VORPAL:
A Middleware for Real-Time Soundtracks in Digital Games

Wilson Kazuo Mizutani

Thesis Submitted
to the
Institute of Mathematics and Statistics
of the
University of São Paulo
for the
Masters Degree in Computer Science

Program:
Computer Science

Advisor:
Prof. Fabio Kon

This research is supported by
CNPq, Brazil

São Paulo, November, 2016

VORPAL:
A Middleware for Real-Time Soundtracks in Digital Games

This is the original version of the thesis written by the candidate Wilson Kazuo Mizutani, as it was submitted to the Evaluation Commitee.

Acknowledgements

I would like to thank all sound designers and musicians who helped me better understand their work and how my research could help them. Among them, my greatest thanks goes to Dino Vicente, who showed immense interest in my work and tagged along for a whole year, always believing in the potential contributions it could bring to soundtracks in games and in general. If not for him, the game we developed to validate the developed technology would be that much less rich and actually representative of what a real sound designer needs.

I am also grateful for everyone at both the Systems and Computer Music research groups at the Institute of Mathematics and Statistics of the University of São Paulo who helped me in the many obstacles towards my Masters title. Special thanks go to Antonio Goulart, who showed me the works of Farnell; to Pedro Bruel, who introduced me to `libpd`; and to Lucas Dario, who chose to participate in my research for his final course monograph. Also, many thanks to my advisor, Fabio Kon, and to professors Marcelo Queiroz and Alfredo Goldman, for their support and insight.

Abstract

Although soundtracks play an essential role in the experience delivered by digital games, there are a number of design restrictions it suffers from due to technology limitations. This is specially true for real-time effects, a natural demand in the interactive media of games. Developers may either implement their own solutions each time, rely on proprietary software, or neglect the soundtrack altogether. Besides, even the best commercial tools support only sample-based audio, which is one of the main causes for the aforementioned design restrictions. Thus, this thesis proposes *VORPAL*, a free software game audio middleware implementation that focuses on procedural audio instead – while maintaining the possibility of sample-based audio – as a more accessible and adequate tool for composing real-time soundtracks for digital games. The middleware, inspired by its commercial predecessors, is divided in two main pieces of software: an audio engine and a soundtrack creation kit. The audio engine comprises a native `C++` programming library, which games and game engines can be linked to to play and control in real-time soundtrack pieces created using the soundtrack creation kit, which consists of building blocks provided as *Pure Data* abstractions. We have interviewed and partnered with professional sound designers to validate our technology, and came to develop a proof of concept game called *Sound Wanderer*, which showcases the possibilities and limitations of the *VORPAL* middleware.

Keywords: digital games, dynamic audio, adaptive music, real-time soundtrack, audio middleware, game soundtrack.

Resumo

Muito embora trilhas sonoras desempenhem um papel essencial na experiência criada por jogos digitais, elas sofrem de diversas restrições de projeto causadas por limitações tecnológicas. Isso afeta principalmente efeitos em tempo real, que são uma demanda natural na mídia interativa dos jogos. Desenvolvedores precisam optar entre implementar uma solução própria caso a caso, investir em software proprietário, ou simplesmente negligenciar a trilha sonora por falta de opção melhor. Além disso, mesmo as melhores ferramentas comerciais trabalham apenas com áudio baseado em amostras, o que é uma das principais causas das ditas restrições de projeto. Portanto, esta dissertação propõe *VORPAL*, um middleware livre para áudio em jogos que foca em áudio procedural – mas mantém compatibilidade com áudio baseado em amostras – como uma ferramenta mais acessível e adequada para a composição de trilhas sonoras em tempo real para jogos digitais. O middleware, inspirado em seus antecessores comerciais, é dividido em duas principais componentes de software: um motor de áudio e um kit de criação de trilhas sonoras. O primeiro é constituído por uma biblioteca de programação nativa em `C++`, com a qual jogos e motores de jogos podem se ligar para reproduzir e controlar, em tempo real, peças da trilha sonora criadas usando a outro componente, que é um kit de blocos de construção providos como abstrações de *Pure Data*. Projetistas de som profissionais foram entrevistados e depois trabalharam em parceria com os autores para validar a tecnologia proposta, o que levou ao desenvolvimento de um jogo de prova conceitual chamado *Sound Wanderer*, que demonstra as possibilidades e limitações do middleware *VORPAL*.

Palavras-chave: jogos digitais, áudio dinâmico, música adaptativa, trilhas sonoras em tempo real, middleware de áudio, trilhas sonoras para jogos, games.

Contents

Chapter 1

Introduction

Digital games are computer programs fundamentally different from conventional applications. The latter's purpose is to produce a result for the user, like sending data through a network, finding the optimal solution for a mathematical problem or even compiling a program source code into object code. Games, on the other hand, produce no results. Their value lies in the experience the user builds while interacting with the program execution, as defended by Schell [Sch14]. This concept is relevant enough that there is a constant search for even more and more innovative or unexplored ways of intensifying the player's immersion in the narrative universe of the game, as with, for instance, augmented reality [VN14]. Usually, the comprehension of what happens within a game is mainly done through a visual representation, but – like in movies and other audiovisual media – the accompanying sound plays an equally important role in the desired description of the exposed content. Soundtracks can be used, for instance, to emphasize an action scene or to suggest some character's intentions without the need to state them explicitly in the dialogue [Mat14]. These are all effects that stimulate a player's immersion into the gaming activity, contributing to the fulfillment of the experience.

However, the role of soundtracks in games is not the same as in movies, even with the undeniable similarities, mainly because of the interactive nature of games [Men13, KC08]. In other words, since the player is able to continuously influence the game narrative – by jumping, escaping, or simply standing still – it is not possible to know beforehand which events will occur, when they will occur or *if* they will occur. This means that whatever is to be presented – be it visually or audibly – can usually only be determined instants before it actually happens. When this time frame available for a computational response is small enough, we say that it comes in *real-time*. For graphics, it is the pixel matrix displayed that must be rendered in real-time; for the soundtrack, it is the sound wave, which we address in Chapter 3.

In this chapter, we introduce in general terms the problems and research challenges of real-time soundtracks in digital games, as well as the role of our work inside this context. In Section 1.1, we describe how real-time behaviors are introduced in games throughout the industry in order to point out the core issues we wish to tackle, besides giving a brief exposition of related problems, currently available solutions, and inherent challenges of the area. Then, Section 1.2 formally states our objective and how we both approach and evaluate our proposed solution.

1.1 Motivation

The development of digital games is a multidisciplinary process. Programmers, artists, composers, game designers, interface designers, sound designers, scripters, producers, and analysts are only a few commonly seen jobs in a game development team [Gre14, Sch14]. Typically, a project starts with the definition of the core concept of the game (a unique mechanic, an instigating narrative, or even just the continuation of a successful franchise) and how that will make it fun and profitable. These and all subsequent decisions are kept organized in thorough documents or, more commonly, in a single Game Design Document, which is the industry standard for recording and communicating through text and concept arts the current state of the game project to keep the whole team on the same page. During what is known as the *pre-production phase*, the key features of the game are established through iterative testing and prototyping, and only then the *production phase* properly begins, with each professional being assigned his or her responsibilities. Depending on many factors, it might actually take months before the many pieces can be put together into fully "playable" software.

For the composer and sound designer part, a contribution mainly in the form of cautiously mastered sound files is traditionally expected. That, however, requires the precise definition of when and how each file is supposed to be played during the game application execution. Without any sort of specialized tool, any desired real-time behavior for the soundtrack can only be implemented by the programmers, since they are the ones in control of the computational capabilities of the game. Even with good communication among team members, the composers and sound designers will seldom be able to directly manipulate how their own creation emerges from the game.

Let us consider an example. The original *Super Mario Bros.* (*Nintendo*, 1985) – see Figure 1.1 – had a few very simple real-time mechanisms in its soundtrack. Besides the commonly present sound effects – which must be timely triggered whenever the player gets hurt, or grabs an item, or interacts in any way with the game world –, the background song also required real-time manipulation under a few circumstances. One would be when the stage timer went from 400 units down to 100: a warning jingle would immediately interrupt the background song, followed by the same song back again, but in an urgingly faster tempo. The other case was when a super star item had been collected. After the usual sound effect, a very exciting song would substitute the stage theme, rushing the player to make the most out of the temporary invincibility bonus. Both of these event triggers can only be detected from within the game code, since it cannot be known *when* they will occur until the corresponding interactions actually happen during gameplay. The composer or sound designer in this case would only be able to provide the files of the stage and invincibility songs (plus the fast tempo version of the stage song if it is not possible to change this from the code) and describe for the programmer under what conditions were they supposed to be played. Then they would have to wait for the programmer to implement it (possibly restructuring many parts of the software architecture in order to fit the trigger routine in) and only after that would they be able to check whether the result is satisfactory or not.

That would make for a rather easy case, because the sound effect and jingle interventions allow the switching between songs to sound more natural than it would if one song suddenly started

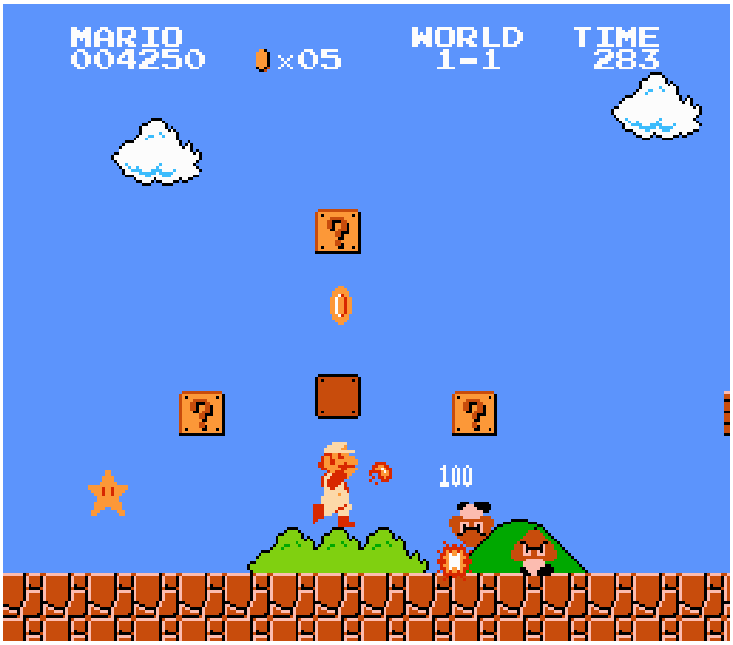


Figure 1.1: An in-game screenshot [Nin85] of *Super Mario Bros.* (*Nintendo*, 1985). On the top-right corner, the countdown for the stage is displayed – the player loses if it drops to zero and when it reaches 100 the background music starts playing in a faster tempo. To the left, the shiny star entity is a collectible item which causes the player's avatar to enter a temporary invincibility mode – it overrides the background music with its own theme for the duration of the item's effect.

playing over the other. A more smooth approach can be seen, for instance, starting from the *Nintendo 64* (*Nintendo*, 1996) generation of episodes from the *Legend of Zelda* franchise, where the background ambiance cross-fades into more tense themes whenever the player's character engages an enemy [KC08, chapter 8]. That is, instead of imposing sudden exclamatory sounds to disguise the change of musical content, the first song merely fades out while the next one fades in, without a clear moment for when one ends and the other starts. Boss battles from *Legend of Zelda: Twilight Princess* (*Nintendo*, 2006) even feature two or three separate musical moments in their soundtrack, transitioning from one another either through a cross-fade or a silence period. These effects require different levels of audio mixing, which, again, are only accessible through game code, and in a more invasive way than the *Mario* example since it would require audio control *over multiple game frames* instead of a single punctual trigger that requires no follow ups in other parts of the code (the concept of game frames will be further explained in Section 3.3.2).

There are many other real-time behaviors a soundtrack can have besides song and ambiance transitions. In real-time strategy (RTS) games such as *Age of Empires II: The Age of Kings* and its expansions (*Ensemble Studios*, 1999) – see Figure 1.2 – many things happen at the same time during gameplay. Castles are built, soldiers attack, villagers get lost, gold mines are depleted, etc. Since sound effects are used to call the player's attention to these events, it is possible that so many sounds are being played at the same time that some of them are missed by the listener. If an important one – like a "being attacked" warning – goes unnoticed, it could be quite frustrating for the gaming experience. Thus, there is a need to enforce a priority order among

these sounds, possibly dropping out the less relevant ones in some situations in order to guarantee that the crucial ones will be properly heard. In 3D games, another commonly used effect is sound spatialization, which uses audio manipulation to simulate how everything would sound in a real three-dimensional space. To implement this, it is necessary to understand how sound propagates and reverberates in different kinds of ambient, and then use the player's position and movement to calculate the resulting hearable audio effects.

Figure 1.2: An in-game screenshot [Stu99] of *Age of Empires 2: Age of Kings* (*Ensemble Studios*, 1999). During a match, dozens of game events can occur at the same time and most come with an assigned sonic signal to help the player keep track of them.

It is actually possible to take this discussion one step further. All the examples so far are based on the development model we described where composers and sound designers deploy their work in the form of sound sample files, which is the industry standard [WL01, Far07] (see also Sections 3.4 and 4.2). If we let go of this restriction and consider that **sound could also be derived procedurally**, many other possibilities arise [Far07, Far10, chapter 22] (see also Sections 3.3.4 and 3.4.5) which are of particular interest to this work.

Most of these mechanisms are actually very commonplace for musicians and sound designers. The difference in the production market of movies, for instance, is that they normally do have control over how their media is to be exposed to their audience. Since the soundtrack is usually one of the last parts of a film production [Mat14], it is natural to craft and master it around the fixed visual progression of the recordings, establishing cross-fades and emphasizing the most relevant sounds at will, as well as carefully applying spatialization according to the known spatial relations between actors and sound sources. In the end, a single soundtrack sequence can be delivered for synchronized playback with the video. **As for digital games development, since many**

decisive soundtrack factors can only be apprehended at run-time by the software alone, real-time control is left at the mercy of the technological limitations of the underlying platform, and, as we will explain during the rest of this section, **it is at least questionable whether the currently available methods and tools solve this issue in the most efficient manner from a game production perspective**.

