DYNAMIC SOUNDSCAPE COMPOSITION IN GAME CONTEXTS

by

DURVAL NUNO SIMÕES PIRES

BSc, University of Coimbra 2012

A DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF INFORMATICS ENGINEERING

Department of Informatics Engineering
Faculty of Sciences and Technology

UNIVERSITY OF COIMBRA
Portugal

2012/2013

Supervised by:
Licínio Roque

Abstract

Most sound design techniques and tools used today were adopted from linear mediums, proving insufficient to cope with the dynamic nature of videogames. Moreover, Middleware tools, which are an alternative to deal with this challenge, can be out of range for indie developers, due to their price and required know-how.

In this dissertation, we propose a Dynamic Soundscape Composition system, which aims to enhance soundscapes experienced during gameplay, by means of a holistic approach to Sound Design. The system implements soundscape composition techniques that follow principles from Acoustic Ecology, specifically, the notion of healthy soundscape.

This composition module is supported by a set of composition techniques which attempt to deal with the dynamic nature of the medium. Also, the module composes the soundscape in a dynamic fashion, by following the designer's intentions defined in the game code by a new API, also presented in this work. The system provides designers the chance to characterize the soundscape by means of this accessible and intuitive API. This API empowers designers by allowing intentions for sound design to be materialized using sound design vocabulary, as opposed to plain programming procedures.

Following specifications for the system's architecture, the API was validated by a formal usability test. Results of the experiment showed the API, although requiring time for an adaptation process, can be seen as an accessible alternative in terms of describing and implementing soundscapes in videogame contexts.

Table of Contents

List of Figures

List of Tables

List of Appendices

Glossary

AC-3 – Audio encoding algorithm.

Acoustic Ecology – Is the study of the relationships between the acoustic environment, orsoundscape, and those inhabiting that environment, with emphasis on balance and on the sense of the context.

Ambiance – Sound Layer which represents sounds of the environment.

Audio-only game – Game that consists mainly on sound, usually with no visual component.

Attenuation – The attenuation that sound suffers according to the distance between the source and the listener.

Designers – In this dissertation, designers should be interpreted as either game designers, programmers, sound designers, or any type of designers who have the required programming skills to use the API.

Dialogue – Sound Layer which represents all forms of discourse presented along the game.

Diegetic - Sound whose source is visible on the screen or whose source is implied to be present by the action of the film.

Digital Signal Processing - Digital Signal Processors take real-world signals like voice, audio, video, temperature, pressure, or position that have been digitized and then mathematically manipulate them. They are commonly used to create or modify audio signals.

DTS – Company that works on the area of digital sound, most famous by their high quality audio codec named DTS Digital Surround.

Frequency Modulation - A method of transmitting information using a radio-frequency carrier wave. The frequency of the carrier wave is varied in accordance with the amplitude and polarity of the input signal, the amplitude of the carrier remaining unchanged.

Foley – Sound Layer which represents sounds that characterize an entity or event.

Game Engine - System designed for the creation and development of video games.

Gameplay – The interaction the player has with a game.

iMUSE – It is an interactive music system created by LucasArts video games. It synchronizes music with the visual action in the game, and transitions from one musical theme to another.

Indie developers – Developers which commonly have no support in the game industry (i.e., with no publisher). Tipically consist in small teams that operate with meager budgets. Their context of development usually gives more space for innovation and for taking risks with new approaches.

Indie games – Games developed by indie developers, usually completely on their own.

Middleware - Audio Middleware tools try to aid sound designers in their work, by giving a more accessible interface and more powerful features than hard coded sound implementations.

MPEG-4 BIFS – Scene description language for audiovisual content.

MO3 – Music file format.

MOD - File format used primarily to represent music**,** which allows the selection of instruments, and definition of patterns describing how the instruments should play.

Music – Sound Layer which represents all musical sounds in the game.

Musical Instrument Digital Interface – It is a technical standard that describes a communication protocol, used by digital interfaces, that allows a wide variety of electronic musical instruments, computers and other related devices to connect and communicate with one another.

Nintendo Entertainment System – An 8-bit console made by Nintendo, released in 1983.

Non-diegetic - sound whose source is neither visible on the screen nor has been implied to be present in the action.

Non-Playing Character (NPC) – Any character that cannot be controlled by the player.

Panoramic – The distribuition of sound between left and right channels in a stereo sound.

Pattern Language – A set of design patterns that support a discourse specific to a domain, which facilitates communication between experts.

SFX **–** Acronym used for Sound effects. Also, Sound Layer that represents sounds that are created ("invented") so that they may be associated to an entity or aspect whose own sonic expression does not exist or is not perceptible [Alves 2011].

Sound Engine – System that executes all the functionalities regarding sound in a videogame.

***Sound Layer* –** Semantic categorization of sounds, as presented by Peck in [Peck 2001]. He proposes five: Ambiance, Dialogue, Music, Foley and SFX.

***Soundscape* -** is the group of sounds which compose a determined sonic environment (is the acoustic manifestation of ‘place’).

Soundscape Composition – As supported by Acoustic Ecology, soundscapes can be understood as musical compositions, in which everyone is both listener and composer. In the specific field of sound design, it can be perceived as the preservation and enhancement of a soundscape that respects the principles defended by Acoustic Ecology theory.

Virtual Studio Technology – Is a type of interface that allows the integration of many types of audio related tools (synthesizers, effect plugins, etc.) into other software programs.

Acknowledgements

My first words of gratefulness must go to my supervisor, Professor Licínio Roque, for all the advices and encouragement while accompanying me throughout this journey. Without his wealth of knowledge, support and friendship, I would not be able to advance in my research, as well as growing as a researcher and as a person. I must also thank him for his patience, for always answering my innumerous questions, and for opening my horizons regarding videogames, sound, and science research.

I thank Valter Alves for sharing his endless knowledge of sound in videogames. His good will was priceless and every minute spent debating with him was infinitely enlightening.

I also have to gratefully acknowledge my colleagues from LabC62, at the Department of Informatics Engineering of the University of Coimbra, for all the feedback and enthusiasm that always shared with me. I have to make a special mention for Luís Pereira and Rui Craveirinha for continuously supporting me, and for making me look to the videogame medium with a whole new perspective.

I have to thank Bárbara Furtado and Hernani Costa for aiding in many different tasks, and for always being available to help me when they were not obliged to it. Finally, I have to thank my family and my girlfriend Matilde Bourhis for all the patience supporting me.

In memory of Dário Teixeira. We miss you.

1 Introduction

Since the birth of videogames, sound and music in games has evolved rapidly. While sounds from the first arcade machines were merely designed to attract attention, current games feature dynamic soundtracks to elevate their profound experiences, along with context aware sound effects that contribute to players' emotional involvement [Peerdeman 2010]. As videogames have been acquiring, in the last years, legitimacy as a form of artistic expression, they have originated many aesthetic, moral and technological debates [Dodds 2008].

The study of videogames by the academic community is a relatively recent phenomenon, one which requires a careful reconsideration of the established theories and approaches to sound in media, as well as the research of new ones. Researchers should be wary of theoretical imperialism and the colonization of game studies by theories from other fields, especially from linear types of media [Kerr 2006]. Although past literature in the area of musical technologies argued towards a total technological determinism [Katz 2010; Theberge 1997], recent approaches refuted that argument and defend a mutual influence hypothesis rather than dominance between technology over aesthetics. In Video game audio, albeit many technological constraints deeply influenced some design choices, programmers (and later, sound designers) always came up with creative alternatives in order to overcome and to aestheticize those limitations [Collins 2008].

However, audio keeps being rejected as one of the most important features in a game, both in academic studies and from commercial products [Wolf and Perron 2003]. Still, game audio is still an aesthetic choice of metaphors, an arrangement of content that completes the video game as a totally integrated work of art [Peerdeman 2010]. Different solutions regarding different games must be researched, in order to allow the audio component of a game to be taken as seriously as it deserves.

1.1 Motivation

Similarly to Dodds, we defend that, just as games can be purely visual experiences, and promote visual-aesthetic awareness, they can equally promote sonic awareness [Dodds 2008]. Unfortunately, few games promote attentiveness to sound, and those who do it, only do it for brief moments, what prevents the player from really appreciate the soundscape. Every time this happens, part of the sound designer’s hard work is wasted.

Maybe one of the reasons for this negligence regarding game audio is that it is content which is not as easy to show off, as are graphics in a screenshot. Another reason may be our natural environmental awareness characteristics, which follow a path of least effort for each sense, as shown in Southworh’s study [Southworth 1967]. In other words, each sense contributes the minimum information necessary, unless it is being relied upon exclusively [Truax 2001]. This can help us understand why gamers rely so much on visual perception, and so little on auditory stimuli. As long as players “see” the game, they will listen to it as little as possible.

It is up to sound designers and game developers to create games which promote auditory awareness, while offering dynamic experiences with profound semantic value in their sonic component.

1.2 Context

Game Sound Design is an area the Information Systems Group has invested significantly in the past years having developed a Pattern Language for Sound Design in Games (PL4SDiG) [Alves 2011]. If in sequential media like cinema, Sound Design is already a hard task, in digital games it is made even harder by the fact that sound composition needs to be considered in an interactive context that is open to redefinition by player actions [Young 2012].

Current engine support for sound mostly focus on basic sound play and mixing techniques, possibly considering positioning and specificity of several sound sources as well effects that change over time. Besides bigger studios which have resources to buy expensive middleware solutions, game programmers need to manually control each sound source, setting its play timing and modifications according to what’s happening in the game scenario.

Sound Design is a technique that focuses at a higher level and considers which sound elements serve each semantic purpose, as well as how these should be combined to produce desired effects. Soundscape design is a set of concepts and methodologies that try to support a holistic perspective of sound design in context and has been the foundation for developing the aforementioned pattern language.

1.3 Objectives

This dissertation's main goal is to propose an approach for Dynamic Soundscape Composition (DSC) in games with a holistic perspective to sound. Videogames being a non-linear medium, just the task of defining a soundscape specification API (Application Programming Interface) to aid this composition is quite complex. This API uses concepts extracted from Acoustic Ecology theory (see Acoustic Ecology), and from Alves's previous work on a PL4SDiG [Alves 2011]. Additionally, we designed a systematization of possible techniques to be used in DSC. In order to be able to test the aforementioned API, and to verify the feasibility of the proposed architecture, we also modeled and prototype a DSC module (DSCM) as a proof of concept to be integrated in a game engine. Furthermore, in the future this proof of concept can be used to test the proposed techniques.

With this module, we intend to develop game engine support for developers to approach the problem of sound in games using these concepts, enabling experimentation of solutions for the problem of dynamic soundscape composition in games. While this study does not aspire to have a definite solution to the problem, it could show some indications of how dynamic soundscape composition could be approached in the future.

2 State of the Art

In order to understand what dynamic soundscape composition is in the context of videogames, it is important to visit some theories, practices, technologies and works which cover several fields of study relevant for sound design and for games as media.

2.1 Videogame Audio History

This resume briefly describes the evolution of sound design and implementation techniques in videogames, growing from an experimental and programmer based sound synthesis process (in its inception phase) to a controlled and very well defined process, normally done by experts from fields like music and film.

2.1.1 The beginning

Long before the birth of videogames, sound was already a key part of some gaming experiences, like slot machines. Sound effects (i.e., bells) were used to attract attention to the machines [Lastra 2012] and to originate the feeling of achievement after a successful or almost successful play.

Many years later, one of the most important artifacts which made the sound of Video games famous was Pong, due to the unmistakable sound of its paddles hitting the ball. It is interesting that, although many of the sounds that were used at the time were strongly constrained by technology limitations, they were advertised to the public as realistic. This trend towards realism influenced not only sound design, but almost all of the fields of game design. However, sounds were very difficult to program on early machines, due to hardware limitation, which affected both sound generation and memory limitation.

Only in the late 70’s music started to be used in Video games, being Space Invaders [Taito 1978] one of the most notorious examples. Following this trend, arcade manufacturers began to include dedicated sound chips in their circuit boards, usually used for tone generation [Collins 2008]. Game systems started to have a dedicated processor to deal with sound, which ended the concurrency between music and sound effects, which was one of the biggest limitations. Nevertheless, mixing was rarely a consideration, so it was not unusual to hear music and sound effects to clash with each other.

With the invasion of home consoles in the 80s, sound chips continued to evolve and to allow more creativity to programmers. Metroid [R&D and Systems 1986], for NES (Nintendo Entretainment System), is a great example. As its composer Hirokazu Tanaka explains, its music was composed not as just background music, but as part of the game's world, without any distinction between music and sound effects [Collins 2008]. However, home consoles still had memory limitations (besides from others), so looping was the solution for most composers. The length of a loop was usually related to the game's genre, and to the game state (boss battle, difficulty level, etc.). Transitions between loops were dealt in different ways: either the loop was designed so that the last section would fit with the beginning of the loop, or a small transitional sequence was used in between looping. Still, most transitions would often cut abruptly [Collins 2008]. This technique continued to be used for many years, and is still being used nowadays (i.e., puzzle games and mobile gaming).

Other ways of overcoming technology limitations were random sequence composition (used in some Commodore games), algorithmic variations on the composition [Games 1984]. This can be considered one of the first attempts to dynamic videogame music composition, even if it was not influenced by players' actions (Adaptive sounds). The fact that in many games the programmer was also the sound designer and composer (who didn't have formal music training), influenced largely the aesthetic of the sound from that era.

2.1.2 Game Audio Evolution

The 16-bit brought some advances like Frequency Modulation (FM) (which allowed a greater number of even more realistic sound effects), and Digital Signal Processor (DSP) (which supported a large number of effects and MIDI (Musical Instrument Digital Interface) instruments and would become very important in the next generations of consoles). Nevertheless, the structure of sound design remained very much the same from the 8-bit era.

At the same time, some computers started to support MIDI compositions, which eased the life of composers who didn't understand anything of programming languages. Another novel approach that came out for PCs (Personal Computers) was LucasArts' iMUSE (Interactive MUsic Streaming Engine), a system to allow composers to create (more) dynamic pieces. One of the creators, Michael Land, referred: ‘‘*the thing that’s hard about music for games is imagining how it’s going to work in the game. The iMUSE system was really good at letting the composer constantly test out the various interactive responses of the music: how transitions worked between pieces, how different mixes sounded when they changed based on game parameters, etc. Without a system like that, it’s much harder to conceive of the score as a coherent overall work*’’ [Mendez 2005]. While MIDI sequencing was directed for linear music, iMUSE allowed composers to create branching and conditional clauses in their pieces, called decision points, in which some predetermined conditions were tested. iMUSE available actions included instrument changes, looping capacity, panning, etc. iMUSE helped to set a precedent for music to be more responsive to players’ actions, distinguishing game music from that of linear media. Another important advance in the 90s was 3D sound, which allowed games to inform players about what is happening around them [Miller 1999]. It was in this era that Video game audio started to be seen more seriously as part of the game development process.

The next generation of consoles witnessed major improvements in almost every aspect of hardware specs. However, consoles like Playstation, which relied on Redbook audio (like PCs), gained more channels and higher sound quality at the cost of dynamic adaptability and interactivity [Collins 2008]. This implied more quick fades and hard cuts. One innovation brought by another console of this era was Nintendo 64’s MusyX, a program to allow Nintendo developers to compose music (similar to iMUSE).

The following generations of consoles brought some key features of present game audio like multichannel surround sound standards like AC-3 (Audio Coding 3), DTS and Dolby Digital, and more powerful DSPs [Collins 2008]. Later, Nintendo put a speaker on the Wii’s controller, which offered developers new ways of involving the player in the game.

2.2 Current Practices in the Context of Game Sound Design

In this sub-chapter, we will make an overview of the current practices in the context of game sound design. Firstly, the current videogame sound design process in the industry is going to be explained (Current Videogame Sound Design Processes). Next, the future role of mixing will be debated (Mixing). After that, some remarks about the dynamic nature of game audio will be made (see Dynamic Nature of Game Audio). Finally, we will cover the growing debate about the pros and cons of both sample based and procedural audio (Sample Based vs Procedural Audio).

2.2.1 Current Videogame Sound Design Processes

Usually, the sound design process follows a group of steps necessary for almost every game, almost like a waterfall-like approach, what makes it quite an inflexible work process. First, it is important to determine the game's theme and genre, and how sound will relate to gameplay, being a good practice to define cue point entrances, game state changes and how sound will be sensitive to game variables (Spotting). It is normally valuable to match the rhythm of gameplay with the sound [Collins 2008]. Next, a list of assets (sounds) needed is done by the sound designer. This task should be coordinated with the emotional rhythm desired for every section of the game. Different layers (foley, ambiance, music, etc.) should be taken into account during this phase. After its creation, sounds are inserted in the audio tool (if one is used) by the sound designer, which then defines rules and behaviors for the playback and synthesis of sound. These rules are based on interactive events which are linked between the sound engine and the game code [Chan et al. 2012].

All of the previous tasks should respect the technical limitations of the target game system and the tools and technologies available for audio implementation and integration. As Karen Collins refers: "*Sound design, dialogue, and music are as much about integration as they are about composition, and the ways in which the sound is implemented greatly affect the ways in which these sounds are received*" [Collins 2008]. A partnership between the technical side (audio programmers, etc.) and the creative side of audio (composers, sound designers, etc.) is absolutely vital, and without it, a game will only ever achieve average audio (John Broomhall).

2.2.2 Mixing

Although in the past mixing was not even a possibility for game audio, it is becoming an extremely important aspect of sound design for games, being even predicted that in the future, the role of "Game Mixer" will exist (Guy Whitmore, [Cavers 2011]). When it started to be applied in this media, it used to be done only in the post production phase (as in linear types of media). With the arrival of Middleware tools, real-time mixing became easier through features like mixer snapshots (group of parameter values that can be applied instantaneously in a single command), which take advantage of a tool's own bus hierarchy.

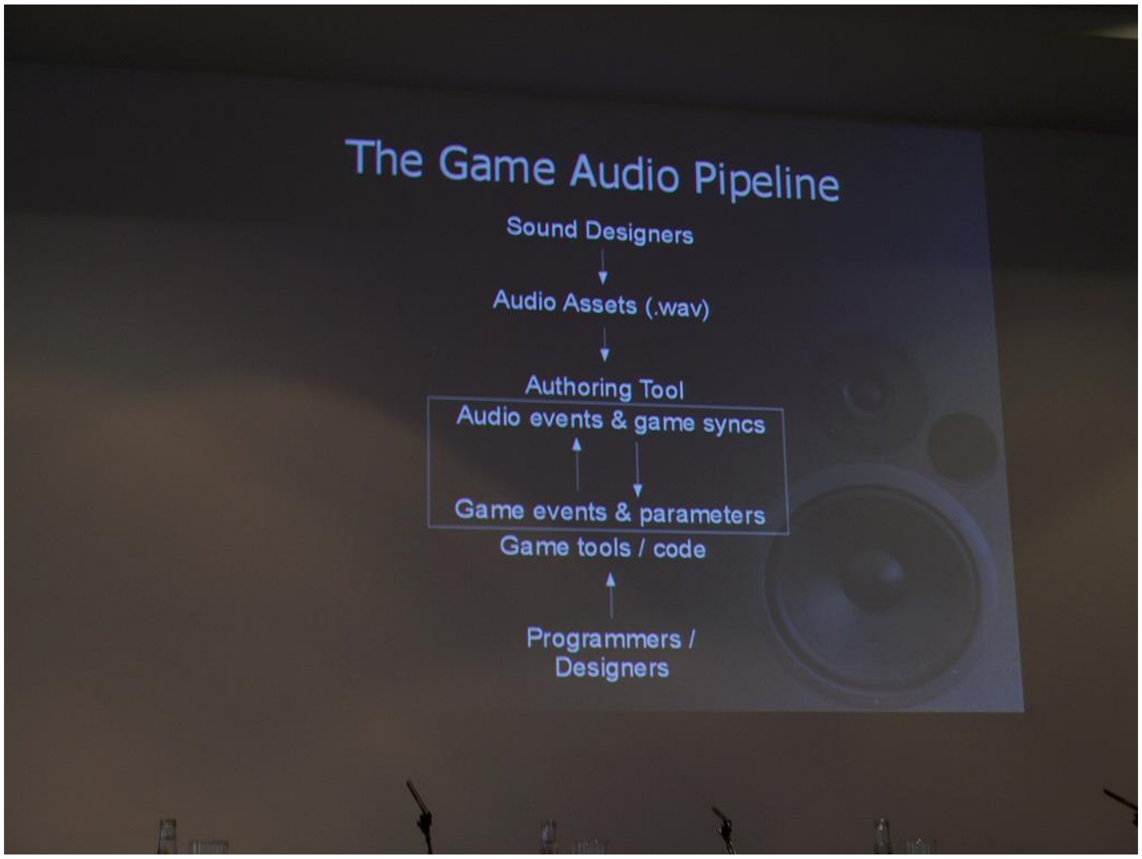


Figure 2.1 - Common Game Audio Pipeline [Cavers 2011]

Real-time mixing allows sound designers to work many aspects in an earlier stage of the game development process. It enables controlling a game's dynamic range (difference between the quietest and loudest volume), as referred by Kristofor Mellroth, Fable II's audio director ([Bridgett 2009; Studios 2008]), or reducing/eliminating specific frequencies from a determined sound in order to avoid superposition of sounds in a specific frequency range. One of dynamic mixing's most important feature is that it can be used to control volumes of different sounds (i.e., to duck every other sound when a dialogue is occurring). An example of the importance of mixing (and ducking) can be found in Little Big Planet (LBP) [Molecule 2008]. In Kenneth Young's (LBP audio director) words: *"Interestingly, despite the fact the characters speak with gibberish voices, it sounded weird not ducking other sounds for them. Before the fact I assumed it wouldn't matter what with their voices not containing any explicit information, but not focusing on their voices whilst they are "speaking" makes what they are saying (i.e. what you are reading) feel inconsequential. I guess that's a nice example of sound having an impact on your perception, and highlights the importance of mixing."* [Bridgett 2009]. On the same line of thought, Wwise's [AudioKinetic] product director Simon Ashby refers: *"The main complexity remains the interactivity, where the mixer has to take into account various different styles of gameplay; the soundtrack emerging out of a single game played by a Rambo-kamikaze gamer is way different than the one from a stealth type of gamer even though it is the same game using the same ingredients."* [Bridgett 2009].

Ultimately, the mix should be invisible to the player. They should not hear anything being turned down, or changing volume. The mix must not distract players, it should instead inform the player narratively and not just mirror what's seen on screen [Cavers 2011], and help them to focus their attention on what is important in an interactive, forever changing dynamic world.

An overview of today's most important mixing techniques [Bridgett 2009] are:

Grouping - The ability to assign individual sounds to larger controller groups.

Auxiliary Channels - These are extra channels, usually representing effects or different output paths such as headphone monitors, to which other designated channels can be routed. In videogame contexts, may be used to send the sound from a particular channel to a software reverb, also running in memory in real-time.

Fall-off – Relationship between sound's volume and effects, and its distance to the listener.

Passive Mixing Techniques - Values which, once set-up, attenuate parameters, volumes or filters of the content 'automatically' (examples: 3D volume fall-off curves of positional sounds and occlusion filtering settings of 3D sounds).

Active Mixing Techniques - This describes systems which allow greater control over sound parameters and the ability to completely override a passive system for a specific moment in time. These overrides often take the form of mixer snapshots in which parameters at the channel or bus level are redefined and then returned to normal once the event has finished.

2.2.3 Dynamic Nature of Game Audio

The intrinsic nature of audio in videogame contexts imposes a new approach to the classical categorization of sounds used in films: diegetic (sound whose source is visible on the screen or whose source is implied to be present by the action of the film) or nondiegetic (sound whose source is neither visible on the screen nor has been implied to be present in the action).

The fact that the player is part of the sound playback process, stand in need of a new type of classification. Besides diegetic and nondiegetic, game audio can be divided into nondynamic and dynamic, being that dynamic sound can still be divided in two other sub-categories: adaptive and interactive [Collins 2008].

- **Nondynamic sounds**
  - These are the situations in which the player does not have any type of control over the sound composition.
- **Dynamic sounds**
  - **Adaptive** - These are sound events which are affected by gameplay, but not directly affected by the player’s movements and actions.
  - **Interactive** - These are sound events which are affected by gameplay, and can be directly affected by the player’s movements and actions.

This dynamic nature is incremented because, in most games, players can interact with sounds in a wide variety of ways, which allows them to have many different functions. While in some games, sound is supposed to be a crucial part of the gameplay mechanic (like in stealth or rhythm-games), in others, sound is only decorative.

The biggest difficulty of linear sound composition in videogames is the lack of ability to match the actions occurring on the screen. This makes it hard for sound to fulfill the role it was supposed to have inside the game’s context. Sound is commonly used to alert players to something they do not see, to identify goals and objects of interest. Without sound, it is much more difficult for the player to understand symbols given by the game, to understand the game’s environment and mood, and to make sense of all the information he is absorbing through vision.

2.2.4 Sample Based vs Procedural Audio

As referred in Videogame Audio History, sound design and implementation techniques in videogames, started as a sound synthesis process, but with the advent of technology, mimicked the music and movie industry and became a sample-based process. This is largely influenced by the eternal search for realism found in the videogame industry, which always looked like a profitable perspective [Low 2001]. However, the current trend is to diverge from those techniques, as Robin Beanland refers: *"We need to move away from film, and develop tools that allow us to focus on interactivity"* [Cavers 2011]. Currently, sound designers record sounds as audio data, which is processed and manipulated using Digital Audio Workstations (DAWs) [Rutherford 2012]. Sample-based audio is natural to linear media such as film and music. However, it is very difficult to coordinate it with videogame's dynamic nature. That is why mixing and other features that Middleware tools offer are so important: to make linear audio work within the dynamic context of videogames. Even so, it may be concluded that game audio tries to reuse work processes and tools from mediums with different aesthetics, with linear principles [Lykke 2008]. This gives the sound designer greater quality on the sounds at his disposal, but one could argue it limits the possibilities for variation, and do not take advantage from the medium intrinsic dynamic nature.

It is important to point that procedural audio should not be viewed as the definitive answer to sample-based limitations, but should instead be seen as one of the possible alternatives/complements. In a nutshell, it is an approach where sound is not locked to time, but is instead locked to a defined setting [Lykke 2008]. This setting is defined through rules and parameters, which produce audio in real-time according to them. Changes inside the game directly affect sound generation because they are input for the aforementioned parameters. In this way, sound must be treated as a process of creation rather than playback manipulation of existing data [Rutherford 2012].

Procedural audio is greatly related to game audio rapid prototyping and experimentation [Knight 2011; Lykke 2008; Paul 2007; Paul 2008; Paul 2010]. Prototyping allows the use of tools to rapidly experiment interactive audio generation without involvement of a game audio coder [Paul 2007]. While prototyping, one should not be afraid of failure, should not spend too much time prototyping and is advised to find different solutions for the same problem [Gray et al. 2005]. In the future, the line between prototyping and implementation may fade as audio implementation tools begin to resemble more closely with tools previously reserved for prototyping [Paul 2007]. Clint Bajakian, Senior Music Supervisor at Sony Computer Entertainment America reinforces this idea: *"Artist creates conditions, rules and procedures, not necessarily the audio itself"* [Bajakian 2004].

The workflow of the creation process is also different between the two approaches. Sample-based audio is related to a sequencer paradigm, procedural audio is rooted in a synthesis paradigm. While in the former the sound designer arranges audio in a linear time-wise fashion, in the latter audio must be treated differently, as something that is shaped in real-time using paradigms normally used by programmers, while keeping in mind efficient and communicative aesthetics for good sound design. Moreover, the process becomes less waterfall-like and more iterative.

One could argue that procedural audio allows sound to adjust to the action taking place in the game, instead of what game developers and sound designers had foreseen, as in sample-based audio. Although it is surely a more responsive approach, it is not easy to tell if that is really better. A more dynamic sound generation also makes it difficult to predict what will occur in a specific event. However, the game/sound designer can think that if the player does not hear what he has foreseen, the emotion he desired to provoke would be lost. Additionally, rigid mappings between game parameters and sound synthesis can limit creative possibilities. Are we simply giving more variation possibilities to sound designers, or are we withdrawing artistic expression from them and their work? The answers to these questions depend on a sound designer's ability to use the best of both alternatives, even existing some procedural approaches that use samples [Rutherford 2012].

Another advantage of procedural audio is that reduces the storage and RAM (Random Access Memory) requirements for audio [Stevens and Raybould 2011], because audio is not played, but generated. On the other hand, while sample-based sounds have a fixed computational cost, in procedural audio, the more complex a sound is, the more computational work is required to produce it. Farnell also refers that with sample-based audio, there will always be a limited number of sounds per object/event, and that those sounds will be stimulated/generated in a limited number of ways [Farnell 2007].

