XIX century with Richard Wagner. Wagner believed that the future of all the arts laid in a combination of all of them, the Gesamtkunstwerk – "total artwork" – and lead off a new goal for the following artists: find the ultimate union of the Arts.

Since World War II it has been taken the most important steps to the urge of the personal computer. It is worth highlighting two contributions both for the technical advances and concept definition: the ENIAC, the first general-purpose fully electronic computer built by the US military and, Vannevar Bush's description of Memex as a device for mechanised private file and library that operates literally *as we may think* [Bus02].

Not long after, in the beginning of the sixties, the first steps were taken to merge art with the computer when the scientist Billy Klüver conceived the notion of collaboration between the artist and engineer for the creation of new ways of art and technology that would be harder to achieve without the participation of both. In the seventies, when Alan Kay presented the "Dynabook" as a "metamedium" that unifies all the media in only one interactive interface, the desire of the unification of the Arts has been brought to digital world.

In the forties, John Cage had evolved the former goal of the integration of the arts: it would be not only the join of all the art mediums known so far but also the join with the public that appreciates it. He integrated - eliminated - the audience in the play itself, i.e., the audience no longer exists as it is an element of the play. Immersion, the participation of the spectator becomes a key aspect of the resulting piece. Based on these concepts Allan Kaprow defined the *happenings*, an "aesthetically planned group interactivity in a composed environment". Which in turn has influenced many artists and scientists in the pursuit of new ways to interact with the computer.

This search reached its peak with the desire of creating "virtual environments" that emerged at most their public. This way, appeared the digital multimedia, fruit of those findings. Randall Packer and Ken Jordan [PJ02] identify the following concepts as its fundamental characteristics:

Integration: The combining of artistic forms and technology into a hybrid form of expression

Interaction: The ability of the user to manipulate and affect his experience of media directly, and to communicate with others through media

Hypermedia: The linking of separate media elements to one another to create a trail of personal association

Immersion: The sensation of being in a simulation or tridimensional space

Narrativity: Aesthetics and concept that result in non-linear stories

Communication technologies are always evolving with society and the society is evolving with them. Besides the internet revolution, mobile phones also had an important role since late 20th century. Now both paths have joined, as mobile phones gained the ability to connect to the Internet. A few years ago, tablets appeared and immediately got a place on the society and are expected to prevail since mobility and ubiquitous connectivity empowers the popularity of this devices. Smartphones and tablets are being mass consumed and projections from IDC[1] and Garnet[2] predict that tablets will catch PCs in sales in the next three years [Hin11].

These changes captured the attention of contemporary artists and they have included them in their artworks either as secondary or as main element of the piece. And so, makes mobile platforms also targets of this thesis.

For the matter of this thesis, the focus is not directly on digital multimedia but in the means used to develop works that fill entirely or partially in that description. So, we can summarize the main features of the resulting works in the following characteristics:

Interactive: The user behaviour influences the behaviour of the application and the other way round

Audiovisual: Both vision and hearing are stimulated simultaneously

[1] http://www.idc.com/getdoc.jsp?containerId=prUS23032211
[2] http://www.gartner.com/it/page.jsp?id=1800514

Real-time: Instant perception of the application reaction to the user behaviour

Desktop and mobile platforms: Not only aimed to the desktop platforms (e.g., personal computer) but also to the mobile platforms as mobile phones and tablets.

1.2 Problem Statement and Goals

The technological achievements previously mentioned are a fruit of the democratization of technology. By democratization, is meant that there is no need to be a specialist to operate a computer (or other computational device) and to have access to one. Technology is open: usable and affordable. There are software tools available that helps non-technical users to produce their own applications, but to take the most of the computer benefits one needs to get deeper knowledge into the computer.

In the seventies, even when computers did not have a graphical user interface there still were some important movements with this thought. Ted Nelson [Nel87], in ComputerLib/Dream Machines calls for a deeply and open understanding of the computers. A complete knowledge of computers opens our minds on how to creatively use the computer to reach our goals. Nelson, in his book, also explores the potential use of the computer as a media tool, which was controversial for the time.

A few years later, Alan Kay [Kay02] talking about the Dynabook concept says,

> the ability to 'read' a medium means you can access materials and tools generated by others. The ability to 'write' in a medium means you can generate materials and tools for others. You must have both to be literate". This idea was introduced in the Dynabook as one of its objectives to capacitate its users (children) to create their own programs.

By the same time, Myron Krueger [KGH85], develops Videoplace (an innovative digital artwork), and introduces the notion of the artist as a "composer" of interactive computer art. That implied that artists had to know how to program, and in the 1970s it was hard to achieve [3].

Computers and software have continued to evolve, as we said in the beginning of this chapter, but the same ideas continue to be applied.

John Maeda in his book Creative Code, highlights that despite software tools are easy to use, they hide the potential of the computer, (re)calling the importance of programming, of "talking" to the computer in its own language. In 1999, motivated by these thoughts Maeda created a programming language with an easy syntax meant to teach artists computational design, it was called "Design by Numbers"(or DBN).

In 2001, Casey Reas and Ben Fry, two students of Maeda, created Processing [RF07]. Processing is an IDE and a programming language similar to DBN but more powerful. It began with the purpose of teaching computer programming within a visual context but it rapidly evolved

[3] Krueger was a scientist and an artist.

to a state-of-the-art tool for the creation of professional creative projects. Its foundations are in Java, using a simplifying version of the Java language, which makes it perfect to set people off on programming and it is straightforward to use by people with a programming background.

Processing is one of the first open-source projects designed to simplify the development of graphical and interactive applications so non-programmers could create them. The applications developed with this tool are Java Applet subclasses so they can be easily exported to the Web.

Processing also helps its users to get connected with each other and sharing their work. It has a vast active community that contributes to the growing of Processing. The community has been providing a large collection of modules that enables Processing to work with additional media types like audio, video, and electronics. Processing has been so influential that its ideas have been expanded to other areas. Arduino [4] a popular electronic prototyping tool have a similar mission and its environment is an adapted version of Processing.

In 2005, Zach Lieberman and Theo Watson took creative and artistic computer expression to a "lower level", launching an open-source C++ library, called OpenFrameworks[5]. Until then, there was not a simple enough library with the bare minimum to do audio-visual creation in C++. The available libraries had several targets (game development, musical creation, etc.) but there was not any library focused on people with low programming skills that pretended to develop creative works. OpenFrameworks encapsulates many popular C++ libraries and simplifies its interface to offer a intuitive use.

Today, the offer for digital artists has enlarged and continues to grow. In 2010, the Barbarian Group released Cinder[6], a C++ library similar to OpenFrameworks, but more efficient as it pretends to take the most of the each platform and rely the least on third party libraries. Another C++ tools are Polycode[7] that can be used as standalone API or as scripting language and the fresh Pocode[8]. NodeBox[9] is an example of an application to develop Python based visuals.

Besides these text-based programming languages there are graphic-based programming languages, with older origins. Max/MSP/Jitter[10] and PureData[11] are visual programming languages for interactive music and multimedia. As both are significant tools in the audio community, they will be described further in Section 2.3. Vvvv[12] and Quartz Composer[13] also follow some of the ideas of Max and Pd but they focus on image rather than sound, though all of them offer multimedia composing.

These are just a few of the tools available, just to mention the most relevant names.

[4] http://www.arduino.cc/
[5] http://www.openframeworks.cc/
[6] http://libcinder.org/
[7] http://polycode.org/
[8] http://www.pocode.org/
[9] http://nodebox.net/
[10] http://cycling74.com/
[11] http://puredata.info/
[12] http://vvvv.org/
[13] http://developer.apple.com/technologies/mac/graphics-and-animation.html

Sound is the most significant element in human communication through speech and music. It is very important to be present in an interactive application to add more expressiveness to that work.

Most of the creative tools presented lack a fair offer on this topic, comparing to the image offer. An overall look to the list of modules (primary or contributed) can give us a rough image of this dissimilarity. Sound is just one category, whilst image is divided in various categories: 2D and 3D graphics, animation, computer vision, and physics (particles, fluids).

These audio libraries offer synthesis tools, expand the file formats supported by the tool, offer real-time audio input and output, and provide sound effects and filters. Also, for audio analysis, they mainly provide Fast Fourier Transform (FFT) algorithms. Though these algorithms are very important in analysis tasks, they just prepare the data to be transformed/analyzed by decomposing it into a series of smaller data sets.

Some libraries provide bindings with specific audio libraries like CSound (Section 2.3.1), SuperCollider (Section 2.3.6) or even with Max and Pd (Section 2.3.2). But these audio libraries focus mainly in synthesis, adding just a few analysis operations.

There is an obvious need in audio analysis. The FFT though is very useful and is the basis for many audio analysis algorithms, to construct them is a very complex task and is necessary to have knowledge on signal analysis.

In Max and PureData, there is also a deficiency on analysis and, opposed to the previous tools, it has a shortfall on image manipulation. Despite offering different functionalities, (2D/3D graphics, video, effects, physics), their use is very limited.

As we can see, among the various tools that can be found, there is not any tool that incorporates all the mentioned aspects (Section 1.1) successfully. So, the main objective of this study is to improve these tools providing a means to perform audio analysis.

Most part of the public that develops this kind of applications has few knowledge of programming or none at all, so the tools used for these projects are simple but efficient.

The goals of this thesis are:

- Explore and study current audio tools
- Identify and specify a solution for the needs identified above;
- Develop a proof-of-concept application

Despite the implementation of a solution be in the horizon of this study, a complete implementation is not our prime concern. We focus mainly in the analysis of the problem but considering its implementation. To sum up, with this study we aim to fill the audio gap without compromising simplicity.

The choice of the subject for this study holds with my personal motivations in terms of arts and music, and by spotting that in fact there is a flaw in the available tools to construct those kind of work.

1.3 Document Outline

This study is structured in five chapters. The first chapter, this one, provides the reader the background and scenario for the basis of this thesis, as well as the goals to achieve.

The second chapter presents the most relevant audio environments and their differences.

Next, in the third chapter, it is presented the proposed solution, the options taken and their reasons and a general description of the resulting system.

The fourth chapter, evaluates the system showing its pros and cons, as well as suggestions to overcome the drawbacks found.

The final chapter summarizes and discusses the ideas developed throughout this document and states the main conclusions of this work. It also presents some limitations in carrying on this study and outlines some directions for future research.

Chapter 2

Audio Tools

The purpose of this section is to provide some information about a selection of relevant audio tools and discuss their concepts.

Before that, some concepts are introduced to clear what we mean by music and sound, as well as it is presented a historical perspective of computer audio tools to better understand computer music and digital audio evolution.

2.1 Music vs. Sound

Through this chapter, the words “sound” and “music” are used apparently with the same meaning, but there are differences between them. There is a lot of discussion about the definition of music and when a sound is music. Although it is not the objective of this work, we will do an overview through that discussion.

Using simple concepts we can say that a sound is anything that you can hear, such as whisper, speak, an engine running, etc. If we have a sound or sounds that is or are too loud, annoying, or not intended then we have what we call a noise. But if that sound or sounds are organized on purpose we have music. Music is pleasant but noise it is not; we enjoy hear music but we don’t enjoy noise.

But while it seems clear the distinction between noise and sound or music, the doubt arises when “sound and music” is used.

On the surface, music is sound, but this is not enough of a definition because there are many sounds that are not, or at least not necessarily, music. For Schmidt-Jones [SJ] music is the art of sound, what is according to Merriam-Webster’s Dictionary[1], that music is "the science or art of ordering tones or sounds in succession, in combination, and in temporal relationships to produce a composition having unity and continuity".

[1] http://www.merriam-webster.com/dictionary/music

If we ask people what music is, we will get a variety of answers and they are all correct. Music may mean different things to different people and perhaps it will say something about the detailed mechanics of music, such as instruments, notes, scales, rhythm, chords, harmony, bass and melody, that matches with a part of the explanation [Dor05, Ale07].

In the last years music has been studied from a biological point of view. Music, as a language, is a universal human trait. Throughout human history and across all cultures, people have produced and enjoyed music. In addition, music is still music even if it evokes different emotions in different people or no emotions at all. Many people say that music is sound that evokes an emotional response because it is part of what music means to them; however, some sounds can evoke emotions, but hardly anyone would describe them as music (e.g. a child crying, a scream, a dog barking). Music is something that makes us feel good. Music creates emotions or interacts with emotions [Per06, Ale07].

According another perspective, sound consists of vibrations that travel through a medium such as gas, liquid or solid. In particular sound is a compression wave, which means that the direction of propagation is aligned with the direction of the motion that is being propagated. Music is sound that's organized on many different levels. Sounds can be arranged into notes, rhythms, textures and phrases. A fundamental component of music is the note. A tone, the kind of sound you might call a musical note, is a specific kind of sound. The vibrations that cause it are very regular - all the same frequency and the wavelength and a certain duration. Melodies, a sequence of notes played in musical time, can be organized into anything from a simple song to a complex symphony. Beats, measures, cadences, and form all help to keep the music organized and understandable. But the most basic way that music is organized is by arranging the actual sound waves themselves so that the sounds are interesting and pleasant and go well together [Dor05, SJ]

To conclude a classic example is our grandparents' opinion about rap music because most rap artists do not sing. Rap it is a musical form that lies in between song and ordinary speech. The main feature of rap is that the music has a spoken component, and this spoken component has rhythm, but it does not have melody [Dor05]. Furthermore, a piece to consider is John Cage's 4'33" which divides many opinions. So, to close this discussion and to be clear, "sound and music" in this chapter is not trying to say that music is not sound. On the contrary, it highlights that something can also be used to create music, i.e. sounds that follow some sort of pattern or have a purpose. Or to keep it simple, without trying to define music, sound in this case means "digital audio signals".

2.2 Some Historical Notes

Computers have their roots in ancient times but it was just by the late of 19th century that the evolution of the "machine that carries computations" had started to advance rapidly. By that time, there were also being made the initials steps to record audio and, in music programming.

By the early 20th century, semi-programmable instruments and early-electronic instruments proliferated and some were quite popular, like the *pianola* and the *theremin*.

In the 1950's computers began to synthesize sound, and by 1957, Max Mathews revolutionized computer music with the Music-N languages that helped popularizing it and introduced various key concepts that are still used nowadays.

Music-N is a generic name to refer to the series of languages developed by Mathews and also to the model they follow. These languages specify sound synthesis modules, their connections and time control.

In this model the concept of "unit generator" is introduced. It is the basic functional component of the system. Unit generators produce or process audio signals, and besides audio input and/or output, they also have control inputs to control its parameters. The combination of many unit generators creates a *patch* or *instrument* which defines a sound. A collection of instruments is an *orchestra*. Another relevant aspect is the *score* that defines what and when to play with additional definition of control changes. Music-V, was the last of these series by Mathews, and the first music programming language to be implemented in a high-level general purpose programming language – FORTRAN – whereas past versions were written in an assembly language. This change helped its widespread and further developments.

Audio programming systems continued to evolve with the advance of computers and programing languages.

When the UNIX operating system emerged, the CARL System was developed based on it. With the rise and popularity of C language, other systems appeared like CMix [Pop93] and CSound (see Section 2.3.1), and both are still used today. A similar tool is Common Lisp Music (CLM) [Sch94] that due to the characteristics of the LISP language made easy to represent many hierarchical music structures.

Also, new approaches like graphical programming turned out to be very popular. The Max languages (discussed in Section 2.3.2) and Kyma [SJ88, Sca02] are some examples.

With the widespread of technology new paradigms have risen and more systems have been developed and they continue to expand.

2.3 Audio Environments

In this section some audio environments are introduced. It is used the word "environment" in order to generalize the tools presented because they are very different: some of them are music programming languages, others are libraries and some are complete applications.