1.1.1 Related problems

In the context of game development, this reality is not exclusive to soundtracks. It is a problem directly related to what is known as *data-driven design*. When a game project *does not* rely on a data-driven design, what happens is that every change to images, sounds, texts, stages, skins, weapons, characters, and everything else in the game will require a source code intervention of some sort, turning the programming process into a fatal bottleneck for the software production [Rab00]. The solution presented by the data-driven approach is to allow non-programmers in a game development team to input their work into the game through data files – hence *data-driven design*. While somewhat obvious, the whole point of this idea is to really take it to the last consequences and turn as much as possible from the game into data instead of leaving it as code.

We shall illustrate this concept through a problem analogous to the real-time soundtrack problem (yet much simpler). In bi-dimensional games, one type of visual animation is the frame-by-frame rendering of bitmap images. These images can represent characters, weapons, furniture, and many other scenery and narrative-related elements. An image that represents a single object or entity is commonly known as a *sprite*. By sequentially rendering different frames of the same character in different poses, an illusion of movement can be achieved. The frames can be organized in different files or in a single one composed of sub-images for each frame – a format also known as a *sprite sheet* or a *texture atlas*, like in Figure 1.3. With that, producing an animation is a matter of specifying the frame boundaries, the order in which they are to be rendered, the speed of the animation, and then writing a very simple piece of code to manage the timing of the frame changes.

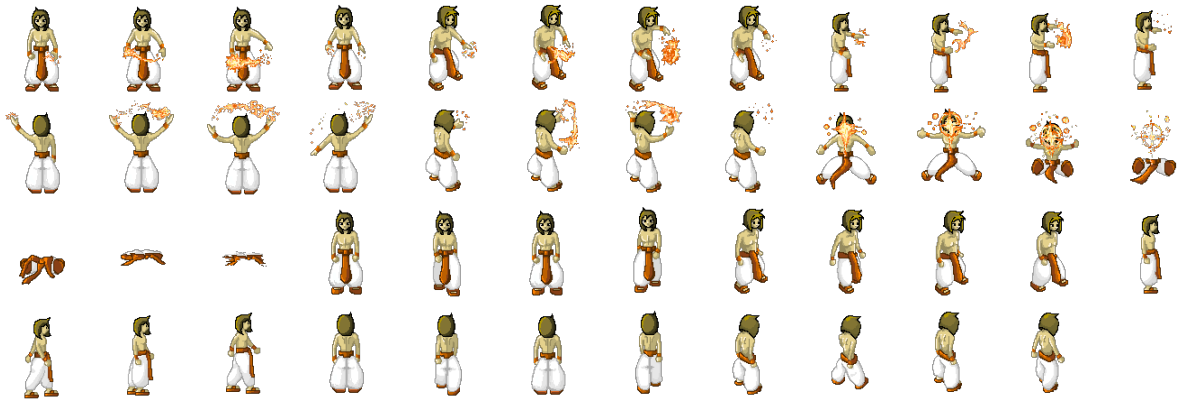


Figure 1.3: A sprite sheet or texture atlas from the hero character Kha [USP10] in *Horus Eye* (*USPGameDev*, 2010). By sequentially rendering parts of this image during the game execution, an illusion of movement is created, and an *animation* is produced.

The problem starts when this sort of asset inevitably needs to change. It can be the image itself, the size of the frames, the amount of frames per second in the animation, their order, etc. These are all relatively simple parameters that are easy to change in a well written software, but the sheer amount of assets in a game can easily require that the game be recompiled too often and end up jeopardizing the development process [Rab00, Gre14, Chapter 6]. The data-driven solution would be to write these parameters into a separate (meta)data file, and have the animation code read from it. With that, we have essentially provided the game with (graphical) real-time behavior in the form of an input file that any artist can write. Notice how this method combines the use of what one could call "static data" (the sprite sheet) with the use of "behavioral data" (the parameter file).

This pattern shows up in practically every other asset type in a game. With 3D meshes, you need parameter descriptions for material bindings, key frames, animations, orientation specification, etc. For in-game text, a proper localization-aware schema is required. The same applies for voice acting samples, which brings us back into the audio domain. As we saw in the previous section, sound samples need to be properly labeled and to have their playback conditions known. More often than not, some sort of filter must also be applied to them (be it a simple volume control or a complex reverberation effect) and thus should be specified too for the game to execute them correctly. As a final example of the "behavior data" pattern, game development tools usually feature a way to construct the stages or maps where the player will be able to navigate. These game-specific data come particularly filled with behavior information: cut-scenes, traps, collectible items, surprise attacks, triggers, puzzles, etc. We invite the reader to consider the fact that, ideally, the soundtrack should be able to reflect all of these in-game activities.

However, some of these "asset description" files quickly become inappropriate for manual editing. When it comes to this, the team might choose to use a specialized software tool for authoring this kind of content. For instance, *Aseprite*[1] is used for "pixel art" sprites and animations, while *Blender*[2] is directed at 3D modeling and animation. These tools are then responsible for exporting all the data files that will be loaded from the game code later. This might demand the use of external programming libraries that know how to interpret the exported files. Designing an in-house format could save the project some extra dependencies – the trade-off analysis is left to the team's judgment. Some examples of this sort of library are `libpng`[3], which is used to process compressed image files, and *Assimp*[4], a powerful library that can handle "various well known 3D model formats in a uniform manner".

As can be expected, not everything in a game can be described through sheer data. When behavior specification requires a file format as complex as a fully fledged programming language, game developers turn to *scripting languages* (for instance, *Lua*[5] [Wik]) as a compromise between computational capability and software integration flexibility. In Section 4.3, we will come back to this design approach when discussing our solution to real-time soundtracks.

[1] http://www.aseprite.org/
[2] https://www.blender.org/
[3] http://www.libpng.org/pub/png/libpng.html
[4] http://www.assimp.org/
[5] https://www.lua.org/

1.1.2 Available Solutions

Having said all that, it should be now clear what we meant with "real-time control is left at the mercy of the technological limitations of the underlying platform". How the game software will support sound reproduction and its necessary behavior description data will directly affect the possibilities and difficulties in implementing a real-time soundtrack. There is a wide spectrum of features one might want for a game technology according to what kind of real-time behavior is intended.

The most common feature available is sample playback. It is the bare minimum a tool can offer for a digital game soundtrack. From the *RPG Maker* series[6] to minimalist frameworks such as *LÖVE*[7] and on to industry-standards such as *Unity*[8], as simple as it is, with some creativity and extra effort, there are numerous real-time behaviors one can do with this. For instance, the independent (*indie*) game *Faster Than Light* (*Subset Games*, 2012)[9] has a pair of sample files for each of its in-game music pieces. These pairs are essentially two versions of the same song, with one of them intended for "exploration" situations, and the other for "combat" situations. The versions in a pair are the same except for a few extra instruments in the "combat" version, mostly percussive instruments. This allows the game to cross-fade between them very naturally, making it sound like a percussionist is really reacting to the game narrative. The obvious drawback is that the game ends up using at least double the amount of disk storage for its music assets.

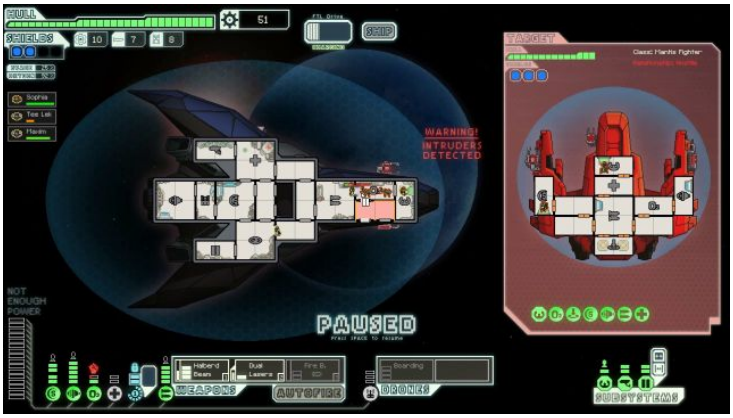


Figure 1.4: An in-game screenshot [Gam12] of *Faster Than Light* (*Subset Games*, 2012). The background music cross-fades between an "exploration" version and a "combat" version according to the game narrative.

This is what we will call an *ad hoc* solution. By directly implementing the desired features on top of a simpler tool, developers can take the first step towards real-time soundtracks. Depending on the game scale and budget, this method will probably be enough, but will come at the sacrifice of (1) allowing the sound designers to manipulate the soundtracks themselves (no data-driven support) and (2) restricting the sound design to audio sample juggling, since that is all the underlying technology permits.

[6] http://www.rpgmakerweb.com/
[7] https://love2d.org/
[8] http://unity3d.com/
[9] http://www.ftlgame.com/

As such, solutions that are able to go beyond require more powerful technology. There are two directions a development team can choose from this point. They can either build their own tool for real-time soundtracks (possibly relying on the sample-oriented features described above) or adopt a specialized audio middleware ready-for-use and pay the eventual monetary and learning costs. For the purposes of this section, we are interested in the latter[10].

First, given this whole discussion on the game development process and how the contents can be easily shipped into the game software, it is important to understand how available middleware solutions integrate with the development environment. We will properly present game development tools in Section 3.3.3, and we will be describing in more detail the main audio technologies for games in Section 3.4. But, for now, let us broadly understand how they work and in what ways this research field can complement their capabilities.

In Section 1.1.1, we discussed how some assets in a game could be accompanied by a "behavior data" file to specify how that asset should go into the game execution, including any real-time effects it might demand. An easy application of this pattern to the *ad hoc* sample-based solution explained above is to use these extra files to express sample playback as a more dynamic element of the soundtrack. For instance, we could group various samples for gunshots and state on the behavior file that whenever a gun of the appropriate type is shot, a sample from that group is randomly chosen to be played (following a uniform probabilistic distribution). With this, we would have made, at some level, an abstraction of how an individual sound effect behaves within the game narrative in real-time. There are many other possibilities, and some are more commonly used than others. What these middleware systems do is provide an interface that eases the manipulation of these soundtrack abstractions (the *soundtrack editor*), and then provide a mechanism (typically an *audio engine*) that imports them into the game software together with the audio samples and enables their playback. The diagram in Figure 1.5 illustrates this workflow.

Of course, this is all still under the assumption that the main data format for audio is sample oriented. This is true for most available tools, a fact that comes with an unavoidable fallback. Since audio samples are generally treated as atomic units of data, there is a limit to how much one can bend them for real-time behavior. This incurs in a non-trivial design restriction which, as we will see, most of the game audio technologies fail to overcome. Farnell [Far07] speaks at length about the disadvantages of sample-driven audio compared to procedural audio, for instance (we will go back to this topic in Section 2.1). All in all this means that, as far as available solutions go, there is much one can do with the technologies at our disposal (both commercial and noncommercial) but there are some things that are just not feasible or practical to do with them. Delving further into this technological gap is one of the intentions of this thesis, while also keeping in mind the success of other tools given their use under appropriate conditions.

1.1.3 Challenges

While the last sections broadly approached the problem of interest in this thesis, here we will briefly discuss some commonplace challenges inherent to this field of research. We have divided

[10] Throughout this thesis the use of the term "audio middleware" (or just "middleware") refers to software systems that integrate the design of a game soundtrack with the corresponding game engines or applications. This meaning may not match the usual meaning attributed to middleware in general, but is, nevertheless, the convention used in the game audio industry – and we follow it to avoid confusion.

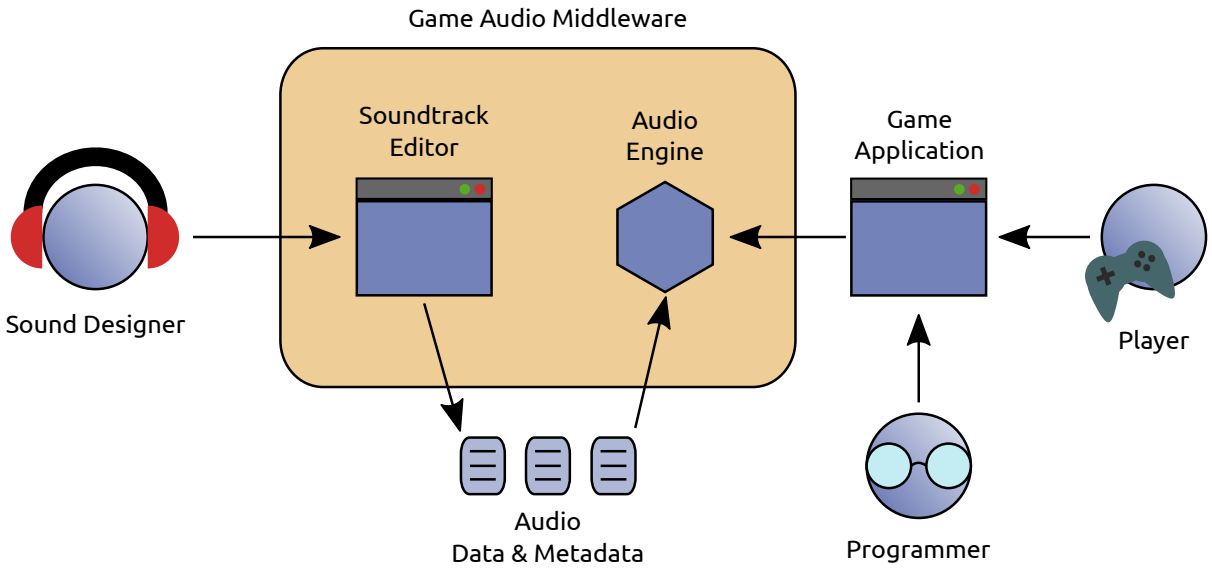


Figure 1.5: A diagram exposing the workflow of a typical game audio middleware. The sound designer uses the *soundtrack editor* to provide audio data in the form of both samples and behavioral specifications, while the programmer loads these assets into the game and plays them using the *audio engine*.

them in two groups: technical challenges and practical challenges. The first ones are those that can be tackled from a Computer Science perspective by proposing algorithms and other computational solutions. The second ones, though definitely not unrelated, lie somewhat beyond its scope.

A first technical challenge is one we have already hinted at when we described *Faster Than Light*'s soundtrack implementation. When the sound design choices lean towards heavy use of audio samples, specially for music, it is only natural to provide variability through the run-time combination of different samples. One could expand on the approach from this game, having an audio sample for each instrument in each tune of the game, and then mixing them during gameplay according to a given set of rules (which could be arbitrarily complex). However, since the storage size of an audio sample is proportional to its duration and not its instrumental composition (more on that in Section 3.1), this method can easily multiply the game storage size beyond acceptable limits depending on the target platform.