Fournel summarizes procedural audio's usefulness: *"Procedural content generation is used due to memory constraints or other technological limitations. It is used when there is too much content to create, when we need variations of the same asset and when the asset changes depending on the game context"* [Fournell 2010].

2.3 Sound Computing Architecture

In a nutshell, there are three main architectures/approaches to videogame audio nowadays:

- Middleware Architecture
- Sound Lib/API Architecture
- Procedural/Dataflow Architecture

Different architectures offer different levels of definition of an actual sound's behavior. The more on the sound's behavior the game sound designer is able to describe, the less specifics the game audio coder will need to guess about. Behaviors' descriptions can range from a static description (such as the amount of pitch shift to randomly utilize) to a much more detailed dynamic scripting of a behavior (such as how the engine loops transition between one another in a car engine model). Currently, game audio tools are best at describing parameter ranges rather than allowing for the definition of dynamic behaviors [Paul 2007].

2.3.1 Middleware Architecture

Audio Middleware tools try to aid sound designers in their work, by giving a more accessible interface and more powerful features than hard coded sound implementations. Middleware lets sound designers link sounds to game objects, such as animations), scripted events or areas. While before there was always the need for a programmer, with middleware that is not the case anymore [Brandon 2007]].

Nowadays, Middleware tools give more power to sound designers with less complexity, allowing more dynamic soundscapes while diminishing production costs, both monetary and time related. However, many studios still have their own audio pipeline solution, due to monetary and technical constraints [Kastbauer 2010]. Still, Wwise and FMOD are widely used by major studios and some indie developers [Cavers 2011].

Features like parameter controlled DSP effects, dedicated prototyping environments, sound prioritization and real-time parameter controls, are extremely useful in order to create a more dynamic audio composition. Additionally, these tools allow the sound designer to work not only in parallel with the initial stages of game development, but also to develop sound to multiple platforms simultaneously.

Sample-based audio is natural to linear media such as film and music. However, it is very difficult to coordinate it with videogame's dynamic nature. That is why mixing and other features that Middleware tools offer are so important: to make linear audio work within the dynamic context of videogames. With the arrival of Middleware tools, real-time mixing became easier through features like mixer snapshots (group of parameter values that can be applied instantaneously in a single command), which take advantage of a tool's own bus hierarchy. Most games use a priority system to control which mixer snapshot prevails over others (as it can be seen in Little Big Planet and Heavenly Sword [Bridgett 2009; Theory 2007].

The usage of some Middleware tools prevents the necessity to define behaviors on game code, which will be easily defined by the sound designer in the authoring tool. This separation between game code and audio behaviors is highly positive for rapid prototyping and fast adaptation to changes.

2.3.2 Sound Lib/API Architecture

This approach was used in the past, and consists in using low-level sound APIs to program all sound related behaviors in the game code. This is not an easy task, and usually do not allow sound behaviors as complex as those which can be seen in games developed with the aid of middleware tools.

Nowadays, this approach is used in three situations: big developers who want to develop proprietary tools (instead of using commercial middleware tools), small developers who do not have money or knowledge to use middleware solutions, or when developers are looking to do something that is so out of the box that is just not possible in established tools *(*Jonatan Crafoord, [Cavers 2011]).

2.3.3 Procedural/Dataflow Architecture

Procedural audio is a philosophy about sound being a process and not data (as explained in Sample Based vs Procedural Audio) [Nair 2012]. This type of architecture is closely related to rapid prototyping and experimentation. Usually, the sound engine is built through a graphical programming language and communicates with game code through some interface like Open Sound Control (OSC) messages. Similarly to middleware tools, in this architecture, there is little code regarding audio inside the game code. All the behaviors (although not so complex as the ones developed in some middleware tools) are defined through dataflow modules which receive parameters from the game.

Procedural architectures are also closely related to experiments with sounds from particular objects or events. The main principle behind this approach is that, with the constant improvement of hardware, it is much more valuable for the sound designer to synthesize a sound through real-time mixing and real-time DSP effects, than having to record many variations of a sound, store each one in different files and having to load them during the game.

Most of the tools which follow this architecture offer a graphical programming environment. This type of environment offers objects (like visual boxes) that do a specific task. It is up to the sound designer to add them to a visual canvas and connect them. By combining objects, create interactive and unique software can be created without ever writing any code.

2.4 Sound Technologies

Different solutions may arise when it comes to game audio implementation. These solutions can be divided in three main groups: Middleware Tools, Low-level Sound Libraries/APIs, and Procedural/Dataflow Tools.

2.4.1 Middleware Tools

In this section, we will list all of the middleware tools that we find relevant to analyze for the sake of this study.

Wwise

Wwise is a Middleware Tool which is supported in an Authoring Tool and a Sound Engine which must be coded in the game to link it with the sound assets. Wwise's approach tries to ease the work of both sound designers and audio programmers by redefining the production workflow for audio and improving pipeline efficiency. While audio objects, which represent the individual sounds in the game, are created and managed exclusively within the Wwise application by the sound designer, game objects and listeners, which represent specific game elements that emit or receive audio, are created and managed within the game by the programmer. In a nutshell, Wwise's production pipeline can be summarized to: Audio Creation, Simulation, Integration, Mixing, and Profiling.

Wwise's most important features are innovative DSP effects, the definition of playback behaviors triggered by events, which are used to determine which sound, music, motion or piece of dialogue is played at any particular point in the game. Many actions can be linked to a specific game event and can affect more than one group of objects. In order to control and organize most of these entities, Wwise makes use of Hierarchy Mixers, which is an evolution from traditional mixing techniques where different instruments were routed to a bus, so that you could control their sound properties as a single mixed sound. This allows the sound designer to group sounds, motion and music objects in such a manner that creates parent-child relationships between the various objects [AudioKinetic].

However, Wwise's Sound Engine is very limited without using its Authoring Aplication, which makes it difficult to use as support for new applications which need only a Sound Engine to render its sound.

FMOD

FMOD started as a simple audio-engine but has evolved into a powerful videogame audio middleware tool, used in a large number of games, like Little Big Planet and Heavenly Sword [Bridgett 2009]. It is partitioned in different products that can be used in parallel to give powerful features to both the sound designer and the audio programmer. For instance, FMOD Studio is an authoring tool and run-time engine that allows audio content creation for games, with an interface that will resemble more a professional Digital Audio Workstations than existing game audio tools. Some of the features that contribute to this DAW feeling are a powerful multi-track event editor and a mixing desk with pro effects for mastering [Firelight]. Another of Studio's main features is the possibility to create, edit, mix and profile content live, which speeds up the sound designer's tasks. FMOD Studio also offers professional DSP effects.

FMOD Sound Engine provides great power to the audio programmer, ensuring maximum sound quality with features like floating point calculations, full 32bit interpolation and advanced compressed sample and streaming support. 3D positioning and HRTF (Head-Related Transfer Function) can be achieved easily through the API and Virtual Voices management using 3D distance and priority properties [Brandon 2007].

XACT

XACT is a proprietary middleware tool from Microsoft, provided with XNA, the game development framework from the same company. Similarly to XNA, supports development only Microsoft platforms. XACT is far more limited than Wwise or FMOD. Even so, it offers some features that can be useful to the sound designer. Sounds can be grouped in Wavebanks (cues that can contain more than one sound file), Sound Cues (Objects to which events and variables can be assigned) or Categories (to which some effects can be applied). Variables can be used by programmers to change Real Time Parameter Controls (values that can be changed in run-time and are input to some function to be computed), and some DSP Effects can be used by the sound designer. It relies a little more on the audio programmer than the aforementioned tools [Brandon 2007].

Psai

Psai is a recent interactive audio middleware tool that focuses on dynamic music composition, reactive to players' actions. Being music a linear medium, and most of the interactive mediums highly non-linear, creating interactive music for games is a complex task. Psai creators claim that it is the only middleware in the world fully geared to preparing and creating highly adaptive game music [Periscope Studio]. Although this statement is not exactly correct, Psai's approach does have some novelty in the way it tries to solve the problem at hand.

Psai consists of guidelines for the conception and production of both the game and the interactive music created for it. Additionally, Psai communicates with the game and controls the music. Combining a special musical AI with a novel production process, its engine tries to make the game's music follow the intensity of the plot and the actions. With Psai, the composer doesn't have to spend hours preparing transitions between different music tracks or ambiances, because Psai's logic will make all the decisions and choose the best transitions to make for the music be always coherent with the game's current state.

Psai core is integrated with games as a DLL (Dynamic Link Library). The code just has to trigger Psai core with simple commands, and Psai controls the music automatically, while intelligent triggers modulate the music. Programmers have to create one function to create the mood change, although it is bundled with logic modules made for specific game genres.

Sadly, there is not much information available about Psai besides its website. In fact, there is not even a concrete list of compatible platforms (it is only said to be compatible with PC and consoles).

Miles Sound System

Miles Sound System is one of the oldest middleware tools available in the gaming industry. Its age is shown in some of its main characteristics. For example, it has the smallest set of designer tools and almost all of its functionalities have to be implemented through the SDK (Software Development Kit) by programmers [RAD]. However, most of its functionalities can be programmed with less lines than in other engines, and Miles is considered the most reliable and robust audio engine around. By current standards, Miles appears to be a little dated. Without real-time parameter controls and some sort of GUI (Graphical User Interface) to let you organize files beyond a simple directory structure, it does not offer much in the way of additional features beyond DSP filtering and 3D sound. Nevertheless, sound designers continue to use it mainly because it is fast, solid and has great support with fast response to e-mails [Brandon 2007]. Unfortunately, it offers no free license of any kind.

Unreal Audio System

Nowadays, Unreal Engine is one of the most used Game Engines [Stevens and Raybould 2011]. Its built-in solution for audio is called Unreal Audio System. One of its main features is that the audio design follows the methodology behind the design of all the other components, which is, building the levels inside the game world while roaming around freely [Brandon 2007]. In this way, sound objects can be attached to game objects that are visible in the game world, which makes it easier to implement 3D sound positioning, spatialization and attenuation. Another advantage of this audio solution integration is that sound behaviors can follow the scripted events defined with the engine's scripting language: Kismet [Epic Games]. This way, the sound designer can take advantage of complex events and actions defined previously by another team member. Many audio effects are at the disposal of the sound designer (i.e., pitch control, modulation, etc.), and extensive debugging tools can be used to monitor resource usage.

Cry Engine Sound System

CryEngine is a Game Engine designed by Crytek and is primarily for use in first-person shooter video games. In a similar fashion to Unreal Engine, CryEngine has a built-in audio solution which offers many features like in game mixing and profiling and a data-driven sound system that guarantees multi-platform compatibility and individual performance optimization. Other important features offered by this solution is the possibility to configure dynamic sounds that react in a complex manner to parameters such as distance or time of day through real time DSP effects. It can also be defined events that will influence the composition of the game music, allowing the score to react to any desired game event, and the creation of a non-repetitive environmental ambiance [Crytek].

Sounds can be added directly onto blended animations to improve (for example) the implementation of foley effects. Once again, as similar to Unreal Engine's Kismet, Cry Engine supports a graph language that eases the creation of scripted events. This way, the sound designer can reuse Flow Graphs used for logic and physics.

2.4.2 Sound Libs/APIs

In this section, we will list all of the Sound Libs/APIs that we find relevant to analyze for the sake of this study.

Marmelade Audio

Marmalade SDK is a cross platform, software development kit for mobile devices. Its main feature is the high portability level that it enables, which allows development and deployment for different platforms without having any kind of restrictions [Marmalade]. Relatively to audio support, Marmalade provides a 24-channel software sound mixer, support for different audio formats like wav or pcm. It is not provided much more info, but from an analysis made to Marmelade's API Reference, it appears to have a low-level support of audio, enabling only basic commands like play, stop, etc. There are available on the web some user-made Sound Engines, however, they are at an early stage of development, allowing only basic usage, normally sufficient for mobile games.

Open-AL

OpenAL (Open Audio Library) is a cross-platform audio API. It is designed for efficient rendering of multichannel three dimensional positional audio. Its API style and conventions deliberately resemble those of OpenGL (Open Graphics Library). Early versions of the framework were open source, but some of the later revisions are proprietary [Wikipedia].

In order to achieve a good three dimensional positional audio, OpenAL uses source objects that contain properties like velocity, position, direction, etc. However, only a single listener can be defined. The rendering engine takes into account factors like distance attenuation, Doppler Effect, etc. Due to its OpenGL oriented skeleton, very little additional work is required to integrate 3D sound in an existing OpenGL-based 3D graphical application. Some versions support HRTF (Head-Related Transfer Function) mixing, which amplifies the realism of spatialized sound. In addition, it also offers high quality effects and filters, as well as support for multi-channel sound sources.

SDL

Simple DirectMedia Layer (SDL) is a cross-platform, free and open source multimedia library [SDL]. Although its audio support appears to be very basic and limited, it has been used in some small indie games and prototypes. Unfortunately, the only information available is the possibility to convert formats if they are not supported by hardware, and that it is designed for custom software audio mixers. It appears that this library does not offer all the mixing functionalities that it may be needed for the project at hand.

SFML

SFML (Simple and Fast Multimedia Library) is a portable and easy-to-use API for multimedia programming that provides low and high level access to graphics, input, audio, etc. [SFML]. It is an object oriented alternative for the SDL. Its main audio features are hardware acceleration, 3D sound spatialization and multi-channel formats (stereo, 4.0, 5.1, 7.1, etc.). It appears to be a little more powerful than SDL, although, once again, it appears that may not offer all the mixing functionalities needed.

BASS

BASS is a free for non-commercial use audio library that provides developers with powerful and efficient sample, stream, MOD and MO3 music, as well as recording functions, delivered in a compact DLL that won't bloat your distribution [Un4seen]. Its main audio features are: support of different types of audio streams; support for multiple channels; 3D sound and DSP effects. It appears to be one of the most powerful standalone Sound Lib.

irrKlang

irrKlang is a free for non-commercial use high level sound engine and audio library [Ambiera]. It has base features known from low level audio libraries, as well as others like a sophisticated streaming engine, extendable audio reading, single and multithreading modes, 3D audio emulation for low end hardware, multiple roll off models, etc. Additionally, irrKlang also offers many sound effects such as echo and chorus, performance and memory management, and low level audio output manipulation (i.e. panning and volume).

Hekkus Sound System

Hekkus Sound System is a small and fast sound engine for mobile platforms (among others), specially designed for games [Maniero]. It is free for non-commercial use. There is little information available about Hekkus, but it seems to be a recent project that, has been receiving updates and revisions in the past few months. The main features offered by Hekkus are a fast mixer routine which allows unlimited music and sound channels and an accessible API. Hekkus is strongly focused to mobile gaming, which means that it may not be powerful enough to deliver all the requirements that may arise from the project to be developed.

JUCE

JUCE is an all-encompassing C++ class library for developing cross-platform software [Raw Material Software]. Besides playing audio streams, JUCE has more interesting features like support for mixers and tone-generators. However, it seems that it may be too limited for the requirements that may arise from the project's specification. In addition, the details of how audio is implemented in the framework are not available to the user, which can make it difficult to change low-level details in the library.

PortAudio

PortAudio is a cross-platform, open-source library for real-time audio input and output [PortAudio]. The library provides functions that allow the acquisition and output of real-time audio streams from the computer's hardware audio interfaces. PortAudio is used to implement sound recording, editing and mixing applications, software synthesizers, effects processors, music players, internet telephony applications, software defined radios and more. PortAudio offers many low-level functionalities and the source code is at the programmer's disposal. However it does not support a very important feature: 3D Sound.

Audiere

Audiere is a high-level audio API. It can play many file formats, as well as many OS-native audio API (DirectSound, WinMM, OSS, etc.) [Audiere].

Although it does not have much information about it available, it claims to offer an easy API, volume, pan and pitch modification and noise generators. Without knowing more about it, it is hard to classify it as a powerful possibility to create a dynamic soundscape engine.

Kowalski

Kowalski is a data driven, portable, high level API for real time audio. It was developed primarily to support the development of games and other interactive applications where audio plays a crucial role. Kowalski's approach tries to provide a data driven system that separates content from code in order to ease the management of audio content [Stuffmatic]. Its approach is based on a hierarchical mix bus system and it is the core of the Kowalski Project.

Kowalski's has many great features like: a good 3D audio support with many effects like distance attenuation, Doppler shift and positional panning, support for mix buses to allow a better control over mixes, and the capacity to save Mixer snapshots and to switch between different snapshots. Other features that are valuable are audio level metering, which makes it possible to keep track of the output levels and detect clipping, and an approximate sample clock API that facilitates the synchronization between audio and visuals [Stuffmatic].

Kowalski's main advantage over other libraries is that its data driven approach keeps code and data separated, which translates into complex content not implying complex code. This approach clearly tries to mimic some features normally offered by sound middleware tools, but that usually are not provided by sound libraries. Consequently, all of this makes Kowalski excel over other sound libraries, although it is still under development. Ultimately, The Kowalski Project can be seen as a hybrid approach between the middleware and the sound library layers.

2.4.3 Procedural/Dataflow Tools

In this section, we will list all of the procedural/dataflow tools that we find relevant to analyze for the sake of this study.

PureData

Pure Data (Pd) is a real-time graphical programming environment for audio, video, and graphical processing. It is supported by almost any platform available and, inclusively, new projects have been made to increase the number of supported platforms, like libpd [libpd], which enables Pd to have access to functions that would be better realized as a procedural piece of code rather than the modular coding style that Pd tends to encourage [Gauthier].

Pd was created to explore ideas of how to allow data to be treated in a more open-ended way and opening it up to any kind of applications, independently of what type of content they provided (audio, graphics, video, etc.) [Puckette]. Pd's gives the sound designer a great amount of freedom and helps him to express his creativity. Its popularity has been growing and it has already been used in commercial videogames [A Game Development Blog 2008]. Besides, works like [Farnell 2010] show that sound synthesis through Pd can play a major role in the sound design scene in the future.

Pd has a modular approach, which means that its reusable units of code written natively in Pd (called patches) can be extended by other modules or even other programming languages. However, being Pd also a programming language, its patches can also be used as standalone programs and freely shared among the Pd user community, requiring no other programming skill to be used effectively [Puckette]. Pd was always designed to do control-rate and audio processing on the host central processing unit (CPU), but rapidly became very useful in the creation of sound synthesis and signal processing through a digital signal processor (DSP). In the last few years, Pd has been used to create Procedural Audio effects to be used on videogames, and in some cases it has been used to do all sound design of a game. Usually, the binding between game code and Pd is done through Lua or OSC. As it can be seen in [Paul 2010], one of the advantages of using Pd to render a game's audio is the possibility of modifying both sample data and sound behaviors in real-time without a lengthy recompilation stage, which allows for rapid iterative game audio sound design (although some Middleware Tools allow similar functionality).

Max/MSP

Max is a visual programming language for music and multimedia, highly modular, with most routines (native Max code) existing in the form of libraries [Cycling 74]. Similarly to Pd, Max's user community is constantly creating new extensions that enhance Max's capabilities. Much like most procedural audio tools, Max does not impose one way to create. Instead, it gives tools to the sound designer to allow him to develop its own rules and ideas. Max allows the creation of tone generators, sample-manipulation, synthesis tools, high-quality filters, spectral processing, real-time recording, etc. Max (and Pd) do not only work with sound, VSTs (Virtual Studio Technology) or MIDI, but can also work with graphics and all sorts of input and output. An example of an audio-only game running just on Max/MSP is DeepSea [Wraughk]. Max has also been used for many game audio prototyping experiences, as it can be seen in [Knight 2011; Lykke 2008].

AudioMulch

AudioMulch is software for live performance, audio processing, sound design and music composition [AudioMulch]. Its premise is to allow the user to create by patching together a range of sound producing and processing modules. Nevertheless, unlike other patcher-based programming environments, AudioMulch's modules perform high-level musical functions, allowing the user to avoid creating things from the ground up using individual oscillators and filters. Some of the categories of high-level modules are: signal generators, effects, filters, dynamics processing, mixers, VST and Audio plugins.

Unfortunately, AudioMulch does not appear to have any available interfaces with external applications, leaving it almost as a standalone application. Its high-level functions could be useful to ease the learning curve of learning Pd or Max, however, its user-friendly approach could mean more limitations on its features than those of Pd or Max. Ultimately, AudioMulch appears to be better as a live performance tool than as a sound design one.

CLAM

CLAM is a full-fledged software framework for research and application development in the Audio and Music Domain [CLAM]. It provides a conceptual model and means to perform complex audio signal analysis, transformations and synthesis. It also provides a uniform interface to common tasks on audio applications such as accessing audio devices and audio files, thread safe communication with the user interface and DSP algorithms recombination and scaling. It can be used as a library to program applications but also can be used through graphical tools to build full applications without coding.

CLAM splits the processing in modules, so that it can be recombined as a network. Data flow can be controlled using different communication patterns, and networks can be executed within different environment backends like real-time applications, audio plugins, etc. It is possible to graphically prototype software, being very easy to integrate the result in a pure code project. All processing modules, user interface elements and backends create by the user community can be shared and extended via plugins.

CLAM is usually used for tasks such as audio analysis and synthesis (specially spectral analysis/synthesis) and Music Information Retrieval, Spectral processing, Spectral Modeling, Tonal analysis, Rhythm analysis and manipulation, etc. Additionally, CLAM is one of the few graphical programming tools that have a specialization plugin which provides many different algorithms to render 3D sound.

To summarize, CLAM appears to be a very powerful tool, especially in signal analysis tasks, which can be useful to the sound engine if frequency or pitch analysis turns out to be a requirement. However, it may be a little difficult to integrate with another Sound Library needed to produce some high level features like 3D sound spatialization.

2.4.4 Conclusion

A comparative table of all the tools analyzed in the previous sections can be found in Appendix A. This table has more details about each tool that, for the sake of briefness, could not be referred in this overview of the available tools.

2.5 Related Research

Sound Design is a rich area which can be approached from many different angles. Normally, works on this field of study only address a sub-problem due to its intrinsic complex nature. The two subsets which are closer to the work proposed in this dissertation are the sub-areas of audio-scene description and soundscape composition.

Regarding solely a musical approach, Eigenfeldt tries to build knowledge into autonomous agents to allow them to produce artistically interesting and compositionally satisfying soundscape compositions [Eigenfeldt and Pasquier 2011]. Using pre-analyzed soundscape samples, the agents try to avoid crowded spectral areas while maintaining a rich musical interaction with each other. Following Truax's soundscape composition guidelines [Truax 2002], according to the authors, "*A generative soundscape system must combine audio recordings in ways that rely upon an understanding of those recordings spectral components and semantic contexts*". A similar approach, through evolutionary engines, is proposed in [Fornari et al. 2008].

In [Macanulty and Durity], Macanulty and Durity propose a Contextually Driven Dynamic Music System for Games which provide musical selection, mixing and effects, which can be controlled dynamically or automatically through a logic system. a music playback system that includes opportunities for musical selection, mixing and music effects that can be controlled dynamically, and also by providing a logic system that can make control decisions based on game play. Thus, the system's architecture has a two layered approach to dynamic music - the logic layer, and the playback layer. This system is still under development, and has to show more to demark itself from similar systems.

Talktome [Yiannis 2012], is a project developed by a student in Berklee College of Music in Spring 2012, as part of my senior project in Electronic Production and Design. It is a prototype for a game audio middleware built entirely in Max/MSP and controlled by Unity3D, using Unity's tech demo, AngryBots. Talktome uses game events to define the game's level of intensity, in order to choose what musical cues to play. The difference from other similar prototypes is that the cues are chosen by a probabilistic algorithm. This way, the sound designer can control the probability value to different cues, and control the game's musical variations.

Finally, An overview of the evolution and future of adaptive game music can be found in [Young 2012]. It shows how much videogame music composition has been growing, and it illustrates various examples of well succeeded approaches in commercial videogames.

Regarding works which do not focus solely the music layer of sound design, in [Chan, Natkin, Tiger and Topol 2012], inspired by MPEG-4 BIFS [Scheirer et al. 1999], the authors use a scene description language (COLLADA) which factorizes common elements needed to describe both visual and auditory information. However, as referred in [Scheirer, Vaananen and Huopaniemi 1999], and in contrast with visual scene graphs, an audio scene represents a signal-flow graph describing digital-signal-processing manipulations. So, the objective of the work was to link auditory and visual information together to allow sound design to be developed closely to graphic assets. Additionally, a first approach to the concept of Soundscape is introduced, in order to give a sense of aural depth and a more human feeling to sound [Truax 2008]. The main achievement of this work is the independence between the proposed architecture and the Sound API, which allows the usage of different solutions.

Similarly, Game Audio Lab [Went et al. 2009] is a framework for academic purposes which enables rapid experimentation of dynamic sound design in gaming contexts. Through the mapping of gaming variables to composite variables (variables that express meaningful information about the game that is not available otherwise), researchers can easily adapt sound and music in real-time during gameplay. The framework's architecture separates the audio engine from the game code, so that designers can modify parameters and engine architecture in real-time. However, sound instances that are not affected by this process are played through the game's original sound engine. This approach requires the game code to be modifiable, which can sometimes be an obstacle. Another example of this kind of experimental rapid prototyping approach is Paul's Pure Data Sound Driver [Paul 2007; Paul 2008; Paul 2010], which supports SFX (Sound Effects), speech, adaptive music, among other types of sound applications. Once again, communication between game code and the sound driver is achieved via network through Open Sound Control (OSC) [Open Sound Control].

However, although not being a work directly related, the video SoundWalkers [Castro 2009] must be mentioned because it helped to better understand some concepts behind Soundscape Theory. Another good example of a work that embodied Soundscape Theory principles is [Alves 2011], which presents a different patterns of sound design in videogame contexts. Influenced by the Acoustic Ecology principles defended by Schafer and Truax (see Acoustic Ecology), Alves argues that sound design in games benefits from being embedded in the overall game design, and still have a great potential to be unlocked. This approach is based in design patterns and it is assisted by a deck of cards. This deck can help game/sound designers early in the design process to come up with creative ideas regarding sound usage in their game. All cards have relationships with others. Although the deck does not have any kind of layering/grouping system to help in the organization of the different patterns, the large number of relationships adds great semantic value to this work.

2.6 Acoustic Ecology

In the following sub-chapters, we will present the main inspiration for the approach we followed in this work. Acoustic Ecology theory argues that the sounds of an environment should be perceived as a whole, and understood as an ecologically balanced entity. In a soundscape, sounds are not arbitrary but instead a complex system of relationships between its inhabitants, and between them and that environment, with implications on timing and on auto-regulation when emitting those stimuli.

2.6.1 Soundscape Definition

Although its definition may vary from author to author, a soundscape is the group of sounds which compose a determined sonic environment (is the acoustic manifestation of 'place') [The Canadian Encyclopedia]. This concept was coined by composer, writer, music educator and environmentalist, Raymond Murray Schafer. During the 1960s he founded the World Soundscape Project (WSP), intended to work as an educational and research group. Moved by its awareness of the degradation caused by man to its sonic environment, he wrote two educational booklets regarding noise pollution: The New Soundscape and The Book of Noise. Despite some improvements achieved by these works, Schafer felt that a more positive approach had to be found [Truax].

Many works and studies began to be published by the WSP, most of them consisting of recordings and analysis of different locales in the world. The objective of this work was to develop the study of this novel field, soundscape ecology. Instead of just referring the problems of noise, Schafer tried to analyze different soundscapes, their properties, and how could they be protected and improved [The Canadian Encyclopedia].

According to Schafer, "*a soundscape has some sounds which are more important either because of their individuality, numerousness or domination*", and is composed mainly by three types of elements: **Keynote sounds**, **Signals** and **Soundmarks [**Schafer 1993**]**.

- **Keynote sounds -** It is the fundamental tone of a composition, according to which everything else modulates. Although not usually listened to consciously, they influence the behavior and rhythms of those who hear them, and allow other sounds to be distinguished in the soundscape (*The visual perception metaphor of figure and ground, the figure being that which is looked at while the ground exists only to give the figure its outline and mass* [Schafer 1993]).
- **Signals –** Are the foreground sounds which are listened to consciously. Using the aforementioned metaphor, they are figure rather than ground. They are intended to be listened, often as warning devices like bells or sirens.
- **Soundmarks –** Refers to a community sound, unique or with qualities which make it specially regarded or noticed by the people in that community. It helps to make the acoustic life of a community unique.

WSP's following work centered on Soundscape studies which tried to unite multiple isolated disciplines that studied Sound with their own frameworks and languages. The main challenge was in the discovery of the missing interfaces between the different fields, in order to unite people from different backgrounds to work towards a healthier sonic environment for our world.