There is a vast collection of audio environments but here, as it was previously mentioned, it will be just describe the relevant ones that are:

Actively maintained: They are kept updated, provide help on learning them and to solve problems that might appear. Also, to reduce the risk of, the application we develop with it, becoming prematurely obsolete.

Open-source: They are transparent and it is possible to modify (add functionalities to) the framework in order to suit best our needs.

Multi-platform: They are portable across (at least) the major platforms and have no (or little) restrictions to be easily used with platform-dependent applications.

Max and Aura do not fill these requirements entirely but they have made important contributions for the development of audio applications and research.

2.3.1 CSound

CSound [Ver86] is a textual-based programming language created by Barry Vercoe in the late eighties that still is widely used today.

CSound, as the name suggests, is a C implementation of the Music-N paradigm. By the time it was created, C was becoming a trend and the Internet was expanding rapidly contributing to CSound popularity.

As a descendent of Music-N, it separates the definition of the orchestra (in ".orc" files) from the score (".sco" files). It also separates audio and control rates in order to get better computational efficiency.

CSound allows the design and synthesis of sound and digital signal processing and supports real-time audio [wsr].

Although its success, it is not very popular amongst composers as it demands previous programming knowledge.

2.3.2 Max and Pure Data

Max (popularly known as Max/MSP) [Puc91, Puc02, Zic02] and Pure Data [Puc96] follow the Music N paradigm. They are the most popular environments these days.

Max was originally developed in the eighties by Miller Puckette at IRCAM[2] in order to enable composers to create interactive music. By 1989 it started to be commercial and since 1999 it has been commercialized and maintained by David Zicarelli's company, Cycling'74.

Initially, Max was just a visual programming language and IDE for music with focus on control external sound synthesis workstations. By 1996, Puckette released Pure Data (Pd), a redesigned and open-source version of Max that aims to explore Max's concepts further in order to expand data treatment including graphics and video data.

Pure Data inspired Zicarelli to develop a Max extension called MSP, standing for either "Max Signal Processing" or Miller S. Puckette initials. MSP allows real-time digital signal manipulation, creation of synthesizers and effects. More modules were created, but MSP and Jitter (a later released one) remained the most popular. Jitter added the possibility to work with image (creation and manipulation of 2D/3D graphics, real-time video processing, etc.). Recently (by January 2012) Max has released a new module called "Gen" that is meant to write patches computationally efficient as if they were written in C or GLSL[3] code.

[2] Institut de Recherche et Coordination Acoustique/Musique: http://www.ircam.fr/

[3] OpenGL Shading Language (http://www.opengl.org/registry/doc/GLSLangSpec.4.20.6.clean.pdf)

Pure Data works on Win32, IRIX, GNU/Linux, BSD, and MacOS X whereas Max is only available for MacOS X and Windows. Pd though it is simpler to use (for some complex tasks patches are simpler) than Max, the latter has a more appealing and usable interface.

Both are very similar and interoperable to some extent. They share the same objective in audio analysis: be easy to learn and have a predictable and acceptable behaviour that does not jeopardize its performance.

IRCAM also released a Java-based version of Max, called JMax [DBC+98].

2.3.3 Nyquist

Nyquist [Dan97] is a programming language based on Lisp for sound synthesis. Nyquist was created in the beginning of the 1990's by Roger Dannenberg as an evolution of previous systems, Artic and Canon. Currently it runs under Linux and other Unix environments, MacOS, and Microsoft Windows.

Though it is a Music-N language, it has the particularity of removing the difference between the orchestra and score, i.e. between the sound synthesis definition and the events that trigger them. So, the order of execution is not explicit but allows more flexibility when manipulating sound. Other advantages of Nyquist, are a customization of behaviours considering different contexts, and having a rich offer that deals with time and transformations (e.g. scaling and stretching sounds).

2.3.4 Synthesis Toolkit

The Synthesis Toolkit (STK) [CS99, SC05] is a set of C++ classes that aims to be a cross-platform programming environment for music synthesis and audio processing rapid prototyping.

Perry Cook originally developed STK in the 1990s with later contributions by Bill Putnam and Gary Scavone. STK is open-source and has been distributed since 1996.

STK has (mostly) platform-independent C and C++ code that can be implemented on any platform and it is easy for the user to use and extend. It also allows real-time synthesis and control.

2.3.5 Aura

Originally, Aura [DB96, DR95] was a framework developed by Roger Dannenberg for real-time sound synthesis that targeted portability and flexibility.

Currently – AuraRT – has broadened its objectives to create interactive multimedia (with focus on music) and has developed a subproject – AuraFX – for flexible signal processing.

Though Aura is not currently available – it is "undergoing a rapid change"[4] – it is listed here because of its attributes: encapsulation of platform-independent code, dynamic instantiation of new instruments, asynchronous control and multiple sample rates without compromising efficiency, and two ways to combine *unit generators* (or *ugens*, units that encapsulate algorithms) in *instruments* (à la Music-V or wrapping a network of *ugens*).

[4] http://www.cs.cmu.edu/~auraRT/

2.3.6 SuperCollider

SuperCollider [McC02] is an IDE and programming language created for real-time audio synthesis. It was created in the late 1990's by James McCartney. It is open-source (under GPL) and cross-platform.

SuperCollider programming language has it syntax based on SmallTalk and C++ and follows the Music-N paradigm. It differs from most of Music-N languages, as it is a comprehensive and general-purpose object-oriented language that has a wide range of specific functions for music, signal processing and sound synthesis. Also there is no distinction between the *orchestra* and the *score*, as Nyquist. It is a powerful language with an optimized synthesis engine.

In its last and current version (SuperCollider 3 or SC3) it made a clear distinction between the language and the synthesis engine that intercommunicate by Open Sound Control (OSC) messages (see Section 3.4.2). One of the obvious advantages is that programmers can use just the SuperCollider server (called "scserver") as an external tool to perform audio synthesis, as long as it conforms to the protocol defined by the *scserver*.

2.3.7 SndObj

Sound Object (or SndObj) [Laz00, Laz01] is an C++ audio processing library that provides the means for synthesis and sound analysis and allows file I/O in real-time.

SndObj has three main characteristics:

- Encapsulation – SndObj encapsulates all the processes involved with sound creation, manipulation and storage
- Modularity – Processing objects can be freely associated to perform a specific function
- Portability – Most code is platform-independent (except for the real-time I/O classes) requiring simply a (POSIX compliant) C++ compiler

The class library is founded on four base classes: SndObj, SndIO, Table and SndThread, which deal correspondingly to sound processing objects, sound I/O objects, mathematical function-tables and thread management.

Sound Object is free software (GPL) for Linux, Windows, MacOS X (but with no MIDI at moment), Silicon Graphics machines and any OSS-supported UNIX.

2.3.8 Marsyas

Marsyas [TC99, Tza07, TJC+08, BT05] was designed such that the common elements of the algorithms would be similar to architectural blocks so that the integration of different techniques under the same interface/framework would be easier.

The first versions of Marsyas focused mainly on sound analysis, and for the development of the 0.2 version, besides enlarging the scope of functionalities (e.g. synthesis) it also included some of the concepts introduced by some of the sound analysis tools presented in this section:

sound synthesis model presented by Synthesis ToolKit, flow architecture similar Clam, ChucK's patching model and some Aura's ideas.

Marsyas also introduces a new concept: *implicit patching* that contrasts with the *explicit patching* of the former tools. Roughly, it relates to how processing blocks connect to each other: it describes the composition of these objects rather than the connections between in/output ports between each element that form them. The use of implicit patching besides solving many problems at implementation level and memory management, improves the use of Marsyas, making the development of new features easier and faster.

2.3.9 Clam

Clam [Ama04, Ama07, AAR02] is a C++ framework with focus on the study and development of music and audio applications. It is licensed under GPL and it is cross-platform (tested on Linux, Windows and MacOS). It offers means to do complex signal analysis, transformations and synthesis.

It started as an internal project of the Music Technology Group at the Universitat Pompeu Fabra[5] to organize their repository of audio-processing algorithms.

Clam has the following features:

- Comprehensive – besides the audio processing classes it includes all necessary classes to build an application
- Extensible data types – can handle a wide variety of complex and simple data types
- Two operation modes – as a regular C++ library or as a prototyping tool

It also offers black and white-box capabilities and a synchronous data-flow and a asynchronous event-driven flow (control-flow).

2.3.10 ChucK

ChucK [Wan08, WC04, WFC07, WC03] is "a concurrent, on-the-fly, audio programming language" designed by Ge Wang and Perry Cook. It is cross-platform (runs on MacOS X, Windows, and Linux) and distributed by the terms of GPL.

Initially it was built for real-time audio synthesis performance and experimentation but it latest version also allows analysis.

It follows a timing model that concurrently represents audio synthesis and analysis with flexible and arbitrarily detailed control. Also, the language semantic allows operating on multiple, dynamic and simultaneous control rates.

ChucK's 'on-the-fly' feature has made ChucK an important tool for live-coding.

[5] http://mtg.upf.edu/

2.4 Discussion

The tools presented are very similar as all follow a graph model (at least at the core level), so they can connect their basic objects and build a complex network.

Some of the environments presented, use a graphical programming language whereas others use text-based languages (and some even provide both). In graphical programming the data and control flow abstraction is represented directly. It shows the connections and the objects together in the same view, and some of them even the flow running. Text-based systems lack this representation and the object definitions and connection statements are often in different parts of the application. However, specifying a complex algorithm is usually easier and tidier to express and read in a text-based code.

Most music and sound tasks can be implemented in either way, though some may be simpler to specify in a particular language than in another. But this is not only applied to graphical/text-based languages but also among the different languages that form those categories as they also follow different models. After all, what defines each one of them are the paradigms they follow to achieve their goals. Nevertheless, for developing efficient applications or creating algorithms with low-level details it is more advisable to use a text-based programming whereas visual programming, is more appropriate for prototyping.

Max and Pd are clearly visual programming languages, but Clam also offers a basic way to model signal processing systems and to edit Clam networks with the "Network Editor". The remaining tools none of them provide a system for that matter except for Marsyas that currently it is being developed a similar tool to Clam's "Network Editor".

Within the tools that employ text-based programming language, SuperCollider and ChucK provide their own specific audio language, while the remaining use general-purpose languages. Nyquist uses Lisp; CSound uses C and, SndObj, Marsyas and Clam use C++. The implementation of a specific language involves much effort. Besides developing a compiler it is useful to have their own editor, and they should introduce remarkable features. Also its concept should be valid in such way to justify its development effort and to attract new users to learn it and use it. A specific language has the advantage to address the matter directly in terms of representation, control and efficiency. Using a general-purpose language to build a library or framework is not as complex, seeing the fact that they already offer a set of abstractions that can be used for a specific purpose. Also you may take the inherited advantages of using a already proven language, such as existing various-purpose libraries, its popularity etc.

Looking at the environments whose primary function is sound synthesis – CSound, Max, Pd, STK, Nyquist, SuperCollider, ChucK – most of them offer some basic sound analysis. CSound, Max, Pd, and SuperCollider provides a variety of FFT tools but rely on black-box objects just to cover the basic tasks. There is not a clear concern on offering means to extend with new analysis modules. Also, for the design of low-level analysis tasks a precise control over time is necessary, which is hard to achieve. Nyquist, on the other hand, besides performing FFT/IFFT it allows some low-level analysis and spectral manipulation. ChucK takes it further by providing classes designed

for analysis and also applies on them some of the concepts it uses for synthesis.

For the analysis-centred tools presented – SndObj, Marsyas and Clam – all of the three support basic synthesis operations. To name a few functionalities, they perform some synthesis methods like additive, spectral and granular synthesis, generating different kinds of waves and noise. Between the three, SndObj has the smallest offer, not only on synthesis but generally comparing the three tools. Marsyas has the largest offer on analysis, processing and general tools. It also offers a wide range of real-time functionalities, and through Python or its own "Marsyas Scripting Language" it can be used for rapid prototyping. Regarding SndObj these functionalities are also available through Python but aimed to sound synthesis. Clam also have a set of few algorithms with real-time support and for rapid prototyping can be used its "Network Editor" tool, previously mentioned. It is worth recalling ChucK as it also has real-time support and it is meant for rapid prototyping.

2.5 Summary and Conclusions

In the 1950s, Max Mathews began a computer music revolution with the release of Music-N languages. Through the evolution of computer programming languages and their new paradigms, computer music has also grown with it.

Today there is a large and diverse collection of audio tools. Their diversity crosses the models and concepts they follow, the programming languages they use: specific or general, visual or textual, their purpose and scope.

Chapter 3

A Proposal for a Possible Solution

The following section describes the solution for the problem described in Chapter 1 and how it was develop. We begin by stating the solution goals, then the options made to achieve it and lastly what we have designed.

3.1 Introduction

We present the design of an Open Sound Control (OSC) message-based interface for Marsyas[1].

The main goal is to develop a Marsyas module that enables other tools to easily interact with it, so that it could be used at different levels depending on the application we pretend to develop, for example:

- Manipulate some sound
- Do intermediate calculus
- Extract sound/music features
- Classify a sound/music

The expected end use of the module we are developing is to support audio manipulation in creative tools, like the ones mentioned on the Chapter 1.

Marsyas should not only address the problems identified on those tools, as it could also be used as an auxiliary tool for other audio environments in order to perform some complementary tasks.

For the matter of this thesis, we will focus on the first use-case because its users it need most. Also, they have a bigger interest in developing applications/works that come in the path of "real-time audio-visual and interactive applications" either for the mobile or stationary environments.

[1] The reasons for choosing this type of solution with Marsyas and OSC will be later explained after understanding the general objectives to achieve

We believe, that achieving this first goal, the second one will be *partially* included. We cannot say *completely* as they involve different kinds of usage.

3.2 User Description

The target audience are people that produce digital and creative works using tools like the ones described in Section 1.2 (e.g. Processing) and, wish to include some audio analysis tasks in their works but do not feel that those tools support them as they should.

This group is mainly composed by visual or musical artists, composers, hobbyists or students that have basic or no programming skills nor signal processing knowledge. These people need out-of-the-box functionalities, that address their needs in a straightforward way and do not make them think about how they accomplish those tasks.

Nevertheless, even if they want to perform the same tasks, they certainly would be using them in different contexts and environments so they must be adaptable.

Just keeping these goals on mind, the specification would define a module that targets those points too directly and might place barriers on experienced users that ask for more than black-box functionalities. So, it is important to keep an eye on the users that could apply this module for working with other audio libraries or wants to carry out more advanced tasks.

Working with audio libraries is different from working with creative tools, as is most likely it would be used by more experienced users that would need a more flexible set of functions.

This second user group is formed by composers eager to find new sound spaces and/or ways to create music and to find new tools to conceive their musical concepts. And also, by sound engineers and music researches that might need to complement their research tools with audio analysis and need a reliable interoperability.

Making a system suitable for both groups is not exclusive, as many goals are similar. The main objective is to simplify. Not in the amount of functions available for the users nor the intrinsic system but in the concept behind and the processes to accomplish the tasks.

The main focus will be users that, as previously mentioned, will be using the selected audio tool[2] with their creative tools, but keeping a mind open to the ones that will be using it with other audio libraries.

3.3 Design Goals

From the ideas previously expressed, we can summarize a set of design goals in the following points:

[2] In the introduction of this chapter was already mentioned that Marsyas was the chosen audio tool. In this section we try to expose the characteristics of the user that uses creative (or other audio) tools. These characteristics are independent of the tool selected

Functionality Offer an adequate set of functions that the adopted audio tool[3] offers

Portability Platform-independent and interoperable with different tools

Usability Easily operable and understandable

The first goal, functionality, should be clarified. One might think that this module should perform all the functions provided by the selected audio tool. But that is not true. Throughout this document it is said that the module defined will allow the selected audio environment to *complement* the audio analysis tasks of some other tool. It should be noticed that, likewise, we did not reduce the functionalities to be provided just to the ones that performed some sort of audio analysis.