A second technical challenge is that naïve implementations and usage of real-time sound applications often cause audible clicks when a change occurs to some ongoing sound. This happens because the abrupt reshaping of a waveform produces a sonic artifact. The common solution is to try as much as possible to transition between different "audio states" smoothly by relying on techniques such as cross-fades (when there is enough memory for an extra buffer) or careful handling of signal manipulation [E01]. To some extent, it is possible to embed this kind of robustness into the audio middleware itself but, depending on how much control is given to the sound designer, it might be impossible to work around all the misuses of the tool features.

Another technical challenge is that of the dynamic level of audio detail (LOAD) [Far07, Far10]. When too many sounds compete both for the player's attention and the computer resources, a good compromise to make is to decrease the quality of less relevant sounds as to reduce their toll

on the machine performance, thus producing a more complete and efficient sonic landscape. This technique is an industry standard in computer graphics, specially with the advent of geometry shaders [DT07], and corresponds to an active research topic for audio processing [DPS+15]. The main challenge here is in establishing the computational costs for each available option.

Regarding practical challenges, the first one is none other than the (un)predictability of game soundtracks. No matter how well a game audio middleware deals with real-time behavior, it cannot make any promises about how it will sound during actual gameplay because, again, there is no way to predict all possible outcomes for even simple games. There are, however, ways to mitigate this problem. One would be to provide an environment for experimenting with the soundtrack "reactions", by manually feeding it with data that would usually come from the game execution. When the complexity of such interaction becomes impractical to simulate, a more elaborate alternative is to support a run-time connection between the game application and the soundtrack editor, allowing the sound designer to experiment with it while the game is running. Even then, handling untested or unpredictable scenarios will depend entirely on the sound designer's skill to extrapolate the performance of his or her creation.

The next practical challenge is that, one way or another, each new game audio middleware comes with yet another skill set the sound designers and programmers will have to acquire (more so for the former). The choice of using one in a game project should take into consideration not only the monetary costs, but also the learning costs of introducing a new technology to the production pipeline. This issue brings to the table research areas such as user experience, computer-human interaction and interface design, to mention a few. We will come back to this topic in the next section.

Finally, a very important practical challenge is that when you design any kind of digital content creation tool, the features and controls you leave at the user's disposal cannot help but influence the aesthetics of the final product, in some ways more than others. For instance, composing a music with *MuseScore*[11] naturally leads to a more traditional score-based content, while using *LMMS*[12] will likely make it easier to produce themes closer to electronic music. One may not exclude the other, but even then it is risky to state that the underlying technology does not impose tendencies and limitations to the general aesthetic of the contents derived through its use.

1.2 Objective

From what we have seen in the sections so far, producing real-time soundtracks in digital games is not a far fetched proposition, but it does comes with many practical challenges. The gap between programming and sound designing activities lead to slow validation cycles, and the limitations of the technologies involved impose restrictions over possibilities and difficulties associated to soundtracks. As such, it becomes clear that there are promising opportunities in the employment of computational solutions to simplify, automate, and expand the production of real-time soundtracks in game applications. So far, we have shown the concern for both powerful real-time

[11] https://musescore.com/
[12] https://lmms.io/

mechanisms and easier integration protocols between the software and sonic domains of game development.

With that in mind, the objective of our research is to design, develop, and evaluate a game audio middleware for real-time soundtracks with the purpose of empowering sound designers by following (mainly) Farnell's proposal for procedural audio [Far07]. All the source code of our middleware and accompanying examples is available as open source software under the *Mozilla Public License Version 2.0*[13] at https://github.com/vorpal-project. A more detailed description of the middleware architecture is available in Section 4.4, and its most relevant implementation details, in Chapter 5. For now, it suffices to say that, according to the general architecture in Figure 1.5, the middleware consists of two core components:

1. A digital content creation tool for real-time soundtracks; and
2. An audio engine capable of playing such content from inside game applications.

1.2.1 Intermediate Goals

In order to achieve the main objective of this work, there are a series of intermediate goals we need to achieve first, which we shall list and describe in this section.

Digital Soundtrack Representation

Digitally, a real-time soundtrack cannot be represented solely by audio sample files. Some other kind of file format is needed to describe the dynamic behaviors of the soundtrack. Additionally, this format not only has to be processable by the audio engine part of the middleware, but it also has to be *editable* from the digital content creation tool. Thus, one of the sub-objectives of this thesis is to choose or design a file format for digitally representing real-time soundtracks, and develop the necessary features in the middleware to read from and write to this format.

For example, consider the sprite sheet animation case from Section 1.1. Aside from the sprite sheet itself (which could be in PNG, JPG, or other texture formats), the game application may need an additional file describing the animation frame sequences. It could start by listing all frames in the sprite sheet, indicating their position, width and height inside the pixel matrix, and then it would define each animation in the sheet by giving it a name and assigning a sequence of frames (referenced by indices from the previous list) to it. Optionally, it might also specify an animation speed (in Frames Per Second or FPS) for each animation entry. That would be a "real-time animation" format, albeit very simple and not very interactive. There are also endlessly possible syntaxes (for plain-text formats) and data layouts (for binary formats) that could be used to express it. Regardless, it serves the purpose of illustrating what we mean by a "digital soundtrack representation" format.

Soundtrack Authoring Interface

Along with the digital soundtrack representation format, we will also require an interface for writing files in that format. Here, we use "interface" in a very broad sense. For instance, if we

[13] https://www.mozilla.org/en-US/MPL/2.0/

decide for some plain-text format, any text editor could be used as the "front-end" for our digital content creation mechanism. The idea is to find a convenient balance between usability and implementation overhead, as well as good integration with the rest of the middleware.

Real-Time Audio Engine

The flip side to an authoring interface is the piece of software able to consume the digital content it produces. In our case, the game audio engine mentioned in the main objective above fulfills that role. Its purpose is to not only read files formatted according to the adopted digital content representation, but also to execute the corresponding behaviors during application run-time – which here means effectively *playing the soundtrack*. The reason we call this an "audio engine" will be made clear in Chapter 3, when we study both game engines and digital audio.

Media Integration

Lastly, depending on how the other intermediate goals were accomplished, the middleware might require different levels of media integration among its components. The two main points here are the export-import protocol between the authoring interface and the audio engine – that is, what steps the sound designers must take to make the soundtrack they created available for the programmers to load into the game application – and the interface's capability to play a preview of the soundtrack in it – i.e. *outside* the game –, which might even require that it runs an instance of the audio engine from within it. In other words, this sub-objective points out that the interaction flow between the middleware components must be properly addressed at some point during this work.

1.2.2 Contributions

This work provides two major contributions to the field of real-time soundtracks in digital games. First, by making the game audio middleware into a research goal, and by publishing it with a free software, open source license, we will be feeding the game development community with an unprecedented implementation reference to this class of middleware, since all other ready-to-use alternatives are commercially licensed or patent protected, besides being closed source. Second, our solution follows Farnell's proposition for procedural audio, which also has not been done before in the form of a full-fledged game audio middleware – there are either ad-hoc implementations for individual games, or generalized tools for non-game applications. As for minor contributions, we present punctual solutions for the following implementation issues in game audio middleware (which we explain and address in the corresponding sections):

1. Procedural soundtrack abstraction (Section 5.2.1);
2. DSP management (Sections 5.2.2 and 5.3.1);
3. DSP-playback mixing with 3D localized audio (Section 5.2.3); and
4. Real-time audio synchronization with a host engine (Section 5.2.4).

1.2.3 Validation

This research is validated in three different forms: **Basic Feature Support**, **Advanced Feature Support**, and **Usability**. The purpose of the first two validations is to verify that our main objective is satisfied – the production of an effective real-time soundtrack middleware for digital games. The third is used mainly as a study for future improvements or alternate implementations to the approaches used here. Next, this section describes in greater detail each of these validation methods.

Basic Feature Support

The bare minimum expected from our middleware is the possibility of developing an actual game with a real-time soundtrack. As such, this first validation method consists of basically proposing and developing such a game using our technology. To avoid an overly artificial or contrived design in the game to showcase the capabilities of the middleware, we decided to establish a partnership with a professional sound designer whose needs and opinions would direct the game contents and features. The resulting title was *Sound Wanderer*, which we further explain in Section 6.1.

Advanced Feature Support

Since we could not possibly contemplate every feature a sound designer might need with a single game, and it would not be practical to keep developing games until all of them were verified, a different kind of validation was used in this work for what we are calling *advanced feature support*. By contacting and interviewing other professional sound designers in local game studios, we identified common needs in their workflow. We join those requirements with others from the literature, thus composing a list of real-time variability approaches to game sound. We then use them as a checklist for our middleware's feature pool. The full list is presented in Section 4.2, and their fulfillment is addressed in Section 6.2.

Usability

Finally, even if our solution did provide all the necessary features for real-time soundtracks in any possible game, it would be for naught if no sound designer was able to understand its user interface and workflow protocols. At the same time, we could not prioritize usability over the core features of the middleware, since completing a working functional system to validate our research comes first. As such, this last validation step was done solely for the purpose of identifying improvements that can be done to the system in future works. The method itself was of qualitative nature: we took notes of all impediments and advantages that came up during the development of *Sound Wanderer*, specially from the sound designer's point of view as a user, and then critically analyzed them in Section 6.3.

1.3 Text Organization

In this chapter we have presented the context and motivation for our work, as well the main challenges involved. Then we stated our primary and secondary objectives, pointing out where

our contributions lie and how we intend to validate the developed middleware. Chapter 2 presents other relevant research works that address similar problems. Next, in Chapter 3, we provide an overview of many concepts and tools required to better understand this research field and to guide the middleware implementation itself. Chapter 4 describes our computational solution for real-time soundtracks by addressing many of the issues raised during this first and second chapters. Then, in Chapter 5, we describe the implementation choices and details of the proposed middleware, relying heavily on what Chapter 3 covered. Chapter 6 presents the results in the form of the proposed middleware and how it meets our validation demands. Finally, Chapter 7 finishes this thesis by discussing our observations along the research and directing readers towards future works.

Chapter 2

Related Work

Real-time soundtracks in digital games is no novelty. Both commercial products and academic works have already shown many ways to achieve that result using a variety of strategies. This chapter intends to give a brief overview of these solutions (leaving more relevant and detailed explanations to other chapters) in order to better understand what is already out there and verify how they might help our work or how our work could improve on them. It will complement the solutions we described in Section 1.1.2 and serve as a guideline for how we design our own solution in Chapter 4.

To organize this discussion, we divide the related works based on how they approach our research topic. Their interest might lie in the technique, in the critical evaluation of current methods, in underexplored applications, besides other concerns. There are even a few cases we chose to talk about that do not even really reach for the same problem we want to solve, yet present a relevant intersection between their investigations and our needs. As such, Section 2.1 surveys works that regard audio in games broadly but which we will analyze with real-time behavior in mind. Then, Sections 2.2 and 2.3 very briefly expose more in-depth works relating to two powerful ways in which real-time soundtracks could improve game applications both technically and aesthetically. The last part, Section 2.4, discusses those not-so-related works we have mentioned.

2.1 Game Audio in General

Scott [Sco14] writes about the work of composers in game soundtracks, highlighting their role in the promotion of the sonic engagement of players. He explains that composing music intended to go along with visual media adds a whole new level of complexity to the already nontrivial task of writing melodies. Particularly, one of the main challenges pointed out by Scott is dealing with the nonlinearity of narratives in this audiovisual format. For instance, it is hard to avoid making the player listen to the same melody for hours of interaction without it becoming monotonous or repetitive. In that sense, he states that **it is important that a soundtrack be dynamic, serving as a support for the user's activities** and that even simple changes can help achieve this. The evolution of technology and the technique of embedding audio in digital games has increasingly worked towards suppressing this problem, but at the cost of demanding greater

digital dexterity from composers and sound designers. More and more of these professionals seek tools like the ones described in Section 3.4, and Scott recognizes that understanding the computational process behind the playback of audio in games is an important step to them. Complementing this discussion about the cohesion between sound and technology in games, Meneguette [Men13] says[1]:

> Regarding the production of such [interactive] sonic situations, dependent on the player's nonlinear action, there is an interesting challenge for composers and researchers of dynamic audio: how to describe the potential situations involving game sonority, taking into consideration aesthetic and technological issues, relating them to programmable parameters in the system, giving them a semantic nature and making the sonority correspond to an emotional and rhythmic directing in the [game] scene, suggesting a potential for the player's action. It could be a question with no easy solution, but it looks like a fruitful research path to be walked, taking care to keep logical coherence at each cross-roads of the route.

As we can see, both these authors help reinforce the motivation in our reserach. Another one we have already mentioned is Farnell [Far07], who actively defends that relying strictly on sample-based audio is both aesthetically and technically restrictive. In his book, *Designing Sound* [Far10], he proposes procedural audio as a more appropriate alternative, which the author defines as "non-linear, often synthetic sound, created in real-time according to a set of programmatic rules and live input". This includes DSP, synthetic sound, generative sound, stochastic sound, algorithmic sound, among others. Farnell offers a full collection of procedural audio solutions to various sound effects that could be used in games. Some of the main advantages of procedural audio he presents are:

1. **Deferred form:** since audio decisions are made at run-time, they can better adapt to the in-game context than if the sound designer tried to "predict" how it would go. Mixing can be specified by priorities, for instance, leading to a more flexible method of highlighting different parts of the soundtrack.

2. **Default forms:** by interacting with the game physics, sound effects could use physically-based sound synthesis as the default behavior (more on that in Section 2.3), while more relevant sonic events are replaced by carefully crafted sounds. This way, there is a good trade-off between quality and productiveness.

3. **Variety:** since procedural audio leans more naturally towards real-time behaviors, its results can easily vary according to the in-game context, avoiding Scott's concern for the soundtrack becoming monotonous or repetitive. This works for both sound effects and music, but with different approaches to each (see following sections).

4. **Variable cost and LOAD:** as mentioned before, being able to manipulate the soundtrack in real-time allows the game software to balance it out and save the computational resources

[1] Author translation from Portuguese.

at its disposal (mainly CPU and memory). This could be combined with advantage 2 by making default forms be parameterized according to resource restrictions, taking much of the work out of the sound designers' and programmers' way.