2.6.2 Acoustic Communication

Barry Truax, a former Schafer student and member of the WSP, during his Soundscape studies, felt that, although sound is a vibratory motion, what is important from a human perspective is its effects as a form of communication. In his own words, Acoustic Communication "*is the most general way to describe all of the phenomena involving sound from a human perspective*". His objective was to understand what Sound's role was in the complex relationship between people and the environment and to try to protect and improve the existent Acoustic Communities (Soundscapes in which acoustic information plays a pervasive role in the lives of their inhabitants) [Truax 2001].

There were two pillars behind this approach to Sound: the notion of context ("*a sound means something partly because of what produces it, but mainly because of the circumstances under which it is heard*" [Truax 2001]), and the idea that sound, the listener, and the environment are not isolated entities with isolated connections, but are instead a complex system of relationships in which everything interacts and influences everything. The flow of communication is bidirectional, since the listener is also a sound maker. To summarize, context give us understanding of how a sound functions, knowing that its role is the mediation and creation of relationships between listener and environment.

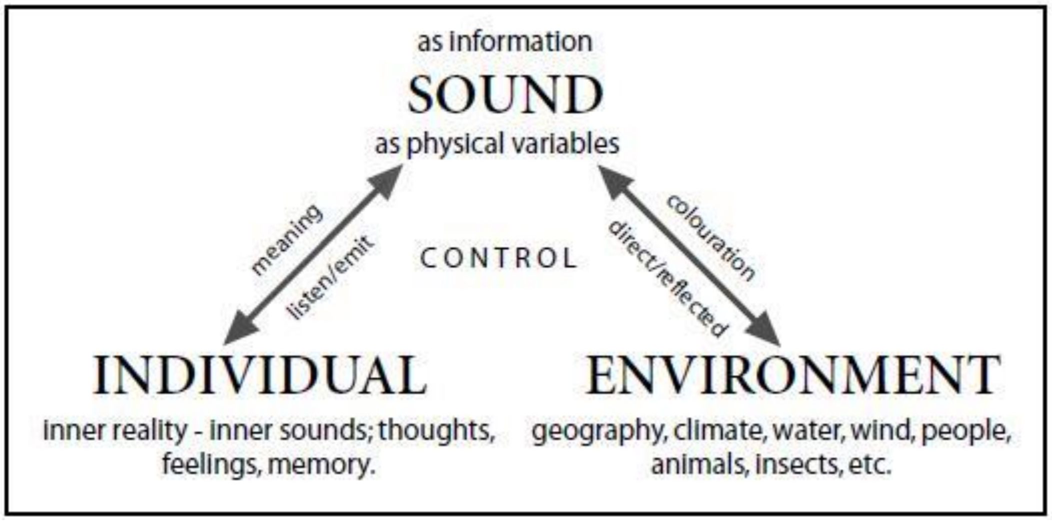

as information
SOUND
as physical variables
meaning
listen/emit
CONTROL
colouration
direct/reflected
INDIVIDUAL
inner reality - inner sounds; thoughts, feelings, memory.
ENVIRONMENT
geography, climate, water, wind, people, animals, insects, etc.

Figure 2.2 - The mediating relationship of an individual to the environment through sound (modified from [Truax 2001]) [Acoustic Ecology 2000].

While the Energy Transfer Model, the model used by most disciplines dealing with sound, focus on the energy transfers that produce a determined sound, the Communicational approach focus on the information communicated by that sound. Similarly, while the former implies the notion of hearing (processing acoustic energy in the form of sound waves and vibration), the latter makes use of the notion of listening (processing sonic information that is usable and potentially meaningful) [Truax 2001].

Listening is a key element in the communicational model, because it is the interface through which we obtain information from the environment. Listening habits should be considered as important as sound making ones. It should also be referred that, in order to maximize the quantity of information perceived by different listeners in different situations, different types of listening must be taken into account. While sometimes one consciously search the environment for clues (Listening-in-search), in other situations one is ready to receive information, even his directions is directed to something else (Listening-in-readiness). Moreover, there are situations where one almost ignores a sound, but is still aware of its occurrence (Background Listening). These are just three examples, but what is important to retain is that different types of listening must be taken into account for different situations.

In this approach, another vital concept is that memory does not simply stores a sound, but stores a pattern composed of a sound plus its original context. In Truax's words: "*Recalling the context may revive a memory of the sound, and the sound, if heard again, usually brings the entire context back to life*" [Truax 2001]. Due to this, the auditory system is always comparing retrieved information to patterns stored.

Truax also divided Sound in three systems of acoustic communication: Speech – Music – Soundscape. As we move along from left to right, the specificity in meaning decreases and the semantic level gets more complex, depending more on the relationship between elements, and between the elements and the whole.

2.6.3 Acoustic Design

According to Barry Truax, "*The concept of ''acoustic design'' refers to the analysis of any system of acoustic communication that attempts to determine how it functions. Criteria for acoustic design are obtained from the analysis of positively functioning soundscapes*" [Truax

2001]. It may involve changing both aspects from the environment and/or aspects of the listener (i.e. listening habits).

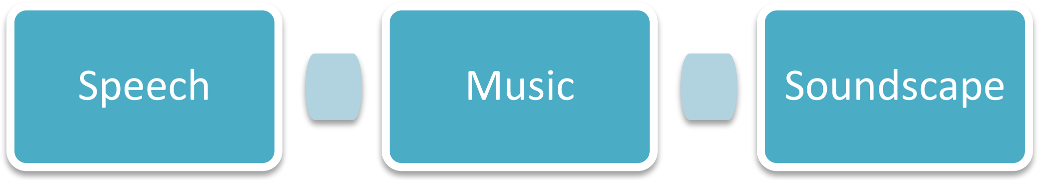


Figure 2.3 – Systems of Acoustic Communication Continuum [Truax 2001]

A functional acoustic system will certainly have the following three properties:

- ***Variety*** - Different kinds of sounds and their variations should be present and clearly heard. These sounds should be "rich" in acoustic information.

- C***omplexity*** - It exists within the sounds themselves and in the types and levels of information they communicate. Familiarity with the environment empowers listeners with the ability to decode and interpret subtleties in the sounds which are not recognized by novice listeners.

- **Functional balance** – It should be the result of spatial, temporal, social, and cultural constraints on the system's variety and complexity, in order to keep a functional equilibrium. However, the system's ability to defend itself from artificial and human perturbations is very limited.

To summarize, the physical properties of a sound are its natural ecological balance system. However, without the environment's constraining forces, there would be too much complexity and sensory overload would prevent effect information exchange through sound. This equilibrium mechanic tries to prevent sounds whose energy is predominantly in the same part of the spectrum to be heard at the same time. Contrarily, sounds in distinct ranges may be heard clearly even if they have different intensity levels.

2.6.4 Soundscape Composition

At Simon Fraser University, along with the evolution of these concepts, Barry Truax and many others involved with acoustic design started to mix electroacoustic techniques with soundscape recordings, which resulted in a new style of electroacoustic music [Truax 2008]. This genre is characterized by the presence of recognizable environmental sounds and contexts, the purpose of being able to invoke the listener's associations, memories, and imagination regarding the soundscape [Truax 2002]. According to Truax, it is essential for the composition to play with the listener's associations between the recordings, because the lack of apparent semantic relationship between sounds prevents a syntax from being developed in the listener's mind [Eigenfeldt and Pasquier 2011].

2.6.5 Acoustic Ecology Ideal

Acoustic Ecology is becoming so important that nowadays it is being leveraged with landscape ecology in order to create a new field of study called Soundscape Ecology [Pijanowski et al. 2011; Truax and Barret 2011].

Summing up, Acoustic Ecology's ultimate goal is to restore equilibrium to malfunctioning soundscapes, which became too damaged by the changes brought by the modern world. Natural balancing forces cannot cope with factors like electroacoustic technology, and it is up to every one of us to start a change. If we accept to live in a lo-fi soundscape, one with low information to offer, sound will become something the individual tries to block, rather than hear [Acoustic Ecology 2000]. Therefore, Acoustic Design must include all elements within the soundscape, including humans and their listening habits. We are all part of a system, so, Acoustic Design is not just about changing the environment, it is about changing with it.

2.7 Conjecture on Soundscape Composition in Games

This State of the Art chapter gives an overview over the actual state of Sound Design in gaming contexts. Since the birth of videogames, game audio has evolved tremendously, although far from reaching its full potential. The industry has been struggling against the dynamic nature of game audio with linear medium techniques and tools, with Middleware tools being an effort to fight that trend. However, even with those tools, the process of foreseeing all the situations that can occur in a game under many circumstances is extremely complex. Moreover, market pressure prevents developers to have freedom to experiment different approaches. Additionally, indie development teams sometimes do not have the know-how, or the time and money to spend on very expensive audio middleware solutions.

Nonetheless, new techniques and tools have been developed and explored, including among the research community. However, most of them only focus one specific sound layer (mostly game music). The few that go beyond music, usually think of soundscape as ambiance, simply applying different styles like time of day or noise level. These approaches seldom explore sound as an information carrier, as suggested by foundation of Soundscape Theory. In addition to changing music, ambiances or other sound layers, regarding the soundscape as a meaningful part of game design holds the potential to enhance the exploration of sound in games.

As Steven Spielberg once said *"Sound and music make up more than half of communicating a story, greater even than what you're seeing"* [Dodds 2008]. Dodds also supports the importance of sound: *"Sound is a great sensory stimulus to the player's consciousness and even to the subconsciousness, affecting the mental processes without the player even noticing"* [Dodds 2008]. It is undeniable that sound can play an important role in the gameplay, attracting attention to one's action, anticipate future events, evoke emotional responses, in sum, audio can help to resonate a memorable moment.

However, audio keeps is constantly rejected as one of the most important features in a game (see Introduction and Motivation). It is something that is much harder to exhibit than graphics or an input interface between a player and a console. This leads to sound design being neglected and constantly put in the last stages of the game design process. This negative attitude against sound makes it almost impossible to create positive explorations of sound, and to allow it to have relevance in the gameplay.

There is also the problem of videogame's intrinsic dynamic nature. Sound has always been dealt with in a linear time-wise fashion, when the medium itself (videogames) is not. It is very difficult to create positive sound design explorations with a rigid approach, composed of strict temporal rules that link a specified event to its specific consequence, most of the times ignoring the soundscape's actual state. For many years, the industry has been struggling against this issue with linear medium techniques and tools, with Middleware tools being an effort to fight that trend. Currently, the best solutions available require a great investment, both economically and in terms of acquiring the necessary know-how. However, even with those tools, the process of foreseeing all the situations that can occur in a game under many circumstances is extremely complex. Rob Bridgett, Radical Entertainment’s audio director, worked on Scarface: The World is Yours [Entertainment 2006], and pointed the large amount of time spent tweaking mixer snapshots for many different game events. In total, the game had about 150 individual mixer snapshots [Bridgett]. Additionally, indie development teams, usually, do not have the know-how, or the time and money to spend on very expensive audio middleware solutions. At the same time, big development studios suffer a strong market pressure, which prevents developers to experiment new ideas and take risks. There is both an audience that is in need of alternatives to deal with Sound Design in Games, and another one that is tied to market constraints and cannot answer this need.

Therefore, an opportunity for a holistic approach to Sound Design arises from this conjecture. Alves’s work [Alves 2011] is a first step in this direction, supporting why an Acoustic Ecology mindset would improve Sound Design, and providing guidance for a positive exploration of sound in Videogames. However, the number of patterns and the way they are related difficult their direct application in a soundscape composition solution. We believe that refining some of the concepts supported by Alves, improvements in the dynamic soundscape composition field could be made, possibly giving some indications of how dynamic soundscape composition could be approached in the future.

3 Approach

In this chapter, we start by presenting the research objectives that emerged from the research detailed in the previous chapter (State of the Art), and from the problem definition that arose from the analysis made on the information gathered. Following this, we present what was the research methodology chosen to achieve the defined objectives, and explain why the chosen methodology is the best fit for the proposed objectives. Additionally, we present the initial planning for the project, with the respective milestones expected for each month. Following this, we detail the changes that were made to the planning, and list of what was in fact done during the second semester. Lastly, we report the architecture and general structure for each of the expected outputs of this dissertation: the API, the systematization of soundscape composition techniques, and the dynamic soundscape composition module.

3.1 Research Objectives

From the actual state of game audio emerges a necessity to find new ways to deal with the medium's intrinsic dynamic nature, and to approach it in a holistic way. So, this project's main goal is to suggest a possible approach to the dynamic soundscape composition challenge. We intended to develop a game engine support tool for developers to approach the problem of sound in games using Acoustic Ecology concepts, enabling experimentation of solutions for the problem of dynamic soundscape composition in games.

Firstly, we aimed at defining a soundscape specification API to be used in gaming contexts. Starting with a pre-defined number of categories (verbs) that represent common sound design exploration patterns [Alves 2011], and also using other soundscape theory concepts, we hoped to find a simple solution to which new categories can be added, in order to semantically enrich the concepts behind our specification API. Then, we wanted to propose a systematization of techniques for dynamic composition in gaming contexts. Although different games have different vibes and use different soundscape styles, we believed that different techniques could be detailed in order to be further applied. Finally, in order to be able to test the aforementioned API, and to verify the feasibility of the proposed architecture, we modeled and prototyped a DSC engine module as a proof of concept to be integrated in a game engine. Furthermore, in the future this proof of concept can be used to test the proposed techniques. This module allows rapid prototyping and experimentation in order to allow developers to more easily test their creative ideas.

It is important to refer that our objective is not to offer similar quality, durability and features of current middleware tools. The system is intended for early stage rapid prototyping, in order to empower small game developers to integrate sound design explorations in their projects.

3.1.1 Soundscape Specification API

Currently, game audio implementation requires very specific programming knowledge, both in terms of coding complexity, and the concepts behind the implementation. It is a process that usually requires designers and programmers to work close together to be able to communicate, which may not be easy due to unfamiliarity with each others' vocabulary and background. Additionally, communication between game engines and middleware tools are made through event systems. These events are defined in the authoring tool by the sound designer, being the programmer in charge of triggering them in the game code whenever necessary. The only identification that these events have is their name, having no semantic information attached to them.

What we hoped to achieve with the creation of this Soundscape Specification API was to create a more designer-friendly solution for the creation and characterization of acoustic elements in a game. The primary goal of this characterization is instrumental, namely that of providing the information that will allow the soundscape composition techniques (see Systematization of Soundscape Composition Techniques) to dynamically operate on sound sources, so that a healthy soundscape can be composed by the dynamic soundscape composition module. Another goal of this API is to work as a means for having an expression of the game's sound design intent, in a more easily understandable form. We hoped to offer capabilities to declare the designer's intentions regarding the game's soundscape, while keeping the code relatively understandable and presenting a low learning curve. These intentions were based not on strict, imperative rules, but instead in a more declarative fashion.

3.1.2 Systematization of Soundscape Composition Techniques

During the evolution of Sound Design in videogame contexts, the trend has always been to adopt techniques and tools from linear types of media like music and cinema. Specialized professionals from those fields created content that was then glued to the game by a programmer. Although Middleware tools have introduced in the last years new techniques and methods to circumvent this problem, we believe that there are still different approaches to be explored, especially by developing techniques and tools made from scratch with dynamic soundscape composition in mind. We hoped to define a list of techniques that attempt to cope with the dynamicity and potential unpredictability intrinsic to the medium, offering guidance to the dynamic composition module.

3.1.3 Dynamic Soundscape Composition Module

As it has been introduced in State of the Art, it is pertinent that a soundscape adapts to the dynamicity of the gameplay to ensure it is communicational value and that it contributes positively to the overall experience. In turn, that requires that designers are able to inscribe the dynamic behaviour of the soundscape that they design.

We exposed that middleware solutions are those that better fulfil such goals, but they still present some major obstacles for a broader adoption by the community of practice. One such obstacle is the typically high price, which renders them unviable for small developers. Additionally, the sophistication of such tools also brings complexity that requires a learning curve that may not justify their introduction in small projects.

Additionally, small teams, in indie development scenarios, usually do not have access to a sound design expert. The vast independent game developer community remains challenged by small budgets and lack of know-how while trying to integrate sound in their games.

Being so, we argue that it is pertinent to contribute with conditions that may augment the prospect that the average developer can take advantage of the exploration of sound in game design. The availability of tools that ease upright integration of sound in games could make it interesting for developers who are not experts in sound design to venture into the practice, either by themselves or while working together with designers. Even if such appropriation solely fosters a greater awareness for the potential of sound in game design, we believe it is fairly arguable that it would be a valuable contribution. Finally, to add to such pertinence, it is worth noticing that small teams actually constitute the majority of game developers. In that sense, contributing for the empowerment of such a massive force of creativity could benefit both the industry, and the gamers.

Therefore, we hoped to design a solution for supporting the dynamic enhancement of a game soundscape, while addressing the goals and issues stressed in the previous paragraphs, through an holistic approach such as the one rooted in Acoustic Ecology. Using the knowledge created by the previous objectives, we intended to design and prototype a run-time dynamic composition module for a game engine. This module serves as a proof of concept of the proposed system's architecture, and we hope that, in the future, it will allow us to test, evaluate, and improve all the techniques defined in our systematization. A game scenario was used to verify how the engine composes the soundscape autonomously in a dynamic fashion, as well as to test the integration of the proposed API with game code. This module allows rapid prototyping and experimentation in order to allow developers to more easily test their creative ideas.

3.2 Methodology

Design Science Research (DSR) is a research methodology characterized by iterative design and formative research. The main difference, when compared to more conventional educational research methodologies, is that it changes the role of design in the whole process. In DSR, design is important not only on the evaluation of theories, but also on their development.

Instead of having a group of theories and principles that are followed blindly during the design process, DSR encourages iterative cycles of problem definition, design, implementation, and evaluation that originate data to be used in the following design iteration (see Figure 3.1 - DSR's iteration steps).

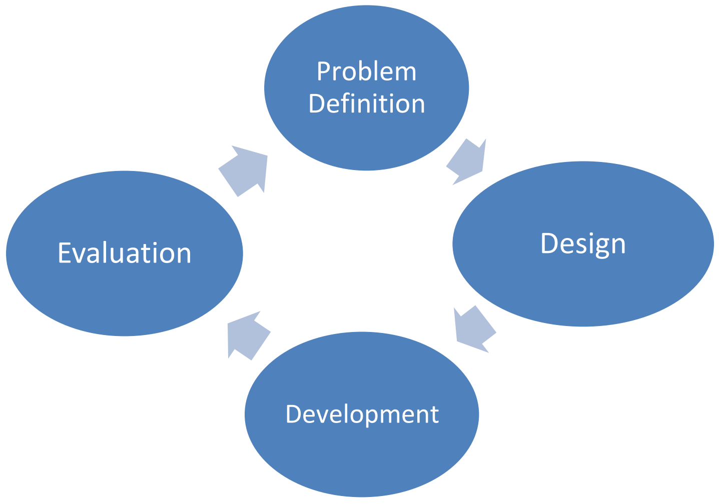


Figure 3.1 - DSR's iteration steps

This approach allows a constant refinement of theories, design process and its outcomes, eliminating the boundaries between design and research. DSR's complete framework can be seen in Figure 3.2.

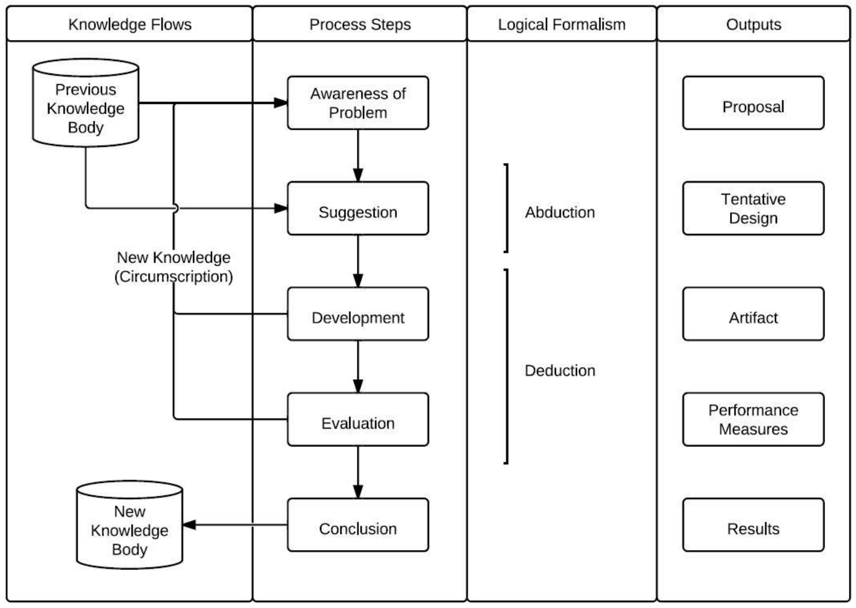


Figure 3.2 - DSR's Framework [Hevner 2004]

This methodology fits this project because it is oriented for the production of knowledge. While other methodologies were created with software production in mind, in DSR, software creation is simply part of the knowledge production process. Being the goals of the project not only the development of a prototype, but also the definition of a specification API and the systematization of soundscape composition techniques, DSR provides a methodology that fits perfectly the proposed objectives. In order to fully understand the adequacy of the DSR methodology to the project to be developed, Table 3.1 can be consulted.

Step	Description	Dissertation
Awareness of problem	It is where the problem is defined and the value of a solution is supported. The output for this step is the State of the Art Report, which helps to define the problem.	In our project, this step corresponds to the first two tasks in the Gantt chart: State of the Art Report and Definition of Problem and Methodology. However, it is important to refer that the problem definition could be refined with data collected from any iteration.

Suggestion	The previous step’s output is used to define the objectives and requirements for a solution. In this step, a first approach to a solution is carried out, from which can be originated interaction models and architecture proposals that complement the tentative design. This solution is achieved through abduction, supplementing what is not known through intuition. It is clearly declared what is known, and what it is not. So, as in abductive reasoning, the premises (tentative design) do not guarantee the conclusion (valid solution).	In our project, this step corresponds to the tasks named Initial Design and Prototyping. During the prototyping task, each iteration creates new data that will be used to refine the objectives and requirements, the specification API, the DSC techniques and the DSC solution applied to the module.
Development	This step consists in the implementation of the proposed solution, by resorting to Software Engineering processes, embodied in the DSR approach. The output from this step is a software artifact. If the artifact was successfully built, the first part of the proof of concept was completed.	In our project, this step corresponds to the task named Prototyping. The design/implementation/evaluation cycles have the duration of one month.
Evaluation	After the development step, the artifact is tested and its results are compared to the objectives proposed in Suggestion. It usually requires the usage of metrics and analysis techniques. If there is no model of evaluation to the specific artifact or the specific objectives that are being designed, the evaluation methods have to be created in the Suggestion step. If the results are successful, the second part of the proof of concept is completed. The knowledge originated from this step is then used to refine the previous steps and is used in the next iteration cycle.	In our project, this step corresponds to the task named Evaluation. It is important to refer that after each iteration, this evaluation occurs but in a more informal way, being only during the Evaluation task that a formal evaluation is carried out. This is due to the project’s tight time constraints.
Conclusion	This step consists in the presentation of the knowledge originated from the whole process (Statement of Learning). The output can vary from concepts or models, to methods or prototypes	In our project, this step corresponds to the task named Statement of Learning, where the outputs of the projects are going to be presented.

Table 3.1 - DSR's steps detailed

In this subsection we explained what research methodology was adopted and the reasons behind that choice. We truly believed that the principles that support the Design Science Research methodology are the ones that cope better with the dissertation’s research objectives.

3.3 Planning

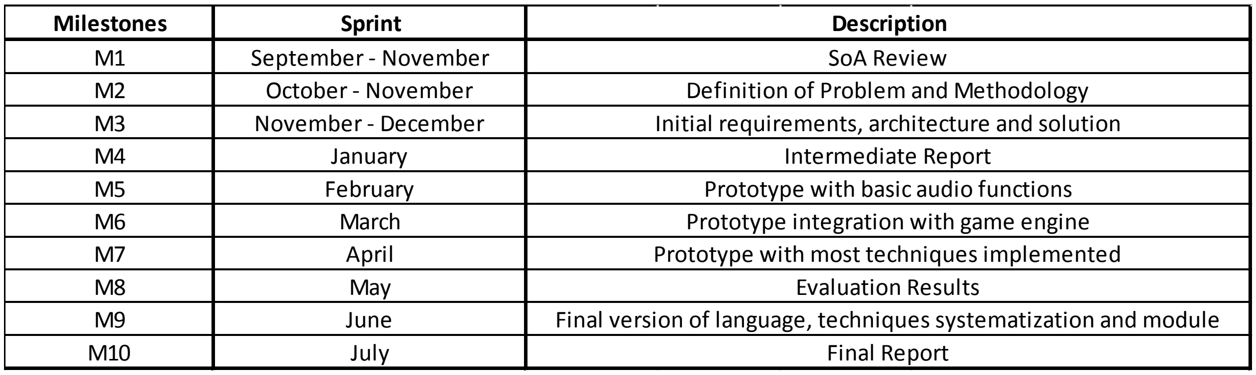
In this section, is described the planning proposed for this dissertation, covering both the first and the second semester. The planning of the second semester is going to remain similar to the one presented in the intermediate report, in order to allow comparison with what was indeed executed during the second semester (presented in Execution). In order to better understand it, it is recommended to consult the Gantt chart presented in Appendix B. The project’s tasks defined in detail in the aforementioned chart. It is important to refer that although this dissertation does not follow a scrum methodology, milestones were projected as a result of one month sprints. These monthly sprints embody an iteration of DSR’s iteration steps (see Figure 3.1 - DSR's iteration steps). However, due to the nature of the work that was produced, the first two months of the first semester did not follow this monthly approach.

The first semester started with the research and documenting of the State of the Art. This was one of the longest tasks (it took approximately three months), mainly because it is the foundation for the work to be produced further on the dissertation. This task was divided in four research components: Soundscape theory, Audio support in game engines, Audio tools, and Soundscape composition. After one month of State of the Art research, the details of our project started to be defined. The knowledge that it was being gathered in the previous task was helping to comprehend and define what the problem that should be addressed was, as well as some assumptions and milestones were starting to be defined. Following the aforementioned specification process, an initial design attempt was put on course. At the same time the goals and requirements were being defined, we started to study different iterations of a possible architecture for the sound engine. The last subtask of the initial design attempt consisted in defining a first specification for a possible solution. This solution consisted in a list of steps the engine will need to do, similarly to defining the different parts of a complex algorithm. The last month of the first semester consisted in transposing all the knowledge obtained in the previous months to the Intermediate Report.

The second semester’s main task is the implementation of the prototype. It is important to refer that this process includes some theoretical tasks, like the continuous definition and refinement of the concepts behind our specification API and the systematization of techniques. After a brief period of time to test further the tools that will be chosen for the implementation process, the prototyping process will start by the implementation of basic audio engine functions (like play, stop, etc.). After that, these basic functions will be integrated with the game engine that will serve as test-subject for this dissertation's work. This will serve as skeleton for the next step in the prototyping process: the implementation of soundscape composition techniques. Lastly, the prototype will suffer some refinements that will be a direct consequence of an evaluation process that will be developed previously. The aforementioned evaluation task will be divided in two parts: evaluation performance, and analysis of the results. Finally, the writing of the final statement of learning will be done, which encompasses the writing, review, and finishing of the dissertation final report.

3.3.1 Milestones

Although this dissertation does not follow a scrum methodology, milestones were projected as a result of one month sprints. However, due to the nature of the work that was produced, the first two months of the first semester did not have any milestone defined. Obviously, being this dissertation a Design Research work, there can be adjustments in both the milestones' dates and their expected output. The list of milestones can be consulted in Figure 3.3 - Milestones Table, and will be explained in the following subsections.



Milestones	Sprint	Description
M1	September - November	SoA Review
M2	October - November	Definition of Problem and Methodology
M3	November - December	Initial requirements, architecture and solution
M4	January	Intermediate Report
M5	February	Prototype with basic audio functions
M6	March	Prototype integration with game engine
M7	April	Prototype with most techniques implemented
M8	May	Evaluation Results
M9	June	Final version of language, techniques systematization and module
M10	July	Final Report

Figure 3.3 - Milestones Table

Milestone 1 – State of the Art Review

In this milestone, the objective was to have all the research about the topics that would compose the State of the Art, and to complete a draft of it. Although some information was still being added to the chapter after this date (even during the second semester), this milestone was successfully accomplished.

Milestone 2 – Definition of Problem and Methodology

The expected result from this milestone was a first specification of the problem to be solved, and a first thought about what milestones would best fit the project at hand.

Milestone 3 – Initial requirements, architecture and solution

Following the specification process performed in the previous milestone, this milestone consisted in transforming data into a more formal output result. In other words, the goal was to define a list of possible requirements for the project, design the diagram of the proposed architecture, and to define in a more formal way, the first solution for the problem that was projected.