It does not make sense to provide *all* functionalities, as most of them might never be used as the "complemented" tool might already provide them. As well it does not make sense to reduce the scope because it underuses the adopted audio tool. The functionalities must be ranked by priority. The audio analysis and processing functions are critical, and the synthesis functions very important. The rest of the priority of the functions is related to the importance they have on the use of those tasks. Functions like I/O operations are more important than arithmetic ones.

To be successful it also should be platform-independent, or at least work over the major operating systems, and compatible with different tools and applications particularly the ones described in Chapter 2.

Lastly, it should be “easy” at every aspect: easy to learn, easy to use, easy to understand, easy to maintain and easy to extend.

The names and conventions used should be consistent and self-explanatory, i.e., good readability. There should be a natural correspondence between the words used and resulting actions, and the same word should have the same meaning in different context. Also, to reduce learning time, it should also follow customary naming conventions and simple words/phrases so it can be easily retained [Jac04, Blo06].

The actions performed and the results obtained should not produce surprises for the user, so they must be clear and assure the user understands the state reached and possible alternative actions to perform [Jac04, Blo06].

In order to reduce the user’s use of boilerplate code, it should provide easy implementations of common tasks. But still have a set of simple methods to allow the user to do everything he needs [Blo06].

The graphic shown in Figure 3.1, shows us the importance of each design goal. In that graphic, the most important design goal is usability, followed by portability and lastly, functionality.

The Functionality goal is kept last as it reflects the completeness of the relation between the functions provided by the module and the functions of the audio tool it "wraps". As previously said the audio analysis, processing and synthesis functions should be provided.

[3] In the introduction of this chapter was already mentioned that Marsyas was the chosen audio tool. In this section we try to expose the design goals of the module that aims to complement sound tasks performed by creative (or other audio) tools. These objectives are independent of the tool selected.

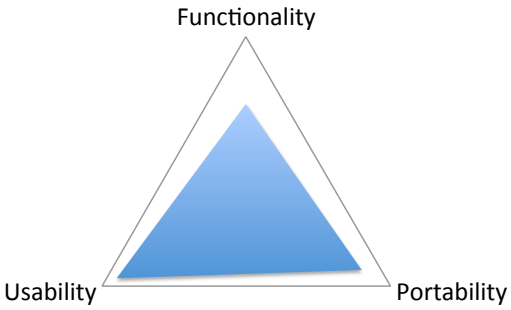


Figure 3.1: Design goals map

Portability is an important goal to take into consideration. The module/selected audio tool should keep up with the platform shift (desktop to mobile) mentioned on Section 1.1 and provide ways to run over them.

Usability is the main goal to achieve, it is the one that influence the most what abstractions should the design provide. Also, usability, is the most distinctive aspect of creative tools, so this module should work conformably with those tools.

3.4 Design Decisions

As mentioned in the beginning of this chapter, the plan is to develop an “OSC message interface for Marsyas”. This section will state the reasons for choosing Marsyas and an OSC message interface.

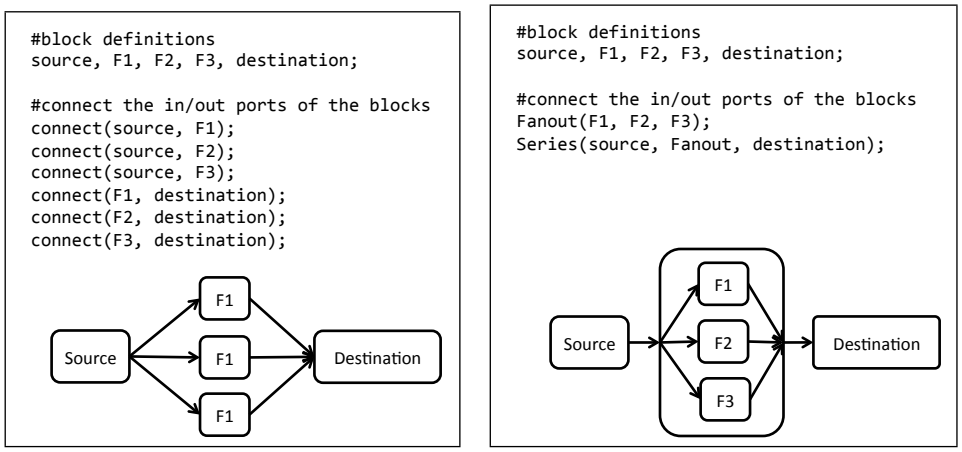
3.4.1 Marsyas

In Chapter 1 we said that creative tools (like Processing, OpenFrameworks, Cinder) have a shortfall on audio analysis functionalities, neither the tools themselves nor the use of external libraries[4] provide the user a reasonable set of analysis and feature extraction methods. So, it was considered the audio environments presented in Chapter 2 whose capabilities focused on analysis. Among them, the choice fell on Marsyas.

Marsyas has the largest collection of analysis and processing features, as well as a fair set of synthesis methods. Also, the real-time functionalities it provides were the factor that contributed the most for Marsyas being selected.

It uses implicit patching which, recalling Section 2.3.8, not only solves many problems at implementation level and memory management, but also improves the use of Marsyas making the development of new features easier and faster because you do not need to express "explicitly" how Marsyas components’ inputs and output connect with each other (Figure 3.2).

[4] Of course, specific libraries can be included but they do not meet the principles of those tools mentioned Section 1.2, namely simplicity in its use.



(a) Explicit Patching (b) Implicit Patching

Figure 3.2: Differences between Explicit Patching (a) and Implicit Patching (b). (adapted from [wspa])

Section 3.2 describes the target audience for this system and it is not expected that users will develop their own analysis algorithms. So, it would seem that some of Marsyas natural characteristics (see section Section 2.3.8) might be irrelevant. But, the model it follows is replicated on the solution, so it provides an understandable mental image of the system and consequently be straightforward to use.

Also, for users with more instruction on computer music and audio fields it keeps an open door for exploring the tool further. Lastly, it can also be used to complement other audio environments that lack analysis functionalities.

An interesting fact about Marsyas is that it is very popular among academic and industrial projects, for instance, Last.fm[5] uses it for rapid prototyping MIR tasks.

3.4.2 OSC Message-passing Interface

Initially, we intended to choose one amongst the creative tools (Processing, openFrameworks, Cinder, etc.) and develop a library that would wrap Marsyas in a simpler interface. But, after exploring that possibility, a message interface seemed more appealing. It brings more advantages for Marsyas and already covers some of the goals mentioned in Section 3.2:

Versatility: Easily adaptable to different number of applications (they just need to follow the same protocol) and environments.

Stability: If the client crashes, the server keeps working; if the server crashes the client is able to manage the recovery. Very important on live performances.

[5] http://www.last.fm/

Portability: Marsyas can be external to your computer, so there is no need for installation, it allows remote control, and makes it suitable for mobile platforms by reducing the memory and processing constraints of those platforms[6].

This model transforms the Marsyas framework work as an audio processing server or, as we prefer to call it, a service *provider*, and the application that uses it is the *client*.

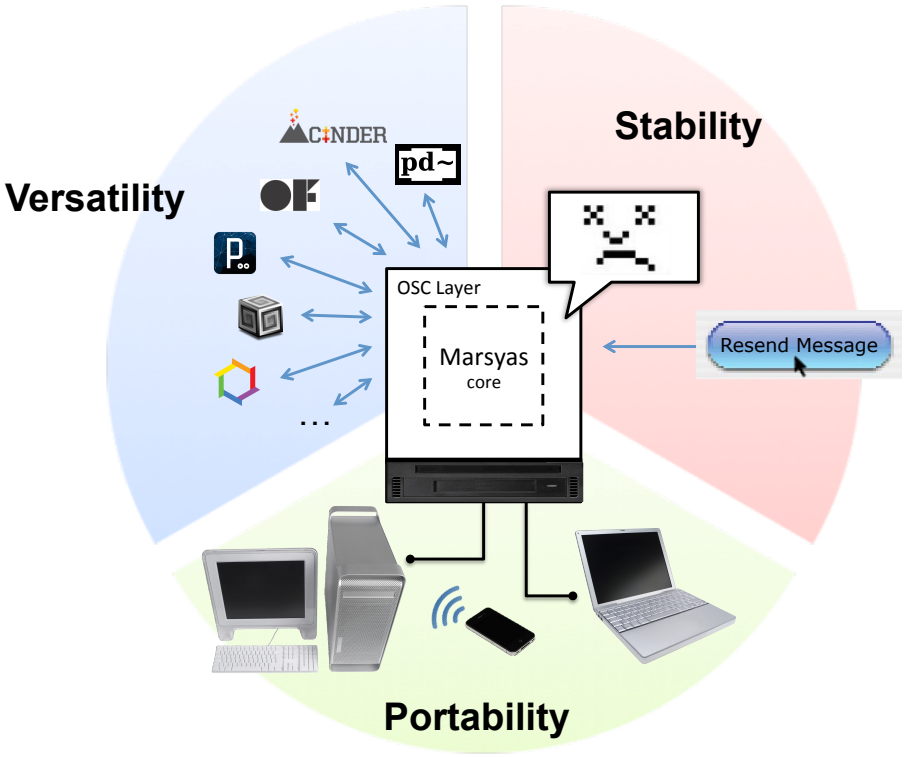


Figure 3.3: Versatile for being adaptable to different applications, portable because those applications can be running on a different platform that the one with Marsyas, and stable for being fault tolerant.

The most important drawback of the adoption of this model is the latency introduced. Firstly, similarly to SuperCollider (SC3), the latency introduced by the messaging process: a small and usually unnoticed delay. Secondly, depending on which output is selected for the audio playback (see Section 3.5.3), a low audio latency could be introduced.

The message format followed is OSC – Open Sound Control [Wri02, FS09, SFW10]. OSC is an open, transport-independent, digital-media content format developed for "communication among computers, sound synthesizers, and other multimedia devices that it is optimized for modern networking technology".

[6]Marsyas (with this module) can be included in the final applications and run on those platforms, but this is an alternative that brings other advantages.

OSC appeared in 1997 by Adrian Freed and Mathew Wright at CNMAT[7] as a development of a previous protocol – ZIPI[8] – that addressed many MIDI shortfalls and also "represented recommended and actual practice at CNMAT" [FS09]. The features of OSC as stated in [wso] are:

- Open-ended, dynamic, URL-style symbolic naming scheme
- Symbolic and high-resolution numeric argument data
- Pattern matching language to specify multiple recipients of a single message
- "Bundles" of messages whose effects must occur simultaneously
- Query system to dynamically find out the capabilities of an OSC server and get documentation

By "open" we mean not only, that there are no license requirements, patented or protected intellectual properties but also that it makes no restrictions to the use of it.

Another huge advantage of OSC openness is its adaptability to different communication protocols [SFW10]. Since most audio tasks are time-critical, OSC is mainly used over UDP/IP.

UDP – user datagram transport – does not provide mechanisms to assure the delivery of the data packets. Also, delivered packets may arrive out of order, where in that case OSC timestamps may be used to recover the original order.

On the other hand, TCP – transmission control protocol – provides reliability in the transport and delivery but is significantly slower than datagram transport due to the overheads introduced. For TCP communications it is necessary to have a means representing the beginning and end of a packet. OSC does not define a means for it, as it can have any number of encapsulating messages. Therefore the OSC packets should be framed with extra data that indicates the packets boundaries. Two ways of achieve that and to adapt to other transport types, are indicated in [SFW10].

OSC is widely spread and the NIME [9] community uses it extensively to rapidly develop encodings for particular purposes. Also, most of the tools, creative or audio, presented in Chapter 2 are OSC compatible. Though all of the mentioned audio tools have an OSC interface, it is worth building a specific interface for Marsyas instead of using one of those environments for its distinctive features mentioned on the previous section.

3.5 General Description

3.5.1 Main Architectural Concepts

The interface simplifies the use of Marsyas though maintaining its unique features to preserve its strength.

[7] UC Berkeley Center for New Music and Audio Technology: http://cnmat.berkeley.edu/
[8] Zeta Instrument Processor Interface:http://archive.cnmat.berkeley.edu/ZIPI/
[9] International Conference on New Interfaces for Musical Expression: http://www.nime.org/

Marsyas follows the data-flow programming concept. Computation works as a data-flow graph where nodes can apply transformations to or be the source of the data that flows through the arcs, i.e. the communication channels. One of the features of data-flow programming is to easily create in our minds a conceptual model of the system, which enables the users of our system to rapidly create and/or understand models.

Marsyas provides various "building blocks" as data-flow components that can be assembled to form algorithms, that in turn can be arranged to compose more complex algorithms. Furthermore, new "building blocks" can be easily added to extend the framework.

These “building blocks” (or data-flow nodes) mentioned before are called *MarSystems*. The most relevant MarSystems are:

Sources and Sinks: To deal with input and output;

Synthesizers: To create audio;

Analyzers: To translate audio blocks into other blocks;

Processing: To perform processing operations, such as filters and envelopes.

Composites: To assemble other MarSystems in various ways (series, parallel, etc.)

Marsyas distribution also comes with a series of command line applications that throughout this document will be referred as “MarApps”. These applications have different functions from soundfile interaction (e.g. playback) to auditory scene analysis. And also GUI Applications that are called through command-line, that let you, for example, visualize a collection of songs.

Many of the existing MarApps were developed by the Marsyas community. These functionalities should be included and be used similarly to the MarSystems and their differences abstracted from the users.

For the message interface, how we call the dataflow nodes is a bit different, so we can differ simple MarSystems from complex ones. Firstly, a MarSystem is just called “block”. This may bring some confusion to Marsyas users but for new users, it is simpler to understand. Also, this has some influence in naming the commands provided, and making them clearer to the user.

The simplest processing block is a “unit” and can have three types: processing, analysis and synthesis. These are equivalent to the MarSystems mentioned above. However, these units are more general, as a means to include the functionalities provided by the MarApps. So:

Processing User can modify its audio input and output

Analysis User can modify its audio input

Synthesis User can modify its audio output

All of them have an identifier, controls, input and output – though one or none of these last two may be inaccessible. The controls are Marsyas mechanisms that allow changing the MarSystem functionality while running.

Other existing type of block is a "composite" that functions like the MarSystem Composites. They provide different methods for connecting various MarSystems in a network, in addition to link them in series. Besides the unit's parameters they also have a list of blocks, and both input and output must be accessible.

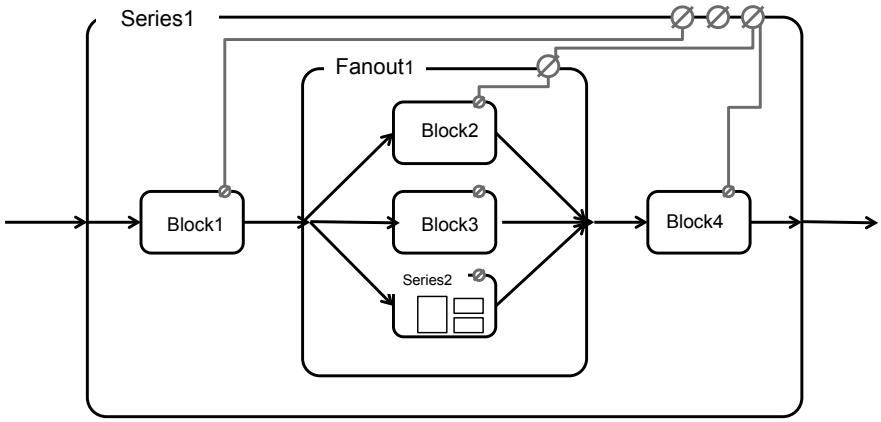


Figure 3.4: An example of a Marsyas network with controls.