Another particularly important work on game audio is Collins's book, *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design* [KC08]. She carefully presents the evolution of game sound throughout the decades since their birth in the second half of the twentieth century. There are two particularly remarkable parts in her text which we bring to attention here. The first is that, curiously enough, game soundtracks were actually procedural in nature at first. In the industry origins, much as it is today, the sonic features of digital games depended on the technology underneath them. The difference was that technological limitations back then were still clearly stuck at the hardware level. Computers could not play long audio samples directly, and relying on analogue synthesizers was the standard solution. **This meant that sound effects and music were programmed instead of recorded**. Naturally, this led to the first real-time, interactive soundtracks in game history. Then in the 1990s, when *Compact Disc Digital Audio* (CD-DA) came around, sample-based audio replaced hardware synthesizers and remained prevalent since then.

The other part of Collins' book we found most relevant to our work was the chapter on *Compositional Approaches to Dynamic Game Music*. She brings up nonlinearity in games by highlighting that "games are largely unpredictable in terms of the directions the player may take, and the timings involved". One of her focuses was on game narratives with tree branching structures, where well defined decision points can cause the gameplay to diverge and how they are particularly engaging moments of the experience. Collins states that "Planning to respond emotionally to the visuals or narrative in terms of audio at these junctures, then, is incredibly difficult, and yet incredibly important". Lastly, the author enumerates ten approaches to variability in game music, mostly directed at symbolic representations of sound (more on that in Section 3.1.2):

1. **Variable Tempo:** accelerate or decelerates the music tempo.
2. **Variable Pitch:** shift whole sequences of notes in the frequency domain. This was historically used as a way to provide musical variation while saving memory.
3. **Variable Rhythm/Meter:** change the rhythm of a music score (from 3/4 to 4/4, for instance). The author did not find any practical examples of this technique, though.
4. **Variable Volume/Dynamics:** change the music volume, generally in order to highlight it or to stop it from distracting the player from some other event more worthy of his or her attention.
5. **Variable DSP:** change which and how filters are applied to music. As an example, some action-intensive games use low-pass filters to indicate a distance in consciousness from the player's avatar, specially when it is hurt or disabled in some manner.
6. **Variable Melody:** algorithmic composition, which the author herself admits being "still very much in its infancy".

7. **Variable Harmonic Content (Keys and Chords):** change how the music harmony interacts with the melody. Collins presents a couple of real-world examples of this technique.

8. **Variable Mix:** change which channels are mixed into the music and how they are so. We've had the example from *Faster Than Light*, but the author also mentions *Super Mario World* (*Nintendo*, 1990) and *Super Mario 64* (*Nintendo*, 1996), to which we could further complement with *Banjo-Kazooie* (*Rare*, 1998) and many others.

9. **Variable (Open) Form:** randomize the sequencing of the music. Somewhat challenging to achieve while preserving aesthetic integrity, but brings advantages such as avoiding Scott's monotony and providing inexpensive (in terms of storage) variety.

10. **Branching State-based Music and the Transition Matrix:** assuming your soundtrack is divided in playback states ("indoors", "escaping", "dying", "triumphant", etc.), this approach maps all possible musical transitions between them in a matrix such that the transition between state A and B is written to the cell in row A, column B of the matrix.

As a last (non-academic) reference for game music, the *YouTube* channel *ExtraCreditz*[2] – which publishes videos on game design in general from the perspective of a professional consultant in the game industry – presents an interesting duality in their chapter on *Video Game Music* [Ext]. They state that both "catchy", memorable themes, and more abstract ambiances are equally important to digital game soundtracks. The first ones help provide an identity for the game and its characters, and makes the player remember them years after he or she has played it. But since that can fall into Scott's issue of being repetitive and eventually monotonous if not unpleasant, this sort of strong melodic composition should not be applied carelessly (for instance, the stage lengths of the game should be considered). The second kind, abstract ambiances, is particularly appropriate for horror games, where the absence of musical structure can cause the necessary (intended) discomfort in the player's experience, and is also useful for activities in-game that are too long or that require the player's undivided attention.

The works of both Farnell and Collins provide explicit, punctual features a real-time soundtrack game technology should support. As such, we will come back to them in Section 4.1 to help compose the specifications our middleware must meet, as explained by our validation criteria described in Section 1.2.3. The basic idea is to either map these lists directly to middleware features, or to make the middleware capable of producing an equivalent result through lower level mechanisms provided to the user.

2.2 Music Automation

As we mentioned in Section 1.1, one of the main bottlenecks for real-time soundtracks in digital games development is the necessity for inserting all the in-game cues to sound events into the game code manually. This is specially difficult when conditions for the cues are not promptly verifiable, or are even too subjective to write into explicit code. For instance, one could wish to play a "sad" theme whenever a character is going through some narrative ordeal, or when the

[2] https://www.youtube.com/channel/UCCODtTcd5M1JavPCOr_Uydg

game and sound designers want to engage the player through the "sad" theme in key narrative interactions. We have seen this in *Faster Than Light* (where the player is sonically instigated during combat interactions), but since there are only two "emotional music states" and the cues are straightforward, the implementation poses no major issue.

In search for ways to better escalate this method, some research lines explore ways to automate the translation of in-game narrative interactions into sound events taking into consideration the relation between a desired musical aesthetic and the effects it has on the player's experience. Bearing in mind that any emotional interpretation of a music track depends on many cultural factors [Mat14], these works propose solutions in the form of rule sets that dictate how certain "emotions" or "moods" should be reflected onto the soundtrack, and are thus able to offer computational methods for automating the real-time process of applying such rules.

Livingstone *et al.* [LMBT10] proposes the **Computational Music Emotion Rule System** (CMERS), which consumes MIDI[3] files and modifies their musical parameters (such as tempo, key, attack, brightness, etc.) in real-time as they are played for the user. To do that, CMERS receives, as input, a stream of points from a bi-dimensional space which represents, in an obviously oversimplified manner, the "emotion" intended to be conveyed through the music. In this space known as the **Two-Dimensional Emotion Space** (2DES), one of the axis indicates whether it is a positive or a negative sentiment – its valence – while the other axis indicates whether it is an active or passive sentiment – its arousal. The system uses the octant in which the valence-arousal points lie to determine what changes to apply onto the music track. These changes are the same no matter which MIDI file the system is processing, they follow music-emotion patterns the authors harvested from an extensive bibliographic collection. The mapping between the space regions and the musical effects are shown in Figure 2.1.

A similar bi-dimensional representation is used by Eladhari *et al.* [ENF06], except it is used to indicate the musical mood of game characters instead of the emotion to engage the player with. In this version of the space, one axis reveals the character's inner mood (that is, regarding itself) and the other reveals its outer mood (regarding those around it). The way the points are applied to the played music also differs from Livingstone *et al.*'s CMERS: the space assumes the discrete form of a matrix – called **Humor Matrix** –, with each of its cells being associated with a prerecorded variation of the character's theme. The matrix can be seen in Figure 2.2. This alternative solution has the audio fidelity advantage of working with samples instead of MIDI, but on the other hand it demands the previous effort of composing all musical variations of each humor cell. One interesting aspect of Eladhari *et al.*'s work is that the authors also propose a method to automatically relate in-game interactions with mood matrix positions using spreading activation networks, molding what they call the **Mind Module**.

While these proposals do lie somewhat outside the scope of our work – specially for dealing with music and composition issues that go beyond the reach of computer science – they do show the kind of musical manipulation a sound designer or composer might be interested in and thus should be supported by a game audio middleware. They even describe effective implementations of these features. With that, we see that having control over volume, pitch, timbre, tempo, attack, and melody might be necessary to direct the musical intention in a game. Besides, some

[3] https://www.midi.org/

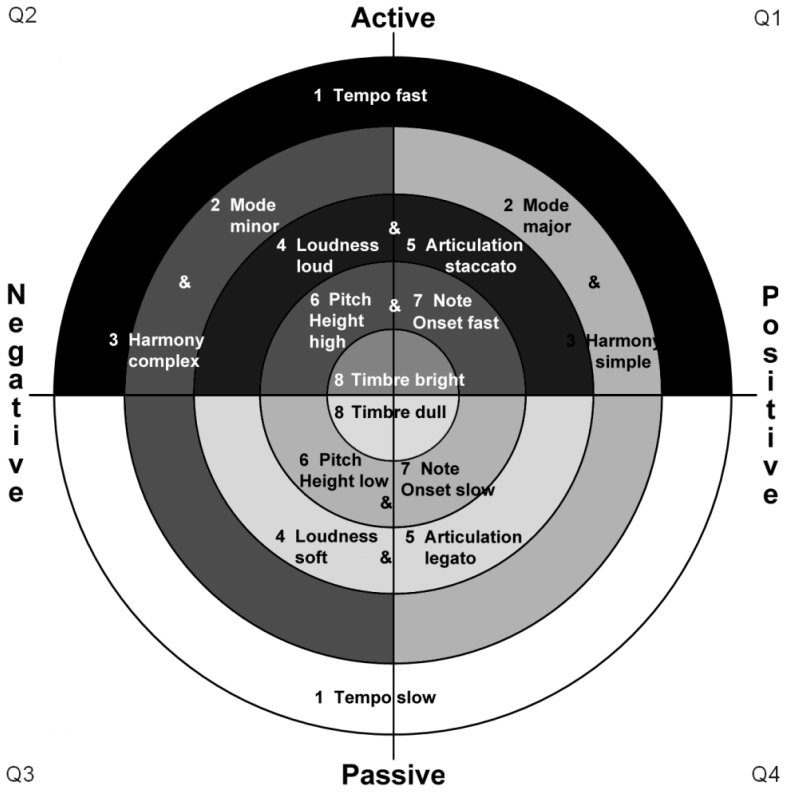


Figure 2.1: 2DES: musical emotion space proposed by Livingstone *et al.* [LMBT10]

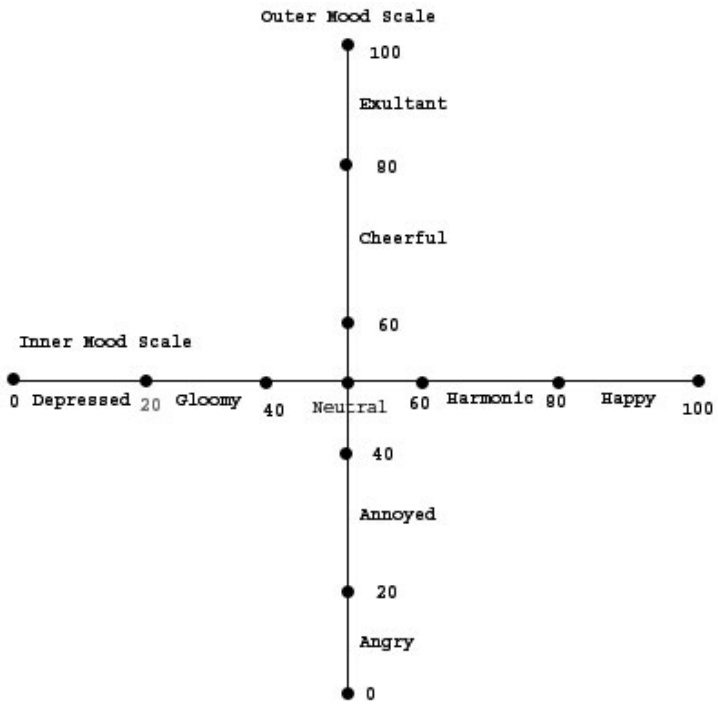


Figure 2.2: *Mood Matrix* proposed by Eladhari *et al.* [ENF06]

of these were also mentioned by Collins, which reinforces their relevance. There is a trade-off to consider here, though. Even though some musical parameters are promptly controllable by post-processing filters – such as volume and timbre (to some extent) –, others are not if we keep relying on prerecorded samples only. The music structure itself must change during playback if one wishes to manipulate its melody, tempo or attack. Livingstone *et al.*'s CMERS achieved that by using MIDI, while Eladhari *et al.*'s Mind Module chose to depend on prerecorded samples. The former is more versatile, but comes with the typical MIDI drawbacks (which will see in the next chapter), while the latter demands more work from the composer, even though it produces audio with more fidelity [Mat14].

2.3 Physically Based Real-Time Synthesis

Following one of Farnell's main ideas for procedural audio in games [Far07, Far10, part IV], another line of research lies in the development of algorithms for physically-based real-time synthesis and processing of sound effects. Essentially, works in this area try to simulate how a sound behaves according to (a selected group of) laws of Physics that apply to it. They extract geometry and material information from objects and the structure of virtual spaces in games and derive the corresponding sound, reverberation and propagation effects, among others.

James *et al.* [JBP06] provides in his work a method for real-time synthesis of the sound that comes from the vibrations in solid bodies with complex geometries by approximating their shape modeled by multi-thousand vertices and triangles into a few dozen key points which would produce an equivalent sound. Bonneel *et al.* [BDT+08] implements an optimization for modal sounds (like the ones from James *et al.*'s work) by using their frequency domain representation instead of the usual time domain implementations from previous works. These contributions allow the user, upon interaction with a virtual scene, to hear perceptually adequate sounds when a pile of boxes falls apart or when an explosion is triggered – all without the need to keep an endless library of sound effect samples.

Once the sounds originating from game objects are at our disposal, the next expected feature of physically-based audio is to simulate how that sound propagates and reverberates in a given virtual space in-game. These techniques are analogue to physically-based ray-tracing in computer graphics [PH10], in which a wave path (of light or sound) is computed by considering the geometry of scene props and their respective diffuse and specular reflections, among other effects. Doing so in real-time while keeping an adequate level of quality is one of the main concerns of ray-tracing methods, due to the sheer amount of rays cast. Nevertheless, real-time propositions are available in literature, such as the works from Taylor *et al.* [TCAM09] and Raghuvanshi *et al.* [RSM+10].

These techniques can be very expensive to implement and process, requiring deep understanding of other knowledge domains such as Physics, Acoustics, and Numerical Analysis. Nevertheless, with the prevailing "realism" oriented aesthetic of digital games in the market, their demand might keep increasing. In our research, we strive for a compromise in this regard by stating the middleware specifications in a way that make it possible to implement these algorithms, but we leave them out of the core system features and allow their inclusion through extension mechanisms.