Milestone 4 – Intermediate Report

The last milestone of the first semester consisted in writing the rest of the intermediate report.

Milestone 5 – Prototype with basic audio functions

At the end of the second semester's first month, it is expected to have an audio engine prototype that performs basic audio functions. At this point, decisions about software must be made and this milestone serves to familiarize with the chosen tools, as well as to create the basic functions that will serve as support for the more high level concepts, and goals, of the following milestones.

Milestone 6 – Prototype integrated with example game scenario

The expected output of this milestone is to have full integration between our basic prototype, and the chosen game engine that will be used. From this point on, new layers of complexity can be added to the sound engine, which will only require small changes in the game code, being the communication channels between the two engines completely built.

Milestone 7 – Prototype with most techniques implemented

At this point, most techniques should be defined and implemented. This will allow the next milestone (Evaluation results) to be achieved without delays. Moreover, the number of techniques not implemented at this stage, will surely influence the rest of the project. Due to this, it is an extremely important milestone.

Milestone 8 – Evaluation results

In this milestone we should be finishing the evaluation process. This will give us some answers about the approach taken and will allow us to take conclusions that we hope will contribute to the soundscape composition field of study. Additionally, it will allow refinements on the artifacts that will improve the quality of the dynamic soundscape composition module.

Milestone 9 – Final version of API, techniques' systematization and DSC module

This milestone marks the ending of the prototyping phase. All the artifacts that are expected to result from this project must be completed. This also includes documentation and other extra tasks needed to complete the artifacts.

Milestone 10 – Final report

The last milestone of the project is the delivery of the statement of learning. In other words, consists in writing, reviewing and completing the final report.

3.3.2 Execution

When the second semester began, we knew there were important decisions that had to be made in order for the prototyping phase to start. Architectural and technological issues were still preventing us to start. Also, these issues could have impact on the solution that it was being designed. Therefore, we preferred to think wisely, even if that would create a little delay on the project. Due to this decision, the prototyping phase only started in March, instead of the projected on the intermediate report (February). Furthermore, the deliverables of each milestone were changed. The new milestones that were defined were:

- **Sprint #1 - March –** Modified version of the game Blindfold; Alpha version of the DSCM, only with communicational modules prototyped.
- **Sprint #2 - April –** Final specification and prototyping of the API; Beta version of the DSCM, with the audio renderer fully implemented, and remaining modules' skeleton prototyped; One heuristic implemented (Context heuristic); Two scientific papers for the Audio Mostly 2013 conference.
- **Sprint #3 - May –** A version of the DSCM with all the heuristics implemented.
- **Sprint #4 - June –** Final version of the DSCM with refined heuristics,testing procedures, analysis of its results, and the writing of the final report.

All these milestones were achieved, and their output is going to be detailed in the following chapters.

3.4 Dynamic Soundscape Composition Solution Architecture

In the following sub-chapters we will detail the architectural and design choices made regarding each of the outputs expected for this dissertation: The soundscape composition API, the systematization of soundscape composition techniques, and the dynamic soundscape composition module.

3.4.1 API Design

As the main goal of this project is to propose an approach for Dynamic Soundscape Composition in videogame contexts through a holistic perspective to sound, the only interface between the target audience and the module is the API and the concepts that support it. Similarly to any other kind of design (i.e. user interfaces), this API and the theoretical concepts behind it followed some pre-defined guidelines. What we hoped to achieve was an accessible, easily understandable, but and the same time resourceful API, that would allow designers to experiment and to enrich their games with rich sound explorations and healthy soundscapes. Similarly to the DSC module, this API was programmed in C#.

One of the main goals of the API is to allow designers to create and characterize acoustic elements in the game. The primary goal of this characterization is instrumental, namely that of providing the information that will allow the heuristics to dynamically operate on those sounds, so that a healthy soundscape can be composed.

Another goal of the use of this API is that it can also work as a means for having an *expression* of the game's sound design, in a form that attempts to be easily understandable. We tried to enhance legibility through a judicious naming of classes and methods, and choice of parameters and expected values. This should help designers to keep a good perception of their decisions while working on a design, with advantages also for the maintenance of that design. But it should also serve as a format to communicate that design to other people, whether it is to discuss ideas within the development team, or to share designs among projects.

Equivalently, another objective for this API is that it presents a low learning curve. This is consistent with our primary motivations for investing on this research proposal, which in turn resulted from the perception that the available solutions that can be used to support the design of healthy soundscapes, particularly middleware tools, are typically characterized by high learning curves. The ease of use of the API is also important to support prototyping and test design ideas.

The ruling guideline behind the design of this API was to think of it not as a language in itself, but as a means to give expression to Alves's pattern language, as well as to other principles found on Acoustic Ecology theory, which served as inspiration for this holistic approach.

This holistic perspective over game audio is a shift in the way sound implementation is foreseen. A parallel can be drawn between some programming paradigms and the proposed approach. What is being proposed is a change from an imperative mentality (imperative programming), to a more declarative mentality (declarative programming). Rather than defining strict orders like "Event A, play sound B", the sound designer should only define sources and contexts which contain high-level directives that will serve as input to the soundscape composition module.

Theoretical Concepts

In order to enable the translation of some Acoustic Ecology principles to an API to be used in game code, we had to define key theoretical concepts that designers should be aware of in order to project soundscapes in a more easily understandable fashion.

- ***Source*** – It is the concept which represents a sound source, and it is the key concept behind the API. Every sound of the soundscape should have a source representing it. Every source has a number of properties that enables designers to shape them as they intend to. It is the information stored in those properties that enables the composition module to reason about how to compose the game's soundscape.
- ***Layer*** – The concept of layer is a categorization of sounds according to their semantics, as referred by Peck in [Peck 2001]. The five different layers defined by Peck are: Ambiance, Dialogue, Music, Foley and Sound Effects. This is one of the properties that will serve to personalize sound sources. Additionally, many of the actions that the sound module will apply, will be done to specific layers. Therefore, the layer chosen by the designer to identify the source will have impact on how it will be treated by the composition module.

- ***Agent*** **–** Besides the concept of layer, we wanted to be able to associate each source to another type of identification. During game design process, the concept of character is one of the key elements. Therefore, we decided that could be useful and intuitive for designers to be able to associate a source to a specific agent in the game. However, it is important to refer that the term "Agent" is just an abstraction that was defined. In other words, it is not mandatory to use a game character in this property, as designers are free to associate sources with whatever term they prefer. To sum up, this is a free tagging system, which can be used by designers, though we labeled it Agent because it is a type of utilization that we think it can be both simple and useful.
- ***Pattern*** **–** The concept of pattern is informed by Alves's work on his Sound Design Pattern Language [Alves 2011] [Alves and Roque 2011]. Although in his work, the large number of patterns translates into different categories (i.e., sound explorations, sound layers, guidelines, etc.), in this work, patterns should be understood as sound behaviors that will be taken into account by the DSC module while maintaining the soundscape healthy. It is important to refer that while some patterns have to be associated with a source, there can be stand alone patterns which affect the whole soundscape. The list and explanation of all the developed patterns can be consulted in Soundscape Composition Techniques.
- ***Listener*** **-** A listener represents the "microphone" inside the game world. In other words, it represents what is heard by the player. There are two properties that influence what the playear hears: the listener's position and direction. These values are needed for the sound engine to make the calculation needed to recreate 3D sound behaviors.
- ***Context*** **–** It is the second most important concept of the API. Most of games nowadays have a large number of elements operating simultaneously, which means that, at a given moment, there can be a large number of sound-producing actions occurring. However, only some of them are relevant for the player. What we pretend to offer with this notion of context is to clearly differentiate between relevant sounds (in context), and sounds that are not relevant (out of context).

- ***Exclusivity*** – This is a concept that is attached to context. If a context is exclusive, it means that sounds that are out of context are not going to be heard at all, while if the context is not exclusive, they are going to be attenuated, but not totally muted.

Functionalities

In a summarized manner, the functionalities of the purposed API include: creation, deletion and management of sound sources; creation, deletion and management of contexts; control over some properties of the listener, such as position and direction; and, the creation and management of stand alone patterns. In each of these cases, details on entities are provided when they are created. Furthermore, in order to uniformize the usage behind all the features offered by the API, we tried to make most functionalities follow the same simple steps: creation; initialization; and play/stop requesting. Details about each of the functionalities offered by the different classes are explained in the following subsections.

Source

- **Creation of a source** – It instantiates a source object to allow the programmer to use the functionalities it offers.
  - *new Source(name, layer, agent, position, pattern, sound, loop);*

- **Initiation of a source** - It requests the DSC module to run all the low-level and internal procedures necessary for this source to be at the module's disposal.
  - *InitiateSource();*

- **Request to play a source –** It requests the DSC module to play this source.
  - *PlaySource();*

- **Request to stop a source** – It requests the DSC module to stop this source. It offers the an option regarding whether the source should be paused, or stopped.
  - *StopSource(pause);*

- **Change source's position –** It requests the DSC module to change the position of this source.
    - changeSourcePosition(position);

- **Change source's sound file -** It requests the DSC module to change the sound file associated with this source.
    - changeSourceSound(soundFile);

- **Change source's looping option -** It requests the DSC module to change whether this source should loop or not.
    - changeSourceLoop(loop);

Context

- **Creation of a context** - It instantiates a context object to allow the programmer to use the functionalities it offers.
    - *new Context(name, type, elements, exclusivity);*

- **Initiation of a context** - It requests the DSC module to run all the low-level and internal procedures necessary for this context to be at the module's disposal.
    - *InitiateContext();*

- **Request to activate a context –** It requests the DSC module to activate this context.
    - *SetContext();*

- **Request to deactivate a context –** It requests the DSC module to deactivate this context.
    - *StopContext();*

Listener

- **Creation of the listener** - It instantiates a listener object to allow the programmer to use the functionalities it offers.
    - *new Listener(position, direction);*

- **Change listener’s position** - It requests the DSC module to change the position of the Run-Time Player’s listener (see Major Functional Units).
    - *ChangeListenerPosition(position);*

- **Change listener’s direction** - It requests the DSC module to change the direction of the Run-Time Player’s listener (see Dynamic View).
    - *ChangeListenerDirection(direction);*

Pattern

- **Creation of a pattern** - It instantiates a pattern object to allow the programmer to use the functionalities it offers.
    - *new Pattern(name, type);*

- **Initiation of a pattern** - It requests the DSC module to run all the low-level and internal procedures necessary for this pattern to be at the module’s disposal.
    - *InitiatePattern();*

- **Request to activate a pattern** - It requests the DSC module to activate this pattern.
    - *PlayPattern();*

- **Request to deactivate a pattern** - It requests the DSC module to deactivate this pattern.
    - *StopPattern();*

A formal and complete version of the API’s specification can be consulted in Appendix F.

Guidelines

There are some guidelines which should be considered before using the API. The code is intended to be clean, and easily understandable. Still, it is up to the programmer to choose in which part of the project he wants to introduce the code. It is recommended to have an initialize function (common in most games), which serves as "headquarters" for all the elements regarding sound used in the game. This way, programmers can look at this initialization area, and, by analyzing the declarations, can have an idea of what type of sources, contexts, and patterns are being used, and why. It is important to refer that programmers should always initialize their assets with their corresponding initialize methods, in order for the engine to create and prepare them to be at the programmer's disposal.

Also, programmers are free to trigger those elements from anywhere, though commonly the triggering events belong to the game's logic. The API encourages programmers to reuse previously defined assets. In other words, as the idea is to simplify sound implementation, programmers are encouraged to use features as the changeSourceSound() method, which changes the sound file associated, while keeping all the other properties associated with that source. Features likes this prevent the obligation of keep creating sources for any new sound that is needed in the soundscape. The main idea is to rationalize assets in order to keep the code clean, simple, and to increase code performance.

In order to allow a smooth sound implementation in their games, programmers should take in account some particularities regarding some of the functionalities offered by the API. For instance, it is mandatory for programmers to define the listener before activating sources, contexts or patterns. Otherwise, calculations regarding positional sounds will fail, as the listener's information is missing. This information is essential in order to calculate the panoramic and attenuation values that sounds should receive before being delivered to players' ears.

Another particularity refers to the method changeSourceSound(). Due to its nature, needs to stop the former sound file associated with the source, and to associate the new one. However, the new sound file does not start to play automatically. It is required for the programmer to call (again) PlaySource() method for that to happen.

Similarly, programmers should be aware that, as the engine only supports one active context, whenever they call the method setContext(), it automatically deactivates the previous context, and activates the one which called the method. This design decision was made so that programmers can achieve their intentions with less lines of code (in a big project, which requires contexts to be activated frequently, imagine the number of times that programmers would need to call deactivate context in order to activate a new one).

It is important to clarify one aspect regarding the class Pattern. As referred in Theoretical Concepts, there are two types of patterns: those which have necessarily to be associated with a source, and those which can be activated on their own. The class Pattern only refers to the latter. For those patterns that need to be associated with a source, they are only referred to in the constructor of a source, and are selected as a string field. This design decision was supported by the objective of trying to achieve the functionalities with the minimum number of lines of code possible. We thought about giving the Pattern class another name, but it would be inconsistent with the knowledge that supported our approach, as we would be calling another name to some behaviors that, in theory, we always refer to them as sound patterns.

3.4.2 Soundscape Composition Techniques

As referred in Research Objectives, we hoped to define a list of techniques that attempt to cope with the dynamicity and potential unpredictability intrinsic to the medium. These techniques consist in a list of common practices in game audio that we believe to be useful in order to achieve a healthy soundscape (see Acoustic Design). The list of techniques is expected to offer guidance to the dynamic composition module, and is expected to be reusable and further updated. Their usage will be available to programmers in the form of patterns to be associated with sources. The main challenge in this task resides in being able to translate the relevant knowledge on Acoustic Ecology and videogame sound design, into algorithmic heuristics. After a great amount of research on the aforementioned issues, we defined a list of heuristics that would implement these techniques onto the DSCM, in order to maintain the soundscape healthy.

Heuristics

In order to build the theoretical concepts that support the heuristics, we resorted to Soundscape Theory (see Acoustic Ecology), and Alves's pattern language [Alves 2011], to inform its definition, because it provides us with contexts of use of sound, and consequently can be instrumental in the characterization of events. The proposed module of heuristics hopes to cope better with the dynamicity and potential unpredictability emerging from the gameplay, and the consequent superimposition of sounds being emitted. Semantically, these heuristics monitor sounds that the game logic determines that *would* be playing (*active sounds*), and decides whether, and how, they should be played (according to, besides other elements, the sources' patterns). The heuristics may modify the acoustic parameters of the sounds that they send to the run-time player (e.g., volume and filters). It is also conceivable that the heuristics modify the timing of sounds that they send to the run-time player (e.g., postponing, sequencing, synchronizing). It is important to refer that all these heuristics are run whenever a source with its correspondent pattern is ready to be played by the DSCM, with the exception of Context, Silence, and Murch's Encoded-Embodied, which are not used in association with a source.

This approach resembles what happens in a natural environment, where (some) sounds exist whether or not the listener gets to hear them. In the following subsections we will enumerate all the heuristics implemented, and detail their impact on the soundscape being composed.

Context

As referred in Theoretical Concepts, the concept of context was used to clearly distinguish the sounds that are relevant for the gameplay at each moment. As this is one of the major struggles game audio faces nowadays, this heuristic was an attempt to help keeping audio semantically valid in relation with the gameplay. So, this particular heuristic allowed us to experiment the application of *contexts* to the balancing of the soundscape.

Specifically, this heuristic consists of a solution given to sound sources in accordance with them being or not part of the soundscape's current context. It allows the interpretation of a gameplay context, in a way that, at a given moment, the sounds belonging to that context deserve a different treatment – typically, more emphasis – over other active sounds. A very simple instance of this heuristic is to allow a single context to be active at any moment, and to attenuate the volume of every sound not belonging to that context to, e.g., 50%. Still, other more complex instances could be coded into the heuristics container.

As to contexts, we defined three categories so far, which can be created through calls to the API:

- **Agent contexts**. When designers create agent contexts, they enumerate the agent entities that compose it. In turn, agents may be associated to sources when these sources are created. Examples of agents may be game characters, objects, places, or any other entity that may be convenient.

- **Semantic layer contexts**. When designers create semantic layer contexts, they enumerate the sound layers that composite it. We have been adopting the following layers for categorizing sounds: *Dialogue*, *Foley*, *Sound Effects*, *Ambiance*, and *Music* [Peck 2001]. When a sound source is created, the sound layer it belongs to has to be set.
- **Ad-hoc contexts**. This is the most versatile category of context. When designers create ad-hoc contexts, they enumerate sound sources that compose it, by their own name. Being so, this type of context is also potentially interesting as a tool to sketch and test other categories that might become included in the set.

In the three categories of contexts, the designer can define which sounds should be in context simply by listing the respective selectors, i.e.: a list of agents, a list of layers, or a list of source names, respectively. Consequently, only the sounds matching the criterion are considered to be in context.

In Figure 3.4, we represent examples of the effect of the heuristic on active sounds. Each of the three parts of the figure refers to the case of each of the categories of contexts, defined above. The circles represent sounds that, according to the game logic, should be playing in the depicted moment, and their color represents the agent with which they are associated. The icon next to the circles represents the actual rendered volume level of each source. The circles marked red are the ones that belong to the current context, which, in those examples, would be heard at full volume, while the others would be attenuated. In the first example, we illustrate an agent context, in this case selecting sources solely associated to the agent represented by blue circles. In the second example, we illustrate a semantic layer context, in this case selecting sources solely associated to dialogue. In the third example, we illustrate an ad-hoc layer context, in this case selecting sources explicitly chosen by the designer (by their name, not represented in the figure).

Thoughts

Thoughts are widely used nowadays in videogames, and their objective is to reveal what a character is thinking of. They allow game designers to express messages in a diegetic way (explained in Dynamic Nature of Game Audio), as well as to obtain emotional explorations through sound (i.e., enhance empathy between the player and a character). Additionally, the

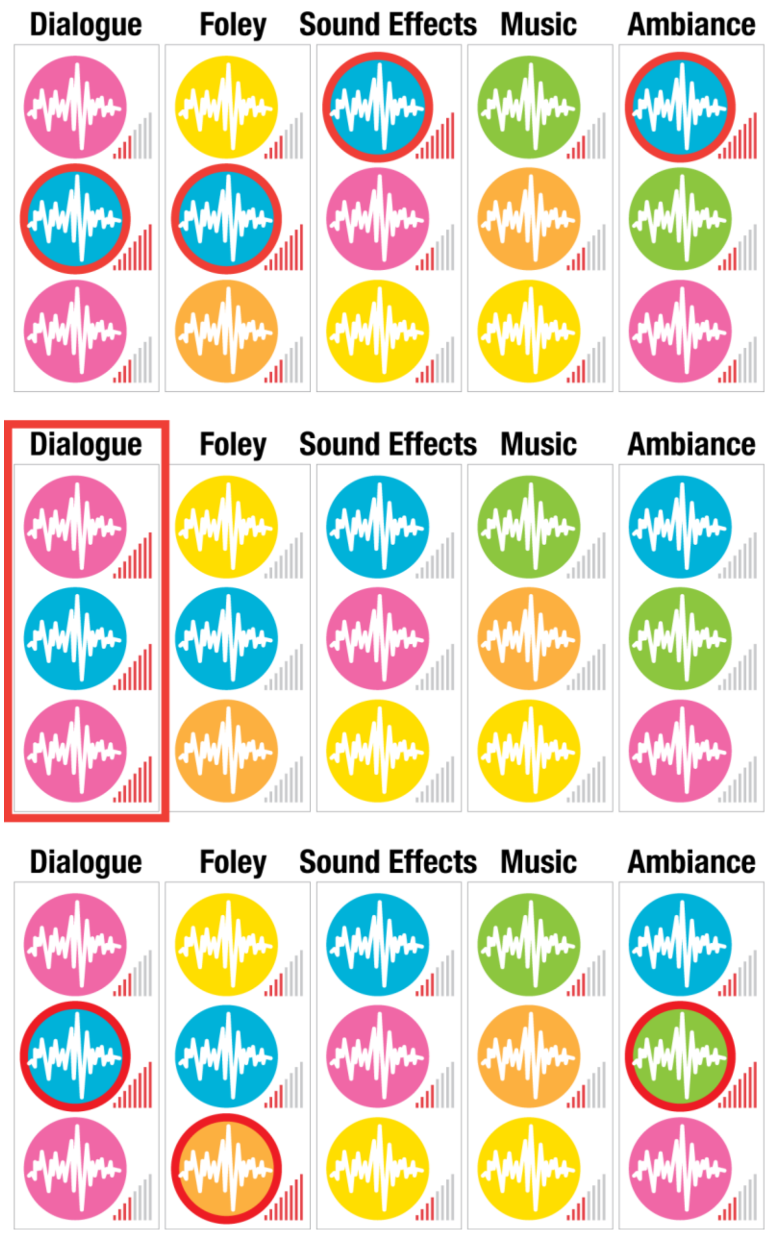


Figure 3.4 - Examples of the effect of the context heuristic

associated introspection contributes to inspire and maintain a sense of immersion in the game experience [Alves 2011]. We wanted to offer the designers a way to represent thoughts as a verbalization inside the head of the character. In order to achieve it, this heuristic attenuates the current volume of every source by 90%, keeping at full volume only the source to be played (which is associated with the pattern "Thoughts"). In addition, an acoustic effect (Echo Filter) is applied to this source in order to achieve the aforementioned "inside the head" feel.

Silence

Silence is one of the most powerful tools to be used in sound design, though very hard to dominate. Its use is usually associated with emotional explorations through sound, especially negative emotions or representations of peaceful moments. However, silence can be achieved in many ways, not necessarily implying absence of sound. Therefore, this heuristic needs to be considered simply as one approach to silence implementation on sound design. Its effects consist in an attenuation of the current volume of every source from the Ambiance, Music, and SFX layers by 90%. The justification behind this design decision is due to the importance that Dialogue has in any game situation, and because when Foley does not have sonic feedback, it usually breaks the player's immersion in the game experience.

Awareness

In most games, designers resort to sound in order to aid the signaling some relevant aspect of gameplay. Recurrently, there is the necessity to expose some gameplay-related aspects of a situation which demand special attention. There are many situations in which these sounds of awareness can be useful, either to evidence a problem, inform about a state, or emphasize an opportunity or to reinforce the outcome of an action. The ultimate objective is to effectively produce changes in the player, being instrumental in influencing the player's behavior. Due to the temporary effect usually associated with this type of sound exploration, this heuristic attenuates the current volume of every source by 90%, while keeping at full volume only the source to be played (which is associated with the pattern "Awareness"). However, unlike the heuristic Thoughts, this effect only lasts for a predefined number of seconds (7). This value can be easily modified in order to meet the designers' needs.

Dialogue

In videogame contexts, Dialogue can consist on any type of discourse presented throughout a game, being used for many different goals, as to communicate aspects related with both gameplay and story. Its importance lays on the humanization it transmits to the characters, enhancing the emotional connection between them and players. Due to its importance, we knew it was imperative to have an heuristic that could always guarantee that Dialogue would have major importance during the soundscape composition process. Therefore, this heuristic attenuates completely Foley and SFX sources, it attenuates the volume of every Ambiance source by 90%, the volume of every Music source by 80%, and, finally, it attenuates the volume of every Dialogue source by 70%. As in the aforementioned heuristics, the source to be played keeps its full volume. The differences in the attenuation values are justified by the different importance of each layer in a dialogue situation. While Foley and SFx are usually not important in these situations, usually the music and the ambiance of the scene are not completely muted. Moreover, while we do not desire the other dialogue sources to difficult the perception of the source to be played, it is still desirable to allow the other characters to signal their presence through their dialogue.

Footsteps

Footsteps are among the most used patterns in sound design [Alves 2011]. They are a type of Foley that is essential to give personality and uniqueness to characters, while providing awareness in many gameplay situations. Footsteps are often is exaggerated, when compared to a real life situation, because, when not present, they can break the player's immersion in the game experience. It is one of the most important forms of sonic feedback that players should receive as output to their actions. Due to movement usually being the most basic capacity of any character controlled by players, they usually react negatively to the absence of feedback regarding that same action (movement). As a result, this heuristic verifies if sources with the pattern Footsteps associated with them have, currently, a volume level below 50% of their full volume (due to other patterns acting on the soundscape). If that is verified, the volume of the source to be played is kept at 50% of its full volume. Otherwise, it plays with the volume currently defined for the Foley layer. This heuristic tries to prevent the aforementioned absence of sonic feedback regarding player movement.

Contextual Music

In videogames, Music has been repeatedly used to characterize specific contexts, being either levels, regions in the game's world, specific types of enemies, etc. Therefore, contextual music allows the improvement over the older concept global music for each level, or world. Each piece of music should contribute to each particular moment along the experience, exponentiating the fit between the music being played, and the situation being experienced by the player. It may also contribute to the variety of the soundscape, allowing players to "take a break" from a "global" music piece being played. For these reasons, this heuristic verifies if the current level for the Music layer is below 50% of their full volume. If that is verified, the source to be played (which is associated with the contextual music pattern), is allowed to play at 50% of its full volume. Otherwise, it plays with the volume currently defined for the Music layer.

Achivement, Failure, No Can Do

Achievement, Failure and No Can Do are special types of SFX which we found important to highlight. Since the beginning of videogames, these were the most used SFX. Achivement is used to signal all positive happenings during gameplay, whether are cause by item collecting, checkpoint reaching, or, more recently, trophies that reward players for a panoply of different reasons. Similarly to Awareness, they represent something relevant, having the additional importance of making the player feel important, due to its performance. On the other hand, Failure has the same degree of importance, though transmitting the opposite semantics, used usually to signal players' bad decisions or poor performance. No Can Do was a semantic messaged created to inform players that the action they are trying to perfom is not possible due to some reason, though the message is not semantically as strong as Failure. In Alves's words, "*This type of sound is mostly informative; it does not reflect a judgement on the action of the player – although it informs about something that cannot be done and, as such, that is not interesting repeating*". Due to the importance of this type of sounds, this heuristic objective is simply to assure that, independently of the soundscape's current state, context, and patterns acting over it, the source to be played (which is associated with one of these patterns) will be played at full volume.

Murch's Encoded-Embodied

As referred in Acoustic Ecology, balance is one of the pillars that support a healthy soundscape. A soundscape absent of sound, is as useless as a soundscape overcrowded with sounds, in which this abundance prevents sound to carry information. So, it was clear for us that layer density was a very important issue with which we had to deal. After researching this matter for previously defined models regarding sound density, there was one work which were enlightening: Walter Much's Encoded-Embodied spectrum theory [Murch 2005]. Supported by the idea that the left and right hemispheres of the brain are used to process different sounds, he argues that, if you devise your mix according to this 'Encoded – Embodied' spectrum (illustrated in Figure 3.5), you can accommodate far more audio content than if you were mixing to the rule of “two point five rule”, itself pioneered by Murch (which defends that there should only be two main sounds and a small element of something else at any one point in time in a film) [White].

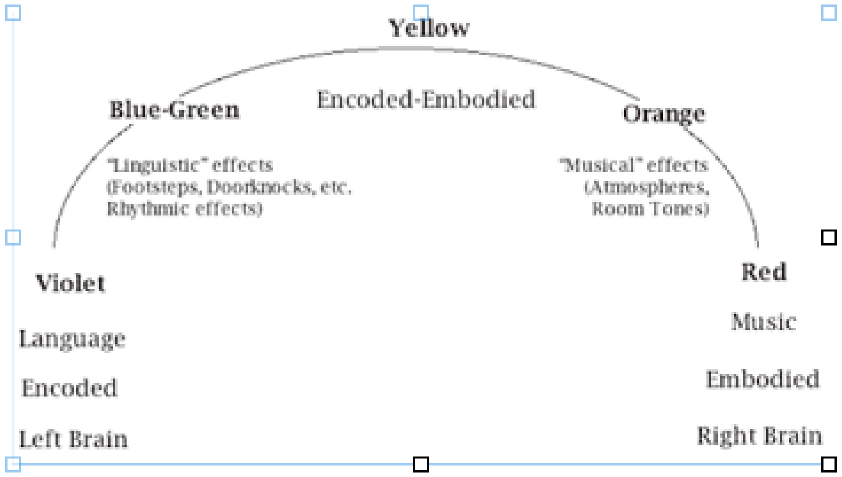


Figure 3.5 - Walter Murch's Encoded-Embodied spectrum theory

After analyzing Murch’s theory, we made a simplification from the approach proposed by Munch, and correlated each of the colors defined by him, with the sound layers we use in the DSCM. So, we decided to associate Violet with Dialogue, Blue-Green with Foley, Yellow with SFX, Orange with Ambiance, and, finally, Red with Music. Munch defends that, at most, two sounds by “color” can be supported simultaneously. Therefore, this heuristic implements this principle, preventing sounds that would violate this rule from playing. However, it should be noticed that, in order to give freedom to designers, this heuristic can be easily switched on/off. Moreover, although violating the rule, we allow designers to change the maximum number of sources supported by the “colors”.