The most complex block is a "network" (see Figure 3.4), which is a set of blocks that has a purpose, i.e. it performs a task. Like the composites, it also has a list of blocks. A network can be considered as a series of blocks.

Yet there are two main differences: firstly, like the *units*, it does not need to have both input and output. This may seem odd at a first glance but for the synthesis of a complex sound, an audio input it is not really need. Also, there must always be at least one network to perform a task.

A *unit* can be (is) a MarSystem network since most of the MarSystems and MarApps provided by Marsyas are networks. This means we can build a *block network* equal to a *unit* (MarSystem). The only difference between the two is that, *unit* follows a black-box model and the *block network* a white-box model. We cannot manipulate (e.g. delete) the components of the unit as you can in the network.

The diagram in Figure 3.5 represents the relation between the presented identities and helps consolidate these ideas.

3.5.2 Messaging

3.5.2.1 OSC Specification

The OSC structure is very simple yet powerful. Figure 3.6 represents the structure of OSC, the elements mentioned are [Wri02]:

OSC Packet: The main OSC transmission unit. It consists of its contents and the size of 8-bit bytes of the contents. The contents could be an OSC Bundle or an OSC Message.

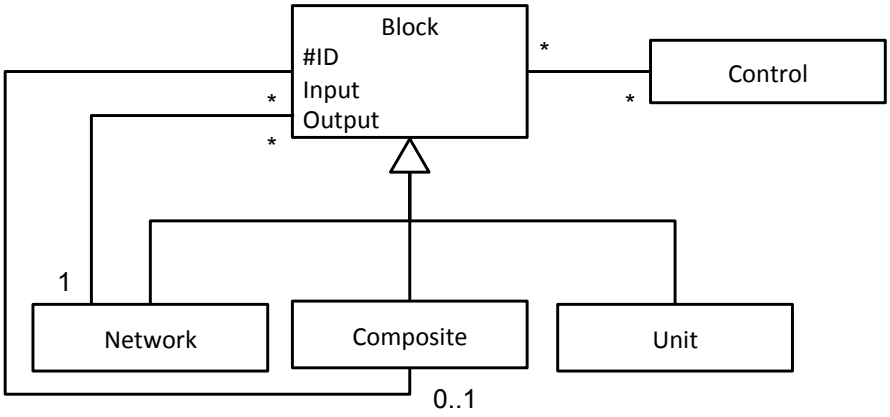


Figure 3.5: Relationship between the different entities.

OSC Bundle: The first byte of an OSC Packet is an OSC-string with "#bundle". This is needed to easily distinguish an OSC Bundle from an OSC Message. The following 64 bits are reserved for an OSC Time Tag. Then, zero or more OSC Bundle Elements could follow it. An OSC Bundle Element consists of its size and contents, as an OSC Packet. This means that an OSC Bundle is recursive, its contents should be an OSC Message or an OSC Bundle.

OSC Time Tags: Currently just provides space for a time tag in the stream, defines its units and only requires that the least significant bit be reserved. It had a specific structure in OSC 1.0 but in OSC 1.1 it was considered unnecessary [FS09].

OSC Messages: Consist of an OSC Address Pattern followed by an OSC Type Tag String, followed by zero or more OSC Arguments.

OSC Address: OSC-string beginning with the character '/'. The path-traversing wildcard – "//" – enables matching across different branches of the address tree at any depth.

OSC Type Tags: OSC-strings beginning with the character ',' followed by a sequence of characters corresponding to the sequence of OSC Arguments. The permitted types are summarized in Table 3.1.

OSC Arguments: Contiguous sequences of binary representations of the relevant data to transmit.

There are many open-source C++ libraries available that already offer an OSC implementation, so there is no need for the system described in this chapter, to implement one of its own. One of those is *OSCPack*, a set of C++ classes that provides an easy but robust way to treat OSC packets.

OSCPack is stable and has been tested on the three major OSs: Windows, MacOS and Linux. OSCPack is already used in some Marsyas applications, and it is also provided in the Marsyas distribution, so it makes a strong candidate to be used in the task of OSC interpretation.

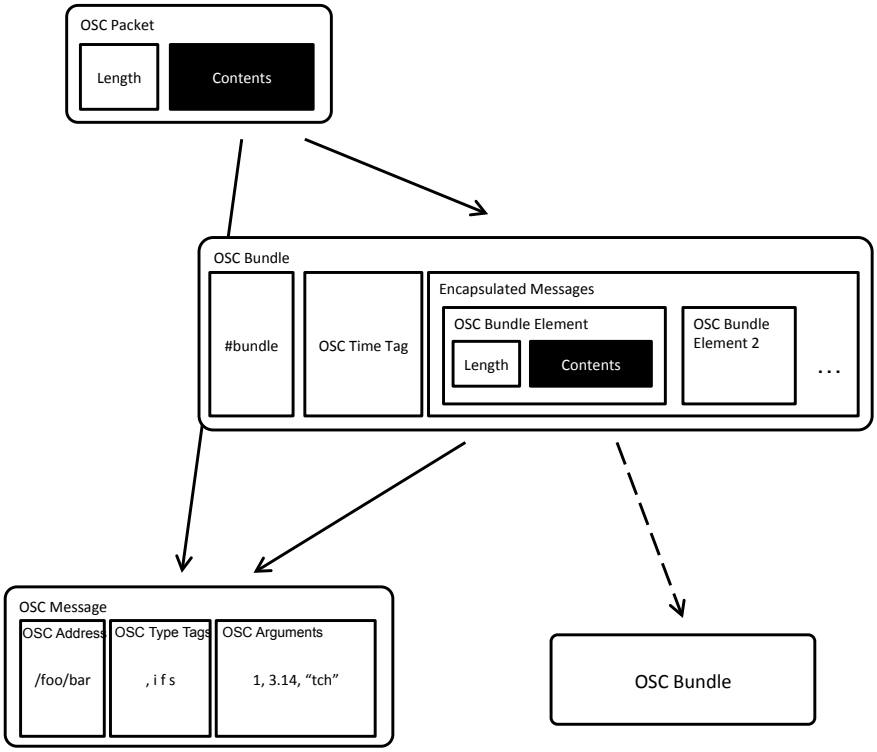


Figure 3.6: OSC Message structure (adapted from [SFW10])

3.5.2.2 Command Description

The commands are placed on the address field of the OSC message. The advantage of OSC is that it offers freedom to defined them as we wish. Nevertheless, in [SFW10] the authors present some design approaches and give some considerations about audio control data. The descriptive styles they present are:

- RPC (Remote Procedure Call) style - uses small address spaces, followed by a list of parameters and respective arguments that take a function-like form.
- REST (Representational State Transfer) style - transmits the current state of an entity.
- OOP (Object Oriented Programming) style - objects have attributes and methods that changes them.
- RDF (Resource Description Framework) style - entities described using relationships.

Table 3.2 presents some examples of these styles.

The style adopted for the command description is an adaptation of the OOP style. RPC and RDF styles are too strict and their messages are too long even for a simple task. Also, though RDF is the "most powerful of the alternatives (...) it makes no semantic assumptions about the data structure" [SFW10] , which is a relevant factor for enabling abstraction. REST is a familiar style, but as it is meant to describe state-machine systems, ends being restrictive in a dynamic

i	Integer: two's complement int32
f	Float: IEEE float32
s	NULL-terminated ASCII string
b	Blob, (aka byte array) with size
T	True: No bytes are allocated in the argument data.
F	False: No bytes are allocated in the argument data.
N	Null: (aka nil, None, etc). No bytes are allocated in the argument data.
I	Impulse: (aka "bang"), used for event triggers. No bytes are allocated in the argument data.
t	Timetag: an OSC timetag in NTP format, encoded in the data section

Table 3.1: OSC Type Tags (from [SFW10])

RPC	REST

`/setgain`
`(channel number = 3)`
`(gain value = x)`
	`/channel/3/gain (x)`

OOP	**RDF**

`/channel/3@gain (x)`
or
`/channel/3/setgain (x)`
	`/channel,num,3`
	`/op,is,set`
	`/lvalue,is,gain`
	`/rvalue,units,dB (x)`

Table 3.2: OSC messages examples using different control data styles (from [FS09])

environment as the Marsystems Network, where there are two types of flow (control and data). The OOP style suits best between the styles presented, as it follows a concept close to the one of the Marsyas allowing greater liberty for defining the command messages.

The adaptation made to the OOP style was reducing the address to the method call so the inclusion of the "identifier" field is optional. The next example shows the adaptation (1.) of the OOP style (2.):

1. **`/setgain`** `(x)` or **`/setgain`**`/channel3 (x)`
2. **`/setgain`** `(x)` or `/channel/3`**`/setgain`** `(x)`

This is a way to keep simple tasks simple, as most part of the uses will not be building complex networks. So, identifiers will not always be needed to use. In the previous OOP example (Example 2.) the "/channel/3" could be omitted but it would not maintain the same structure, as it would not be as intuitive. In the adapted-OOP style, the identifier field is on the right, so looking at the structures of the Example 1., the user gets the impression that both follow the same structure and that the "identifier" is a field that can be "added" because as we read from left to right, it is more natural. A way to maintain the structure in the Example 2. would be having a reserved address word that substituted it (e.g. “default”), but this method would complicate a simple task. The user would have to know that there is a “word” that he had to use and to memorize it (the word and the procedure).

The OOP style allows the use of pattern matching operators, as the *wildcard operator* defined in Section 3.5.2.1. It is defined the "*" operator to be used either in the address or in the arguments space. It enables "one-to-many mappings between patterns and groups of messages" [SFW10]:

`/setgain/channel1 (4)`

`/setgain/channel2 (4)` If there are only 3 channels: `/setgain/* (4)`

`/setgain/channel3 (4)`

The command description used has got most of its inspiration on the SuperCollider server synth engine [McC]. Its main structure is similar to the one adopted:

`/command (arguments)`

This definition, also enables an understandable use of nested calls. The use of OSC-Bundles is applied to the command definition in two ways:

- Call a series of methods within the same message to be applied in the same order to the parent-message command result:

  `/open/channel3 (/setgain (x) /wait (y) /setgain (z))`

- Define various methods as parameters of the parent-message:

  `/setgain/channel3 (/getgain/channel4)`

Also, this structure allows an undefined number of arguments:

```
/add/network1 (block2)

/add/network1 (block2 block5 block7 )
```

To sum up, the structure adopted brings the following features, maintaining its simplicity:

- The identifier field is facultative
- Clear use of nested calls
- Use of wildcard operators
- Allows undefined number of arguments

3.5.2.3 Modules

The messages commands are divided into five groups according to their function: Definitions, Units, Composites, General and Responses.

The *Response* module is concerned with the Provider reply messages to the Clients requests.

In this section will be only presented the modules (and respective commands) that directly offer functionalities to the user.

For a complete reference, with examples, auxiliary and provider's response commands and, error responses refer to the Appendix A.

Definitions The *Definitions* module takes care of the Client's session. It allows the Client to register and quit, and to manage his current processes, i.e. networks. Also, in order that the use of the command messages be simple, some definitions are assumed, this modules lets the user change those definitions. Table 3.3 shows the available commands.

`/register`	Register
`/quit`	Quit
`/status`	Close a process
`/close`	Close a processing network
`/delete`	Delete a block
`/cleanall`	Delete all networks
`/output`	Define output standard
`/input`	Define input standards
`/autoplay`	Turn automatic play on/off
`/nopid`	Define what to do when no pid defined

Table 3.3: Definitions Module Commands

Units The *Units* module provides three general commands to call the three different available units: Analysis, Processing and Synthesis. The messages should follow the structure below:

```
/[command] [method_name] [input]
```

In this structure, "command" is one of the commands listed in Table 3.4, "method_name" is the name of the MarSystem or MarApp we want to use over the "input" that can be a music file or the output of other block[10].

Currently this module has the shortest command list, but it is expected to grow. So that the most used MarSystems/MarApps can have their own commands.

`/analysis`	Analyze a sound
`/process`	Process a sound
`/synth`	Synthesize a sound

Table 3.4: Units Module Commands

Composites The *Composites* module, provides means for creating a composite and manipulating it (Table 3.5), i.e. adding and removing its components. The creation of a composite is similar to the example of the units module, and it is possible to define directly its position in the network (i.e. defining its input/output connections).

```
/composite [composite_type] [/in, input] [/out output] [elements]
```

The "composite_type" is the name of the MarComposite we want to use. The "/in input" and "/out output" are auxiliary messages to connect the input and output ports of the composite. Lastly, comes the "elements" which are a list of blocks. These last three message components are optional.

`/composite`	Create a composite
`/compositeadd`	Adds a block to a composite
`/compositeremove`	Removes a block from a composite

Table 3.5: Composites Module Commands

General This module, as its name suggests, provides general operations over the blocks, from modifying a block control to trace a block (record its changes). But, functionalities like closing processes or deleting blocks are in the Definitions module because they are managing tasks, i.e., control the processes running in a session. Table 3.6 lists some of these definitions.

One interesting function is the ability to control a network like a tape recorder, with functions like stop, play (start) and pause. "Forwarding" and "backwards" are also available and control the

[10]The next section, 3.5.3, is explained what kind of data can be the "input"

number of "ticks" forward or backwards of a network, i.e., the number of times the data passes from a node to the next/previous node.

It also offers the ability to define events. If the user defines what parameter to watch - the input, the output or a control - that will already trigger an event whenever its value change. For a more specific situation an interval can be defined.

`/setinput`	Change the intput of a process
`/setoutput`	Change the output of a process
`/newevent`	Create an event
`/listevents`	List all events
`/control`	Read/modify a control or list available controls
`/setcontrolrange`	Specify the value range of a control
`/setcontrolevent`	Throw a event
`/copy`	Copy a block
`/play`	Start a block
`/stop`	Stop a running block
`/pause`	Pause a running block
`/forward`	Next ticks of a block
`/backwards`	Previous ticks of a block
`/restart`	Restart a block
`/ignore`	Deactivate block
`/run`	Activate block
`/free`	Disconnect block from network
`/dumpTree`	Print process tree
`/trace`	Trace block
`/block2unit`	Convert a block to a unit (black-box block)
`/savemsl`	Save network (block) as a MSL file

Table 3.6: Some of General Module Commands

3.5.3 Input and Output

For Audio input and Output, Marsyas supports the most common audio files formats (.au/.snd, .wav, .mp3, .ogg), real-time audio (as it uses the RTAudio[11] library). These types of data must also be supported through the interface system described in this chapter, but that system should hide the details related to the manipulation of different data from the user so he can use the system the same way with each type.

The resulting data might not be other audio files. As the primary function of the system is analysis and feature extraction, they will not necessarily perform any modification to the original sound file. So, those results depend of what we are analyzing. In order to generalize, we consider that those results are alphanumeric data or "events". The alphanumeric data relates to results that represent time, pitch, number of occurrences, or a class/tag name (e.g. in a genre classification).

[11] RtAudio is a C++ library used for cross-platform real-time audio API

"Events" are notifications triggered by the detection of some occurrence that the user can define. To put it in another way, the analysis result falls on the regular message-passing communication, and this section does not lean over it.

The user may want to perform some Synthesis or Processing task, and that necessarily involves audio data.

So, the scenarios identified for the Client (user) send data to the Provider are:

1a) Send Audio File - Send the audio file

2a) Send Audio File Location - Send the location of the audio file within the local network of the Marsyas Provider

3a) Specify Input Device - Send the name and location of the input device so the Provider can read it

Likewise, the Client will receive the resulting audio:

1b) Receive Audio File - Receive the resulting audio file from the Provider.

2b) Load Audio File - Load audio file from the location provided by the Provider or previously specified by the Client.