2.4 Other Works

There is a multitude of works that focus on innovative ways to use sound in games but do not necessarily cover our concerns for real-time interaction. For the sake of completeness, we will briefly mention here some involved areas which help give an insight into further possibilities for game audio middleware. One such line of research is that of games whose user interface is composed of audio elements (also known as *audio games*), in particular games where the player uses run-time audio input to act inside the game narrative [PH07, PH08]. An interesting work from Nelson and Wünsche [NW07] points out three other unusual approaches to game audio. The first is using pre-processed sonic input for generating game content or controlling its mechanisms such as dictating the speed of enemy attacks based on the music rhythm or what kind of actions the player can do based on the currently playing instruments. Another approach is making in-game player actions produce individual sounds that, together, may compose a musical piece of its own – that is, turning the games into instruments themselves. There is a similar work in this vein by Furlanete *et al.* [FMM08]. The third approach is about how DJing techniques could be applied to game soundtracks, since one of the main skills needed for a DJ is to mix a song with the following one seamlessly.

Among these, using audio as a form of *input* (be it off-line or on-line) does not contribute to our research discussion, since we are mainly interested in improving how the audio is presented to the player, which means manipulating how it is played in the game *output*. On the other hand, the ideas of

1. using game elements and mechanics as a form of music creation, and of
2. applying DJing techniques to music transitions

are both relevant in our context. The first hints at the division of the soundtrack into more granular elements in a way that makes their combinations produce an interactive music composition that follows no conventional form in particular. The second provides a whole music manipulation domain from which one could draw transition methods and possibly bring them into a computational formulation, which could greatly simplify Collins' *transition matrix* method, for instance.

Many of the basic concepts and tools we mentioned along this chapter are discussed in the next chapter. Having provided a better understanding of ongoing research in this area, we can now focus on building the foundations we need to justify and implement our proposed solution in later chapters.

Chapter 3

Concepts and tools

There are three main knowledge domains involved in this research. The first of them is that of digital signal processing (DSP) for audio. In order to familiarize ourselves with the kind of algorithms and data structures we will be using, we need to understand how sound is represented, manipulated and played by computers. The next domain is that of the soundtrack production process, which we will study with the intention of finding the gaps that our middleware is supposed to fill in for. This means that we will not concern ourselves with how to make an appropriate soundtrack, but how to computationally help those capable of doing so. The third and last domain is that of digital games itself. It is important to understand the technological restrictions associated with the development of this kind of software application, specially to know where and when audio features come into play. All of these domains will be brought together when we propose our middleware solution in Chapter 4 and when we present its key implementation aspects in Chapter 5.

The first part of this chapter will present each of these domains in Sections 3.1, 3.2, and 3.3, respectively. The second part, consisting of Section 3.4, will regard currently available or historically relevant game audio middleware and also *Pure Data*[1], which is a more general purpose DSP tool we will refer to in all following chapters. By analyzing what we consider the main solutions for real-time soundtrack in games so far, this chapter will have established the groundwork upon which we will build our own middleware. Finally, as a disclaimer for the content in the rest of this chapter, we clarify that by no means does it intend to provide an exhaustive reference for the approached subjects. When adequate, we will indicate proper bibliographic material for the reader to deepen his or her understanding of individual topics.

3.1 Digital Audio

For something to be manipulated by computer programs, it is necessary to represent it digitally, even if such representation ends up not being exact. That is, we must reduce that something to numbers stored in machine memory. Some representations are based on the real characteristics of the phenomena, while other are more artificial but likely easier for computational analysis or synthesis. With digital audio, the representation of sound is closer to the former. In very simple

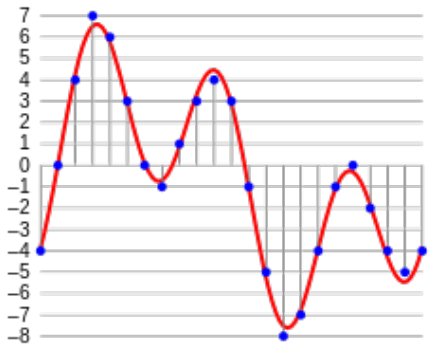
[1] http://puredata.info/

terms, our ears perceive sound by sensing very small variations in air pressure. If we consider this air pressure variation a function over time – which we call a **signal** –, we can use the language of mathematics to talk about it and take a first step towards a more machine-friendly representation. In this case, it would be an $\mathbb{R}^+ \to \mathbb{R}$ function – known as an **analogue signal** –, which can be converted into real air pressure variations with proper equipment (i.e. the sound card of any modern computer). However, computers cannot store real numbers nor represent arbitrary continuous functions, so instead we represent sound as a sequence of numerical values indexed by *instants* in the time domain, which neatly comes down to an array of integers or floating point numbers in memory. We call this a *digital signal*, hence *digital audio*.

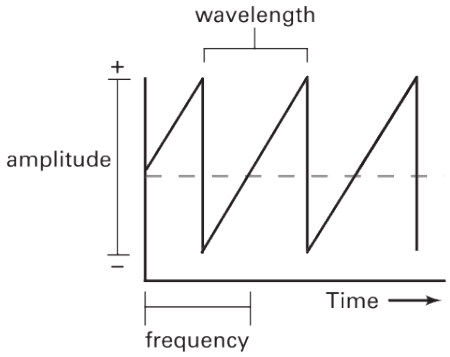
Each individual number in this format is called a **sample**, but this term may also refer to whole sequences of audio signal. The degree of fidelity from data stored this way depends on the resolution used for slicing the time domain and the precision used to store the values related to air pressure in bytes. The resolution basically states how many values are needed to represent a second's worth of audio signal, thus constituting its **sample rate** (usually measured in Hertz). Audio precision is derived from how much memory space is dedicated to the storage of each individual sample – the larger the space, the larger the set of possible values is and thus the greater the precision is –, and whether it uses a fixed or floating point representation – which influences how the precision varies between smaller and greater sample values. These and other concepts are extensively explained in great references such as Roads' book, *The Computer Music Tutorial* [Roa96], so here we will limit ourselves to what is of immediate interest to our research. Also, Figure 3.1a illustrates the sampling process from an analogue signal into a digital one.

Carrying on with the basic terminology introduction (which is used by many tools and programming interfaces), there are certain qualities of sound related to musical aesthetics that can be derived from its signal representations. The **frequency** with which patterns repeat themselves in a signal over time is directly related to the *pitch* we recognize in the corresponding sound, being possibly identified as musical notes or more complex harmony compositions. Alternatively, one may refer to the **wavelength** of a signal, since it is the inverse of the frequency. The maximum local variation in sample values inside a continuous patch of the signal determines how loud we hear the corresponding sound – that is, it determines its **volume** –, and is usually named the signal's **amplitude**. Figure 3.1b indicates each of these measures in the graphics of a sound wave signal.

Computationally speaking, as we have seen, one uses continuous memory spaces to store and represent a sound signal, with each memory unit corresponding to a sample, and their positions, to the temporal progression. When we need more than one channel (to obtain stereo sound, for instance), you can either interleave the samples from each channel over the memory or write one channel until the end before starting the next. Then, in order to play audio represented this way, all we need is to specify the whole format details we used to write the signal with in the first place: sample formats (memory size, fixed or floating point and signed or unsigned numbers), sample rate, size of signal sequence and channel quantity (and whether they are interleaved or not). We then send this data along with the signal to the computer sound card, and it will be capable of converting the digital signal into an interpolated analogue signal and play them through loudspeakers or earphones. In order to access this feature from the device, programmers



(a) Digital audio sampled at a 4-bit precision (blue points) from an analogue signal (red line) [Aqu13].



(b) Basic properties of sound waves: frequency, wavelength and amplitude. [KC08].

Figure 3.1: Basic terminology regarding digital audio and sound waves.

use specialized programming libraries, which in turn depends on the available drivers to send and receive information from and to the sound card. As we explain in Chapter 5, our middleware uses the *OpenAL*[2] Application Programming Interface (API) for this.

An immediate consequence of this digital audio representation is that storing it in the file system can be very straightforward. We can just transcribe the signal from RAM to a binary file, add a header with format specifications, and later load it back to memory to play it. However, this kind of storage has the disadvantage of having a very high space cost: a stereo audio sequence of five minutes sampled at $44100Hz$ occupies 26.46 megabytes, so any simple playlist of a hundred songs would not fit into a typical two-gigabyte pen drive. In order to avoid this, many compression methods have been developed, which may come with or without data loss. For example, the WAV format stores the signal as-is, i.e. without any compression (and thus suffers no data loss), while the *Vorbis* OGG format[3] uses lossy compression.

3.1.1 Digital Signal Processing (DSP)

Nonetheless, these file formats are exactly what we referred to as "sample-based" audio back in Chapter 1: they are *linear*, *atomic* structures which may hinder the interactive nature of games. One way to use them in a more *procedural* way (refer to Section 2.1) is to produce (*synthesize*) or manipulate (*filter*) the signal sample array from inside the software application (as opposed to doing it in a separate digital content creation tool). This is called DSP, a method we have also already mentioned in Chapters 1 and 2. Here we will briefly explain in general terms how these two facets of DSP (synthesis and filtering) can be implemented and used to achieve a few illustrative sounds and effects.

Once we know the relations between the signal structure and the corresponding sound, we can reproduce or invent a variety of timbres of interest, and then use them as sound effects or compose them into more musical forms. We can even imitate human speech[4]. Audio synthesis is

[2] https://www.openal.org/
[3] https://xiph.org/vorbis/
[4] *Vocaloids* are an example of this. See https://en.wikipedia.org/wiki/Vocaloid.

this process of artificially creating a sound. The most classic example is the sinusoidal synthesizer. Since a signal frequency is related to its pitch and a sine wave has only one fundamental frequency, writing it to the sample array with a frequency within humans' audible spectra and then playing it results in a "clean" pitch sound. Following a similar logic, there are many synthesis techniques available. To name a few: additive synthesis, AM/FM synthesis, granular synthesis, etc.

The other side of DSP are filters. Their idea is to process one or more input signals – which may come from sampled sources or outputs of other DSP functions – by numerically manipulating them to produce an output signal with a desired effect. The simplest example is to multiply all the samples in a signal by a fixed number. This effectively reduces or increases the original sound volume, since it scales its overall amplitude. When we ponder a collection of input signals added together using an amplitude filter for each, we are doing a process called **mixing**. As with synthesizers, there are endless possibilities for what you can do with filters. A few examples are: low/high-pass filters, band-pass filters, cropping filters, noise removal filters, etc. More often than not, synthesizers and filters complement each other in a way that makes distinguishing them a blurry and questionable process.

This quick introduction promptly points to some advantages we get when we bring DSP to table together with real-time soundtracks. Since synthesizers and filters are computer procedures, they can be fed with real-time data, which gives us real-time *control* over sound. Going back to our overused example, we can now see that the crossover effect from *Faster Than Light* is a combination of two sampled themes with a mixing filter that receives the amplitude factor in real-time from the time management mechanism of the game (which we will explain in Section 3.3).

Since there are a lot of common pitfalls in handling a raw signal, developers and sound designers usually rely on dedicated tools as much as possible for DSP features. There are many options available, each with its own approach and advantages. Some of them come as applications with well polished user interfaces and a typical digital content creation workflow, such as *Audacity*[5]. Others come in the form of programming languages, providing nearly unlimited freedom at the cost of a harsher learning curve and narrower accessibility, like *CSound*[6], *SuperCollider*,[7] and *FAUST*[8]. *Max/MSP*[9] and *Pure Data* are two particularly interesting examples that try to find a balance between these two extremes. They are both applications that work each with its own *visual programming language* specialized for DSP, with *Max/MSP* being proprietary and *Pure Data*, free software. By relying on the *data flow programming* paradigm, these languages allow sound designers to more easily develop filters and synthesizers to meet their needs. Each "program" or "module" written on these environments are commonly referred to as *patches*. Section 3.4.5 provides a more in-depth description of *Pure Data*, since it is particularly relevant to our research.

[5] http://www.audacityteam.org/
[6] http://csound.github.io/
[7] http://supercollider.github.io/
[8] http://faust.grame.fr/
[9] https://cycling74.com/products/max/

3.1.2 Symbolic Representation

We have mentioned that computational representations have two qualities: how close they are to the original phenomena, and how convenient they are to manipulate. We also said that digital audio is a computational representation that tries to come as close as possible to the original physical phenomena of sound since it literally models a sound wave as a sequence of bytes. Then we presented the concept of DSP, which allows us to mathematically control digital audio. Now, as powerful and useful as that is, there are a few use cases where seeing sound as a signal makes it harder than it is worth it. Notably, it is hard to represent score-based musical compositions using digital audio and DSP. In order to play a single piano note, we might need a synthesizer, a filter to shape the note onset, a chronometer to synchronize the tempo, a separate timer to sustain the note and possibly many other structures. It is far from ideal to have to implement this patterns every time one needs a music routine. If, instead, we encapsulate it in a tool that needs only be fed with notes in their own representation (say, a pitch and duration pair) to play them, it would be much simpler and more productive to write songs (at least score-based ones). That is the general idea behind what is called *symbolic representation* in computer music.

Symbolic representation has a very important historical role in computer music, thanks in great part to an industry standard called MIDI (which we already referred to in Section 2.2), a digital communication protocol used to integrate instruments, digital audio workstations (DAWs), computers and more. In general terms, the MIDI protocol specifies a series of parameterized commands (and their binary representation) for controlling musical devices, from inputting individual notes to mixing different instrument channels. One of the issues with MIDI is that, since it consists of nothing but symbols, it entirely depends on how the target platform is able to generate audio from the commands it receives. For instance, if a MIDI command asks for a certain sax note to start playing, there is no way for it to specify exactly what kind of sax timbre is to be used. It could be a synthesized one or a sampled one, and in each case even more unspecified parameters arise (which synthesizer to use, which brand of sax to sample from, etc.). There are standards that try to mitigate this though – like *General MIDI*[10].