3.4.3 Dynamic Soundscape Composition Module

The DSCM represents the culmination of all the research performed in all the previous stages of the project. All the theoretical knowledge gathered in the State of the Art, from different fields of study, to the iterative approaches to the problem definition and its proposed solutions, will be put to use in this artifact. This module serves as a proof of concept of the proposed system’s architecture, and it allows us to test what differences can the proposed API bring in terms of sound implementation. Additionally, we hope that, in the future, it will allow us to test, evaluate, and improve all the techniques defined in our systematization.

In this chapter, we expose a proposal for supporting the dynamic enhancement of a game soundscape. The proposal consists of a system that moderates sounds being dispatched to the sound engine, with basis on a characterization of the participant sound sources, and on a heuristics module. The characterization of the sources is done by means of an API that we developed. The heuristics translate holistic concepts such as those rooted in Acoustic Ecology.

Approach

This proposal constitutes a lightweight and reusable approach for composing a healthy soundscape, by dynamically regulating the sounds presented to the player, hence avoiding the hurdle of covering each conceivable gameplay state, in the game code. For us to be able to address sound in different states, we felt the need to introduce the concept of *active sounds*. We define *active sounds* as the sounds that according to the game logic would exist, in a particular moment; i.e., sounds that were triggered and did not yet finished. An active sound becomes heard if the heuristics dictate that it should be sent to the Run-Time Player. It can even happen that an active sound eventually finishes without ever being heard.

Our proposal may appear to be inefficient because it seems to consist of tackling a problem after allowing it to happen. Yet, this is a misjudgement because letting sounds become active is computationally negligible, and those sounds do not actually play, until the heuristics determine so (and how). Actually, we conjecture that this approach may turn out to be computationally more efficient than programming sound behaviour for all the predictable situations that might emerge from interaction. One conspicuous reason is that the algorithms that would be used to decide on sound behavior on the game's logic, would be essentially the same as those that we are proposing to inscribe in the heuristics module.

Also, there is no contradiction in cases when the game logic triggers a sound and the heuristics somehow override that decision, for two reasons. First, the game logic could have triggered that sound, precisely, because the developers decided not to make that kind of control at that level, but to rely instead on the heuristics to eventually decide on the actual rendering of that sound. Second, if the heuristics are well defined, their interposition should be legitimate.

On the other hand, we should also emphasize that the adoption of this proposal does not impose that no sound behaviours are controlled elsewhere, such as together with the game logic.

The heuristics operate on the active sounds, regardless of whether they were triggered unconditionally or as result of some prior verification. Provided that there are no incompatibilities in the definitions, there is no reason for not having several “layers” of sound design/implementation, complementing each other. In the extreme case, those multiple layers of decision would be (simply) redundant. That is also why we suggest that this proposal is seen as a complement to a development system and not as a replacement to other audio capable features that the adopted system may have.

Finally, it should be noticed that modularizing the knowledge on composing healthy soundscapes eases its customization, without interfering, and possibly concurrently, with the development of other aspects of the game. Not less importantly, since the heuristics are not bound to any particular project, they may be shared with other designers, which might be a way towards its maturation.

System’s Architecture

In order to reach the system’s final architecture, we performed an iterative process to constantly improve and refine the solutions it were being created. In this section, that iterative process and its different outputs will be presented.

DSCM’s Framing

When this project started, the only thing known was the goal: to create a dynamic soundscape composition engine. So, we had an idea about where would this module be situated in the game’s global architecture. The engine would be situated between the game logic and the audio renderer (see Figure 3.6 – DSC’s Framing).

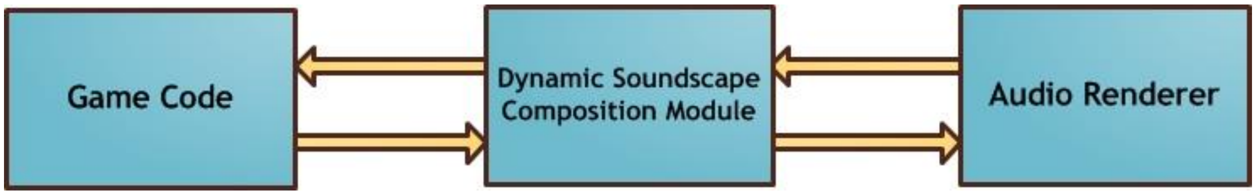


Figure 3.6 – DSC’s Framing

What we hoped to achieve with this solution was an abstraction layer situated between game code audio instructions, and the low-level and technical language used by audio renderers. After the definition of the DSCM’s framing, the architectural work focused on translating the decisions previously taken into a more formal and detailed specification, to further develop the system's architecture.

Major Functional Units

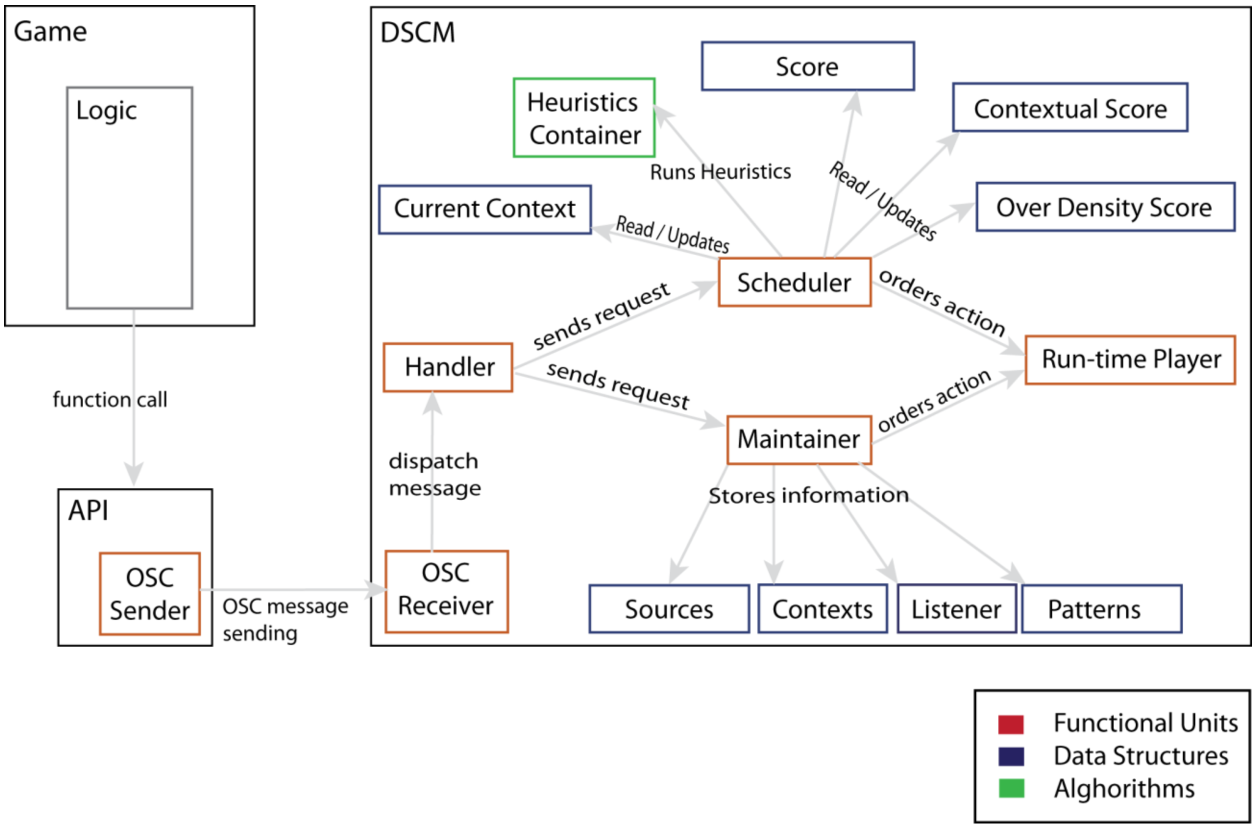
In order to continue to specify a dynamic view of our system's architecture, we started by defining the major functional units of our system. The following bullet points will detail each of these functional units.

- **Communication Interface** – The interface that deals with the communication with both the game code and the audio renderer. In the first case, it consists in a simple client-server architecture that makes use of the OSC communication protocol. This allows the game code and the module to be in separate computers, as the communication travels through network. On a different manner, the communication between the DSC module and the audio renderer is local, through programming code.
- **Request Handler –** It is the element that verifies the type of message received by the communication interface, and that decides to which element that message should be forward to. It is expected to perform functions similar to those of a Servlet.
- **Run-Time Player –** It is the representation of the audio renderer to be used by the DSC module. Its functionalities encompass many low-level audio functions, and are defined to be independent from the audio engine used to render the sound. With this approach, different audio renderers can be tested without having the need to change other functional units of the system.
- **Resource Maintainer** – It is the element responsible for arranging all the resources and initialization procedures that are necessary for the composition process to be able to be performed. Is the support unit behind the composition process performed by the Scheduler.
- **Scheduler –** It is the main element of the composition process. It is the scheduler that coordinates all the resources, analyses the state of the soundscape, and decides when and how to execute modifications upon it. It makes use of the heuristics stored in the heuristics container to aid on the composition process.

- **Heuristics Container -** It is where all the heuristics used by the scheduler in the composition process are stored. It contains not only the information about the list of heuristics available, but also all the behaviors associated with each of them.
- **Resources** – It represents all the resources used in the composition process, namely, the sources, contexts, and patterns created by the designer. These resources are used by more than one functional units of the system, being mainly used by the Resource Maintainer, the Scheduler, and the Run-Time-Player.

Dynamic View

Now that all the major functional units were defined, we wanted to completely define all the internal components of the DSC module, as well as to show the actions of the system during execution. The system's dynamic view architecture can be seen in Figure 3.7.



Following, we will explain the system's architecture presented in Figure 3.7. The box on the top-left represents all the code that is specific to a particular game project. That is where the game logic is, for instance. The bottom-left box represents the API we created, with the classes and methods that implement features made available for sound design. It contains, for instance, the code that creates a sound source or a context. The idea is that developers call such code from the game logic.

The sound engine is represented by the box on the right. The communication between the code in the API and the sound engine is ensured by *Opem Sound Control* messages. OSC is a content format for messaging among computers that is optimized for modern networking technology, offering a high-level of interoperability, accuracy, flexibility. It offers programmers an open-ended URL-style symbolic naming scheme, which enables easy pattern matching during message reception, as well as a very simple support for argument data in the messages. A simple example of an OSC message used in this project is: "/create/source/", being the information related to the source to be created attached as argument.

During the game execution, the *OSC Receiver* forwards the incoming messages (sent by the *OSC Sender*) to the *Handler*. The Handler parses the incoming messages and forwards them either to the *Scheduler* or the *Maintainer*, depending on their purpose. The Maintainer's job is to arrange everything that is to be used in the composition process: all the creation, deletion and edition of sources, contexts, patterns, the listener, and other low-level details. These tasks require the Maintainer to cooperate with the *Run-Time Player*. This cooperation occurs because the data that the Run-Time Player needs to initialize, related to each source, is also stored in the structures controlled by the Maintainer. *Sources*, *Contexts*, *Patterns* and *Listener* are structures, holding information on, respectively: the sources that have been created; the contexts that have been created; the patterns that have been created; and the listener entity. The latter is relevant in case the designer creates 3D sources. These objects are created when the designer uses the API to initialize these kind of entities, and are kept ready to be used until the game is shutted down.

The *Scheduler* coordinates the operationalization of the composition. It deals with stop/start requests regarding sources and patterns, as well as with changes to the active context. Whenever a source is requested to be player, it operates the decisions resulting from the application of a set of *Heuristics*, which in turn take into consideration the *Current Context* and currently active sources in the *Contextual Score*. The latter is a structure that maintains a categorization of active sources according to their semantics (layer). The heuristics can also assess other aspects such as the sources' associated agent, and sound design patterns. The Scheduler forwards orders to the *Run-Time Player,* which actually renders the sounds, also taking into account the information provided by the Maintainer, as explained.

It is important to refer that, during the composition process, the Scheduler makes use of three different Scores, all of them being composed of the five layers explained before: Ambiance, Dialogue, Music, Foley and SFX. The structure simply named *Score*, stores every source that the game logic triggered, and that did not finish or were requested to stop, whether the scheduler decided to make them audible or not. This structure enables the Scheduler to, at every moment, have a complete view of the soundscape that was triggered by the game, independently of what decisions the Scheduler have made over the source. *Contextual Score* is a structure that stores the currently active sources that are "in context", according with the engine's *Current Context*, and that respect Murch's Encoded-Embodied heuristic (when is activated). In practice, this is the structure which holds the precise soundscape which is being heard by the player. The third score, *Over Density Score*, is a structure that, as the name suggests, stores sources that, although being in context with the *Current Context*, belong to a layer which has already reached its limit, according the aforementioned Encoded-Embodied heuristic. This way, whenever a source is stopped or reaches its end in the *Contextual Score,* a source can be extracted from the *Over Density Score* and put on the *Contextual Score*. Independently from the score in which they are inserted, all sources are removed from it whenever they are stopped, or end.

Composition Process

In order to assure that the DSCM's composition process is easily understood by everyone, it may be of utter importance to clarify some issues regarding it. Currently, the module only supports one active context. This means that, every time the game code triggers the method setContext(), the module automatically disables the current active context, and activates the new one. Also, it is important to refer that, the context's exclusivity property, also influences the application of the heuristics. Specifically, if the *Current Context* is exclusive, the heuristics will only be applied to the sources included in the *Contextul Score.* On the other hand, if the *Current Context* is not exclusive, the heuristics will be applied to the sources included in the *Score* (in other words, to all the sources triggered by the game's logic).

Besides the Context heuristic, which is always applied whenever a play request is sent to the engine, the Scheduler also has to deal with a variety of different sources, each of them with its pattern, which results in many heuristics being applied at the same time. In order to manage this complex abundance of sources and patterns, we decided to implement a priority system similar to a stack. In other words, the heuristic that has most priority over the others is the heuristic associated with the pattern from the last source to be played. This means that, whenever a new source is played, the heuristic associated with its pattern may override the settings of the former. However, this only happens if both heuristics have impact on the same layers. Frequently, different heuristics only affect some sound layers. Consequently, this allows more than one heuristic to be shaping the game's soundscape, not only the most recent. Whenever a source ends, the heuristic associated with its pattern is removed from the top of the stack, and the new "first" pattern receives top priority treatment, and so on. Therefore, the game's dynamicity and unpredictability is always matched by this constant update of which heuristics should be more relevant in the soundscape composition process.

4 Results

In this chapter, we will present the results obtained in this dissertation, from both the prototyping activities, as well as the experimental evaluation that was performed. We will start by detailing the preparation for the prototyping phase, following this by explaining meticulously what was prototyped in each month. Also, we will justify the work's prioritization regarding the system's prototyping.

Additionally, we will make a thorough explanation of the experimental evaluation performed. It will be justified why the used testing technique was chosen, it will be explained the experiment's planning, the scenario in which tests were performed, and, finally, we will present and analyse the results.

4.1 Prototyping Activities

As soon as we had a first problem definition and a first solution with sufficient depth, we were ready to start the development activities. Between the work developed in the first semester, and the development activities, it is important to refer that there was still some refinement to be done to the adopted solution, as well as some decisions regarding the technical component of the project.

The first decision to make was related to the type of data structures to be used in the project. This may look a simple question at first, but the fact that the engine have to deal with features that are constrained by time, turned this decision into one of the most delicate to be taken throughout the whole project. Additionally, this decision would have impact on the engine's behavior, especially on the composition process. Therefore, many issues regarding the composition process were also dealt with in this period. A great number of different ideas blossomed from the research performed, but with limited time for development, we had to focus on what appeared to be the most important features to fulfill the project's goals.

Similarly, at this stage we had to close some issues regarding the system's architecture, as well as the technologies to be used. With the data structures and the composition process defined, we could refine and close the architecture proposal we had. In the end, there were some major differences from the architecture proposed in the intermediate report, but that was one of the benefits of the methodology adopted for the project (see Methodology). With all the design and architectural constraints defined, we could finally choose the language in which the engine would be developed. After deciding what would be the best audio renderer to fit the project's objectives (FMOD), we decided to develop the engine in C#, mainly because it would be more easily used for developers using some game-related frameworks (i.e., XNA and Unity). Besides, the chosen audio renderer (FMOD) had a built-in C# wrapper, what resulted in a perfect match for our project.

Finally, the last decision before starting the development phase was the decision about what game scenario would be used to implement and test the DSC module. In order to essay the debuting implementation of our proposal we decided to resort to a game that we designed and developed previously to this particular research endeavor, called Blindfold.

Blindfold is an adventure audio game, in which the player is invited to wear an actual blindfold. The game is projected as a soundscape where players walk through a rich and enigmatic experience, with emotions being evoked not only by the intrinsic acoustic characteristics of the sounds being used, but also by their semantic content, designed to stimulate the sensemaking dimension of the gameplay experience [Pereira and Roque 2012]. The use of the physical blindfold adds meaning to the interface and is consistent with the game narrative.

For the sake of better characterizing the game, in the scope of this dissertation, we include a debug screenshot representing a gameplay situation, in Figure 4.1 - Debug screenshot of Blindfold. Illustrations such as these are meant for development purposes only, since the game is exclusively auditory.

Blindfold was originally developed in XNA, a Microsoft's framework for videogame development. Therefore, the game's sound was implemented using XACT and its authoring tool. It is possible to use XACT both as a low-level API, for simple sound usage, or as a middleware tool. For Blindfold, we used the latter.

What was desired was to try to replicate the sound design of Blindfold, this time using our proposed system to implement it. Additionally, being an audio-only game, Blindfold gave us many different situations with different sound explorations, which would allow experimenting different behaviors related to the patterns to be implemented. Blindfold was a natural choice for this experiment, because we were acquainted to its sound design goals, and it constituted a promising stage to experiment with sound. It is of the outmost importance that this scenario is not understood as a comparison between ‘using’ and ‘not using’ the proposed approach. That could be, indeed, and interesting exercise but it would imply dealing with other delicate aspects that were not central to the intended observations (such as, which exercise should be done first, or how to perform the “same” exercise separately in different conditions).

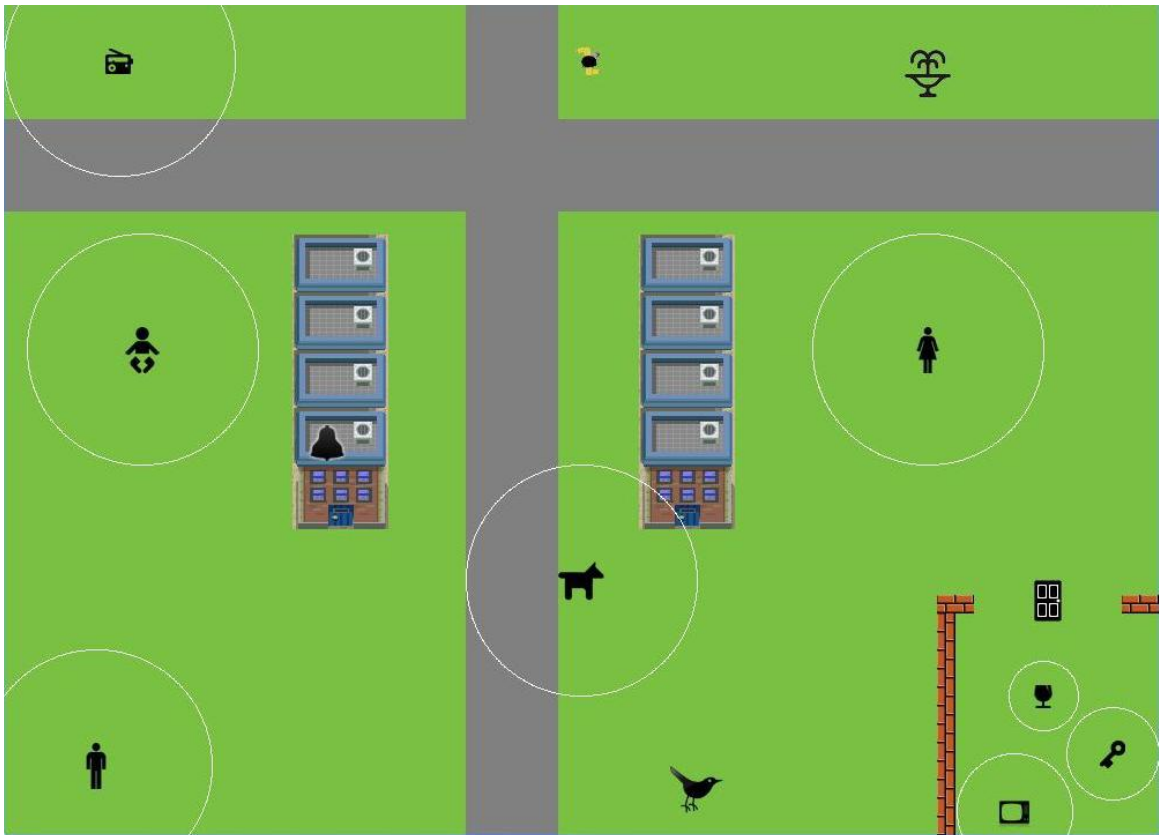


Figure 4.1 - Debug screenshot of Blindfold

4.1.1 Activities Developed

In the following subsections, we will detail the prototyping activities developed throughout the second semester. These activities are divided according to the sprint in which they were developed.

Sprint #1

The development phase started with some modifications performed on Blindfold. Initially, we re-structured the game’s code to be more organized, clean, and to be structured in a more object-oriented fashion. This was necessary for two reasons: two facilitate any changes that we could find necessary to implement in the game; and to make the game's code more understandable, which could be useful later in the project's lifetime, if we decided to perform API-related testing (as it turned out).

In addition, and to finish the necessary transformations to allow Blindfold to be used in our project, we muted the game. In other words, we removed from the game's code all the lines related to the game's sound implementation. This required also some modifications to the game's logic, because, being this an audio-only game, this removal of sound had profound implications in it.

With the completion of these two tasks, the game scenario required for the experiment of the DSC module was ready to be used. The next step in the development process was the prototyping of a simple version of the DSC module, still without the structures needed for any composition process. This first prototype was meant mainly to build the communication infrastructures between the game and the module. Using an OSC open source framework, we developed communication interfaces to be used on both sides (game and module). These interfaces are generic, so, instead of OSC, other communication protocol can be tested in the future. While on the game's side, the communication interface consist in a client ready to send requests, on the module's side, this implementation consisted in a server that is constantly listening for requests from the game. This server, after receiving requests, forwards them the second component of the module to be developed, the Request Handler, which was prototyped in the second half of the sprint. After completing these tasks, we had finished the implementation of the skeleton that would support all the work to be performed in the future.

To sum up, the backlog processed in this sprint was:

- Modified version of the game Blindfold;
- Alpha version of the DSCM, only with communicational modules prototyped.

Sprint #2

With a channel of communication ready to be used, it was now necessary to give the game (or the programmers) a "language" to express their intentions in the game code, and also the OSC address patterns that were going to be used to translate these intentions to a type of information understandable by the DSC module. Moreover, we had to embody the engine with the necessary knowledge to respond adequately to the requests received from the game.

Therefore, in this month we started by developing the API that would be at the disposal of designers and programmers (see API Design). This API is composed of 4 classes to be used by the programmers (Context, Listener, Source, and Pattern), and one support class that is only used by these for in some operations (Utils). After implementing these classes, we had to program the address patterns hidden behind the methods offered by the API. In other words, we had to define the syntax of the OSC requests that each action deployed by the API would originate. After defining the naming logic to be used in the address patterns, we had to implement them on both the OSC sender (game's side), and on the OSC receiver (module's side).This step was extremely important due to OSC's pattern matching verification that occurs whenever an OSC server receives a request. This way, requests that do not match the expected address patterns are automatically discarded.

Following the completion of all the tasks that regarded communicational and linguistics aspects of the system, the next step in the development process consisted in programming two of the main components of the module: the Maintainer, and the Scheduler. At this stage, although they were not being used, we knew which requirements they had to meet, so, we implemented them for the sake of allowing the engine to become more close to the architecture previously defined as soon as possible. Nevertheless, we decided to prototype another of the main components of the module: the Run-Time Player. This component is the interface of the audio renderer. Besides the interface, we also coded the specific implementation of the specific audio renderer used in the project (FMOD). This task involved the implementation of the low-level operations necessary for the management of the soundscape (i.e., play, stop, volume change, sound initiation, attenuation, engine's listener, etc.). Additionally, we also implemented all the data structures projected in the system's architecture, without which the system could not perform any compositional work. These structures include the structures that store all the sources created (Sources), all the contexts (Contexts), all the stand-alone patterns (Patterns), and, finally, the structure that stores all the sources that are playing (Score).

With the core low-level operations of the audio renderer implemented, we could now try to mimic the original Blindfold sound design without any dynamic composition process. In other words, we prototyped a version similar to the original one, but now using our engine, although the play and stop requests received from the game's code would be automatically performed without any additional reasoning. At this stage the engine did not have any scheduling and reasoning process. It may seem that this did not have any interest for the project, but it was an important stepping stone for us to see that at this stage, we were already doing what the previous implementation of Blindfold was doing.

On the second half of this month, we decided to implement the first heuristic: Context. The implementation of this heuristic required the update of the address patterns to be used, both on the API and on the module, as well as the different types of requests that the Request Handler should be ready to respond to. Likewise, we had to create a new structure on the module: the Contextual Score. As referred in Dynamic View, the difference between this structure, and the aforementioned Score, is that, while the latter contains all the sources that the game's logic required to be played, the Contextual Score contains only those who been required to be played and belong to the current context. Lastly, we had to implement the first heuristic behavior in the Heuristic Container for the Scheduler to start composing the soundscape dynamically. This required the method with the expected behavior to be coded into the Heuristics class. Following this implementation, the scheduler class had to be changed in order to run this heuristic before deciding when and how to play a request source. Moreover, the audio renderer implementation had to be updated with code to manage the two groups of sources that emerge from this implementation: sounds "in context", and sources "out of context". In order to achieve this separation, we implemented in the audio renderer class support for audio buses, one of FMOD's most useful features. After this task was completed, we had the first and probably the most important of the heuristics ready to be used.

To sum up, the backlog processed in this sprint was:

- Final specification and prototyping of the API.
- Beta version of the DSCM, with the audio renderer fully implemented, and remaining modules' skeleton prototyped.
- One heuristic implemented (Context heuristic).

Sprint #3

In the last month of the development phase, there was still much work to do. We had still nine heuristics to implement, as well as the creation of situation in Blindfold, in which we could experiment those heuristics. The implementation of the heuristics that were still to be implemented consisted in the enrichment of the Heuristics Container. As each heuristic was being added, new behaviors were being coded on it, which were at the scheduler's disposal. In similar fashion with the work done with the first heuristic (Context), we also had to update the address patterns of the OSC communication interfaces. Obviously, the scheduler was being constantly updated in order to support the crescent number of heuristics that should be run in the composition process, while deciding if, and how, sources should be played. The last structure that we had to prototype, due to one of the heuristics (Murch's Encoded-Embodied), was the Over Density Score (see Dynamic View).

When all the proposed heuristics were implemented, we started to define game situations in which they could be tested. This type of experiment allowed us to test and refine the behaviors associated with each pattern. Blindfold proved to be a good scenario to experiment due to the different situations which contains. We implemented also a system which allowed us to activate patterns on the keyboard, without requiring a specific game event to be met, which allowed to change patterns and contexts on-the-fly, and to compare different contexts and patterns in the same game situation.

To sum up, the backlog processed in this sprint was:

- A version of the DSCM with all the heuristics implemented.

Sprint #4

Although we did not perform any experimental evaluation regarding the heuristics and its behavior, the heuristics implementation suffered iterative refinement, due to our perception of its behavior. For instance, in sprint #3, the heuristic thoughts implemented a reverb effect on the source to be played. However, during the analysis of the heuristics we were doing while testing the DSCM, we realized it would be better to change that effect to an echo effect. Moreover, many of the volume levels that each heuristic performed on each sound layer was iteratively tested and modified until we reach the final values, presented in this document.

To sum up, the backlog processed in this sprint was:

- Final version of the DSCM with refined heuristics.