3b) Specify Output Device - Send the name and location of the output device so the Provider can write on it.

The receiving and sending functions are very similar performing the reverse functions of one on another.

For 1a) and 1b), it would be necessary use the "blob" OSC-type to send/receive the file via message. The 2a), 3a) and 2b), 3b) scenarios are the easiest cases. The path to the file or input device is encapsulated in the OSC-message as string type.

Also, Marsyas can read scripting files (MSL [12] or Python) files. The use of a scripting file is more advanced and not a primary necessity for the regular user. But the use of MSL or Python files is useful for loading networks that either synthesize a sound or construct a processing/analysis network. Or can be used to store a network and just reload the next time:

4a) Send Scripting File - Send the scripting file to the Provider

4b) Receive Scripting File - Receive the scripting file sent by the Provider

5a) Send the scripting file Location - Send the location of the scripting file to the Provider

5b) Load the scripting file - Load scripting file from the location provided by the Provider or previously specified by the Client.

There is one detail that is worth be discussed. Although Marsyas supports different audio files, the networks you construct do not use the "audio file" itself, but it is converted to a matrix called *realvec*, so we could also use this as a format for audio messages.

[12] Marsyas Scripting Language

3.6 Summary and Conclusions

In this chapter, Marsyas is presented as the tool that best covers the flaws pointed out in Chapter 1.

It is also described an OSC interface for Marsyas so it can be used by other applications and tools. Besides suiting best the needs identified, this proposal covers the following aspects:

Interoperability: Achieved using the popular OSC message-format.

Reliability: The message-passing interface makes the Marsyas module independent (service provider) of the tool that uses it (client) which makes fault tolerance easier. Marsyas is also a mature framework with very low frequency of failure.

Usability: OSC message format is very straightforward and easy to understand. The commands provided are also readable and easy to use as they should follow an OSC-address scheme.

Maintainability: This module is easy to maintain as it should have very few classes involved. It only focuses on the management of clients, forwarding the OSC requests to Marsyas, and retrieving its answers. For OSC parsing, a third-party library (OSCPack) is used and the responsibility for sound synthesis, analysis, processing and other calculus is borne by Marsyas core.

Portability: Marsyas is available for the major OS's and have few dependencies. The message-based interface also allows Marsyas to be external to your computer; to use it is only need to send and receive OSC messages over UDP/IP network.

Chapter 4

Discussion of the Proposed Solution

This chapter reviews the work achieved through subsequent applications and emphasizing the importance of sound in interactive applications.

It presents some examples where the specified model can be applied, an use case and its evaluation. A solution for the problems detected is described and it is also made other observations about the protocol.

4.1 Applications

There are various technologies that allow the user to interact through sound. One of the oldest ones being "Clapper"[1], an electrical switch that allows you to turn on and off many electrical appliances, or, not as old, a mobile phone voice dial. Nevertheless, in our day-to-day life it is not very common to use those types of interaction. In most cases, sound is limited by just offering some kind of feedback of our actions.

The sound interaction, as input or output, may not be a viable method in some situations, for example in a quiet place where it would become a disturbance. But in interactive applications from art installations to computer games, sound takes an important role to immerse the audience/users of those applications.

On both cases, the most common ways to interact with sound can be reduced to [Nob09]:

- The user creates a sound that he/she or someone else can hear (e.g playing a piano or using an application that produces sound interactively).
- The user creates a sound input using a tool (e.g. play a MIDI keyboard connected to the computer).
- The user creates a sound that is the input (sound data or speech data).

[1] http://www.chia.com/index.php/the-clapper

- The user talks or says a word to create the input (speech recognition).

The interactive applications that this study is turned to, focus mainly on to the previous first two points: sound as feedback. CreativeApplications.Net (CAN) [2], is an influential and reliable digital art blog where anyone can find references to creative applications. On that website, the main audio function of most of the applications tagged with "sound", is sound synthesis. In fact, it is not very simple to find applications or installations with sound input. The most common type found are digital instruments/synthesizers.

Nevertheless, there are some very interesting applications. Messa di Voce[3] (2003) is one of those. It is a performance and art installation created by Golan Levin and Zach Lieberman, it induces the user to play in a very engaging way. It has twelve different modes of operations, based on the different characteristics of sound it detects and graphics it produces. But the common operation is that the sound we produce creates graphics, that in turn can be manipulated by our movements, and those changes influence the playback of the sound you produce.

Another application with audio input, from the gaming area, is Singstar[4]. SingStar is a karaoke game for the Sony PlayStation released in 2004. It was, and still is, a quite popular game by its social component. It offers different multiplayer modes that allows us to play with or against our friends. This application "judges your performance based on timing, sustain and pitch". Also, one edition of this game ("SingStar 80s"), came with a mini-game: a version of Pong[5], where to move the pad up and down you'll need to produce sounds with high or low pitch, respectively.

These applications are just some examples where audio analysis can be applied. And they can take a larger step in terms of number and functionality scope, with the aid of the tool designed in this study.

Getting useful information from sound is often very complex, and this work makes audio analysis available for all: free, cross-platform and easy to use.

The production of visuals based on music data is very popular in concerts and nightclubs. Marsyas offers a large collection of music feature extraction methods that can be easily applied to the creation of impressive visuals that combine perfectly with music.

Another example of how audio analysis can be applied, could be an application where the users handclaps controlled the music's tempo. This application would involve real-time audio analysis for detecting the claps and the claps' periodicity, and real-time processing to alter the music's tempo.

Marsyas advantages are not only in its real-time capabilities. MIR functionalities, like classifying a music genre, do not always produce results in real-time. Even so, the process data produced while it refines the results can be used, to generate graphics and create a visual composition in real-time.

[2] http://www.creativeapplications.net/
[3] http://www.tmema.org/messa/
[4] http://www.singstar.com/
[5] http://www.pong-story.com/atpong1.htm

It is quite difficult to enumerate the different applications where Marsyas, through the OSC interface, can be used. This, deeply depends on the creativity of the user so, it can be applied in countless ways.

4.2 Use-case: Animata Bassist

This study focuses on presenting a specification of the communication protocol, but in order to validate the choices made, a subset has been implemented. The best way to ensure that the path followed is the best one, is implementing a use case that will help reviewing the work until then and devise the future approach.

This section presents a small application to assess some of its features and drawbacks.

4.2.1 Description

Animata[6] is an open source animation software, that allows real-time control of marionette-like animations.

In this case, Marsyas is used to control the movements of an Animata puppet based on the characteristics of a song. The head of the puppet bobs in sync with the music tempo which is extracted using a beat-tracking algorithm[7]. The pitch extraction[8] makes the left hand going up and down the neck of the bass. Lastly, the onset detection[9] controls the movement of the hand around the bridge of the double-bass (see Figure 4.1) .

All of the needed algorithms are not native MarSystems, they are "applications" which means it is trickier to call them as they follow different structures. One of objectives of this application is to study how to call non-MarSystems algorithms.

The communication between Marsyas and Animata is made using Processing. So the Marsyas server can be placed in one machine and Animata and Processing in another. This allows to test the portability feature of the system and calculate the latency.

4.2.2 Evaluation

The Processing client was quite easy to implement, though some limitations were found.

Firstly, the user needs to know what are the available functions/methods and what they do, to know if they are relevant for the work he is developing and, also needs to know how he can access the relevant information provided by those methods, so it can be useful. This is a standard procedure with most systems and not a specific "step" to take with this tool.

This application makes use of the "ibt", "pitchextractor" and "onsets" applications. The "ibt" application is used for real-time beat-tracking so any time a "beat" is detected the puppet head bobs. The "pitchextractor" application, as the name suggest, is used for extracting the pitch. The

[6] http://animata.kibu.hu/

[7] Induces tempo and tracks the occurrence of "beats", i.e. when listeners 'tap their feet'

[8] Extracts the pitch, the frequency, of a sound.

[9] Detects the beginning of a musical note (or sound).

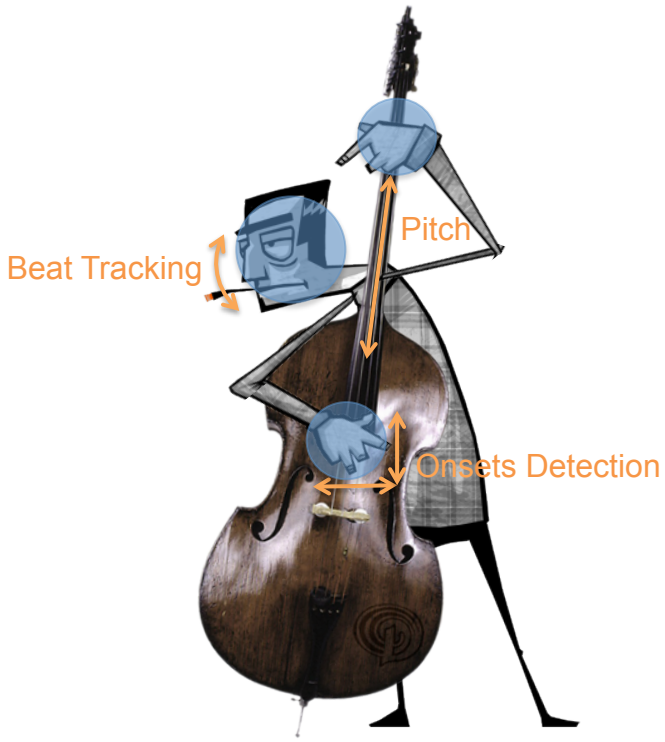


Figure 4.1: Animata doublebass player and respective Marsyas controls.

left hand moves up if the pitch has lowered and moves down if a higher pitch is detected. Lastly, "onsets" is used to move the right hand of the puppet to a random position (within a limited area around the doublebass's bridge) every time a *onset* is detected.

To achieve this, it was only necessary to send three messages to Marsyas (one per method) requesting to throw an event each time an occurrence was detected, following the message structure defined.

Then the results would be extracted by the Marsyas reply message and sent to Animata to change the position of the limb associated with the received message.

To sum up, this messaging method is a very simple way to perform these complex tasks without perceiving it. It clearly encapsulates the intricate calculations involved in these methods.

We began this chapter mentioning that this tool had some drawbacks. As previously said, the user must know where the result of the algorithms are stored and, currently, there is not a support for getting that information. The advantages mentioned before are lost, when this information is searched by the end-user, that has to read and understand the code to find it.

In "pitchextractor" and "onsets" the results are stored in one of their controls (recall Section 3.5.1): "processedData" and "onSamples" respectively. For "onsets" that control stores and updates the time (counting since the beginning of the music) when an *onset* is detected. In "pitchextractor", the control stores a matrix where the pitch value is updated in the position (0,

0). In "ibt" as it was taking a long time to find where those values were stored, would not be coherent with the purposes of this protocol to linger over that search, as it it aims to be simple and not be needed study the algorithm.

This problem also affected the Marsyas OSC implementation, as it made harder to find a general method to get the results pretended.

A second problem emerged from the attempt to make these methods run: synchronization. When working with those algorithms in the normal way, as a command-line applications, they offer several operation modes. All of three perform in real-time, and some of the modes that they offer to output the results are: play the original audio mixed with the detected events, play just the detected events or do not play nothing at all. In this last case, the output file, where the event's times (and values) are stored, is concluded almost instantaneously, while in the other modes they continue to be updated in sync with the music. There is not a mode that without playing any sound could update the output file at the same time those events occur in the music. By saying that there is not a mode, it is also meant that there is not a function that can be called with the same objective. The importance of this, is to simplify the synchronization method of all algorithms, i.e., if the events are detected in sync with the music and all algorithms start performing simultaneously, they will be synchronized.

Of course there are ways to pass around this. One of those is to read the output file, and just send the event OSC-messages to the client whenever the time read is equal to the time of the timer set at the beginning of the algorithm.

4.2.3 Concluding Remarks

A successful implementation of this use-case was already expected to be very difficult to achieve.

Marsyas Applications focused mainly in the end use (through the command line) that in turn is constrained by task they perform. As Marsyas Applications have different purposes they result with heterogeneous structures. This fact pointed out two problems: the need of the user to know where the controls are, and the synchronization of multiple methods.

We could only perform a partial evaluation of the three parameters: usability, functionality and portability. For instance, the first one was partially tested with a very positive result in terms of calling the method, the downside was to find what to call (this last factor prevented testing the last two parameters as they were not successfully implemented).

4.3 A Few Adjustments

The problems detected by the implementation have shown that Marsyas should have a few adjustments.

There are two possibilities to overcome the issues identified: to hide or to adapt.

"To hide" means that without changing code of the Marsyas Applications and just using what is available, wrap up each MarApp under the same interface. They maintain their structure but could be used the same way by the classes that implement the protocol.

The second alternative is to adapt all MarApps to a common structure, not changing the algorithm itself but adding more functions and/or controls to the main MarNetwork. It will also allow the same use by the classes that implement the protocol.

Both of those options demand considerable knowledge of computer music algorithms and will be very arduous to apply those changes as there is a vast collection of MarApps.

To adapt is the most attractive option because it creates an uniform structure for MarApps and makes them more useful in the future. Hiding the details by applying a second layer, can be more interesting since it does not change the code, but not as efficient as it would be if those adjustments were made directly in the code.

Either way, it is necessary to design and put into action a format for future Marsyas Applications. Most audio tools have an OSC implementation that simplifies and diversifies their use. A required format for the MarApps would immediately make them available by OSC messaging without their developers thinking about it.

4.4 Summary and Conclusions

This chapter concludes that the proposed protocol shows promise, but at the present, a complete and detailed analysis and evaluation of protocol could not be made.

Before the implementation of the OSC-messaging system, Marsyas should be subject of a makeover. This involves three aspects:

- the documentation provided should be complete and simple
- there should be a structure for Marsyas Applications to follow
- all the existing Marsyas Applications should be adapted [10] to that model.

Nevertheless, browsing through the command reference (in Appendix A), we evidence that the commands (and its structure) to perform the tasks are quite clear – to use and to understand. It allows the user to easily deal with the application with few knowledge about audio analysis. It means that one of the objectives of this study has been reached.

[10] Adapt here means both of the options mentioned - hide/adapt - not just "adapt" as previously explained.

Chapter 5

Conclusions and Future Work

In this last chapter, we present the general considerations about the work developed. Particularly, it reviews the work accomplished and its assets, and tries to comprehend the factors that may have limited the development and the results of the proposal solution.

This chapter also points some directions for future research of this work and concludes with a personal thought not only about the work itself but also about the experience around it.

5.1 Summary and Contributions

In this study, we examined some audio tools through the point of view of those who use technology for creative purposes. In doing so, a solution was studied and defined so it can improve the systems that are used for the development of real-time and audio-visual applications for desktop or mobile platforms. Those users should be able to straightforwardly apply a wider set of analysis and processing methods to their projects, enriching their audience experience.

The contributions that are most significant in terms of interactive and digital art, as well as in computer music, include:

- An updated look at the most relevant audio tools
- A simple and comprehensive analysis system specification
- A launch pad for a successful implementation

5.2 Limitations

In the development of any study we are faced with unexpected constraints that make us take decisions in order to pursue our goals.

Time is always the first constraint, in particular in this work. It involves, beyond other aspects, specific investigation about the subject itself and about methodology. There is a time necessity, in

order to assimilate and reflect about what we are doing and, to go back and redo it and improve it. Namely, it led to the fact of not being able to test the specification completely.

Despite the fact that the subject of this study is to my liking and although I have some basic knowledge about music, during the accomplishment of this work, a solid musical background combined with a larger knowledge and practice with audio environments, would be a valuable advantage to the enrichment of this work, regardless of my investment in overcoming the difficulties motivated by those aspects.

However, despite of the above limitations, it was possible to achieve a work that, in my opinion, has potential to continue and to prevail.