Curiously, although its hardware use is intrinsically manifest in real-time, in the software level one of the most common applications of MIDI is as a sort of *music score file format* where *one-way sequences* of MIDI commands are listed. Nonetheless, that does not mean it cannot be used for real-time applications, as we have seen from Livingstone *et al.*'s work [LMBT10]. In fact, it actually supports real-time behavior better than sample-based approaches, since its atomic units (MIDI commands) are more granular than sample sequences and its symbolic nature is likely more expressive to humans than raw byte arrays or function graphs, making them easier to reason with in terms of dynamic behavior. As an example, it is trivial to pitch-shift a MIDI sequence in real-time: simply add a fixed number to the pitch parameter of all incoming note commands. On the other hand, pitch-shifting a sample requires some effort to avoid time-stretching it too, even for off-line applications. One could take a step even further and propose that a music be composed algorithmically by picking the note pitches on the fly following a computational process influenced by the in-game context. This approach potentially provides an endless source

[10] https://www.midi.org/specifications/item/gm-level-1-sound-set

of soundtrack material, albeit at the cost of unavoidable aesthetic limitations.

Finally, there are also many tools available for creating digital MIDI content, two of which we have already brought up in Section 1.1.3. *MuseScore* is a What-You-See-Is-What-You-Get (WYSIWYG) digital content creation application based on classic score notation. It can load MIDI files and export projects in sample formats, and one of its strong points are the keyboard shortcuts that make composing quick and relatively comfortable. The *LMMS* application (mentioned in Section 1.1.3), on the othe hand, comes with a DAW-like interface and a piano-roll control for composition. Its main advantage are its numerous sound design features and powerful synthesizers. Both tools are free software. Section 3.4.1 describes one of the first game audio middleware systems that supported real-time soundtrack by relying on a MIDI-like internal representation.

3.2 Soundtracks

It is not always possible to apply traditional soundtrack concepts to games, mainly because such notions derive mostly from the study of linear audiovisual media. Notwithstanding, undermining these concepts would be a mistake. Not only do games often rely on conventional linear scenes (where there is no reason not to follow industry standards for the soundtrack), more often than not the role of sound is the same independently of the (non)linear nature of the narrative [KC08]. For that, we present in this section some basic concepts regarding soundtracks, with a focus on terminology and relevant practices that will help us discuss the issues our middleware tackles. We note, however, that understanding what determines the aesthetic quality of a real-time soundtrack is beyond the scope of this work. Our concern lies in the computational challenges involved in empowering the sound designers to do what they believe is best for the game – and for that, we need to comprehend a bare minimum of their language and workflow.

Matos [Mat14] defines "soundtrack" as being the sum of all *sound effects*, *voice* and *music insertions* that are to be played along with a visual representation of a narrative. Each of these soundtrack dimensions have a different function in the media. *Sound effects* are there mainly to make the image more "real" (whether they are accurate reproductions of real sounds might as well be irrelevant), while conversely tracing a subjective "point-of-listening" [Bas15]. In games, though, sound effects have an additional role: to improve user interface feedback. They can be used to point out noteworthy events, like an enemy being spotted, even if the equivalent event in real-life had no inherent sound; or simply to make some control interactions feel more pleasant, like the sound of a button being pressed inside the game menu when you click on it. This last case is directly related to the concept of *juiciness*, which is not limited to sound feedback but to a limitless set of possible audiovisual experience enhancements in game design [Sch14]. Figures 3.2a and 3.2b show screen captures of games that employ both visual and sound juiciness in their user interfaces. Back to soundtracks, *voice* is composed of the characters' speeches, and as such it is obviously desired that they be clearly audible at all times - leading to some technical implications regarding real-time audio mixing such as overall volume control and sound prioritization. For instance, another famous game in the same line as *Age of Empires II* (refer to Section 1.1) is *World of Warcraft III* (*Blizzard Entertainment*, 2002), where the real-time sound alerts are mostly done

by voice-overs. In order to properly listen to and make out the alert message words, there cannot be too many other concurrent sounds. Lastly, *music insertions* traditionally serve the purpose of filling out the narrative by showing what the image cannot show by itself: the scene mood, a character's intention, a commentary on story events, among other functions [Bas15, Mat14]. In games, music can also be used to (try to) regulate the player's attention: intense themes may cause a sense of urgency, while calm themes may help the player focus on a puzzle.

(a) Main menu in-game screenshot [Ent14] from *Hearthstone* (*Blizzard Entertainment*, 2014).

(b) An in-game screenshot of a menu [Atl08] from *Shin Megami Tensei: Persona 4* (*Atlus*, 2008).

Figure 3.2: *Hearthstone* and *Persona 4* are good examples of games that demonstrate the concept of *juiciness* in their user interfaces.

Aside from this division of the soundtrack by roles, we can classify its individual elements regarding whether they belong to the **diegesis** or not, that is, whether they com from inside our outside the narrative universe. For example, when a character plays a song with a flute or even during dialogues, the space where the sound conceptually comes from is "within" the narrative. This characterizes **diegetic** soundtrack elements. On the other hand, a song that is played only for the audience to hear (a background music) or the narrator's speech (sometimes) come from "outside" the narrative itself (they are actually the means by which the narrative is presented). We say, then, that they are **extra-diegetic** soundtrack elements. In practice, this separation is not always clear, specially in digital games, where **trans-diegetic** uses are common – like when a character speaks directly to the player to explain UI commands, extrapolating the narrative universe or "breaking the fourth wall" [MP11].

As we can see, some sound events in a game are related to diegetic phenomena, which are, in turn, directly or indirectly under the player's control. This effectively makes game soundtracks *dynamic* in nature. In this sense, we can also classify to which degree this dynamic behavior is present in a soundtrack. It is called **interactive** when it responds to immediate actions from the player, such as when he or she presses a button on the gamepad. On the other hand, when sonic changes occur due to the internal mechanisms of the game – like a warning jingle when a stronger enemy appears or when the theme cross-fades into another during transition between stages – we consider it an **adaptive** behavior [KC08]. Thus, it is common for sound effects to be regarded as interactive audio, while responsive melodic progressions tend to be seen as adaptive audio, and both fit into the broader category of dynamic audio.

3.2.1 Traditional Production Process

Having presented some initial concepts regarding soundtracks for audiovisual media, here we will point out some common steps in the production line of a traditional soundtrack for movies, serial shows and animations, while keeping an eye out for the technologies used. Let us start by fitting the soundtrack into the bigger picture of audiovisual production. Matos [Mat14] states that the most common practice is to leave it as one of the last steps, and this is reaffirmed by some of the interviewed sound designers (see Section 4.2). This means that the composer's or sound designer's work is usually based upon a (mostly) complete version of the movie or animation. His or her job is to compose, record, synthesize, and master all the necessary music and sounds, and then manually synchronize them with each cue in the video. Here we already have a crucial contrast to games development: even if the soundtrack is left to the end of the process, the sound designer does not have the leeway of seeking through gameplay to synchronize the samples with the cues, since, as we have seen, not all possible interactions can be foreseen beforehand.

Although we have been restricting our discussion to the point of view of sound designers and sometimes composers, there can be a plethora of other professionals involved. For sound effects, Foley artists and sound effect recordists must provide the necessary samples, carefully molded and recorded in the appropriate studios and with proper equipment. In animated media where there are no "physical" actors in the footage, the production needs to hire voice actors to record the character speeches from. For music, it mostly depends on the targeted aesthetic. The team may opt from synthesis and music bands to whole orchestras. Then the music samples must be recorded, edited, and mixed before being handed together with the other samples to the sound designer or sound editor, whoever is responsible for putting the pieces together onto the final soundtrack for the movie.

As for the technologies deployed, there are far too many to go over in this work, particularly on the hardware side of the process: microphones, recorders, amplifiers, etc. Instead, we will briefly talk about the software tools used for editing and synchronizing sound, since they are the closest to game audio middleware (or rather, game audio middleware are inspired on them). These tools are the Digital Working Stations (or DAWs) we mentioned in Section 3.1.2, and they have two main interfaces: a mixing table and a timeline editor. Mixing tables are used to route the playback of different audio signals to one or more channels. A simple example would be to route the playback of a sample to the loudspeakers of the studio to hear it. Channels can lead to other equipment (like reverberators, amplifiers and other filters) or to specific modules in the DAW (for DSP-based effects). Then, each of the resulting signals can be fed back to the table through another channel, and we can mix them into the final resulting sound. The purpose of the timeline editor, on the other hand, is to provide control over the synchronization of the soundtrack. It can come in a variety of forms, but in one way or another it must display a *timeline* where the user can drop, drag, seek, cut, copy, and paste audio samples, among a myriad of other basic or more advanced operations. Once the sound designer is satisfied with the state of an audio timeline, he or she can then export the resulting audio file in the usual sample-based formats (WAV, OGG, etc.). Two known examples of DAW software tools are *Pro Tools*[11] (proprietary) and *Audacity*

[11] http://www.avid.com/pro-tools

(free software).

To devise the relevant cues in a video clip, the sound designer and the director (or any other authoritative figure in the production) must discuss it together and write down all of them for future reference. Matos presents a few examples of this kind of document, which are mostly crafted in tables that map each sample to a video clip, a timestamp and a cue, among other details [Mat14]. Based on this, the sound designer can estimate the costs for the required resources and professionals, then start his or her work. When it is done, it is usually delivered as a single audio file to be played together with the video. When this step is reached, there might be a need for some last-minute synchronization adjustments caused by extra tweaking to the recordings, until the movie is ready with its final soundtrack.

3.2.2 Real-Time Soundtracks

Now that we have seen how traditional linear soundtracks are produced, this section will discuss real-time soundtracks in general. Aside from digital games, there are a few other cases where we can say a soundtrack falls into this category. Essentially, whenever one wishes to have a sonic form intentionally paired with a live visual performance – be it on a screen, on a stage, on the streets or wherever – real-time behaviors are bound to arise. The key factor here is that it has to be a *live* performance, because otherwise the best alternative is to follow the traditional process from the previous section (after all, both professionals and technological tools have already accumulated decades of experience in it). In a sense, games can be considered live performances where the player is the performer, except there usually is no audience[12].

The real-time behaviors in these performances may come in many forms. For instance, they could happen as planned improvisational interventions, like a solo or an act that depends on the audience's participation. Sometimes, the whole performance itself is designed in a free-form session, depending on what the artists and performers decide to do in the spur of the moment. De Lucca, one of the interviewed sound designers, has many works like this (see Section 4.2.3). Even if a performance is supposed to follow a linear script, human error and other unexpected mishaps can occur which require on-the-fly adjustments and adaptation. A few merely illustrative examples are:

- **Excessive repetition:** specially for soundtracks intended to accompany activities or to include waiting periods. Having a few extra songs or effects can help mitigate this without having to rely on improvisation.
- **Transitions:** as we have seen, simply switching from one song to the next instantly is usually avoided by using cross-fades or silent intermissions. Handling these transitions is one of the common skills required from DJs (see Section 2.4).
- **Polyphony:** when there are multiple sounds being played, like a background music and a person giving a speech, the most relevant at each moment must be prioritized over the others dynamically.

[12] A reality less and less prevalent due to the still growing culture of game streaming on Internet video channels, which might curiously have their own soundtracks played over the ones from the game being played.

Thus, when working in this new paradigm, the production process must adequate itself. There might be a need for rehearsals to experiment with the sound and the music, or even a prerecorded footage to serve as a general guideline. But, most important of all, it can no longer be delivered as single monolithic audio sample. It must be provided as a set of isolated pieces, which must then be put together during the actual performance. For example, consider the soundtrack for the opening ceremony of an event. With all the variables involved, the professional responsible for this orchestration, while in possession of all necessary samples and proper equipment, has to follow each of the pre-established cues to start, stop, and mix each sound in time, improvising when necessary. In order to remember all the cues, they must be discussed and documented beforehand – possibly similarly to the one done in the traditional production process. Here, the main difference between usual live performances and digital games is that in live performances the one responsible for real-time control is a sound professional, while in digital games that job is historically left to the programmer, since he or she is the only one who has sufficient access to the game source code, which in turn is the digital equivalent to the performance equipment except for the fact that it must be *programmed beforehand* and not *controlled during the execution*.

3.3 Digital Games

Games, in their digital form, are technically *soft real-time interactive agent-based computer simulations* [Gre14, Chapter 1]. The "interactive" part implies that the user does not provide all the necessary input beforehand, nor does he or she need to wait until the process execution finishes obtaining a complete output. Whenever the user receives a stimulus from the game, he or she makes a decision based on it and informs the program of that decision, typically through the use of dedicated controllers (like joysticks or gamepads) or common peripherals (like the mouse and the keyboard). Then, the game processes the consequences of that action along with its internal mechanics, characterizing the "computer simulation" part, which mostly regards interactions between virtual entities of the game world, hence "agent-based" simulation. After updating the simulation to the next stable state, the game generates the corresponding stimulus to be presented back to the player, thus starting the interaction cycle anew. This data exchange between game and player repeats itself *continuously* during the application execution, generally with a rate of 30 to 60 Hz, and hence the "real-time" part. "Soft" here refers to the fact that games are not strictly real-time systems, in the sense that they are not critical systems where a failure to meet the real-time requirements may lead to physical harm to human beings[13].

The game interaction cycle goes on until an end condition is met, like the user closing the application window. In multiplayer on-line games, the generalized structure we just described may be more complicated internally, but the overall flow of interaction remains the same. The resulting output in games is usually a mixture of images being shown in screen monitors and sound being played from loudspeakers or earphones. In other words, sound is one of the means through which the player can understand what is happening in the virtual universe he or she has engaged with. For a more in-depth discussion on what a game is on a more conceptual level and

[13] Nevertheless, the quality standards of the game industry still demand high and consistent performance speeds for a title to be considered a professional product.

independently of its (non)physical form, we highly recommend Schell's book, *The Art of Game Design: a Book of Lenses, Second Edition* [Sch14].

Section 3.3.1 details some aspects of the game development process in the same way we have detailed the traditional soundtrack production workflow back in Section 3.2.1. Then, in Section 3.3.2, we delve into the software architecture of digital games, which will be complemented in Section 3.3.3 by a discussion of the main technologies involved in the development process. Finally, Section 3.3.4 will bridge computer games implementation and real-time soundtracks, pointing out key algorithms and data structures involved in making the features we want a reality in digital games.