4.1.2 Work prioritization

There are some relevant issues that need to be pointed regarding the prioritization of the prototyping work developed in this project. One of the questions that were left open after the first semester was the choice of the game scenario to be used. We decided to make that decision first, mainly because this uncertainty regarding the game scenario to be used was mentally disturbing. As soon as the scenario was chosen, we felt as the path to the project's success was now clearer.

The following priority for us was the communicational mechanisms of the system. Being this a distributed system, communication is vital for its performance. Accordingly, we felt that the communication framework was the first thing to be tested and verified, as it would be the skeleton that would support all the more important work to be implemented in the future. In the same line of thought, the API was extremely necessary for us to allow communication between the programmer and the module. This may seem like contradiction, as at this stage we were not running any work on the module's side. However, we wanted the next step to be a simple version of the module, and we wanted to be sure that the communication was already similar to the one projected to the final version, instead of creating stubs just for this intermediate version of the DSC module we wanted to prototype briefly.

As referred in the previous paragraph, the next priority on the scheduling was to have a simple version of the engine, which was architecturally close to the one we aspire to develop, but that still did not do any dynamic composition task. As referred in Activities Developed, this was an important stepping stone for us to see that at this stage, we were already doing what the previous implementation of Blindfold was doing. In addition, this was also a guarantee that the low-level implementation regarding the Run-Time Player was going well.

From this point forward, we had to deal with the most important part of the project. We knew that from now on, all we had to worry was the tasks related to the composition process, and that they were going to be built over a stable version of the engine, which gave us confidence to face the challenges ahead. The option of giving priority to the Context Heuristic was due to its importance for the goals this solution aims to achieve. This heuristic by itself can help designers to control a large number of sources in a very simple fashion, only requiring a few lines of code. After the implementation of this heuristic, we implemented all the remaining nine without any special order, the objective was simply to complete the proposed list of heuristics and to finish the module. After the module was complete, we tested the heuristics in different game situations and refine their behaviors according to the sonic feedback received from these experiments.

To sum up, we follow this prioritization because our mindset since the beginning of the project's planning was: to lay the foundations on which the system would be built upon, create a basic implementation to allow the confirmation of the architectural decisions previously made, and to allow us to have a strong support framework over which we could continuously refine and evolve the main focus of the project, the composition process.

4.1.3 *Expected behavior*

The behavior observed by the DSCM was close to the expectations. The use of contexts allowed the sound composition to offer the player the most valuable sounds in a large number of different situations. However, there is space for improvement in terms of transitions between different contexts. In terms of heuristics, the results also look promising, with many heuristics really helping the sound design in achieving the proposed objective (i.e., the heuristic *Thoughts* really helps to intensify the sensation of "monologue occurring inside a character's head"). In general, the heuristics' behavior was very close to what was expected. Furthermore, the fact that heuristics could be tweaked by designers, allow for a positive improvement and sharing of heuristics. This way, the community can also be involved in the further development of the engine. More information regarding the system's limitations and performance aspects can be consulted in Conclusions and Future Work.

4.1.4 *Requirements*

In this section, we will present the requirements defined in the first semester, and discuss whether they were achieved or dropped.

Soundscape Specification API

R1 - Soundscape Description	Must	The sound designer should be able to define a soundscape using this specification language
R2 - Allow references to contexts of sound explorations	Should	The game engine and the dynamic composition module should communicate using this language. The main goal is to allow references to contexts of sound explorations just by the name of the verbs used.

Table 4.1 - Soundscape Specification API Requirements

Dynamic Soundscape Composition Techniques Systematization

R3 - Allow sound to be an information carrier	Must	Just like argued by Acoustic Ecology's founders, sound should be more than just a stimulus, it should be a communication

		interface, and these techniques should allow sound to be an information carrier between the game and the player.
R4 - Analyze sound’s characteristics	Should	The techniques defined should take into account the intrinsic characteristics of sounds. Instead of just analyzing a sound by name or category, its frequency and volume should be also taken into account.
R5 – Allow coherent audition	Should	The techniques should allow other actors in the game scenario to hear stimuli that is coherent with the current soundscape composition. We want to change the deafness of current game engines. Presently, the intrinsic characteristics of a sound (i.e., power, frequency, rhythm, etc.), or the current soundscape state is not taken into account when the game is deciding what is heard by the actors.

Table 4.2 - Dynamic Soundscape Composition Techniques Systematization Requirements

Run-time Dynamic Composition Module

R6 - React to directives	Must	The module must adapt to the changes being made by the sound designer accordingly, while keeping the effects of sound inside the game coherent.
R7 - Allow run-time prototyping	Must	The module must not require neither the game nor itself to be restarted whenever changes are made by the sound designer.
R8 - Visual Interface	Must	The module must have a visual interface to allow the sound designer to change directives in an easy and intuitive way.
R9 - Statistics	Nice	The interface available to the sound designer could show some statistics about the sound engine in order to give the sound designer another way of analyzing the different soundscape settings.

Table 4.3 - Run-time Dynamic Composition Module Requirements

- R1 – The designer is indeed capable of describing a soundscape through the API that was developed. Although, this was not achieved by a stand-alone language, but through a programming API. The constraints of building a system that would parse a discourse closer to natural language would require much more time than that available for this dissertation. Still, the API's syntax allows programmers to draw some conclusions on the soundscape that was developed in a game just by looking and the declarations and instructions, due to the patterns and agents associated with sources. Additionally, the declarations of contexts also help to describe the soundscape.
- R2 – Due to the changes in the previous requirement, this requirement had to change. Nevertheless, although the system resorts to an API, we still can say that the game and the DSCM communicate through the same language (the patterns, agents, contexts, etc.), and that they allow the engine to understand different sound explorations just by their name. As the engine resorts to the heuristics to answer to each specific pattern, we can say that this requirement was fulfilled.
- R3 – Due to the behaviors (heuristics) implemented in the engine, whenever the engine is playing a source associated with a pattern, it delivers the sound in conditions that were thought to reach a specific pattern. Therefore, we can say that every sound plays in conditions that are defined to allow players to experience a specific type of stimulus. So, it is up to designers, and programmers to make sure they use the API's full potential to deliver not only sound stimuli, but important semantic information through sound.
- R4 – This requirement was dropped due to the amount of work that would require to implement it, and because there were other functionalities that we thought were more important for the system. Nevertheless, we still think that this is a functionality that is deeply unexplored nowadays in videogames, and that can offer developers conditions to create creative sound explorations in their games.
- R5 – In equal manner as the previous requirement, the time constraints of the project did not allow this requirement to be fulfilled. A type of requirement like this would demand a deep research and analysis, because it would involve the game's logic. Moreover, the system would need another channel of

communication for the engine to “talk” to the game’s logic. All of these functionalities are detailed in the section Limitations and additional features/usage for the project.

- R6 – This requirement was fully achieved. The system reacts to all the directives implemented by the programmer, and the sound composition is always done accordingly. In many situations, the directives defined by the programmer can conflict with one another, and is up to the engine to decide the soundscape state (i.e., what should be the pattern with max priority, which should be the current context, etc.).
- R7 – Allow in this dissertation it was not programmed any plugin to tweak the directives in plugin, the system itself is fully prepared for that type of usage, as the DSCM is completely separated from the game’s code, communicating through network (OSC).
- R8 - While the value of this requirement cannot be neglected, the time that would be required to implement it would be large, and between implementing this feature, or the possibility of spending more time adding new heuristics to the engine, we decided to go with the latter.
- R9 – During the course of the development, this requirement was dropped because it was not very important, unlike many of the others. It will make more sense in the future, with a mature version of the DSCM.

4.2 Experimental Evaluation

The main objective for the testing phase of this dissertation was to test how programmers would adapt to the API; how easy it would be for them to understand how to use it, especially the theoretical concepts behind it. Being this one of the main goals of the system, it would be very important to verify if they would have difficulties understanding the mechanisms behind the API. Moreover, it would be useful to test if its syntax could really be more understandable and give information about the design intents that the code is following.

4.2.1 Testing Technique

In order to obtain information regarding the aforementioned goals, we chose to perform a Formal Usability Lab Test. There were various reason to support this choice. Without requiring a large number of participants, it allows the collection of personalized data regarding each of the participants, without external interference. This is an important issue, because what we were trying to achieve was a test that would enable us to detect problems in a controlled environment, with minimal interference from the person supervising the test.

In comparison with Informal Usability Tests or Heuristic Evaluation, its results are considered to be more reliable. Similarly, Field Tests were not a valid alternative because in an internship context, it would be impossible to find an actual project to perform the testing activities.

However, it is not a technique free of problems. Being a formal exercise, participants may feel pressured and not be comfortable with the setting. Furthermore, it requires more time to prepare, as it demands the experience to be designed, the participants to be selected, definition of the tasks to be performed (scripting), preparation of materials, data collection, analysis, etc. Still, we thought that it would be the best solution to achieve our goals for the evaluation experiment.

4.2.2 Test Planning

As referred previously, we wanted to perform an experience that would allow us to test the API that was developed. So, there were four steps while planning the test activities: design the experience, select the profile of the participants, develop the tasks to be performed, and, prepare the materials to be used.

Therefore, we decided that we wanted the API in a programming environment. In other words, we wanted to test its application in real programming code, not only test the knowledge of the participants regarding the API, in abstract. Although the focus of the experiment was the API, which means it was a programming-oriented test, we reflected and recognized that we had to be very careful while designing the experiment. We wanted testers to be focused on questions regarding the API, not stuck on the syntax behind the programming scenario in which it would be tested, or spending too much time understanding particular syntax issues of the programming language used to code on the API. The main objective was to test the logic of its application, how would the testers apply the concepts offered by the API.

There was the option of testing the API by itself, not applied to a specific code from a game scenario. However, that kind of test would be too abstract, informing only whether testers had memorized the syntax. We think that, instead of testing the API in abstract, it would be much more valuable to test also how the players would perceive the relationship between code related to the API, and code from a game. That is why we chose to use methods from n actual game (Blindfold) in this experiment. Nevertheless, it would be very hard for the players to, besides all the knowledge to grasp regarding the API and its usage, have the need to learn about Blindfold, its characteristics, story, etc. It was imperative to not overload the tester with information regarding the chosen game scenario. That was the reason why each method had a detailed description, and each line of code from the method had a comment describing its purpose. Tester should instantly understand the meaning behind each method, as that was not the purpose of the testing experiment.

Likewise, we simplified the instructions' syntax, transforming the C# instructions almost into pseudo-code. The objective of the experiment was not to verify how well testers knew C# and its syntax. What was important was to use the language to make sure testers understood the purpose of each method.

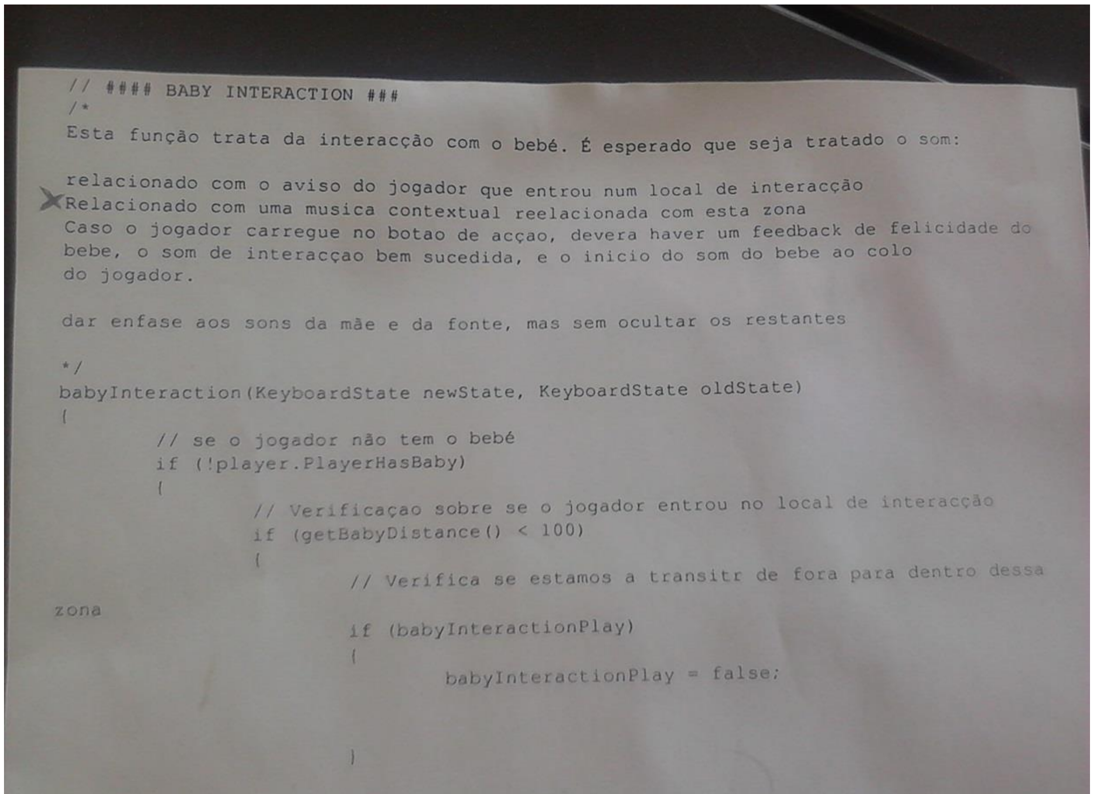
In terms of the participants' profile, the only requirement was that they had to have some programming background. Again, although the code from the methods was in pseudo code, it would still require programming knowledge to understand the logic behind each method. Moreover, this requirement is common to the API, so, it was a logic requirement. The number of participants was 6, 5 male, and 1 female.

The tasks that we desired participants to perform involved all the classes from the API. We wanted them to use all the methods that the API had to offer. Consequently, we had to choose what group of methods, from Blindfold, would offer more possibilities in terms of different usages for sources, contexts, patterns, and the listener. After the selection process, we ensure that the chosen methods would give the opportunity for testers to use all types of contexts, various sources with various agents and patterns, and to edit the listener's properties.

However, in order to achieve all the goals listed in the previous paragraphs, we had to make a decision: the test would not be performed in an IDE. We wanted to simplify the code from the game sound scene, as well as to guarantee that the participants would not be overloaded with classes, project packages, and struggling to other computer/IDE related issues. So, we decided that we would resort to paper cut-outs to perform our experiment. We knew that would not be a conventional solution, but, due to the aforementioned constraints, we truly believed it would be the best approach, in order to guarantee that testers would only be focused on the issues that were important for the experiment.

4.2.3 Test Scenario

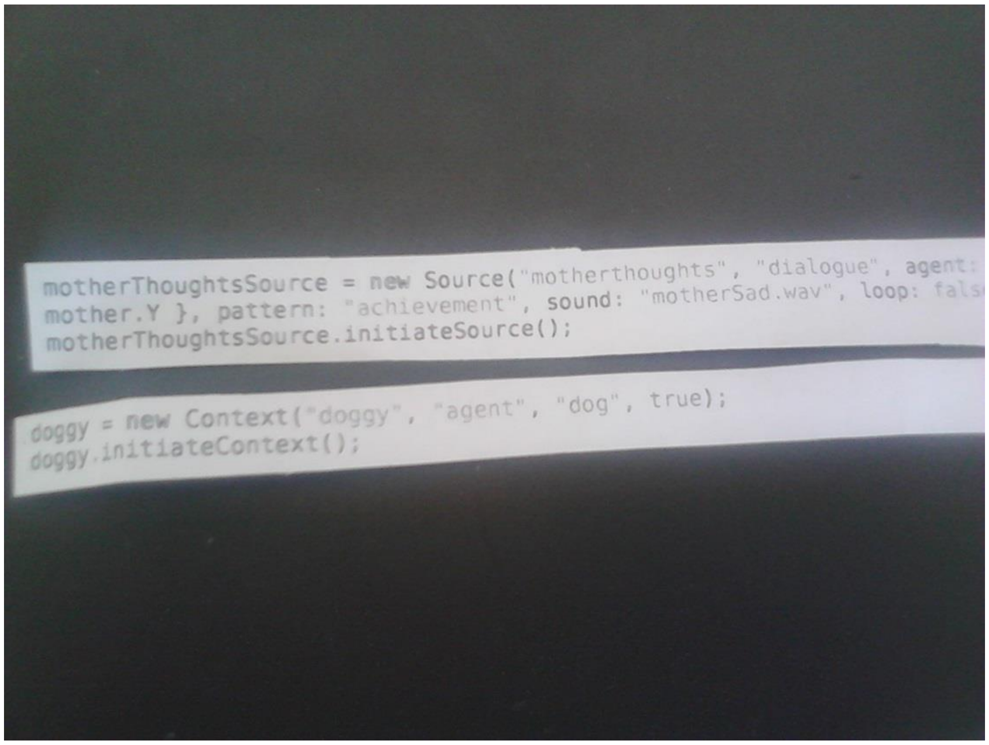
Firstly, each participant started to receive some theoretical lesson about the concepts behind the API. Each of the classes, and methods, were explained both theoretical, and through code examples. This preparation phase took about 15 minutes for each of the participants. The participants were told that the idea was not for them to memorize what was being explained, but instead to try to understand the concepts being demonstrated.



```
// #### BABY INTERACTION ###
/*
Esta função trata da interacção com o bebé. É esperado que seja tratado o som:

relacionado com o aviso do jogador que entrou num local de interacção
Relacionado com uma musica contextual reelacionada com esta zona
Caso o jogador carregue no botao de acçao, devera haver um feedback de felicidade do
bebe, o som de interacçao bem sucedida, e o inicio do som do bebe ao colo
do jogador.

dar enfase aos sons da mãe e da fonte, mas sem ocultar os restantes

*/
babyInteraction(KeyboardState newState, KeyboardState oldState)
{
        // se o jogador não tem o bebé
        if (!player.PlayerHasBaby)
        {
                // Verificaçao sobre se o jogador entrou no local de interacção
                if (getBabyDistance() < 100)
                {
                        // Verifica se estamos a transitr de fora para dentro dessa
zona
                        if (babyInteractionPlay)
                        {
                                babyInteractionPlay = false;


                        }
```

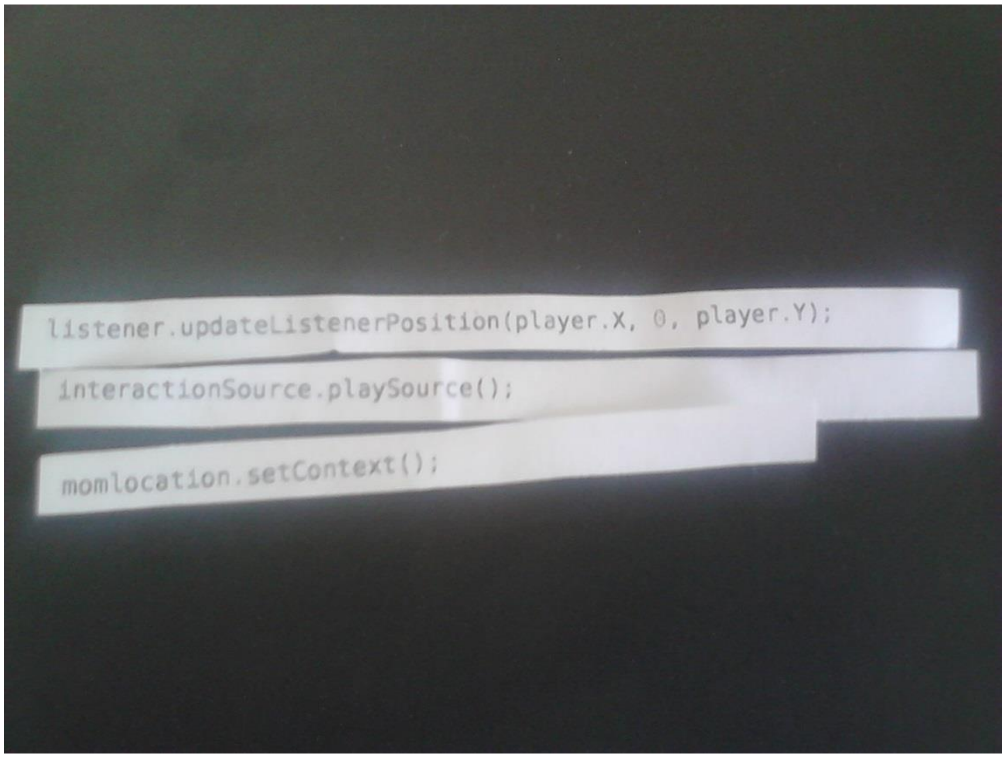
Figure 4.2 - Method's description

Next, participants had at their disposal five methods extracted from Blindfold's code, printed to A4 sheets. However, all the lines of code regarding sound were removed from the methods. Still, these methods had above them a text describing it, and referring which sonic events were expected to be dealt with in it (see Figure 4.2).



```
motherThoughtsSource = new Source("motherthoughts", "dialogue", agent:
mother.Y }, pattern: "achievement", sound: "motherSad.wav", loop: fals
motherThoughtsSource.initiateSource();

doggy = new Context("doggy", "agent", "dog", true);
doggy.initiateContext();
```

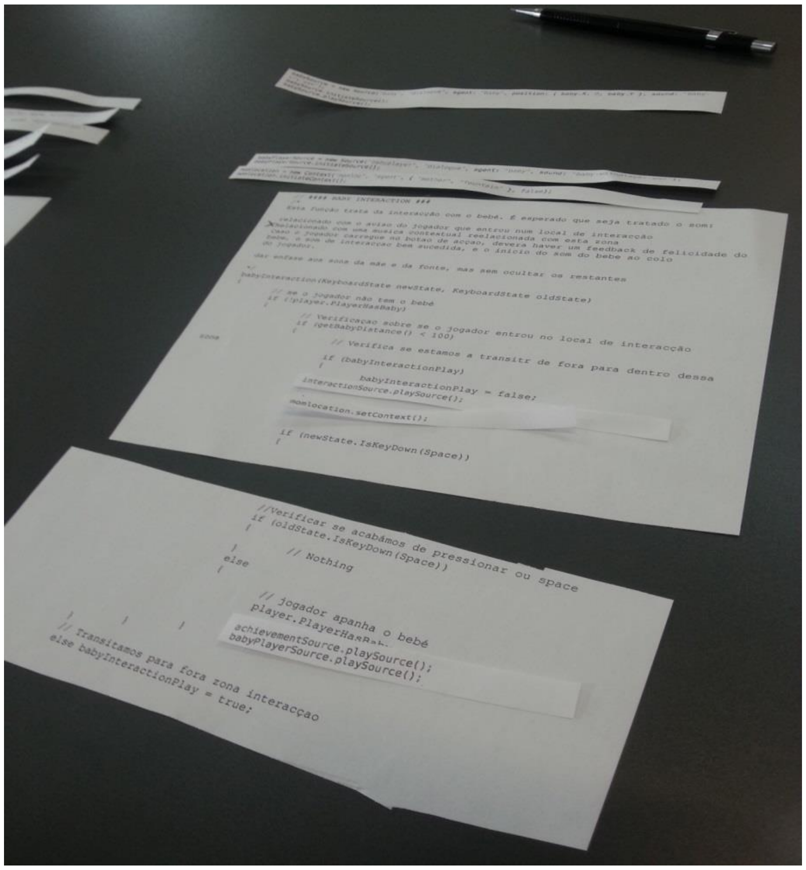
Figure 4.3 - Declaration examples



```
listener.updateListenerPosition(player.X, 0, player.Y);

interactionSource.playSource();

momlocation.setContext();
```

Figure 4.4 - Instructions example

Testers also had at their disposal two types of paper cut-outs: one with the declaration of the sound sources to be used on those functions (see Figure 4.3), and another one which had the actual lines of code to be included on the aforementioned functions (see Figure 4.4). The objective was to put the testers in a position where they would need to read the methods' description, analyse their code, and, by evaluating which sonic events were expected to occur and looking at both the declarations and the lines of code, select from the paper cut-outs the lines of code that belonged to a specific method, inserting them in the right place (Figure 4.5 and Figure 4.6). This would require them to understand the various types of declarations for sources, contexts, patterns and for the listener. Moreover, they would also search on the lines of code, which ones would use the objects they thought would be needed for a specific method, and if the actions performed would achieve the goals specified in the method's description.



```
[illegible]

// #### BABY INTERACTION ###
/*
Esta função trata da interacção com o bebé. É esperado que seja tratado o som:
    relacionado com o aviso do jogador que entrou num local de interacção
 X relacionado com uma musica contextual reelacionada com esta zona
   Caso o jogador carregue no botao de acçao, devera haver um feedback de felicidade do
   bebe, o som de interaccao bem sucedida, e o inicio do som do bebe ao colo
   do jogador.

 dar enfase aos sons da mãe e da fonte, mas sem ocultar os restantes
*/
babyInteraction(KeyboardState newState, KeyboardState oldState)
{
    // se o jogador não tem o bebé
    if (!player.PlayerHasBaby)
    {
        // Verificaçao sobre se o jogador entrou no local de interacção
        if (getBabyDistance() < 100)
        {
            // Verifica se estamos a transitr de fora para dentro dessa
            if (babyInteractionPlay)
            {
                babyInteractionPlay = false;
interactionSource.playSource();
momlocation.setContext();

            if (newState.IsKeyDown(Space))
            {
                //Verificar se acabámos de pressionar ou space
                if (oldState.IsKeyDown(Space))
                {
                    // Nothing
                }
                else
                {
                    // jogador apanha o bebé
                    player.PlayerHasBab
achievementSource.playSource();
babyPlayerSource.playSource();
    }       }       }
// Transitamos para fora zona interacçao
else babyInteractionPlay = true;
```

Figure 4.5 - Method with instructions added by a tester

Each of the five methods explores different aspects of the features offered by the API. One of them explores functionalities related with the listener, while the other four explore the usage of sources with different types of patterns, and the usage of different types of contexts. It is important to refer that not all the paper cut-outs are used in the exercise, existing some of them which are very similar to the ones to be used. The aim of this is to verify if the testers can differentiate the sources by their characteristics, and according to the methods' needs.

We hoped this could inform us about what problems would the testers had, while using the functionalities offered by the API. To obtain the desired information, while participants were performing the test, we were monitoring their performance and taking notes of every relevant event, either "positive" or "negative". Furthermore, if participants had any doubt interpreting the methods, or any question regarding the API, we tried to help them, though only by giving them hints, or helping them understand some specific aspect. The data gathered during the experiments, and the conclusions drawn from it will be detailed in the following sections.

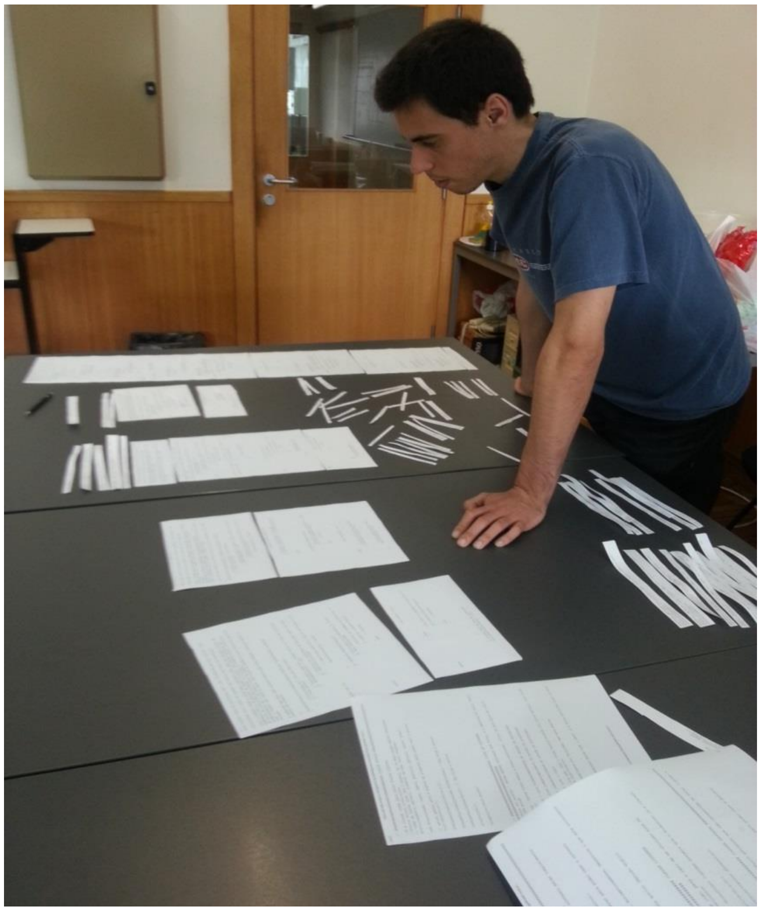


Figure 4.6 - Participant performing the test

4.2.4 Results, Analysis and Improvements

The observation and correspondent collection of information performed during the tests, resulted in a table for each of the participants with a header row similar to the one presented in Table 4.4.