5.3 Future Work

In this section, we look into some possible future directions and applications for Marsyas, opened up by the OSC-interface described in this study.

This is a project that can be enriched by being embraced by a team of IT Engineers and musicians in order to make the most of the capabilities that the presented tool promotes, due to the necessity of having technical knowledge about music.

5.3.1 Conclude the Implementation

The obvious direction this work takes, is to complete the implementation of the protocol. But first, one needs to overcome the barriers identified in the previous chapter which shall result a transformation of all Marsyas Applications so they have an homogeneous structure and a complete documentation. This might be the most important and challenging aspect of all three. It will naturally progress to the second phase: conclude the implementation and testing of the MarApps' OSC interface.

After this point it will be straightforward to develop and test the MarSystems. Giving precedence to the MarSystems Modules focused on Analysis, Processing and Synthesis operations. Because the chief purpose in using Marsyas is to fill the analysis needs of creative and (other) audio environments.

Lastly, it is appropriate to simultaneously develop applications that validate the implementation. They will also complement the documentation as practical examples.

5.3.2 Improve the command reference

The second step to take after deploying the module is to examine the users' suggestions, in order to improve the system's service. With the help of the community, they can share first person testimony of their use and impressions of the system. Besides detecting eventual errors, they can also suggest modifications and provide information about their most common tasks or command compounds they use. That is to say, their feedback will partially help to correct and extend the

command offer with specific tasks. It is very important that the users participate so the system can grow.

5.3.3 Keep up with the trends

Cloud computing is a technology that uses the "cloud" (the internet) to maintain data and applications. It allows users to use applications without needing to install any kind of software. The work they produce is accessible anywhere with any computer with internet access.

Cloud computing has never been so popular. While computers became ubiquitous in many forms (work computers, home computers, mobile phones, etc) most platforms start having the capabilities that allowed us to view and/or process the data you used, but sometimes the software needed to open them was not available. It eventually became unpractical to carry those data and programs with us, and cloud computing was an answer for that as it centralizes storage and processing.

This concept could be applied to music and audio environments and would be very interesting for Marsyas to adopt this paradigm. In Chapter 3, it is said that an advantage for the choices made is portability: Marsyas could be in one computer and the client application on another. So, cloud computing can be seen as an evolution of that approach, and will definitely increase the flexibility and popularity of Marsyas. Some authors of creative digital works might be not interest in all functionalities offered by Marsyas or even expecting to use audio analysis recurrently. This service facilitates the integration of audio analysis in their artworks, in whatever form they follow.

The downside of this idea is that, in order to become practical, it is necessary to invest in infrastructures to offer a feasible computation. This means it is necessary to be funded and/or to charge by this service which clashes with the ideals of Marsyas as an open-source tool.

To avoid this matter, Marsyas could adopt the concept of "volunteer computing" where the computation is spread by the computers of Marsyas users or other volunteers interested in contributing to this cause. It shall be regarded as a support system to reduce eventual costs in the infrastructures, as just relying on "volunteers" compromises availability.

5.4 Final Remarks

At the end of this work, I would like to refer that I was very pleased to work on two of my favourite subjects: my area of professional development, IT and computation, and one of my leisure areas, music.

The chosen topic for the development of this work can be seen from different points of view, and for each one of them can bring up a solution proposal.

I feel very comfortable with the results that I came with. They open a door to everyone with different background to use advanced sound analysis. But, I have reached a turning point that from which would have a lot of interesting work to continue to develop.

Appendix A

Command Reference

This appendix presents the list of commands that the future OSC module should have. There are some considerations to make before presenting them.

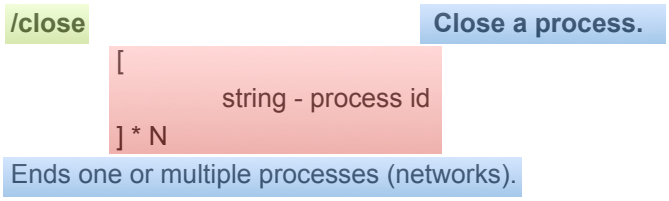


Figure A.1: Command reference structure

In Figure A.1, it is shown the structure used for presenting each command. Highlighted in green is the command address, in red its parameters, and in blue are the short and long descriptions of the command.

The command address sometimes appears with a structure similar to: `/cmd[/pid]` Here "pid" refers to the block identifier where the command result should be applied, and being in brackets means its inclusion is optional. If the user does not define it, the action of the command will be targeted to the last active block, i.e. the block that was more recently used.

Some commands take block identifiers as arguments. In this case, e.g. "`/cmd[/pid] pid2`", it should be clear that the command is, as previously said, applied to `pid`-block and not to `pid2` block. The command does not changes the state of the block with the `pid2` identifier.

In the parameter description in Figure A.1, the parameter, "string", that is between brackets is being *multiplied by N*. This means that this command accepts more than one argument, in fact, it accepts *at least one* argument. There are other options: zero or more (N^{o}) or a specific number of times (e.g. {0,1,..,N}).

Besides this, there are some reserved words that have a special meaning. All of them are defined in the next sections in their context, except for "*" and "RANDOM". The first one, "*", is a "wild card" for blocks and controls. The second, "RANDOM" is to be used with the Units module commands to select a random method for analysis, processing or synthesis.

A.1 User Messages

/clientID **Client message wrap.**

[
OSC Message
] * Nº

In a multi-client scenario, the user must register and wrap its requests with this message. The “clientID” is not a command but the user’s identifier defined by him.

/register/clientID **Register as a Client.**

[
string – address and listening port
] * {0,1}

Request to register. Sends the “ClientID” and allows to define where the Provider must send the reply messages

A.1.1 Definitions

/quit **Quit.**

no arguments

The Client requests to unregister.

/close **Close a process.**

[
string - process id
] * N

Ends one or multiple processes (networks).

/status[/pid] **Print system definitions.**

no arguments

Print Client's system definitions and information about system blocks/networks (which ones are active, etc). If Pid defined, only prints information concerned about that block.

/delete[/pid] **Delete a block.**

no arguments

Delete a block.

/cleanall **Delete all networks.**

no arguments

Delete all unconnected networks.

/stdoutput[/pid] **Define default audio output.**

no arguments

Define output standard. This is, if default speakers or file type/location. If bid defined, it only affects that process.

/stdinput[/pid] **Define default audio input.**

no arguments

Define input standard. This is, if default microphone or file type/location. If pid defined, it only affects that process

/autoplay[/pid] **Turn automatic play on/off.**

string/int - mode

Turn automatic play on/off. Mode:
OFF - never;
ON - always: processing and output;
OUTPUT - just the result of the processing block.

/nopid **Define what to do when no pid defined.**

string/int - mode
[
string/int - block id
] * {0,1}

Define what to do when no pid defined. Mode:
DEFAULT - last active block;
FIRST_ACTIVE - first active block;
FIRST_CREATED - first created block;
LAST_CREATED - first created block;
PID - defined block.

A.1.2 Units

/process[/pid] **Process a sound.**

string - filter/effect name
string/blob - sound file (audio file, audio file path) or output of other process (pid)
[
OSCMessage – connect to other blocks or apply other commands
] * Nº

Apply a filter/effect to a sound or process output.

/synth[/pid] **Synthesize a sound.**

string - source type
[
arguments
] * Nº
[
OSCMessage – connect to other blocks or apply other commands
] * Nº

Synthesize a sound based on the source type defined.

/analysis[/pid] **Analyse a sound.**

string – method name
[

arguments
] * Nº
[
OSCMessage – connect to other blocks or apply other commands
] * Nº
Analyze a sound based on the method defined.

A.1.3 Composites

/composite[/cid] Create a composite.
string/int - composite type
[
/in Declare an input connection
string - identifier of the input block or input sound file
] * {0,1}
[
/out Declare an output connection
string - identifier of the output block or output sound file
] * {0,1}
[
string - identifier of the block inside the composite
] * Nº
Create a composite structure. “cid” is the component’s identifier.

/compositeadd[/cid] Add a block to a composite.
[
string - identifier of the block inside the composite
] * N
]
Add one or more processes to the composite with identifier cid or, if not defined the system creates one.

/compositeremove/cid Removes a block from a composite.
[
string - identifier of the block inside the composite
] * N
]
Remove one or more processes to the composite with identifier cid or, if not defined the system creates one.

A.1.4 General

/setinput[/pid] Change the input of a block.
[

string - identifier of the input process or input sound file
]
Define the input of a block. Connect the input to another network.

/getinput[/pid] **Get the input of a block.**

no arguments
Get the input of a block.

/setoutput[/pid] **Change the output of a process.**

[
string - identifier of the output process or output sound file
]
Define the output of the last active process or, if pid defined, to the process with that pid.

/getoutput[/pid] **Get the output of a block.**

no arguments]
Get the output of a block.

/newevent[/pid] **Create an event.**

string - event name
[
string - mode
[
/OP **Declares a range**
[
int/float - values
] * N
] * Nº
] * {0,1,2, 3}
]
The client receives a message whenever a block changes its in/output or control values, or matches the expression defined. Mode:
IN - monitor the input
OUT - monitor the output
CONTROL - monitor the controls
Operators:
EQUAL - equal to one or more values
NOT - different from one or more values
IN/IN_EX - between two values (inclusive/exclusive)
NOT_IN/NOT_IN_EX - not between two values (inclusive/exclusive)
LESS/GREATER – less or greater than one value

/listevents[/pid] **List the events defined.**

no arguments
Lists the events defined for a block.

/deleteevent[/pid] **Delete the events defined.**

string - event name

Delete events.

/control[/pid] **Read/modify a control or list available controls.**

[string - control name]
[
string/int - control value
] * Nº

If no arguments specified, it lists the controls. If the control name is specified it just reads the value of the control. If the message contains more arguments, it modifies the value of a control.

/listcontrols[/pid] **List the controls available.**

no arguments

Lists the controls of a block.

/readcontrol[/pid] **Read a control of a block.**

string - control name

Read a control.

/writecontrol[/pid] **Modify a control of a block**

string - control name
[
string/int - new control value
] * N

Modifies the value of a control.

/getcontrolrange[/pid] **Get the value range of a control.**

string - control name

Get the value range of that control.

/setcontrolrange[/pid] **Specify the value range of a control.**

string - control name
[
/OP **Declare a value range.**
[
int/float/string - value or expression
] * N
] * N

Set the control value to a random value that respects the rules defined. If that control is mutable, i.e. the control value changes during network processing, it warns the Client whenever it happens.

Operators:

EQUAL - equal to one or more values
NOT - different from one or more values
IN/IN_EX - between two values (inclusive/exclusive)
NOT_IN/NOT_IN_EX - not between two values (inclusive/exclusive)
LESS/GREATER – less or greater than one value

/deletecontrolrange[/pid] **Delete the ranges defined for a control.**

string - control name

Delete the control range of a control.

/setcontrolevent/control_name **Create an event.**

[

/OP **Declare a range.**

[

int/float/string - value or expression

] * N

] * Nº

The client receives a message whenever the control changes its value or matches the expression defined.

Operators:

EQUAL - equal to one or more values

NOT - different from one or more values

IN/IN_EX - between two values (inclusive/exclusive)

NOT_IN/NOT_IN_EX - not between two values (inclusive/exclusive)

LESS/GREATER – less or greater than one value

/getcontrolevent/control_name **Get the events of a control.**

no arguments

Get the events of that control.

/deletecontrolevent/control_name **Delete the events of a control.**

no arguments

Get the events of that control.

/copy[/copy_pid] **Copy a block.**

string/ - identifier of the block to be copied

Copies a block.

/play[/pid] **Start a block.**

no arguments

Start a block.

/stop[/pid] **Stop a running block.**

no arguments

Stop a block.

/pause[/pid] **Pause a running block.**

no arguments

Pause a block.

/forward[/pid] **Next ticks of a running network/composite.**

int - number of ticks

Forward next ticks of the running network.

/backwards[/pid] **Previous ticks of a running network/composite.**

int - number of ticks

Backwards next ticks of the running network.

/restart[/pid] **Restart a block.**

no arguments

Restart a block.

/ignore/pid **Deactivate block.**

no arguments

Turns a block off. It cannot be a block in a extremity of a network.

/run/pid **Activate block.**

no arguments

Re-activates a ignored block.

/free[/pid_father] **Disconnect block**.

[

string – identifier of the block

] * N

Disconnect one or more blocks from a network or composite.

/freeall[/pid_father] **Disconnect all blocks.**

no arguments

Disconnect all blocks from a network or composite.

/deepfree[/pid_father] **Disconnect all blocks.**

no arguments

Disconnect all blocks recursively from network or composite.

/dumpTree[/pid] **Print process tree.**

no arguments

Print process tree.

/trace[/pid] **Trace block.**

[

/trace_time

int - number of ticks or milliseconds

] * {0,1}

[

string - mode

[

string/int - control

] * Nº

] * {0,1,2, 3}

Trace a process for a certain period of time. Receive a message whenever process input, output or controls change their values. If controls specified only tracks those.
Mode:
IN - monitor the input
OUT - monitor the output
CONTROL - monitor the controls

/stoptrace[/pid] **Stop tracing block.**

[
string - mode
[
string - control
] * Nº

] * {0,1,2, 3}

Turns trace off.
Mode:
IN - monitor the input
OUT - monitor the output
CONTROL - monitor the controls

block2unit[/pid] **Convert a block to a unit.**

no arguments

Convert a block to a unit.

savemsl[/pid] **Save network as a MSL file**

no arguments

Save a network (block) as a MSL file.

A.2 Provider Messages

/done **An action has completed.**

[
string – action (command) name
]

Confirmation message of a client's request.

/blockinfo **An action has completed.**

[
string - block identifier
] * {0,1}

Block creation successful.

/fail[/server addr] **An error occurred.**

string - error code
string - error message
[

string/int - error arguments

] * N

An error occurred. For the error code see Section A.2.1 Error Messages.

/status.reply **Status reply.**

/settings **Default definitions**

/in

string

/out

string

/autoplay

string

/nopid

string

/error

string

/confirmation

string

/block **Block list**

[

/block identifier

string – status

[

/blocksettings **Diferent definitions**

[

/SETTING_NAME

string

] * Nº

] * {0,1}

]

Status reply. “SETTING_NAME” refers to a specific block setting definition, i.e. that differs from the general definition.

/status.reply **Status reply.**

/block identifier

string – status

/blocksettings **Diferent definitions**

/in

string

/out

string

/autoplay

string

/nopid

string

/error

string

/confirmation
string
OSC Message – “/ctrl_list” Message
Block status reply (when the block identifier is specified).

/block_info/pid **Block creation success**
no arguments
Block creation success message. The “pid” is the identifier of the block created.

/input.reply **Send the input**
string – input definition
It sends the pid or file location of the input. If it is an audio file it only sends its name.

/output.reply **Send the output**
string – output definition
It sends the pid or file location of the output. If it is an audio file it only sends its name.

/input_event/pid **Input event.**
string - value
Reply of "/event".

/output_event/pid **Output event.**
string - value
Reply of "/event".

/event_list **List of the events defined.**
string - event name
[
string - mode
[
/OP **Declares a range**
[
int/float - values
] * N
] * Nº
] * {0,1,2, 3}
List of the events defined. . Mode:
IN - monitor the input
OUT - monitor the output
CONTROL - monitor the controls
Operators:
EQUAL - equal to one or more values
NOT - different from one or more values
IN/IN_EX - between two values (inclusive/exclusive)
NOT_IN/NOT_IN_EX - not between two values (inclusive/exclusive)
LESS/GREATER – less or greater than one value

/ctrl_list/pid **List the controls of a process**

[
 /ctrl_def **Begin a control definition**
[
string - control name
string - control type
string - control permissions
string - current value
]
] * N
Lists the controls of a process

/ctrl_value/pid/control_name **Return the value of a control**
string - current value
Lists the controls of a process

/crtl_range/pid/control_name **Return the range of a control**
[**/rng,** **Declare a range**
string - bolean set operator
[
int/float/string - value or expression
] * {1, 2}
] * N
Reply of "/getcontrolrange".