3.3.1 Development Process

There are many ways to start a game development process, which are, in turn, influenced by the motivations behind it. Typically, we are talking about a company or studio who wants to sell a new product. This means there is a target audience with a particular need that the game is supposed to satisfy. There are many entertainment aesthetics a game can provide through its mechanics and dynamics [HLZ04] to meet that need, all of which support the overall theme of the game [Sch14]. Once all of this basic parameters are set, the team designed for the project can start brainstorming concepts and ideas for the game. They must experiment with them until an acceptable projection for the game is found, and then the team focuses on turning that vision a reality. The brainstorming and prototyping part of the process is known as the *preproduction stage*, and the implementation focused part, *production stage*. Finding the most appropriate time to shift development from one stage to the other is one of the great challenges in this industry. As the project comes close to a game capable of offering a mostly bug-free and complete experience, development moves to the *release stage*, where the team must put everything together into a single distributable package. After the release, the project may still go on by launching periodic patches for fixes or even new features that could not go into the initial version of the product.

Even if all these stages macroscopically resemble a waterfall-like development process [Roy70], in the sense that it is a one-way production pipeline, each stage can and should [Sch14, Chapter 8] be done via iterative development[14]. Thus, the preproduction stage can iterate over the game concepts and features, using prototypes to evaluate the resulting experience; while the production and release stage can iterate over the implementation using commonplace agile methods [BGM+01]. In all stages, each iteration prioritizes tasks that will mitigate product risks the most. Validating this in games means being able to verify whether a certain feature or mechanism implemented at a given point in production is really aligned with the intended game experience. There is no way to know if a certain soundtrack piece is matching gameplay without play-testing it, for instance. On that regard, it is important that composers and sound designers be able to experiment with their created content *from inside the game* as much as possible.

The professionals involved in the development of a digital game can be grouped into four great fields, each of which we present minimal representatives for[15]:

[14] Some teams actually do go back from production to preproduction, leading to some infamous game projects that last ten or more years to be released – some indeed worth the wait.

[15] In practice, specially in small teams, professionals share multiple roles in the development process.

- **Programming** – computer scientists, software engineers;
- **Art** – graphic artists, sound artists, writers;
- **Design** – designers, scripters, quality assurance; and
- **Production** – producers, directors, product owners.

Our main interest here lies in the "sound artist" type of professionals together with the "programming" field, since they both are the intended user groups of our middleware. We will address the workflow of sound artists for the rest of this section, and programmers in the following ones.

A game soundtrack can involve from a single sound designer to whole teams of professionals with a diversity of specializations. For the musical pieces, the team might employ composers; for sound effects, a Foley artist, etc. Performers and orchestras can contribute recorded songs, and voice actors might provide the material for in-game dialogues. At some point, most of these assets must pass through the sound designer, responsible for distributing and mixing them into the final samples the game software will load and play.

That final step is the one we are most interested in. Compared to other software applications, games are very peculiar when it comes to shipping the binary executable for user consumption. All the 3D models, textures, sprites, sound, music, text, voices, and even stages, items, and other game-specific content must be provided as separate loadable assets and packaged together with the program (because of the need for data-driven design – see Section 1.1.1). To keep track of all these files, some studios go as far as to employ specialized resource management databases [Gre14, Chapter 6]. From the game code, each of the assets is referred to using some sort of identification (like the path string to its location in the file system), then loaded at run-time as needed and eventually released to make the memory space it occupied available again. During the time it is loaded and in use by the game, the code may apply a series of operations over that resource. For instance, we might request a specific walking movement from an animated 3D mesh of a character. Some resources may even be programs themselves, like shaders and scripts. In this regard, when the soundtrack is stored solely in the form of sample files, the programming interfaces normally available for operations over them are those of simple DSP effects: volume control, band-pass filters, pitch shifting, etc.

Going back to the sound designer's work, if the sound resources are all sample-based, this professional must either leave them as close as possible to their final intended playback – since the available run-time operations are limited – or talk it over with the programmers to figure out what effects can be applied via code – and then the result will no longer be under his or her full control. What game audio middleware applications do (as we will see in Section 3.4) is provide a new representation for audio assets, one that comes with "choreographic data", much like a sprite sheet or 3D mesh animation, to give back control to sound designers. They do so by providing both the interface for creating and editing these resources and the computational means to properly load and execute them from the game application without further dependency on programmers. Middleware systems like this effectively *extend* the set of supported formats of the resource management module in the game software, which allows all related professionals in the team to pour new kinds of assets into the project.

3.3.2 Software Architecture

Since all digital games are bound by the cyclic behavior described in Section 3.3, their computational implementation must abide to this paradigm as well. This is done with an endless loop that can only be stopped from the inside. It serves as the basic structure for all the execution flow of the application. Not only that, but it is also responsible for guaranteeing the real-time expectations of the media. In game programming, this architectural pattern is known as the **Game Loop**, a universal solution to the player-game interactive cycle [Gre14, Nys14]. Each iteration or *frame* of the loop is commonly divided in three phases:

1. Poll input events;
2. Update the game simulation to the next stable state; and
3. Present the current state through graphics and sound.

This organization makes explicit the input and output parts of the game code, a fact we will take advantage of when explaining our middleware implementation in Chapter 5. Figure 3.3 illustrates a simple and effective implementation of this variation of the pattern where a synchronization mechanism aligns the simulation steps with "real world" time to protect the game from hardware-specific processing speeds. However, in a more general scenario, there could be many other sub-systems that make the game work as a whole. For instance, there could be artificial intelligence, physics, networking, and video streaming sub-systems, some of which make the concepts of input and output blurry at best.

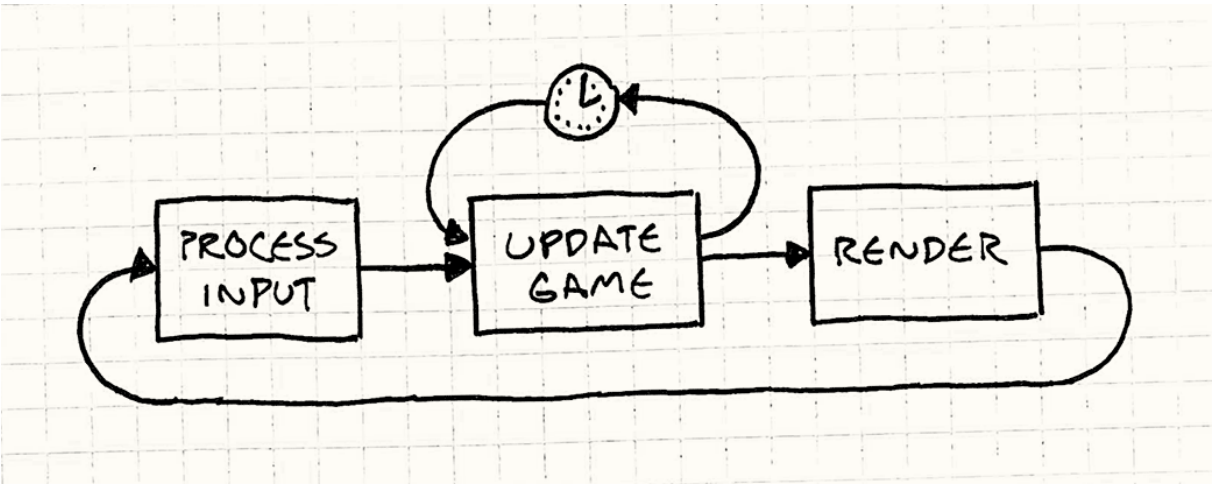


Figure 3.3: A flow diagram of a simplified fixed-frame game loop as proposed by Nystrom [Nys14].

Every iteration, then, the Game Loop requests the services of these sub-systems in order, each with their own demand for real-time scheduling and performance care[16]. In that regard, we can broadly divide the sub-systems of a digital game in two groups according to how they synchronize with the Game Loop.

[16] Multithreaded implementations can either feature parallel Game Loops, each responsible for a different set of sub-systems, or be used to improve the performance of individual frames – for instance, using the Graphics Processing Unit (GPU) of the platform to perform large-scale physics simulation.

1. **Idle-frame synchronization.** The sub-system should be serviced as often as possible, that is, whenever the process is "idle" and can spare the CPU time for it. Depending on what the sub-system function is, it might require information about *how much time has passed since its last service*. An example is graphics rendering, which produces better effects the faster it updates the output, as long as it matches the refresh rate of the display device.

2. **Fixed-frame synchronization.** The sub-system should be serviced between uniform time intervals to remain robust. To achieve this, the Game Loop must wait or skip this service step whenever the time is not right to execute it. A typical example is the physics sub-system, which becomes more robust towards floating point precision issues when the time intervals used vary the least.

In the simplified example from Figure 3.3, the "Process Input" and the "Render" steps use idle-frame synchronization, while the "Update Game" step uses fixed-frame synchronization. We illustrate this in Figure 3.4, where the diagram shows how the "Process Input" and "Render" steps occur more often than the "Update Game" step. This is the case when the game applications does not demand too much CPU time: rendering is fast and may happen most of the time, while game state updates must retain their synchronization consistency.

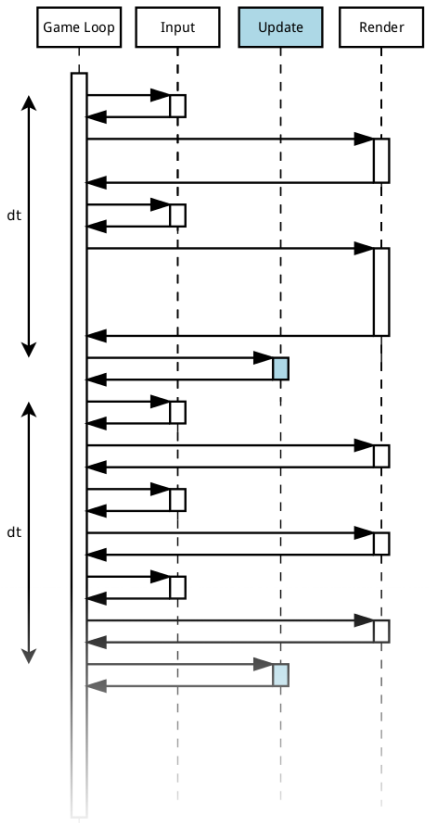


Figure 3.4: A UML sequence diagram of what could happen in the Game Loop from Figure 3.3. The synchronization mechanism must ensure that fixed-frame processing steps occur between similar time intervals.

In very practical terms, this essentially means that each game sub-system comes with a corresponding routine (or method, in Object-Oriented Programming) for requesting its service in the current Game Loop iteration. When the sub-system is idle-frame synchronized, this routine requires a parameter through which the programmer must inform the time since its last execution, since it is the Game Loop that dictates time progression for the software. On the other hand, if the sub-system is fixed-frame synchronized, the routine requires no such information. To illustrate this in explicit code[17], an example of an idle-frame synchronized service would be

```
void GraphicsSubSystem::render (double dt);
```

while an example of a fixed-frame synchronized one would be

```
void PhysicsSubSystem::simulation_tick ();
```

[17] We use C++ for code examples in this thesis. The reason is explained in the next section, Section 3.3.3.

As such, a typical game architecture consists of a core engine, where the Game Loop resides, and a set of sub-systems that must be managed across game frames. Each sub-system, in turn, may have its own internal organization to handle its tasks. A graphics sub-system likely comes with an internal data structure of a tree to represent affine transformations between 3D meshes, for instance. In Object-Oriented Programming, the game architecture could be directly translated into a series of classes, one for each sub-system, and a centralizing `Engine` class to aggregate them all. There are, of course, many other classes that might be needed in the game implementation, specially to provide interactions between each sub-system. For more information on these game programming patterns, we recommend the "Game Loop" and "Component" chapters from Nystrom's *Game Programming Patterns* book [Nys14].

Here we can also see how digital games differ from other interactive applications that rely on endless loops in the base of their architecture too. Since games are *soft real-time simulations*, as much as they are not critically time-bound, the demand for fluid interactive experiences directly impacts the underlying architecture requirements. If the frame frequency is not high enough, the user will not be able to perceive an appropriate feedback of his or her input, or incongruencies between the simulation state and the rendered effects will become all too evident, likely compromising the game experience.

From all this discussion, we can see that the part in a game architecture responsible for audio playback should be a dedicated sub-system like all others. Every time its service is requested, it should compute all relevant audio samples and send them to the device sound card. Gregory's *Game Engine Architecture, 2nd Edition* has a chapter entirely dedicated to audio implementation in digital games [Gre14, Chapter 13]. Taking all this into consideration, it is our belief that a real-time soundtrack sub-system should be idle-framed synchronized. In fact, if it is not, then there is a risk that its service is not requested often enough, causing time gaps in the game execution where there are no audio samples to be played. This either causes abrupt silence surrounded by crackling artifacts, or the endless repetition of the last provided samples, both of which are usually regarded as not very acceptable gaming experiences. Thus, by knowing how much time has passed since the last request, a game audio sub-system can calculate how many samples are needed to compensate for that time difference, process them, and then properly play them through the sound card.

However, a few issues arise when we talk about real-time game audio. For instance, a common optimization in sample-based audio is to eagerly pre-compute all the samples in a given sound file and have them all sent to the sound card as soon as possible. This has the advantage of eliminating any chance of the samples not being ready when the time to play them arrives. On the other hand, if some in-game event happens that is supposed to affect the soundtrack and change what is going to be played, then either we wait for the queued samples to end – which incurs in latency for the player – or we throw away the pre-computed samples and send new ones, assuming that is possible – and we waste the resources spent on processing the samples earlier. Because of this, real-time soundtracks benefit more from lazy approaches, such as the one proposed by Weiner [WL01]. The key question is how many samples should one compute in advance for the audio that will be played until the next game frame. In previous work [MK16], we have observed that the *LÖVE* framework (previously mentioned in Section 1.1.2) restricts data

transfers to the sound card to blocks of 4098 (potentially stereo) samples, and that this causes a latency of approximately 93 milliseconds at a sample rate of 44100 Hz. If we are talking about sound effects, that is well within the perceptible threshold of a regular player. When implementing the audio sub-system for a game using idle-frame synchronization, one should remain aware of these latency issues [LK04].