Event	Type	Task	Importance for user	Possible Solution

Table 4.4 – Header row of the event collection table

As depicted in Table 4.4, the information that was being collected was: the event that was being reported, the typology of the event (inserted afterwards), the task (which was the method that the participant was completing), a classification of what was the importance of the event for the user, and, finally, a possible solution for the problem (when applicable). It was also recorded the time spent completing each of the methods, and a counting of the number of instructions inserted in each method, the number of well-placed instructions, the number of instructions placed in wrong positions, and the number of missing instructions in each method.

From this information, we made various tables to summarize different aspects of the collected data. Firstly, we made a purely performance-oriented analysis, in which we observed the rate of completion of for each of the five methods (presented in Table 4.5). The table has the following structure: the first column contains an entry that represents all the instructions to be inserted in the exercise (from the five methods); the following columns have data regarding the instructions inserted, correct instructions, wrong instructions, and, missing instructions. Moreover, each of these columns is further divided in two: one for the mean, and one for the standard deviation. The values presented in the lines below are in percentage format.

	Instructions Inserted		Correct Instructions		Wrong Instructions		Missing Instructions	
	Mean %	STDDev %	Mean %	STDDev %	Mean %	STDDev %	Mean %	STDDev %
Instructions (24)	21,5	2,5	18,7	4,0	3	5,0	4	2,9
Instructions (%)	89,6	10,5	77,8	16,6	12,5	20,7	16,7	12,1

Table 4.5 - Performance-Oriented Analysis Table

It is important to refer that, due to the number of participants, it is not recommended to make any statistical analysis of this data. In order to approach it in that perspective, a larger number of participants would be required.

Nevertheless, it appears to point towards a tendency for a positive application of the API on the experiment. There is no evidence of testers simply using an overabundance of instructions (even with the standard deviation value, the quantity of instructions inserted does not surpass 100%). Also, there is evidence of a good understanding of the methods functionalities, judging by the results regarding the instructions well placed in the methods. Still, the values regarding instructions misplaced in the methods show some evidence of some confusion while interpreting the code and/or the API. This could be due to a difficulty in understanding code from never seen before game, even with the method's description that was given to participants. However, the comprehension of the API's functionalities, and mechanisms, could have also been misinterpreted by testers, due to this being their first contact with it. Furthermore, the time that they took to complete each method, as well as the verbal commentaries they gave throughout the test, show evidence of a hard first contact with the API.

However, it is hard to ensure whether the main cause of the results was the perception of the API's functionalities, and concepts behind it, or if it was the description of the methods that had more weigh in the exercise's resolution.

Besides this analysis, we also tried to analyze the frequency of events reported during the experiment, according to its typology, and also, according to the task (method) related to it. The typology was defined after all the tests had been performed, according to the type of events that were observed. Furthermore, in this analysis, unlike the performance-oriented one presented previously, the results will be separated by task (method), in order to allow us to see also the differences between them. This data is presented in Table 4.6.

	Type	**BabyInt**	**MotherInt**	**KeyInt**	**DogInt**	**UpdateInput**		***Total***
Frequency of events	Game Interpretation	2	1	1	0	0		4

by Type	Game Interpretation Problem	6	8	3	2	0		19
	Code Interpretation	2	1	0	0	0		3
	Code Interpretation Problem	4	0	2	1	0		7
	API Interpretation	9	10	8	2	1		30
	API Interpretation Problem	5	11	2	2	0		20
	Decoding Data	8	2	1	1	0		12
	Decoding Data Problem	1	4	7	4	0		16
	Exercise Interpretation	2	0	1	2	1		6
	Exercise Interpretation Problem	4	0	0	0	0		4
	Total	43	37	25	14	2		121

Table 4.6 - Event's Frequency Table

It is important to refer that the methods (columns), are ordered by the order that they were completed by the participants (from left, to right). Therefore, the first thing that is possible to observe is that the number of events observed decreases in each method. This can be considered as evidence of a gradual understanding of the API and its mechanics, though it shows a tendency to a difficult first interaction with the API. Being one of the API's objectives an easy learning and usage, there appears to be some work to be done on this matter.

Another tendency that is highlighted by the data presented in the table is that most of the problematic events are centered in three categories: game interpretation problems, API interpretation problems and data decoding problems. The tendency regarding game interpretation problems confirms one of the difficulties that were expected: how to give the participants the background regarding the game scenario (Blindfold), without overloading them with information, preventing them to focus on the API. We have to also consider the possibility of the method's descriptions not being clear enough for someone who did not have any previous contact with the game scenario used in the experiment. Throughout the experiment, many testers had to ask questions regarding the game and its methods, due to their difficult in understanding the game scenario and all its constraints. Maybe there should have been a more deep explanation of the game scenario to be used, or a playtesting time with the game, in order to let players make their perception of the game's mechanics on their own (and, probably, in a more interesting way than by reading or hearing a detailed explanation).

The other major categories of problems (API interpretation and decoding data), are mainly related to the API. Most of the problematic decoding data events involved the API, being either a difficulty interpreting the names of sound files used, or the semantics of the source's names chosen. Also, the fact that players received a theoretical explanation of the API, with only a few code examples, increases the possibility of these problems decoding data. Therefore, this tendency towards a difficult data decoding, reinforces the opinion that the naming conventions of the API's classes, methods and patterns should be considered as a vital part of the API's successful acceptance by the users. Similarly, this reinforces also the opinion that the naming given to both sources and files used in the API need to be semantically expressive.

Additionally, the problems regarding the API's interpretation were focused mostly on declarations and instructions. There were many events that showed evidences of difficulties understanding the different types of contexts, or difficulties understanding some mechanics regarding the API's methods (i.e., understanding that, by being called, some methods stopped the source's reproduction). The relative high number of problematic events, shed evidence on a difficulty on understanding the API's mechanics (as evidenced on the first table). However, the larger amount of problems is centered on one specific method, which may be understood as a method-specific problem. Nevertheless, we should not minimize this issue, and recognize that the API may need some improvements on both its documentation, as well as on its structure. The major problem is how to clearly explain information on both concepts that may be new to the users (i.e., patterns), as well as concepts that are interpreted in a new way (i.e., context). Probably, the easiest way to explain the API's theoretical foundations is by using it in a real project. On the other hand, it is hard to convince someone to try a solution if he does not understand the advantages that he obtains by doing it. Moreover, regarding the API's structure, many events shed light on certain unfamiliarity with the concept of defining all the source's properties in the constructor. There was a certain resistance to being constantly "forced" to look at the constructor to understand the properties of a sound source. However, it is hard to guarantee whether this is due to a problem of the solution itself, or simply because the aforementioned unfamiliarity with the mechanic. Either way, the important fact to retain is the resistance to the mechanic that we thought it would be highly intuitive. On the other hand, it could be argued that the reason why testers had the necessity to continuously look at the declarations was because they were not constructed by them. Still, without a test scenario that allows them to make the integral declarations of objects, it is hard to be certain of which case is true.

Lastly, in order to have a better understanding of which were the API's concepts that were related to the observed events, we made a third table which analyzed this correlation (event-classes, when applied), as well as the event's classification as *positive* or *negative*. This data is presented in Table 4.7.

	API related concepts									
	Sources		**Contexts**		**Patterns**		**Listener**			***Total***
	Positive	Negative	Positive	Negative	Positive	Negative	Positive	Negative		
DogInt	3	3	0	1	0	0	0	0		7
KeyInt	4	0	4	2	1	0	0	1		12
MotherInt	6	8	4	2	1	2	0	0		23
BabyInt	6	3	6	5	1	1	0	1		23
UpdateInput	0	0	0	0	0	0	1	0		1
Total	19	14	14	10	3	3	1	2		66

Table 4.7 - API Concepts Analysis

Analyzing the data presented on the table, the number of negative events regarding sources, on the MotherInt method, reinforces the idea that there was a special difficulty regarding this method (there were also evidences of this fact in the analysis of Table 4.6). Also, although there are more positive events than negative ones, the difference between positive and negative is not expressive enough to allow us to draw any type of conclusions.

As expected, the classes Source and Context have the larger number of negative events. This is comprehensible because there was a larger number of instructions that used sources and contexts, than patterns and listeners. Still, there was also a significant difference in quantity, between instructions and declarations that used sources, and instructions and declarations that used contexts. Therefore, it could be expected a relative significant difference between negative events of each of them. However, the number of negative events regarding sources, and the number of negative events regarding contexts do not present a significant difference. This similarity in the aforementioned values can be interpreted as evidence of a problematic understanding of contexts, and the concepts supporting it.

In fact, the use of contexts requires an understanding of many concepts that may be new for users: *Layers*, *Agents*, and *Exclusivity* (besides sources). This abundance of new knowledge for users to understand may have been too much for the participants to grasp, especially due to the reduced amount of time using the API. Moreover, according to the testers' commentaries during the experiment, the naming chosen for these concepts may not have been the best. Besides *Layer,* which seems appropriate for the concept it describes (though not being natural for the common programmer, requiring explanation), the name "*Agent*" and "*Exclusivity*". When contacting for the first few times with the API, participants commonly had doubts about what was an agent, and did not know if its use should be related with any character from the game. Furthermore, being a concept that is quite abstract even after explanation (what is an agent?), we now recognize there is evidence that another alternative could make the concept more understandable.

Similarly, the concept of "*Exclusivity*" is not one that becomes instantly understood by its name. The term exclusivity may be even more abstract than agent. Therefore, the events observed during the experiment show a tendency for a high-level of difficulty to understand it. There was observable a tendency for participants to always ask what was its meaning, what can be understood as a need for the concept to be memorized, instead of being intuitive as was expected. Once again, it appears to emerge an opportunity to think in some alternative that would ease the perception of the desired concept and functionality.

5 Conclusions and Future Work

During the whole project, many decisions and corrections were made. Every day, the problem definition, the proposed solution, and many other aspects were constantly being refined. Time constraints, technological constraints, domain-related constraints, there was a panoply of factors that were always compelling us to make important decisions. Additionally, the tests that were performed also gave us important data that allowed us to make important considerations regarding the work that was developed, and suggesting possible aspects which could be important to modify.

Therefore, in this chapter we present improvements and design errors corrected during the development of the project. Furthermore, we also detail the limitations of the proposed approach, and propose additional features (and usages) to be added to the solution, in the future. Following this, we present the written work that was created during this project's lifetime, as well as some reflections about the contributions of the work and its learning goals. Finally, we present some Final Remarks on the whole project.

5.1 Improvements and design errors corrected

Throughout the work developed in this dissertation, various improvements and corrections were made, mainly due to the nature of the chosen methodology (see Methodology), which encourages a constant refinement of the problem definition and its requirements. In this section we will detail some of the improvements and corrections made throughout the project.

One of the first doubts we had consisted in how to represent the currently playing soundscape. While at first we thought about using only one structure (*Score*), we soon realized that there were many different states and types of sources that had to be grouped separately. Along with this, the introduction of the concepts "in context", and "out of context", there more variable to take into account while deciding how to store and group sources. Conjointly, we first thought that sources should be grouped by their context. However, the importance that the sources' layers have on heuristics, and the perspective that, in the future, more than one context can be active simultaneously, made us rethink and decide to group sources in the scores by their layer.

As referred before, Alves's work on Patterns for Sound Design [Alves and Roque 2011] [Alves 2011] was one of the most important inspirations for our approach. Nonetheless, the inspiration and guidance that it provided made us be too focused on the concept of pattern, thinking always about how to build a system around it. As we were researching and attempting to draw possible approaches to the solution, it became clearer that there were other concepts that should be prioritized, namely source and context. This allowed us to reach a solution that was not too attached to the need of thinking of sound implementation based on the notion of pattern. Also, we soon find that there were many patterns referred by Alves that would not fit the type of usage that we aimed for the API. We had to focus on which patterns could be easily translated into an algorithmic representation, and that would not be ambiguous in terms of perception about the consequences of their usage.

In the beginning, we looked at these patterns as if we were looking at "how" to do something, how to use sound explorations. Yet, at the same time that our understanding of Alves's patterns was maturing, we realized that we were in reality searching for a way to represent "what" we wanted. With this mindset, we understood that "what" can be related to the designer's intentions (directives), and that the "how" is invisible to the engine's user (it is the knowledge embodied in the engine's heuristic container).

Initially, we thought that the game scenario in which the DSC module would be tested should be new, designer exclusively for that purpose. Nevertheless, the plan was changed for numerous reasons. Firstly, the time to prototype a new scenario, instead of re-using a game already programmed, would be very large. Secondly, it was more easily for us to envision possible situations to test the engine if we had already a mental structure of the game. Finally, instead of a prototype made specific for one or two situations, a game like Blindfold allowed many different game situations to be explored. Also, at first we thought that the game scenario should be a closed 2D space, due to the abundance of sounds in a small space that would allow the management of sources to be perceptible. Furthermore, we thought that the use of 3D sound would compose an extra layer of complexity that would not bring any value to our proposal. As soon as implementation began, we realized that the usage of 2D or 3D sound would not bring any difference, so, that option was another reason to allow us to use Blindfold.

Another aspect that was changed during the project was what we considered to be the importance of a visual interface to tweak parameters and allow an easier prototyping. While the value of this feature is not neglected, the time that would be required to implement it would be large, and between this feature or the possibility of spending more time adding new heuristics to the engine, we decided to go with the latter. Additional information about this feature can be found in Limitations and additional features/usage for the project. In addition, other feature that had to be postponed was the possibility of letting the composition process influence the game's logic, or AI. This would also require much more time to spend solely on this issue than that available on a Master's dissertation. Still, the work developed on this project can be seen as a baby step in the direction of a sound engine capable of unmuting game logic.

Finally, there were two theoretical issues that were changed and that had a major impact on the project's outcome. First, the target audience of the engine started to be sound designers and similar people, which had no programming skills. Nevertheless, independently of how much we tried to simplify the API and make it understandable, we soon realized that some programming skills would be inevitable to use it. Correspondingly, programming knowledge would be needed for the users to read the code from the game, in order to understand where and how to insert the lines of code they desired. This made us broad the spectrum of possible users of this engine. Programmers, game designers, sound designers, all of them can see some utility on the engine, especially in low-budget indie studios. The second theoretical issue that had great impact on the project was the concept of context. Being a concept used in many different areas in a variety of ways, we had long debates regarding this matter. Initially, it appeared that context should be a super complete structure of information that should incorporate many different sources of information regarding the soundscape. Unfortunately, as more information was being clustered to this notion of context that was being projected, the more difficult it was being to conceive a way of reason and make decisions using all the information that was being considered. It was necessary to take a step back in order to move forward. In other words, we had to focus on what types of situations could really earn value with this concept, and what was the information that was really important. After meditating on this matter, the most important usage that we envisioned was the capability of selecting, from all the sources available on a soundscape, the ones that were truly important in a specific moment. From this idea, we always worked towards a notion of context that would allow this selection/focus of the important sources in an easy fashion. This was one of the stepping stones of the project, and one of the features that we consider to be more valuable for sound design in videogame contexts.

5.2 Limitations and additional features/usage for the project

The main task that appears as the most vital to be done on top of this dissertation's work is the further testing and evaluation of all the implemented heuristics. However, in order for the results to be truly enlightening, some requirements should be met while doing the evaluation experiments. It appears to be mandatory that this task should resort to domain experts which can evaluate and give insights on the engine's performance. Unlike computer science researchers who usually do not have any type of sound design background, the expertise that domain experts could bring to the studies would enrich the evaluation process infinitely. Still, an evaluation with these characteristics would surely not be a "one-time" experiment. On the contrary, it would be an iterative process that would require a cyclic process of "testing-evaluation of results-refinement". In the end, this process would allow the improvement of the engine's robustness.

Similarly, another way of increasing the engine's versatility would be to increase the number of heuristics. The immensity of sound explorations which can be used nowadays on videogames creates the need for a constant evolution of the patterns offered by the engine. The greater the number of heuristics at the designers' disposal, the more freedom they have to let their creativity blossom. Additionally, the increase in the number of heuristics available would also translate into more behaviors being coded only through pattern definition. This would make the sound-related code more easily understandable, and prevent the need for to code certain behaviors, which would be automatically performed by the DSC module.

Nevertheless, heuristics are not the only important variable to determine the versatility and robustness of the engine. It would be tremendously valuable to allow the engine to use more than one context and pattern simultaneously. However, research has to be done in order to determine the best way to support more than one context and combine their properties. Equivalently, it would be good to research about the different possibilities to support more than one pattern (what already is supported, but always giving priority to the more recent). As an alternative, one possibility that could be valuable to test in the future is the implementation of a stack system, similar to the one it was implemented for the heuristics, but, in this specific case, applied to contexts. This way, when the programmer popped a context, the next one in the stack would be assumed as the current context.

Sometimes, heuristics have default values associated with them, which influence their output. It would be useful for designers to have a graphical interface that would allow them to tweak this type of values in real-time while testing their prototypes. For instance, the attenuation of the volume levels, or the number of seconds that a specific heuristic should perform, could be more easily tuned this way, in real-time, during prototyping. This feature could even be improved by incorporating a graphical module to show some statistic data regarding the soundscape (i.e., number of sources played, number of patterns, time spent by each context, etc.).

Also, as mentioned before, in this first prototype developed on this dissertation project, we implemented a unidirectional channel of communication to allow the game logic to send messages to the sound engine. It would be potentially useful to have bidirectional communication channel. This would allow other kinds of synchronization between audio and game events, such as making the soundscape composition process influence game logic and its AI module.

In order to maintain "conscience" of what happened before in the soundscape, it would be very useful to have a logging system. A system like this would allow keeping record of every source, pattern and context which occurred in the soundscape, to allow future consultation of that information. This information would allow the scheduler to have in account not only the soundscape's current state, but also the events that occurred in the past.

One of the main features that could be further researched and could bring benefits for the engine would be a new data structure, through which time-related issues like the duration of sources could be more easily stored accessed. In other words, we are talking about a data structure which would allow data to be stored in a timeline-fashion, which would allow better control over both scheduling and planning processes. Furthermore, this is a key aspect in order to allow the engine to schedule events for a specific moment in the future. However, this is not an easy task, as the duration of gameplay is uncertain, which can cause some issues in terms of memory availability after long periods of scheduling. This is certainly an extremely important issue that requires a great amount of work. However, it is also one of the features that could truly improve the composition process.

We are also particularly interested in studying further explorations of this tool for fast prototyping. The system's architecture allows the game code and the audio engine to run independently, communicating through network, which means that they can even run in separate machines. This, for instance, prevents the need to reboot the game every time the sound engine is modified, which provides an expedite way to prototype. For example, the aforementioned idea of creating a graphical interface to tweak values related to the system's heuristics would greatly benefit from this separate run-time prototyping capability.

5.3 Written Work

Since the beginning of this dissertation's work, we had the wish to develop at least one scientific paper related to it. Still, it was not until the beginning of the second semester that we had the possibility to define more clearly what we would be doing in the following months, and, consequently, what could possibly be useful to report in a paper.

Also, we analysed which were the conferences in which the work to be reported would fit better. Clearly, Audio Mostly emerged as the most interesting option, because of both its focus on audio, as well as the deadline for the submission of papers (beginning of May).

Additionally, due to the conference's focus on audio, we decided to make an ambitious decision: do not one, but two papers for the conference. Being Blindfold an audio-only game, it appeared to be also a good fit for the conference. Furthermore, as it was used as game scenario in the development of the DSCM, we thought it would be interesting to make a paper about it. The process would always be interesting in the optic of this dissertation, as it would allow to describe in detail the design process of an audio-only game, as well as the analysis of data regarding previously performed playtesting.

In terms of acceptance, the first paper (Dynamic Enhancement of Videogame Soundscapes) was accepted as a poster paper. On the other hand, the second paper (The Blindfold soundscape game: a case for participation centered gameplay experience design and evaluation) was accepted as a full paper. In the following subchapters we will detail the content of both papers. Additionally, it was requested that we make an interactive demo session to show Blindfold using the DSCM proposed in this dissertation.

Both papers can be consulted respectively in Appendix D and E.

5.3.1 Dynamic Enhancement of Videogame Soundscapes

In this paper, we started by doing a contextual introduction to the game audio domain, referring some of its main issues and challenges (i.e., the medium's intrinsic dynamic nature). It is also summarized the most important concepts behind Acoustic Ecology theory, namely the concept of Healthy Soundscape. Also, we made an overview of the sound computing architecture that, nowadays, support the implementation of sound in videogames.

After contextualizing the reader with the most important concepts which supported the work to be presented, we propose a system aiming at the enhancement of the soundscape generated during gameplay (DSCM), which we set to follow principles from Acoustic Ecology. The paper gives an overview of the system's overall architecture, focusing on explaining the workflow of the proposed solution. It is also given special attention to the explanation of the heuristics' important role in the system. The paper also presents the interface that designers will have at their disposal to inform the heuristics, and characterize the sounds being handled by the sound engine (API), detailing its guidelines and intentions behind its design.

Finally, we present in the paper reflections on an essay where a game was remade using the proposed system, focusing mainly on the one heuristic used in the remake. From the information regarding that essay, we offer some reflections on the conclusions drawn, as well as listing possible features and improvements to be performed in the future.

5.3.2 The Blindfold soundscape game: a case for participation centered gameplay experience design and evaluation

In this paper, we started by doing a contextual introduction to the game audio domain, focusing on audio-only games, as well as research regarding the impact of audio on different aspects of gameplay. Next, we present a model for participation-centric game experience design and evaluation, as well as the methodology used in the exercise that was going to be presented.

The paper reported on a game design exercise that focused on the sensoriality and sensemaking participant dimensions for conceiving and evaluating gameplay experience, by framing design intentions, artifact characteristics and user participation. After analyzing the design case of an audio-only game developed with the help of the aforementioned model (Blindfold), we present the data obtained through an exercise of playtesting performed to 19 participants. Through this exercise we were able to build understandings of user participation in the soundscape constituting the gameplay scenario.

By employing a goalquestion-metric approach we demonstrated the viability of using the participation-centric gameplay model dimensions as a basis for the synthesis of gameplay participation indicators and metrics, and their analysis in the context of interactions with a game as soundscape.

5.4 Lessons Learned

This sub-chapter synthesizes over the lessons learned during the development of the dissertation. We start by referring what we think are the main contributions this dissertations brings to the field of dynamic soundscape composition. Furthermore, we also take some conclusions regarding the expected learning goals for this internship, and whether they were fulfilled or not. To conclude, we make a final statement regarding both the area of work, and the project developed.

5.4.1 Reflections and Contributions

We started this dissertation by characterising the importance of sound design in games. We defend a holistic approach to sound design, which calls for an appreciation of the overall soundscape as part of the manifestation of relationships between entities in the game world, including its inhabitants and the environment, but also extending to player. We funded our arguments on the communication model presented by Acoustic Ecology.

We also argued that designing a soundscape for a game constitutes a great challenge due to the dynamicity of this kind of product. In fact, interaction during gameplay can lead to the activation of events and objects in unpredicted ways or it may be impractical to cover all predictable possibilities, in ways that the triggered acoustic associations still constitute an interesting soundscape. We resorted to the concept of healthy soundscape to emphasize the difference between a composition that retains its communicational meaningfulness from a mere superimposition of whatever sounds may became active at a given moment.

We also reviewed the main architectures currently available to implement sound in games. The review put into perspective several relevant aspects, including the perception that the adoption of a middleware can encompass the means to set healthy soundscapes in games. However, middleware solutions are typically too expensive to constitute a viable solution, particularly to smaller developers operating on low budgets. Middleware sophistication can also become overwhelming and impose a hard learning curve that may not fit the goals and constraints of smaller projects. We believe that both the community of practice and the public would benefit from the empowerment of indie developers to create rich soundscapes.

Being so, we conjectured that it might be possible to conceive a system that would support healthy soundscapes in games without having to resort to a middleware as a means to reach that kind of control. We also favored solutions that would avoid achieving that goal by tweaking or embedding sound behavior related code into the game logic.

We proposed a solution that reduces the need to foretell and code the multitude of possibilities that may emerge from the dynamic nature of gameplay events, in terms of ensuring that the sound stimuli being consequently triggered actually constitute a healthy soundscape. Basically, we allow the sanitization of the soundscape to be moved from the game logic to a module that evaluates active sounds in runtime and uses contextual information to decide on how to let them actually get to the rendering system – and, consequently, to the player's ears.

The intelligence in such module is expressed through heuristics that, in turn, translate principles and concepts from a body of knowledge, including from the field of Acoustic Ecology. The module operates as a sort of "filter" on what would be heard otherwise – although we have been refraining from characterizing it as such, to avoid overshadowing other interventions on the soundscape that go beyond filtering in the strict sense.

In order to allow designers to take advantage of the heuristics, we propose an API for characterizing objects and contexts. This API was also written with the intent of easing the expression of design goals, through a sensible choice of naming and provided methods. This should be instrumental not only for code maintenance but also for sharing and discussing it with other practitioners.

Our proposal is not intended to constitute an alternative to the already established approaches but rather a complement to those approaches. This also reflects on the kind of evaluation we have been performing, in the scope of the adopted DSR methodology. We have been less oriented towards comparisons with other solutions, and more focused on the verification of how the adoption of the proposal may be effective and promising, including in terms of new types of opportunities that it may unveil.

We conducted an essay consisting of redesigning an audio game we had formerly developed, this time adopting our own system. The exercise confirmed the feasibility of the proposal, and also helped to inform and refine the development of the implemented heuristics. It also shown that, in this instance, we were able to achieve the desired acoustic behavior, with just a small amount of calls to our sound engine, through the API, from the game logic. While remaking Blindfold, we could perceive the tremendous impact that sound-related code can have in a game's logic. Also, as the project at hand gets bigger and sound plays an important role in it, it become harder to realize sound's functional role just by looking at the code – even for the authors. Consequently, it also gets harder to maintain the code in ways that the two components (sound-related and non-sound-related) are developed in an independent fashion, in order to avoid errors due to unwanted interactions.

Nevertheless, it appears to be clear that a domain of expertise like this, requires various and different scenario to be used for testing, in order to cover all the features and possibilities that a vast number of patterns can offer. This does not mean that each heuristic should be tested on its own scenario. On the contrary, each heuristic should be tested in various scenarios so that their performance can be analyzed in different settings, in order to perceive which constraints affect each heuristic.

Lastly, the two academic papers developed during this dissertation are also a contribution to the domain. Besides contributing with a design case and evaluation of an audio-only game performed with the aid of Pereira's model [Pereira and Roque 2012] , we contribute with an approach to a participation-centric analysis of gameplay. Moreover, the second paper can also be seen as a contribution to the dynamic soundscape composition field, with a first approach to the architecture detailed in this dissertation.

5.4.2 Conclusions related to learning goals of this internship

The project developed under this dissertation was highly invaluable for me to be
able to grow as student, researcher, and engineer. When I reached professor Licínio Roque to propose to work under his guidance in my Master's dissertation, after working together with him in the EDJ course, I knew that it would be a great opportunity to learn more about one of my greatest passions in life: vidoegames. Specifically, audio in videogames has always fascinated me, specially because music and sound design are other areas of expertise that interests me.

Therefore, the main learning goal of this internship was to grasp a better understanding of which are the current procedures on sound design for videogames nowadays. After the deep research that was made to produce the State of the Art report, and after all the prototyping activities developed, I honestly feel that this goal was achieved. I now understand the different roles that exist regarding game audio, and the responsibilities of each of them. Specifically, I now understand the workflow between the two major roles: the sound designer and the audio programmer. After acquiring this knowledge, I now understand which skills I should develop in order to capacitate myself to work in this area someday. Additionally, I developed my knowledge in the area of dynamic soundscape composition, which still has a long path of research ahead of it. Still, I feel that I acquired many theoretical foundations that are going to be important in this research field, specially regarding Acoustic Ecology. More important that to give a definitive solution, the learning goal was to fully understand the problem, and that was fully achieved.

In addition, I also had some personal goals for this dissertation. Knowing that most of the academic projects involve groups, I wanted to test and to improve my autonomy as researcher and engineer. I wanted to push my limits and boundaries, and tackle by myself, whenever possible, every adveristy that I could find in my way. Although this dissertation has made me evolve immensly in this matter, I still have to thank professor Licínio for guiding me when I needed, and for always being a lighthouse which helped me to navigate in a sea of doubts. Moreover, professor Licínio always promoted the development of my critical sense, encouraging me to evaluate over my own point of view at every moment. This made me become more humble, loosing any kind of problem admiting I may be mistaken, and always being rigorous while analysing both my work, and the work of others.