/ctrl_event/pid **Control event created.**
string - control name
string - control value
Reply of "/newevent".

/crtl_event/pid/control_name **Return the event definition of a control**
[**/rng,** **Declare a range**
string - bolean set operator
[
int/float/string - value or expression
] * {1, 2}
] * N
Reply of "/getcontrolevent".

/trace.reply/pid
/MODE[/name]
value
Reply of "/trace". The “/name” field is used on controls. Mode:
IN – input value
OUT - output value
CONTROL - controls value

/tree[/pid] **Print process tree.**
[
/block/block_id

[
/block/sub_block_id
[etc]
] * N
] * N
Print process tree using depth-first search.

/msl[/pid] **MSL file**
[
string – file location
] * {0,1}
Confirmation message, if the user did not specify the file location, the provider sends it.

A.2.1 Error Messages

err_blockRunning	Block is running. Close first.
err_inUse	Identifier already in use.
err_noProcess	Block or control not found.
err_notAllowed	Operation not allowed.
err_notFound	Block or control not found.
err_readingFile	Error reading file.
err_requestRejected	Error writing file.
err_writingFile	Error writing file.

A.3 Examples

This section presents a few examples on how to use this commands.

[“/register/clt1]	
[“/done” register]	[“/fail”, “err_inUse”, “Identifier already in use.”, “clt1”]

[“/clt1”, [“/quit”]]	
no response	[“/fail”, “err_blockRunning”, “Some processes are running, cannot perform that action.”, “p1id”, “p2id”]

[“/clt1”, [“/process”, “filter_name”, song]	

]	
[“/blockinfo”, “pid”]	[“/fail”, “err_notfound”, “Not found method <filter_name>”, “filter_name”, song]

[“/clt1”, [“/process”, “filter_name”, in_pid]]	
[“/blockinfo”, “pid”]	[“/fail”, “err_notfound”, “Not found block <in_pid>”, “in_pid”]

[“/clt1”, [“/setinput/pid”, in_pid]]	
[“/done”]	[“/fail”, “err_notfound”, “Not found block <in_pid>”, “in_pid”]

[“/clt1”, [“/composite”, composite_name ["/in", in_pid], ["/out" out_pid], pid1, pid2, pid3]]]	
[“/blockinfo”, “pid”]	[“/fail”, “err_notfound”, “Not found composite <composite_name>”, “composite_name”]

[“/clt1”, [“/compositeadd”, pid1, pid2, pid3]]	
[“/done” compositeadd]	[“/fail”, “err_notfound”, “Not found block <pid2>”, “pid2”]

[“/clt1”, [“/control”]]	
[“/crtl_list/pid/”, [“/crtl_def”, “ctrl_name”, “ctrl_type”, “ctrl_permissions”, current_value] * N]	[“/fail”, “err_noProcess”, “The Client has no processes”]

[“/clt1”, [“/setcontrolrange/pid”, “control_name”, [/IN, value1, value2],	

[/NOT_IN, value3, value4]]	
[“/done”]	[“/fail”, “err_notAllowed”, “The control <control_name> cannot be modified”, “control_name”,“p2id”]

Appendix B

Use cases

This appendix presents the some use cases of Marsyas via OSC messages.

B.1 Definitions

Use case name 1.1.1 Register as a new client.
Goal In Context A user asks the provider to register it.
Preconditions -
Successful End Condition The user receives its client identifier.
Failed End Condition The application for new client is rejected.
Primary Actors User
Trigger The User asks the Provider to register as a new client
Main Flow

Step	Action
1	The User asks the provider to be a Client.
2	The Provider verifies it can handle a new Client.
3	A new Client is created.
4	A confirmation message with the Client's id is sent to the User.

Extensions

Step	Branching Action
2.1	The Provider does not verify it can handle a new Client.
2.2	The User is rejected as a Client.

Use case name 1.1.2 Quit
Goal In Context A Client asks the Provider to unregister.
Preconditions The Client is registered
Successful End Condition The Client is registered
Failed End Condition A fail message is received by the Client
Primary Actors Client
Trigger The Client asks the Provider to unregister
Main Flow

Step	Action
1	The Client asks the provider to unregister.
2	The Provider verifies that the Client has no unclosed processes.
3	The Client is unregistered

Extensions

Step	Branching Action
3.1	The Provider does not verify that the Client has no unclosed processes.
3.2	The Client is warned which processes are closed.

Use case name 1.1.3 End processing block.
Goal In Context A Client asks the Provider to close a process.
Preconditions The Client is registered at the Provider. The Client has blocks running.
Successful End Condition The block(s) end.
Failed End Condition Cannot end the block.
Primary Actors Client
Trigger The Client asks the Provider to close some block(s).
Main Flow

Step	Action
1	The Client asks the provider to close a block.
2	The Block is closed
3	A confirmation message is sent to the Client.

Use case name 1.1.4 Check Client status and definitions
Goal In Context Inform the client about the system definitions and its processes (blocks).
Preconditions The Client is registered at the Provider.
Successful End Condition A status message is sent to the Client.
Failed End Condition A status message is not sent to the Client.
Primary Actors Client
Trigger The Client asks the Provider to check its definitions
Main Flow

Step	Action
1	The Client asks the provider to check its definitions
2	The provider send a status message to the client

Use case name 1.1.5 Check block definitions
Goal In Context Give client information about a block.
Preconditions The block exists and it is owned by the client.
Successful End Condition A block status message is sent to the Client
Failed End Condition A status message is not sent to the Client.
Primary Actors Client
Trigger The Client asks the Provider to check block definitions
Main Flow

Step	Action
1	The Client asks the Provider about a process
2	The provider send a status message to the client

Use case name 1.1.6 Delete block
Goal In Context A Client wants to delete one or more blocks
Preconditions The blocks exist, they are owned by the client and are not running
Successful End Condition The block is deleted
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the Provider to delete one or more blocks
Main Flow

Step	Action
1	The Client asks the provider to delete a block.
2	The Provider verifies that the block may be erased.
3	The provider erases the block.

Extensions

Step	Branching Action
3.1	The Provider does not verify that he can erase the block
3.2	The Client is warned the block cannot be closed.

Use case name 1.1.7 Define the default input/output action.
Goal In Context The Client wants to change the default audio output.
Preconditions The Client is registered.
Successful End Condition The default definition is changed
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the Provider to change the default definitions.
Main Flow

Step	Action
1	The Client asks the Provider to change the default input /output definitions.
2	The provider changes them

Use case name 1.1.8 Turn automatic play on/off
Goal In Context The Client does (not) want to play the results automatically.
Preconditions The Client is registered.
Successful End Condition The default definition is changed
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the Provider to change the automatic play definitions.
Main Flow

Step	Action
1	The Client asks the Provider to turn the definition of automatic play on or off.
2	The provider changes them.

Use case name 1.1.9 Define the default action if the client does not define the block identifier in a command
Related Requirements -
Goal In Context The Client wants to change default process that the actions apply when he does not define it
Preconditions The Client is registered.

Successful End Condition The default definition is changed.
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the Provider to change the definition for no block identifier.
Main Flow

Step	Action
1	The Client asks the Provider to change the definition for no block identifier
2	The provider changes them.

B.2 Units

Use case name 1.2.1 Apply a basic filter/effect to a sound file
Related Requirements -
Goal In Context Apply a filter/effect to a sound file and output the result
Preconditions The Provider recognizes the sound file.
Successful End Condition The output soundfile is the result of applying the filter to the input file.
Failed End Condition The output soundfile is (not the result of applying the filter to) the input file.

Primary Actors Client
Trigger The Client asks the Provider to process a sound.
Main Flow

Step	Action
1	The Client asks the Provider to process a sound.
2	The Provider verifies that the request mentions the process ID
3	A new process is created.
4	The Provider verifies that the file type needs to be converted to a matrix.
5	The Provider converts the input file to a matrix
6	The Provider applies the filter/effect to the matrix
5	The Provider converts the matrix to the defined output type
6	The Provider verifies that he needs to play it
7	The Provider plays the output sound file
8	The Provider sends a message to the Client with relevant information about the process (pid, file location, etc).

Extensions

Step	Branching Action
2.1	The Provider verifies that the request does not mention the process ID
2.2	The Provider assigns a process ID to the requested process
2.1	If the input file is a matrix there is no need in converting it.
6.1	The Provider verifies that he does not need to play the resulting file
6.2	Does not play the file.

Use case name 1.2.2 Apply a basic filter/effect to a process
Related Requirements -
Goal In Context Apply a filter/effect to a process and output the result.
Preconditions The process is owned by the Client.
Successful End Condition The output sound file is the result of applying the filter to the input file.
Failed End Condition The output sound file is (not the result of applying the filter to) the input file.
Primary Actors Client
Secondary Actors -
Trigger The Client asks the Provider to apply a transformation to the result of a process.
Main Flow

Step	Action
1	The Client asks the Provider to apply a transformation to the result of a process.

2	The Provider verifies that the process ID field is not empty.
3	The Provider verifies that it needs to create a new process.
4	The Provider creates a new Process.
5	The Provider applies the filter/effect to the process output matrix
6	The Provider converts the matrix to the defined output type
7	The Provider verifies that he needs to play it
8	The Provider plays the output sound file
9	The Provider sends a message to the Client with relevant information about the process (pid, file location, etc).

Extensions

Step	Branching Action
2.1	The Provider verifies that the process ID field is empty.
2.2	The Provider retrieves the last process of that Client.
3.1	The Provider verifies that it does not need to create a new process.
3.1	The Provider applies the transformations to the process itself.
7.1	The Provider verifies that he does not need to play the resulting file
7.2	The file is not played.

Use case name 1.2.3 Synthesize a sound
Related Requirements -
Goal In Context Produce sound.
Preconditions -
Successful End Condition A sound is created.
Failed End Condition A sound is not created.
Primary Actors Client.
Trigger A client asks the Provider to synthesize a sound.
Main Flow

Step	Action
1	A Client asks the Provider to synthesize a sound
2	The Provider verifies that the process ID field is not empty.
3	The Provider verifies that it needs to create a new process.
4	The Provider creates a new Process.
5	The Provider synthesizes a sound.
6	The Provider converts the resulting matrix to the defined output type.
7	The Provider verifies that he needs to play it.
8	The Provider plays the output sound file
9	The Provider sends a message to the Client with relevant information about the process (pid, file location, etc).

Extensions

Step	Branching Action
2.1	The Provider verifies that the process ID field is empty.
2.2	The Provider retrieves the last process of that Client.
3.1	The Provider verifies that it does not need to create a new process.
3.1	The Provider applies the transformations to the process itself.
7.1	The Provider verifies that he does not need to play the resulting file
7.2	The file is not played.

Use case name 1.2.4 Analyze a sound
Goal In Context Analyze a sound.
Preconditions -
Successful End Condition The Client receives the analysis result message.
Failed End Condition A sound is not analyzed.
Primary Actors Client.
Trigger A client asks the Provider to synthesize a sound.
Main Flow

Step	Action
1	A Client asks the Provider to analyze a sound
2	The Provider verifies that the process ID field is not empty.
3	The Provider verifies that it needs to create a new process.
4	The Provider creates a new Process.
5	The Provider analyzes a sound.
6	The Provider verifies that he needs to play it.
8	The Provider plays the output sound file
9	The Provider sends a message to the Client with relevant information about the process (pid, file location)

Extensions

Step	Branching Action
2.1	The Provider verifies that the process ID field is empty.
2.2	The Provider retrieves the last process of that Client.
3.1	The Provider verifies that it does not need to create a new synthesizer unit.
3.1	The Provider applies the transformations to the process itself.

B.3 Composites

Use case name 1.3.1 Create a composite
Goal In Context Create a composite of blocks

Preconditions The composite type requested exists.
Successful End Condition The composite is created
Failed End Condition The composite is not created
Primary Actors Client
Trigger The client asks the provider to create a composite
Main Flow

Step	Action
1	The client asks the provider to create a composite
2	The Provider verifies that he can create a composite (if defined, the in/out block, sound files and contents exist)
3	A composite is created

Use case name 1.3.2 Add/remove block to composite
Goal In Context Add/remove a block from the content of the composite
Preconditions The composite requested exists. The blocks exist and are owned by the Client.
Successful End Condition The block is added/removed to/from the composite
Failed End Condition An error message is sent to the client
Primary Actors Client
Trigger The client asks the provider to add/remove the block to composite
Main Flow

Step	Action
1	The client asks the provider to add/remove block to composite.
2	The Provider verifies that he can create a composite (if defined, the in/out block, sound files and contents exist)
3	The block is added/removed to/from the composite

B.4 General

Use case name 1.4.1 Define block in/output
Goal In Context Connect the in/output of a block to other block or to a audio file source/sink
Preconditions The block exists and is owned by the client. The connected block/file exists.
Successful End Condition The block is connected to that file/block
Failed End Condition The block is not connected to that file/block
Primary Actors Client
Trigger The Client asks the provider to define block in/output
Main Flow

Step	Action
1	The Client asks the provider to define block in/output

2	The Provider verifies that he define it
3	The block is connected to that file/block

Use case name 1.42 Set an event
Goal In Context Set a block evebt
Preconditions The block exists and is owned by the client.
Successful End Condition A message is sent whenever a occurrence is detected.
Failed End Condition The client receives a error message
Trigger The Client asks the provider to be warned when a control/input/output of a block changes its value.
Main Flow

Step	Action
	The Client asks the provider to be warned when a control/input/output of a block changes its value
	A message is sent whenever a occurrence is detected

Use case name 1.4.3 List controls
Goal In Context Get the controls of a process
Preconditions The process is owned by the Client
Successful End Condition The list of the process controls is sent to the Client
Failed End Condition A fail message is sent to the Client
Primary Actors Client
Trigger The Client asks the Provider to list the controls of a process.
Main Flow

Step	Action
1	The Client asks the Provider to list the controls of a process.
2	The Provider sends to the Client the list of controls

Use case name 1.4.4 Read a control value of a process.
Goal In Context Read the value of a control.
Preconditions The process is owned by the Client. The process has the control specified. The control can be read.
Successful End Condition The value of the control is sent to the Client.
Failed End Condition The value of the control is not sent to the Client.
Primary Actors Client
Trigger The Client asks the Provider to read a control of a process.
Main Flow

Step	Action
1	The Client asks the Provider to read a control of a process.
2	The Provider sends to the Client the value of the control.

Use case name 1.4.5 Modify a control value of a process.
Goal In Context Change the value of a control.

Preconditions The process is owned by the Client. The process has the control specified. The control can be written.
Successful End Condition A confirmation message sent to the Client.
Failed End Condition The value of the control is not sent to the Client.
Primary Actors Client
Trigger The Client asks the Provider to modify a control of a process.
Main Flow

Step	Action
1	The Client asks the Provider to read a control of a process.
2	The Provider sends to the Client the value of the control.

Use case name 1.4.6 Get control values range
Goal In Context Get the value range defined for a control
Preconditions The process is owned by the Client. The process has the control specified.
Successful End Condition The value range is sent to the Client
Failed End Condition Can not read the values range
Primary Actors Client
Trigger The Client asks the Provider to get control values range
Main Flow

Step	Action
1	Client asks the Provider to get control values range
2	The Provider sends a confirmation message with the control range to the Client

Use case name 1.4.7. Set control's values range
Goal In Context Set a value range defined for a control
Preconditions The process is owned by the Client. The process has the control specified.
Successful End Condition A new value range is defined
Failed End Condition A new value range is not defined
Primary Actors Client
Trigger The Client asks the Provider to set a control's values range
Main Flow

Step	Action
1	The Client asks the Provider to set a control's values range
2	The Provider set the control value with a random number within that range.
3	The Provider sends a confirmation message to the Client whenever its value gets out that range

Use case name 1.4.9. Copy block
Goal In Context Copy a block

Preconditions The block to be copied exists and is owned by the client.
Successful End Condition The block is copied
Failed End Condition The block is not copied.
Trigger The Client asks the provider to copy a block
Main Flow

Step	Action
	The Client asks the provider to copy a block
	The provider copies the block

Use case name 1.4.10 Manipulate network/block state.
Goal In Context Play, stop, pause, forward, backwards or restart a block.
Preconditions The block exists, is owned by the client and is not running.
Successful End Condition The block's state changes to the one intended.
Failed End Condition The block's state does not changes or changes to a state different than the wanted state.
Trigger The Client asks to change the block state
Main Flow

Step	Action
	The Client asks to change the block state
	The state is changed

Use case name 1.4.11 Turn on/off a sub process
Goal In Context (De)activate a block
Preconditions The Client is registered, the block exists and is owned by the client.
Successful End Condition The block is (de)activated
Failed End Condition The block is not (de)activated
Trigger The Client asks the provider to (de)activate a block
Main Flow

Step	Action
	The Client asks the provider to (de)activate a block
	The provider verifies he can disconnect the blocks.
	The blocks are disconnected
	A confirmation message with to Client.

Use case name 1.4.12 Disconnect one or more blocks from a network
Goal In Context Disconnect one or more blocks from a network without deleting them
Preconditions The Client is registered, the blocks exist and they are owned by the client.
Successful End Condition The blocks are disconnected
Failed End Condition The blocks cannot be disconnected
Trigger The Client asks the provider to disconnect a block
Main Flow

Step	Action

	The Client asks the provider to disconnect a block
	The provider verifies he can disconnect the blocks.
	The blocks are disconnected
	A confirmation message with to Client.

Use case name 1.4.13 Get network tree
Goal In Context The Client wants to be informed of the components of a block (network or composite)
Preconditions The Client is registered, the block exists and is owned by the client.
Successful End Condition The block tree is sent to the Client
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the provider to get the block tree
Main Flow

Step	Action
1	The Client asks the provider to get the block tree
2	The provider sends the block tree

Use case name 1.4.14 Start/stop Trace a block
Goal In Context The Client wants to be informed of the changes of the block's in/output and controls
Preconditions The Client is registered, the block exists and it is owned by the client.
Successful End Condition The block is started/stopped being traced
Failed End Condition An error message is sent to the Client
Primary Actors Client
Trigger The Client asks the provider to start/stop tracing a block
Main Flow

Step	Action
1	The Client asks the provider to start/stop tracing a block
2	The provider starts (stops) sending a message with the values of the in/outputs and controls.

References

[AAR02] Xavier Amatriain, Pau Arumí, and Miguel Ramírez. Clam, yet another library for audio and music processing? In *17th Annual ACM Conference on Object-oriented programming, systems, languages, and applications*, pages 46–47, 2002.

[Ale07] Megan Alexander. Understanding sound vs. music. Web, 2007. http://www.helium.com/items/89130-understanding-sound-vs-music, Last accessed: February 2012.

[Ama04] Xavier Amatriain. *An Object-Oriented Metamodel for Digital Signal Processing with a focus on Audio and Music*. PhD thesis, Universitat Pompeu Fabra, 2004.

[Ama07] Xavier Amatriain. Clam: A framework for audio and music application development. *IEEE Software*, 24:82–85, 2007.

[Blo06] Joshua Bloch. How to design a good api and why it matters. Keynote presentation at the 21st ACM SIGPLAN symposium on Object-Oriented Programming Systems, Languages, and Applications, 2006. http://lcsd05.cs.tamu.edu/slides/keynote.pdf, Last accessed: February 2012.

[BT05] Stuart Bray and George Tzanetakis. Implicit patching for dataflow-based audio analysis and synthesis. In *Proceedings of the 2005 International Computer Music Conference*, 2005.

[Bur98] Phil Burk. Jsyn - a real-time synthesis api for java. In *Proceedings of the 1998 International Computer Music Conference*, 1998.

[Bus02] Vannevar Bush. *As We May Think*. In [PJ02], 2002.

[CMRW03] Nick Collins, Alex McLean, Julian Rohrhuber, and Adrian Ward. Live coding in laptop performance. *Org. Sound*, 8:321–330, 2003.

[CS99] Perry R. Cook and Gary P. Scavone. The synthesis toolkit (stk). In *Proceedings of the 1999 International Computer Music Conference*, 1999.

[Dan97] Roger B. Dannenberg. Machine tongues xix: Nyquist, a language for composition and sound synthesis. *Computer Music Journal*, 21(3), 1997.

[Dan02] Roger B. Dannenberg. A language for interactive audio applications. In *International Computer Music Conference*, San Francisco, CA, 2002.

[DB96] Roger B. Dannenberg and Eli Brandt. A flexible real-time software synthesis system. In *Proceedings of the 1996 International Computer Music Conference*, 1996.

[DBC+98] François Dechelle, Riccardo Borghesi, Maurizio Ceccco, Enzo Maggi, Butch Rovan, and Norbert Schnell. jmax: A new java-based editing and control system for real-time musical applications. *Computer Music Journal*, 1998.

[Dor05] Philip Dorrel. *What is Music? - Solving a Scientific Mystery*. Philip Dorrell, 2005. http://whatismusic.info/.

[DR95] Roger B. Dannenberg and Dean Rubine. Toward modular, portable, real-time software. In *Proceedings of the 1995 International Computer Music Conference*, 1995.

[FRM+99] Déchelle Francois, Borghesi Riccardo, De Cecco Maurizio, Maggi Enzo, and Rovan Butch. Jmax: Demonstration of an integrated environment for real-time musical applications. 23:50–58, 1999.

[FS09] Adrian Freed and Andy Schmeder. Features and future of open sound control version 1.1 for nime. In *NIME*, 2009.

[Hin11] Dion Hinchcliffe. The "big five" it trends of the next half decade: Mobile, social, cloud, consumerization, and big data. Web, October 2011. http://www.zdnet.com/blog/hinchcliffe/the-big-five-it-trends-of-the-next-half-decade-.../1811, Last accessed: February 2012.

[IEE98] IEEE. Ieee recommended practice for software requirements specifications. *IEEE Std 830-1998*, 1998. http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=720574.

[Jac04] Mathieu Jacques. Api usability: Guidelines to improve your code ease of use. Web, November 2004. http://www.codeproject.com/Articles/8707/API-Usability-Guidelines-to-improve-your-code-ease, Last accessed: February 2012.

[Jor03] Ken Jordan. Defining digital multimedia, 2003. http://www.kenjordan.tv/Defining%20Digital%20MM%20Draft.doc, Last accessed: February 2012.

[Kay02] Alan Kay. *User Interface: A Personal View*, pages 121–131. In [PJ02], 2002.

[KG03] Alan Kay and Adele Goldberg. *Personal Dynamic Media*, pages 393–404. MIT Press, 2003.

[KGH85] Myron W. Krueger, Thomas Gionfriddo, and Katrin Hinrichsen. Videoplace - an artificial reality. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 35–40. ACM, 1985.

[Laz00] Victor E. P. Lazzarini. The sndobj sound object library. *Organized Sound*, 5:35–49, April 2000.

[Laz01] Victor Lazzarini. Sound processing with the sndobj library: An overview. In *Proceedings of the 4th International Conference on Digital Audio Effects*, 2001.

[LSDJ08] Michael S. Lew, Nicu Sebe, Chabane Djeraba, and Ramesh Jain. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE*, 96(4):668 –696, April 2008.

[Lyo02] Eric Lyon. Dartmouth symposium on the future of computer music software: A panel discussion. *Computer Music Journal*, 26:13–30, 2002.

[McC] James McCartney. Supercollider server synth engine command reference. Web. http://supercollider.svn.sourceforge.net/viewvc/supercollider/trunk/common/build/Help/ServerArchitecture/Server-Command-Reference.html, Last accessed: February 2012.

[McC02] James McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61– 68, 2002.

[McK10] Cory McKay. *Automatic Music Classification with jMIR*. PhD thesis, McGill University, 2010.

[MF10] Cory McKay and Ichiro Fujinaga. Improving automatic music classification performance by extracting features from different types of data. In *Proceedings of the international conference on Multimedia information retrieval*, pages 257–266, New York, NY, USA, 2010. ACM.

[MWC07] Ananya Misra, Ge Wang, and Perry R. Cook. Musical tapestry: Re-composing natural sounds. *Journal of New Music Research*, 36(4):241–250, 2007.

[Nel87] Theodore H. Nelson. *Computer Lib/Dream Machines*. Microsoft Press, Redmond, WA, USA, 1987.

[Nob09] Joshua J. Noble. *Programming interactivity a guide for Processing, Arduino, and OpenFrameworks*. O'Reilly Media, 2009.

[OGMR10] João Lobato Oliveira, Fabien Gouyon, Luís Gustavo Martins, and Luís Paulo Reis. Ibt: A real-time tempo and beat tracking system. In *Proceedings of the 2010 International Society for Music Information Retrieval*, 2010.

[PAP+06] Stephen T. Pope, Xavier Amatriain, Lance Putnam, Jorge Castellanos, and Ryan Avery. Metamodels and design patterns in csl4. In *Proceedings of the 2006 International Computer Music Conference*, 2006.

[Per06] Isabelle Peretz. The nature of music from a biological perspective. *Cognition*, 100(1):1 – 32, 2006.

[PJ02] Randall Packer and Ken Jordan. *Multimedia: from Wagner to Virtual Reality*. WW Norton Co, 2002.

[PK05] Kylie A. Peppler and Yasmin B. Kafai. Creative coding: Programming for personal expression. 2005.

[Pop93] Stephen Travis Pope. Machine tongues xv: Three packages for software sound synthesis. *Computer Music Journal*, 17:23 – 54, 1993.

[PR03] Stephen T. Pope and Chandrasekhar Ramakrishnan. The create signal library ("sizzle"): Design, issues, and applications. In *Proceedings of the 2003 International Computer Music Conference*, 2003.

[Puc91] Miller Puckette. Combining event and signal processing in the max graphical programming environment. 5(3):68–77, 1991.

[Puc96] Miller Puckette. Pure data: Another integrated computer music environment. In *Proceedings of the Second Intercollege Computer Music Concerts*, 1996.

[Puc02] Miller Puckette. Max at seventeen. *Computer Music Journal*, 26:31–43, December 2002.

[RF07] Casey Reas and Ben Fry. *Processing: A Programming Handbook for Visual Designers and Artists*. MIT Press, 2007.

[RS] Jonatan Rubio and Santi Serrano. Dotnetprocessing documentation. Web. http://dnetprocessing.sourceforge.net/docbook/view/index.html, Last accessed: February 2012.

[SC05] Gary P. Scavone and Perry R. Cook. Rtmidi, rtaudio and a synthesis toolkit (stk) update. In *Proceedings of the 2005 International Computer Music Conference*, 2005.

[Sca02] Carla Scaletti. Computer music languages, kyma, and the future. *Computer Music Journal*, 26(4):69–82, 2011/11/11 2002.

[Sch94] Bill Schottstaedt. Machine tongues xvii: Clm: Music v meets common lisp. *Computer Music Journal*, 18(2):30–37, 1994.

[SFW10] Andrew Schmeder, Adrian Freed, and David Wessel. Best practices for open sound control. In *Linux Audio Conference*, 2010.

[SJ] Catherine Schmidt-Jones. Talking about sound and music. Web. http://cnx.org/content/m12373/latest/, Last accessed: February 2012.

[SJ88] Carla Scaletti and Roger Johnson. An interactive environment for object-oriented music composition and sound synthesis. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPSLA '88, pages 222–233, New York, NY, USA, 1988. ACM.

[Sor05] Andrew Sorensen. Impromptu : an interactive programming environment for composition and performance. In *Australasian Computer Music Conference 2009 : Improvise*, Queensland University of Technology, Brisbane, July 2005.

[TC99] George Tzanetakis and Perry Cook. Marsyas: A framework for audio analysis. *Organized Sound*, 4:169–175, December 1999.

[TJC+08] George Tzanetakis, Randy Jones, C. Castillo, L. G. Martins, L. F. Teixeira, and M. Lagrange. Interoperabilty and the marsyas 0.2 runtime. In *Proceedings of the 2008 International Computer Music Conference*, 2008.

[Tza02] George Tzanetakis. *Manipulation, Analysis and Retrieval Systems for Audio Signals*. PhD thesis, Princeton University, 2002.

[Tza07] George Tzanetakis. *MARSYAS-0.2: A Case Study in Implementing Music Information Retrieval Systems*, pages 31–49. Information Science Reference, 2007.

[Ver86] Barry Vercoe. *Csound: A Manual for the Audio Processing System and Supporting Programs with Tutorials*. MIT Media Lab, 1986.

[Wan08] Ge Wang. *The ChucK Audio Programming Language: A Strongly-timed and On-the-fly Environ/mentality*. PhD thesis, Princeton University, 2008.

[WC03] Ge Wang and Perry R. Cook. Chuck: A concurrent, on-the-fly audio programming language. In *Proceedings of the 2003 International Computer Music Conference*, pages 219–226, 2003.

[WC04] Ge Wang and Perry R. Cook. Chuck: a programming language for on-the-fly, real-time audio synthesis and multimedia. In *Proceedings of the 12th annual ACM International Conference on Multimedia*, pages 812–815, 2004.

[WFC07] Ge Wang, Rebecca Fiebrink, and Perry R. Cook. Combining analysis and synthesis in the chuck programming language. *Proceedings of the 2007 International Computer Music Conference*, 2007.

[Wri02] Matt Wright. The open sound control 1.0 specification. Web, March 2002. http://opensoundcontrol.org/spec-1_0, Last accessed: February 2012.

[wsa] Aura - real time distributed objects for interactive multimedia. Web. http://www.cs.cmu.edu/~auraRT/, Accessed: February 2012.

[wsca] Chuck : Strongly-timed, concurrent, and on-the-fly audio programming language. Web. http://chuck.cs.princeton.edu/, Last accessed: February 2012.

[wscb] Cinder - the library for professional-quality creative coding in c++. Web. http://libcinder.org/, Last accessed: February 2012.

[wscc] Clam website. Web. http://clam-project.org/, Last accessed: February 2012.

[wscd] Creative applications website. Web. http://www.creativeapplications.net/, Last accessed: February 2012.

[wsma] Cycling 74. Web. http://cycling74.com/, Last accessed: February 2012.

[wsmb] Marsyas. Web. http://www.marsyas.info/, Last accessed: February 2012.

[wso] Introduction to osc. Web. http://opensoundcontrol.org/introduction-osc, Last accessed: February 2012.

[wspa] Implicit patching vs. explicit patching - marsyas user manual. Web. http://marsyas.info/docs/manual/marsyas-user/Implicit-patching-vs_002e-explicit-patching.html, Last accessed: February 2012.

[wspb] Pure data - pd community site. Web. puredata.info/, Last accessed: February 2012.

[wsr] Csounds.com. Web. http://www.csounds.com/, Last accessed: February 2012.

[wsr05] Rtcmix - an open-source, digital signal processing and sound synthesis language. Web, 2005. http://rtcmix.org/, Last accessed: February 2012.

[wss] Supercollider. Web. http://supercollider.sourceforge.net/, Last accessed: February 2012.

[Zic02] David Zicarelli. How i learned to love a program that does nothing. *Computer Music Journal*, 26(4), 2002.

REFERENCES