Lastly, there was another learning goal that was achieved: to obtain a better understanding of how academic research is performed. Although having already performed some research work on other courses, this was the most profound and serious approach to research during my academic journey. Specifically, the learning and usage of Design Science Research, allowed me to look at research work with a new perspective, giving more credit to it. Also, it made me understand better how it pushes boundaries and break established concepts, allowing researchers to remain open-minded and highly motivated. Research work trully builds on the past, to bring the future to the present.

5.5 Final Remarks

Sound design continues to grow in terms of importance during the game development phase. As videogames keep evolving, new approaches to sound continue to be explored. What started as a exclusive procedural process, have grown into a methodical and professional task, which now witness some attempts to recover the positive aspects of procedural approaches. The dynamicity of the medium will continue to be a hazardous challenge to overcome, but with research, experiments, and the creativity of the growing indie community, the future has certainly rich and original sonic experiences to be unfolded. Though the work developed in this dissertation cannot allow for conclusive conclusions about if the proposed approach is the best for the problem of dynamic soundscape composition, we think that it has laid foundations for the solving of the issues raised during this work, thus placing us ever nearer to the desired answers.

References

A GAME DEVELOPMENT BLOG 2008. The Beat Goes on: Dynamic Music in Spore A Digital Dreamer, Retrieved October 4, 2012 from http://www.adigitaldreamer.com/game-development/the-beat-goes-on-dynamic-music-in-spore.

ACOUSTIC ECOLOGY W.F. 2000. Soundscape: The Journal of Acoustic Ecology, An introduction to Acoustic Ecology. *Vol.I, Number I*.

HEVNER ALAN R., MARCH SALVATORE T., PARK JINSOO AND RAM SUDHA 2004. Design science in information systems research. *MIS Q.* 28, 1, 75-105.

ALVES, V. 2011. Sound Design in Games Wiki Retrieved October 28, 2012 from http://www.soundingames.com/.

ALVES, V. AND ROQUE, L. 2011. A Deck for Sound Design in Games - Enhancements based on a Design Exercise, In ACE 2011, ACM.

AMBIERA irrKlang Homepage Ambiera, Retrieved October 10, 2012 from http://www.ambiera.com/irrklang/.

AUDIERE Audiere Homepage Sourceforge, Retrieved October 10, 2012 from http://audiere.sourceforge.net/.

AUDIOKINETIC Wwise Homepage AudioKinetic, Retrieved October 10, 2012 from http://www.audiokinetic.com/en/products/208-wwise.

AUDIOMULCH AudioMulch Homepage AudioMulch, Retrieved October 10, 2012 from http://www.audiomulch.com/about-us.

BAJAKIAN, C. 2004. The future of game audio production O'Reilly, O'Reilly Mac OS X Conference 2004, Santa Clara Ballroom (2004).

BRANDON, A. 2007. Audio Middleware Mix: Professional Audio and Music Production, Retrieved October 21, 2012 from http://www.mixonline.com/basics/education/audio_audio_middleware/.

BRANDON, A. 2007. Audio Middleware, part 2 Mix: Professional Audio and Music Production, Retrieved October 21, 2012 from http://mixonline.com/recording/mixing/audio_audio_middleware_part/.

BRANDON, A. 2007. Audio Middleware, part 3 Mix: Professional Audio and Music Production, Retrieved October 21, 2012 from http://mixonline.com/basics/education/audio_audio_middleware_part_2/.

BRIDGETT, R. The Game Audio Mixing Revolution Gamasutra, Retrieved November 2, 2012 from http://www.gamasutra.com/view/feature/132446/the_game_audio_mixing_revolution.php?print=1.

BRIDGETT, R. 2009. The Future of Game Audio - Is Interactive Mixing the Key? Gamasutra, Retrieved November 2, 2012 from http://www.gamasutra.com/view/feature/132416/the_future_of_game_audio__is_.php.

BRIDGETT, R. 2009. The Game Audio Mixing Revolution Gamasutra, Retrieved November 2, 2012 from http://www.gamasutra.com/view/feature/132446/the_game_audio_mixing_revolution.php?print=1.

CASTRO, R. 2009. Soundwalkers Vimeo, Retrieved December 13, 2012 from https://vimeo.com/1737899.

CAVERS, J. 2011. Into Sound - Rewind: AES Audio for Games Conference 2011 Tumblr, http://joecavers.tumblr.com/post/4319520354/rewind-aes-audio-for-games-conference-2011.

CHAN, S.-H., NATKIN, S., TIGER, G. AND TOPOL, A. 2012. Extensible Sound Description in COLLADA: A Unique File for a Rich Sound Design. In *Proceedings of the Advances in Computer Entertainment* 2012 Springer.

CLAM CLAM Homepage CLAM, Retrieved October 10, 2012 from http://clam-project.org/.

COLLINS, K. 2008. *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. Mit Press.

CRYTEK Cry Engine Homepage Crytek, Retrieved October 15, 2012 from http://www.crytek.com/cryengine.

CYCLING 74 Max Homepage Cycling 74, Retrieved October 11, 2012 from http://cycling74.com/products/max/.

DODDS, T. 2008. Playful Audition: the "Everyday" Experience of Sound in Video Games. Bachelor Thesis, University of Technology, Sydney, Australia.

EIGENFELDT, A. AND PASQUIER, P. 2011. Negotiated Content: Generative Soundscape Composition by Autonomous Musical Agents in "Coming Together: Freesound". In *Second International Conference on Computational Creativity* Universidad Autónoma Metropolitana, Unidad Cuajimalpa, México City, México.

ENTERTAINMENT, R. 2006. Scarface: The World Is Yours. Videogame, Vivendi Universal Games.

EPIC GAMES Unreal Engine Homepage Epic Games, Retrieved October 15, 2012 from http://www.unrealengine.com.

FARNELL, A. 2007. Synthetic game audio with PureData AudioMostly, Retrieved November 18, 2012 from http://obiwannabe.co.uk/html/papers/audiomostly/AudioMostly2007-FARNELL.pdf.

FARNELL, A. 2010. *Designing Sound*. Mit Press.

FIRELIGHT Fmod Homepage Firelight, Retrieved October 10, 2012 from http://www.fmod.org/.

FORNARI, J., MAIA JR, A. AND MANZOLLI, J. 2008. Soundscape design through evolutionary engines. *Journal of the Brazilian Computer Society 14*, 51-64.

FOURNELL, N. 2010. What is Procedural Audio? Game Developers Conference, Retrieved November 19, 2012 from http://www.gdcvault.com/play/1012704/Procedural-Audio-for-Video-Games.

GAMES, L. 1984. Ballblazer, Videogame, Atari.

GAUTHIER, P. Java external plug-in for pureData le-son666, Retrieved December 6, 2012 from http://www.le-son666.com/software/pdj/.

GRAY, K., GABLER, K., SHODHAN, S. AND KUCIC, M. 2005. How to prototype a game in under 7 days Gamasutra, Retrieved December 22, 2012 from http://www.gamasutra.com/view/feature/2438/how_to_prototype_a_game_in_under_7_.php.

KASTBAUER, D. 2010. Audio Implementation Greats #2: Audio Toolsets [Part 2] Designing Sound, Retrieved October 29, 2012 from http://designingsound.org/2010/01/audio-implementation-greats-2-audio-toolsets-part-2/.

KATZ, M. 2010. *Capturing Sound: How Technology Has Changed Music*. University of California Press.

KERR, A. 2006. *The Business and Culture of Digital Games: Gamework and Gameplay*. SAGE Publications.

KNIGHT, H. 2011. Procedural Audio and Binauralisation Using Max/MSP and the Unity3D Game Engine Cycling 74, Retrieved November 18, 2012 from http://cycling74.com/project/procedural-audio-and-binauralisation-using-maxmsp-and-the-unity3d-game-engine/.

LASTRA, J. 2012. *Sound Technology and the American Cinema: Perception, Representation, Modernity*. Columbia University Press.

LIBPD libpd Homepage libpd, Retrieved October 10, 2012 from http://libpd.cc/.

LOW, G.S. 2001. Understanding Realism in Computer Games through Phenomenology Stanford Computer Science, Retrieved December 14, 2012 from http://xenon.stanford.edu/~geksiong/papers/cs378/cs378paper.htm.

LYKKE, M. 2008. Procedural Audio in Computer Games Master Thesis, University of Aarhus, Denmark.

MACANULTY, I. AND DURITY, G. Contextually Driver Dynamic Music System for Games Vimeo, Retrieved December 12, 2012 from https://vimeo.com/16034304.

MANIERO, T. Hekkus Sound System Homepage shlzero, Retrieved October 14, 2012 from http://www.shlzero.com/.

MARMALADE Marmelade Homepage Marmalade, Retrieved October 11, 2012 from http://www.madewithmarmalade.com/.

MENDEZ, S. 2005. Music in the Air: Exclusive Interview with Michael Land The Dig Museum, Retrieved November 19 , 2012 from http://dig.mixnmojo.com/museum/interview_land.html.

MILLER, M. 1999. 3D Audio Gamasutra, Retrieved November 1, 2012 from http://www.gamasutra.com/features/19991102.

MOLECULE, M. 2008. Little Big Planet, Videogame, SCEE.

MURCH, W. 2005. Dense Clarity - Clear Density. In *The Transom Review*, TRANSOM Ed. Transom, Retrieved April 5, 2013 from http://transom.org/?page_id=7006.

NAIR, V. 2012. Procedural Audio: Interview with Andy Farnell Designing Sound, Retrieved November 16, 2012 from http://designingsound.org/2012/01/procedural-audio-interview-with-andy-farnell/.

OPEN SOUND CONTROL Introduction to OSC Open Sound Control, Retrieved November 6, 2012 from http://opensoundcontrol.org/introduction-osc.

PAUL, L.J. 2007. Video Game Audio Prototyping with Pure Data videogameaudio, Retrieved October 25, 2012 from http://www.videogameaudio.com/IDIG-Sep2006/GameAudioProtoypingWithPureData-LPaul-2007.pdf.

PAUL, L.J. 2008. Video Game Audio Prototyping with Half-Life 2 Springer Berlin Heidelberg, Retrieved October 25, 2012 from http://dx.doi.org/10.1007/978-3-540-79486-8_17.

PAUL, L.J. 2010. Video Game Audio Prototyping with Half Life 2 Vimeo, Retrieved October 25, 2012 from https://vimeo.com/7122167.

PECK, N. 2001. Beyond the library: Applying film postproduction techniques to game sound design, Game Developers Conference 2001, 20-24.

PEERDEMAN, P. 2010. Sound and Music in Games, Written Work, Vrije Universiteit, Amsterdam.

PEREIRA, L. AND ROQUE, L. 2012. Towards a game experience design model centered on participation, In CHI 12 Extended Abstracts on Human Factors in Computing Systems, ACM, NY, USA, 2327-2332.

PERISCOPE STUDIO Psai Homepage Periscope Studio, Retrieved October 20, 2012 from http://www.homeofpsai.com/.

PIJANOWSKI, B.C., FARINA, A., GAGE, S.H., DUMYAHN, S.L. AND KRAUSE, B.L. 2011. What is soundscape ecology? An introduction and overview of an emerging new science. Landscape Ecology, 26(9):1213–1232.

PORTAUDIO PortAudio Homepage PortAudio, Retrieved October 20, 2012 from http://www.portaudio.com/.

PUCKETTE, M. PureData homepage PureData, Retrieved October 22, 2012 from http://puredata.info/.

R&D, N. AND SYSTEMS, I. 1986. Metroid, Videogame, Nintendo.

RAD Miles Sound System Homepage RAD, Retrieved October 22, 2012 from http://www.radgametools.com/miles.htm.

RAW MATERIAL SOFTWARE JUCE Homepage Raw Material Software, Retrieved October 22, 2012 from http://rawmaterialsoftware.com/juce.php.

RUTHERFORD, S. 2012. Procedural Methods for Audio Generation in Interactive Games Stefan Rutherford, Retrieved December 10, 2012 from http://stefanrutherford.com/Procedural_Methods_for_Audio_in_Interactive_Games.pdf.

SCHAFER, R.M. 1993. *The Soundscape: Our Sonic Environment and the Tuning of the World*. Inner Traditions/Bear.

SCHEIRER, E.D., VAANANEN, R. AND HUOPANIEMI, J. 1999. AudioBIFS: Describing audio scenes with the MPEG-4 multimedia standard. In *Multimedia, IEEE Transactions* IEEE, Multimedia, IEEE Transactions.

SDL SDL Homepage SDL, Retrieved October 23, 2012 from http://www.libsdl.org/.

SFML SFML Homepage SFML, Retrieved October 23, 2012 from http://www.sfml-dev.org/.

SOUTHWORTH, M.F. 1967. *The Sonic Environment of Cities*. Massachusetts Institute of Technology.

STEVENS, R. AND RAYBOULD, D. 2011. *The Game Audio Tutorial: A Practical Guide to Sound and Music for Interactive Games*. Focal Press.

STUDIOS, L. 2008. Fable II, Videogame, Microsoft Games.

STUFFMATIC Kowalski Homepage Github, https://github.com/stuffmatic/kowalski#kowalski.

TAITO 1978. Space Invaders , Videogame, Midway.

THE CANADIAN ENCYCLOPEDIA World Soundscape Project The Canadian Encyclopedia, Retrieved December 19, 2012 from http://www.thecanadianencyclopedia.com/articles/emc/world-soundscape-project.

THEBERGE, P. 1997. *Any Sound You Can Imagine: Making Music/Consuming Technology*. Wesleyan University Press.

THEORY, N. 2007. Heavenly Sword, Videogame, SCEE.

TRUAX, B. The World Soundscape Project Simon Fraser University, Retrieved December 19, 2012 from http://www.sfu.ca/~truax/wsp.html.

TRUAX, B. 2001. *Acoustic Communication*. Ablex.

TRUAX, B. 2002. Genres and techniques of soundscape composition as developed at Simon Fraser University. *Org. Sound 7*, no. 1. Cambridge: Cambridge University Press: 5-14.

TRUAX, B. 2008. Soundscape composition as global music: Electroacoustic music as soundscape*. *Org. Sound 13*, no. 2. Cambridge: Cambridge University Press: 103-109.

TRUAX, B. AND BARRET, G.W. 2011. Soundscape in a context of acoustic and landscape ecology. *Landscape Ecology,* 26(9): 1201-1207.

UN4SEEN BASS Homepage Un4seen, Retrieved October 20, 2012 from http://www.un4seen.com/.

WENT, K., HUIBERTS, S. AND VAN TOL, R. 2009. Game Audio Lab - An Architectural Framework for Nonlinear Audio in Games. In *Audio Engineering Society Conference: 35th International Conference: Audio for Games*.

WHITE, I. Encoded-Embodied: Mixing across a cognitive audio spectrum. In *ICAUDIODESIGN* Wordpress, Retrieved April 3, 2013 from http://icaudiodesign.wordpress.com/encoded-embodied-mixing-across-a-cognitive-audio-spectrum/.

WIKIPEDIA OpenAL Wikipedia, Retrieved October 21, 2012 from http://en.wikipedia.org/wiki/OpenAL.

WOLF, M.J.P. AND PERRON, B. 2003. *The Video Game Theory Reader*. Routledge.

WRAUGHK Deep Sea Wraughk, Retrieved November 14, 2012 from http://wraughk.com/deepsea/.

YIANNIS 2012. Talktome: adaptive/dynamic audio prototyping for video games Tumblr, Retrieved November 26, 2012 from http://www.gameaudiomiddleware.tumblr.com/.

YOUNG, D.M. 2012. Adaptive Game Music: The Evolution and Future of Dynamic Music Systems in Video Games. Bachelor Thesis, Ohio University, USA.

A. Tool Comparison Table

	Game-Driven Musical Scores (Interactive)	Monitor and profile audio in real-time or in Sandbox Environment	Define environmental audio propagation and effects	Platforms supported	Access to engine's source code	Parallel development (platform and language)	Sound Playback Behaviours	Hierarchical audio structures	Event Creation system	2D/3D Positioning and Sound Propagation	Occlusion / Obstruction Support	Dynamic Audio Controls (RTPCs, Swtich, states,etc)	Sound Playback Prioritization	DSP Effects	Multi-Listener Support	Free for non profiting-usage	Mixing and output channels	Number of channels	Game Engines with Integration prepared
Videogame Audio Middleware																			
Wwise	Yes	Yes	Yes	PC, Mac, Nintendo 3DS, PlayStation®3, PlayStation® Vita, Wii, Wii U, Xbox 360	Yes? (On contact)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	?	Just announced as easy to integrate
FMOD	Yes	Yes	Yes	Windows, Linux, Mac, PS3, PSP, Vita, 360, Wii, Wii U, iPhone, iPad, Android, Google Native Client	No	Yes	Yes	Yes	Yes	Yes (HRTF)	Yes	Yes	Yes	Yes	Yes	Yes	Yes	16	CryEngine, Unity, UE3, BigWorld, Vision Engine, Scaleform
XACT	Yes, but cumbersome	No	No	PC, Xbox 360	No	Yes (only the two supported)	Yes (limited)	No	No	Yes	No	Yes	Yes	No	?	Yes	No	?	XNA
Psai	Yes	?	No	PC, ?	?	?	Yes	?	Yes	No	No	Yes	?	No	No	No ?	No	XXXX	?
Miles Soundsystem	?	Yes	?	Windows, Linux, Wii, Wii U, 3DS, 360, PSP, Vita, PS2, PS3, Mac, iPhone, Android	Yes	Yes	?	Yes	?	Yes	with effects, yes	?	?	Yes	?	No	Yes	?	?
Unreal 3 Soundsystem	?	Yes	?	Windows, Wii U, Vita, PS3, Flash, 360, Android, iOS, Mac	No (Udk, UE3 has but is paid)	Yes	Yes	?	Yes	Yes	?	?	?	?	?	Yes	Yes	Yes	Unreal Engine
CryEngine Audio	Yes	Yes	Yes	PC, Xbox 360, PS3, ?	No?	Yes	Yes	Yes	Yes	Yes	?	Yes	?	Yes	?	Yes	Yes	?	CryEngine
Audio API, Libs and Frameworks																			
Marmelade Audio	No	No	No	iOS, Android, Blackberry OS, bada, Windows, Mac, LG Smart TV	Yes	Yes	No	No	No	?	No	No	No	?	?	Yes	Yes	24	XXXX
OpenAL	No	No	No	PC, Android ? Mac ?	Yes	?	No	No	No	Yes (one of them, HRTF)	Yes (through EFX extension)	No	No	?	?	Some versions yes..	Yes, with speaker assignment	?	XXXX
SDL	No	No	No	Windows, Be, Mac, Linux, and many many more (but not mobile)	Yes	?	No	No	No	No	No	No	No	?	No	Yes	Yes	Infinite	XXXX
SFML	No	No	No	PC only?	?	No	No	No	No	Yes	No	No	No	?	?	Yes	Yes.	?	XXXX
BASS	No	No	No	Win, iOS, Android, Linux	?	No	No	No	No	Yes	?	No	No	Yes	Not sure, think not	Free for non commercial use	Yes. Speaker assignment	Multi-channel	XXXX
irrKlang	No	No	No	Windows, Linux, Mac	Yes	Yes	No	No	No	Yes	No?	No	No	Yes	?	Yes	?	?	XXXX
Hekkus SoundSystem	No	No	No	Iphone, Android, Bada, Symbian, NaCl, Win32, MacOSX, Linux	With source code	?	No	No	No	No?	No?	No?	No?	No?	No?	Yes	Yes	unlimited	XXXX
JUCE	No	No	No	Windows, Linux, Mac, iOS, Android	Yes	Yes	No	No	No	?	No	No	No	Yes	No?	Yes	Yes	?	XXXX
PortAudio	No	No	No	Windows, Macintosh OS X, and Unix	Yes	Yes?	No	No	No	No?	No	No	No	Yes?	?	Yes	?	?	XXXX
Audiere	No	No	No	Windows, Linux	Yes	?	?	?	?	?	?	?	?	?	?	Yes	?	?	XXXX
Kowalski	No	No	No	Windows, Linux, Mac, iOS	Yes	No	?	Yes	Yes	Yes	Yes	?	Yes?	Yes	?	Yes	Yes	?	XXXX
Graphical Programming & Procedural Tools																			
PureData	Programatically yes	Yes	Programatically yes	Almost any platform available	Not applicable?	Yes (at least the modules, not sure about integration)	Yes	Yes	No	through patches?	No	Yes	?	Yes	?	Yes	Yes	?	XXXX
Max/MSP	Programatically yes	Yes	Programatically yes	Windows, Mac, ?	No	Yes (at least the modules, not sure about integration)	Yes	Yes	No	trhough routines?	No	Yes	?	Yes	?	No	Yes	?	XXXX
AudioMulch	?	?	Yes	Windows, Mac	No	Yes (at least the modules, not sure about integration)	?	?	No	No	No	?	?	Yes	?	No	Yes	?	XXXX
Clam	?	?	Programatically?	Windows, Mac, Linux	?	No	?	Yes	No	HRTF, among others	?	?	?	Yes	?	Yes	Yes	??	XXXX

B. Gantt Diagram

Design Science Research	Tasks	Sub-Tasks	September	October	November	December	January	February	March	April	May	June	July
Awareness of problem	SoA Report		🟨	🟨	🟨								
		Soundscape Theory	🟦										
		Audio Tools		🟦									
		Related Work			🟦								
	Detailed Proposal			🟨	🟨								
		Problem definition		🟦									
		Assumptions and milestones			🟦								
Suggestion	Initial Design				🟨	🟨							
		Goals and requirements			🟦								
		Architectural drivers			🟦								
		Solution Specification				🟦							
	Intermediate Report						🟨						
Development	Prototyping							🟨	🟨	🟨	🟨	🟨	
		Language and Techniques specification						🟦	🟦	🟦	🟦	🟦	
		Audio engine basic functions						🟦					
		Integration with game engine							🟦				
		Techniques implementation							🟦	🟦	🟦	🟦	
		Refinements based on evaluation										🟦	
Evaluation	Evaluation										🟨		
		Evaluation Performance									🟦		
		Results analysis									🟦		
Conclusion	Statement of Learning											🟨	🟨

C. Initial Composition Plan

			Composition plan				
	Pre-Filtering	Pre-Filtering	Ambiance	Foley	SFX	Music	Dialogue
Eventos	Thoughts	**Agente que origina pensamento? Tipologia? (Pensamento que isola outros sons, ou pensamento que combina com música)**	**(Se agente origina é o PC)**Diminuição de volume e/ou LPF **(Senão)** apenas atenuação		**(Se agente origina é o PC)** Diminuição de volume do SFX; LPF; **(Tipologia)**Certos SFX podem estar presos a um certo pensamento, em certos momentos	**(Tipologia)**Volume da musica pode ou não ser diminuído (depende do impacto emocional que desejado)	**(Se agente origina é o PC)**Eco e/ou Reverb nos pensamentos, diminuição de volume e LPF no restante diálogos.
	Sound Effects	**Agentes envolvidos? Características do som? Tipologia? (Ouch!, Achievement, Iminent Death, No Can Do, Failure)**			(**Características do som?)**Verificar frequência para ter certeza que não se tapam? Poderia ser intercalado com outros que estejam a ocorrer? Mudar o Pitch(mas isso pode mudar o seu significado e impacto).	**(Tipologia)**Se SFX for causado por uma acção importante ou com um certo significado, é acompanhada por uma breve música (Contextual Music do Valter? (:) que teria de ser mixada juntamente com a musica a tocar	SFX ocorre mais baixo ou ocultado por um filtro
	Silence	**Que agente/fonte/layer silenciar? Tipologia? (período de silenciamento?)**	O Ambiance pode ou não ser cortado, sendo que muitas vezes é limitado a um mínimo, para acentuar impacto do efeito desejado	Foley não costuma ser tocado, pois ajuda a dar enfâse ao "Silence"	Pode dar enfÂse, mas apenas se o SFX tiver o mesmo papel que o Foley na cena em questão	Em muitas vezes a música não é usada, mas também pode ser a única camada activa se retirarmos todas as outras	Algum efeito talvez? (DSP)?
	Awareness	**Agentes envolvidos?(quem ouve, quem foi ouvido) Características do som ouvido? Tipologia?**	=	=	=	=	**(Tipologia)**Por instantes, tudo pode ser dminuído/ofuscado para que seja perceptível o novo som que entrou no horizonte acústico do jogador; **(Tipologia)**Outras vezes é muito suave só para que o seu aparecimento na Soundscape e desaparecimento possa ocorrer sem causar grande impacto ou quebra na concentração do jogador, sendo que o seu aparecimento normalmente é sinal de um determinado estado, que depois deixa de existir conforme o som desaparece; **(Tipologia)**Pode ser intervalada
	Ambiance	**Características do som? Tipologia?**	**(Tipologia)**É preciso verificar se é apenas um acréscimo de sons ou modificação do ambiance já existente, ou se é um ambiance totalmente novo; e é preciso fazer um fade in/out gradual, que pode ser regulado autónomamente, ou de acordo com movimentação do jogador				**(Tipologia)**Normalmente o diálogo tem primazia sobre tudo, mas o ambiance pode ser usado para enfatizar pedaços de diálogo, como no Patapon
	Dialogue	**Agentes envolvidos? Tipologia?**	**(Tipologia)**É normalmente entregue mais alto que as outras layers, mas existem certos usos como conversas outdoor em andamento que se tornam mais credíveis com uma diferença menor entre diálogo e ambiance (ex: red dead redemption Valter)	Cuidado para não ofuscar diálogo	Cuidado para não ofuscar diálogo	**(Tipologia)**Música, mesmo quando mais baixa, pode dar um toque único ao diálogo e fornecer-lhe alguns pontos estéticos, que sem música ou com outra, seriam totalmente diferentes	
	Foley	**Agente envolvido? Tipologia?**					
	Footsteps	**Agentes envolvidos? Tipologia?(Correr, andar, superfície)**	**(Agente+Tipologia)**Usually more strong than the real life perception	=	=	=	=
	Music	**Características do som? Tipologia? Tipologia?**	**(Características do som? Tipologia?)**Normalmente a música tem prioridade sobre o ambiance, mas se for conseguida uma interligação/metamorfose entre a música e o ambiance, pode ser conseguido um impacto extraordinário. Para tal, normalmente é usado algum jogo de dynamics (volume) para criar uma "dança" entre as duas camadas	**(Tipologia)**Por vezes, esta layer é ignorada para que a música possa envolver em pleno o jogador numa determinada situação	**(Tipologia)**Por vezes, esta layer é ignorada para que a música possa envolver em pleno o jogador numa determinada situação	**(Tipologia)** Transições devem ser o mais suave possíveis	**(Tipologia)** Muitas vezes usadas para enfatizar o sentimento que se quer transmitir com um determinado diálogo ou frase

D. Dynamic Enhancement of Videogame Soundscapes

Available in the annexed file AM13_1.pdf

E. The Blindfold soundscape game: a case for participation centered gameplay experience design and evaluation

Available in the annexed file AM13_2.pdf

F. API Documentation

```
class Context
   {
        //Properties
      public string Name { get; set; }
      public string Type { get; set; }
      public ArrayList Elements { get; set; }
      public bool Exclusive { get; set; }

        //Constructor
      public Context(string name, string type, ArrayList elements, bool exclusive)

        //Methods
      public void initiateContext()

      public void setContext()

      public static void stopContext()
   }
}

// Support Class
class Utils
   {
      public static string doubleArrayToOscString(double[] position)
   }

class Listener
   {
        //Properties
      private double[] position;
      private double[] direction;

        //Constructor
      public Listener(double x, double y, double z)

              public Listener(double[] position)

        //Methods
      public void updateListenerPosition()

      public void updateListenerPosition(double[] position)

      public void updateListenerPosition(double x, double y, double z)

      public void updateListenerDirection(double[] direction)

      public void printListener()
  }

class Pattern
  {
        //Properties
      public string Name { get; set; }
      public string Type { get; set; }
      public bool Active { get; set; }

        //Constructor
      public Pattern(string name, string type)
```

```
            //Methods
        public void InitiatePattern()

        public void PlayPattern()

        public void StopPattern()
    }

class Source
    {
            //Properties
        public string Name { get; set; }
        public string Layer { get; set; }
        public double[] Position { get; set; }
        public string Pattern { get; set; }
        public string Sound { get; set; }
        public bool Playing { get; set; }
        public string Agent { get; set; }
        public bool Loop { get; set; }

            //Constructor
        public Source(string name, string layer, string agent = null, double[] position = null,
string pattern = null, string sound = null, bool loop = true)

            //Methods
        public void changeSourcePosition(double[] position)

        public void changeSourceSound(string sound)

        public void changeSourceLoop(bool loop)
    public void initiateSource()

    public void playSource()

    public void stopSource(bool pause)
}
```