3.3.3 Tool and Technologies

Since the main aspects of the general structure of games and the types of digital assets involved (like textures, music, and 3D meshes) are mostly the same across titles, it is to be expected that a number of development tools have been released during the last decades to simplify the process of making this sort of entertainment software. Some of them are *on-line*[18] tools, in the sense that they work alongside the game application, that is, they are part of the game executable (possibly thanks to one or more dynamically linked libraries). Others are *off-line* tools, meaning that they are part of the game production pipeline, but not part of the final product. There are also tools that are *on-line* and *off-line* at the same time. The most important examples of on-line tools are **game engines** (or **game frameworks**[19]), which we will discuss soon. Many programming libraries that provide access to lower-level resources of the computer are another common example of on-line tools (like the already mentioned `libpng`). As for off-line tools, we have already mentioned a few:

- ***Blender*** – 3D modeling and animation
- ***Aseprite*** – pixel art painter
- ***MuseScore*** – WYSIWYG music score composer (Figure 3.5)
- ***LMMS*** – DAW-like electronic music composer (Figure 3.6)

These are tools used to produce game assets, which is one of the parts of the game production pipeline. Other off-line tools whose byproduct are not assets include code versioning systems (such as *Git*[20]), team management helpers (like *Trello*[21]), continuous integration systems (such as *Travis-CI* [22]), among many others. Aside from the programming oriented ones, off-line tools in game development are there to allow non-programmers to input their work into the game application without having to touch the code, and thus these tools require intuitive and polished graphical user interfaces, in contrast to the Command Line Interfaces (CLI) and APIs programmers are used to (and in fact often work more efficiently with).

Game engines have a particularly important role in the game industry, since they not only accelerate the production of new titles, the effectively provide the possibility of game making to those who otherwise would never be able to. In very simple terms, a game engine is an

[18] Not to be confused with the concept of being connected to some remote application through the Internet.
[19] It might be debatable whether these two terms really mean the same thing, but in this thesis we treat them so.
[20] https://git-scm.com/
[21] https://trello.com/
[22] https://travis-ci.org/

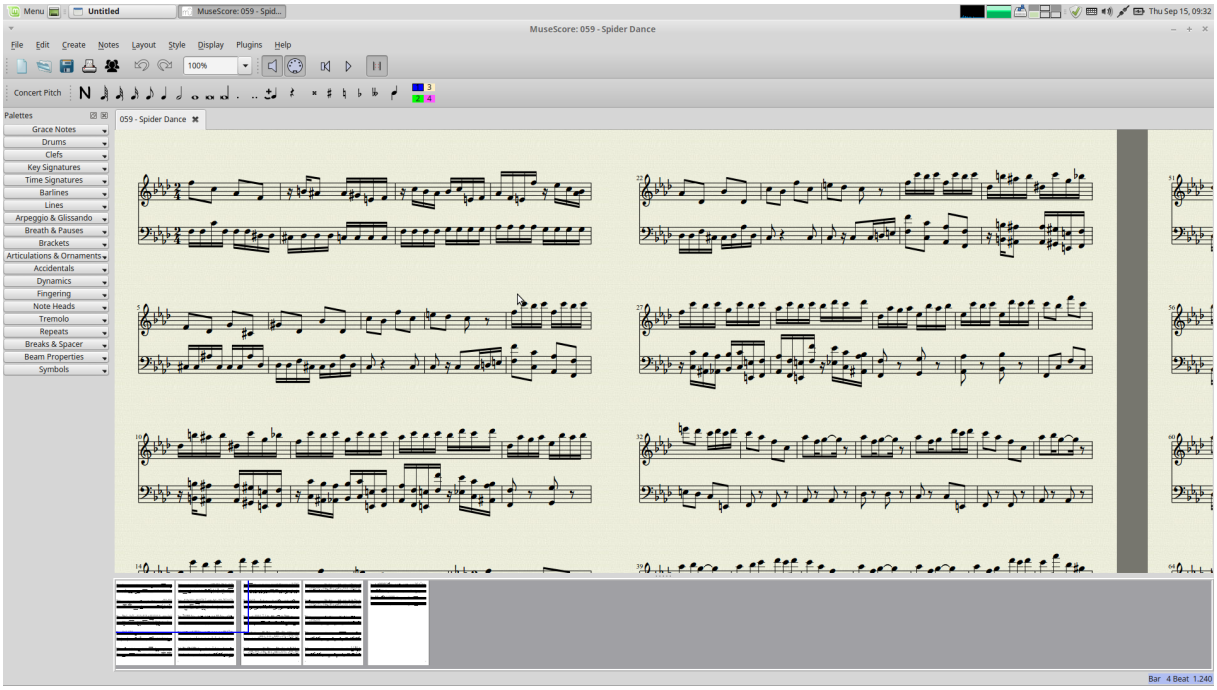


Figure 3.5: A screen capture of the user interface from *MuseScore*, a WYSIWYG music score composing tool.

intentionally decoupled part of a game software that you can reuse to develop different titles, typically of a similar genre, simply by switching *only* the "missing", title-specific part. The more we leave in the engine, the less effort we need to make other games with it, but the more specific the engine becomes. The title-specific part could be composed by either extensions to the source code – like inheriting a class and implementing methods to dictate how the game uses the engine features – or the consumption of other domain specific file formats – like descriptions of game stages, items, and cut-scenes. Engines that work by extending the code are closer to Johnson's definition of a software framework [Joh97], while the ones that expect the game to be provided as a data input conform to Gregory's proposal that "arguably a *data-drive* design is what differentiates a game engine from a piece of software that is a game but not an engine" [Gre14].

Generally, data-driven engines belong to the hybrid group of game development tools we mentioned, providing an off-line visual interface for building game content in addition to its on-line features. *Unity* (which we already mentioned on Section 1.1.2) is a famous example of this, but in the last year we have seen a growing community around a free software alternative, *Godot*[23]. This kind of game engine is the main reason so many indie studios are able to publish titles even if they lack the programming expertise present in larger, older studios. Figure 3.7 shows the off-line interface of *Godot*. An extensive list of game engines, both commercial and free software, can be found on-line at Wikipedia[24]. There are some specialized game engines that do not to provide *all* the backbone necessary to develop a game, but rather are designed to perform

[23] http://www.godotengine.org/
[24] https://en.wikipedia.org/wiki/List_of_game_engines

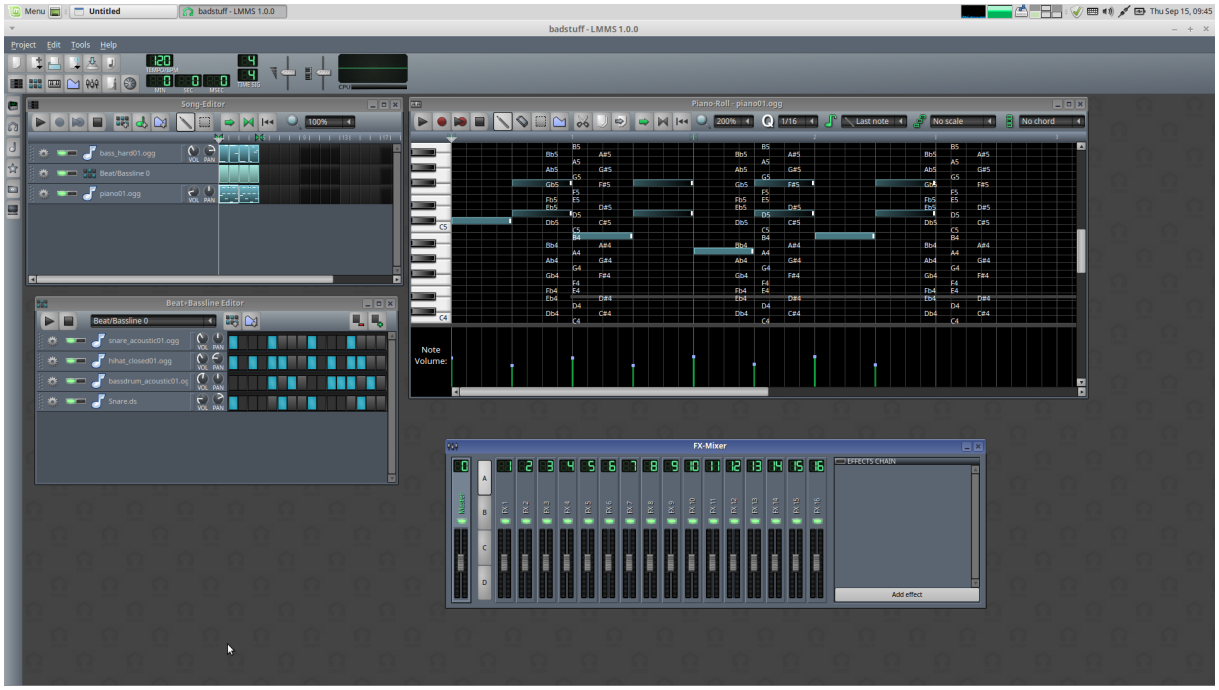


Figure 3.6: A screen capture of the user interface from *LMMS*, a DAW-like electronic music composing tool.

a very specific task very well. A classic example is the *Open Graphics Render Engine* (*OGRE*)[25] – as its name suggests, it deals only with the graphical rendering part of a game engine.

However, engines alone cannot make all kinds of games by themselves. For instance, if we need a 3D model of a character, we are most likely to rely on more specialized tools such as the already mentioned *Blender*. Then, we have to *import* that model into the game project through the engine interface (be it off-line or on-line). In the case of 3D models in particular, each tool has its own format (e.g. *Blender* writes models to `.blend` files). Unless the tools can export the models to a widespread format – such as *Wavefront*'s *OBJ*[26] or *Khronos*' *COLLADA*[27] – the engine might require additional plug-ins to be able to import meshes from each kind of 3D model file format. When there is one or more prevalent formats for a type of media (like WAV for lossless uncompressed audio sample files) it is much easier for the developers to include support for importing them in the engine.

When the asset formats are too specific, the team might prefer to use specialized programming libraries that are already able to load from and save to these formats. This could, evidently, demand the licensing of third-party software. Sometimes, the digital content creation tool is already distributed with the necessary binaries to load their exported files into other programs. Such is the case of some game audio middleware systems we will analyze in Section 3.4.

On that matter, one of the key features of game audio middleware is their hybrid off-line and on-line nature. This is directly related to what we discussed back in Section 1.1. In the traditional,

[25] http://www.ogre3d.org/
[26] http://www.martinreddy.net/gfx/3d/OBJ.spec
[27] https://www.khronos.org/collada/

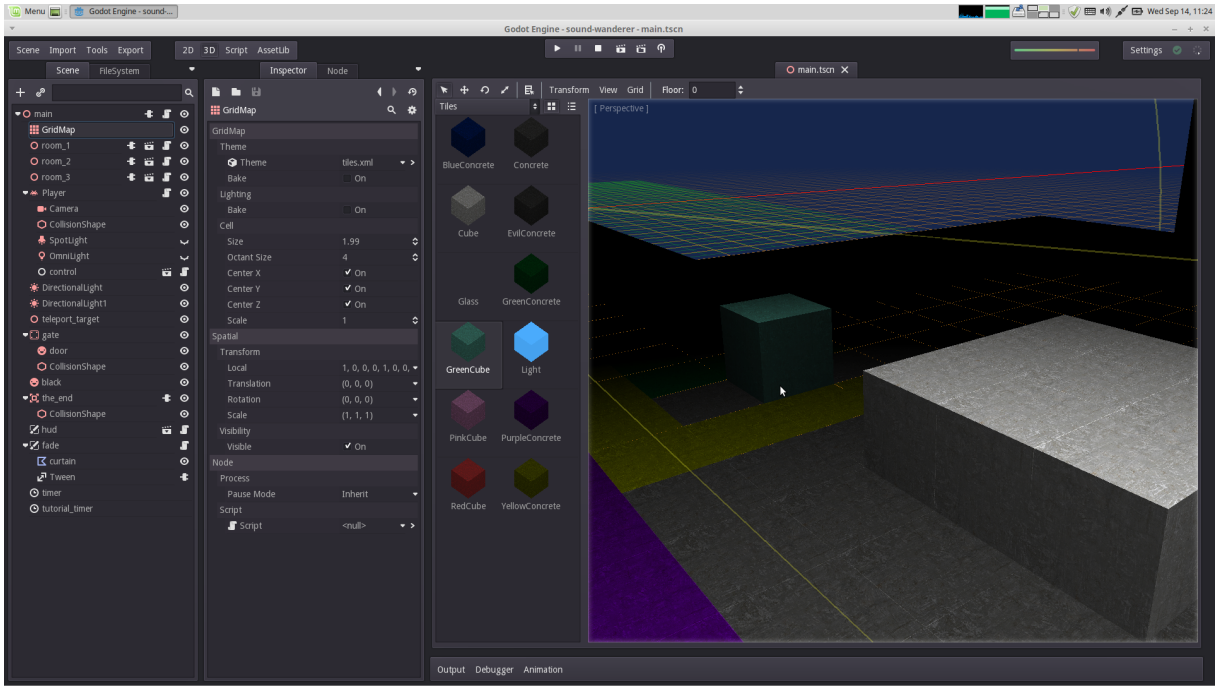


Figure 3.7: A screen capture of the user interface from the *Godot* engine. The game being worked on is *Sound Wanderer*, a prototype we developed for our research. Read more about it in Section 6.1.

sample oriented approach, there is a disconnection from the off-line and on-line tools used. One would use, for instance, *LMMS* to produce a certain music track for the game. While inside *LMMS*, the composer or sound designer has full control over how the music sounds. But then he or she has to *export* it into a common format, say, OGG. Then, an on-line tool combination like `libogg`[28] plus *OpenAL* could *import* that music file into the game. This means that all real-time effects available are limited by what the on-line tools can do, and not the original off-line tool which the music was created with, besides the already mentioned problem that now only programmers can control how the music is played. When we use a middleware, since its off-line and on-line interfaces are integrated, the same effects the sound designer has access to will be present during game execution, and both importation and exportation can be done without losing information from the original content, since we do not need to convert it into a third-party format. All of this, however, would not be a problem if there was a widespread real-time interactive sound format (an issue we will address in Section 4.3).

One problem we have not discussed yet regarding the vast set of available game development tools is platform support. Starting with game engines, there has been a historical predominance of restrict support to *Windows PC*. This has changed in recent years, a phenomenon generally attributed to *Valve*'s adherence to *Linux* support (remarkably their release of *SteamOS*[29]) and the growth of the mobile market under the *Android* platform. Even then, there are still numerous limitations. The *Unity* game engine, which is arguably the most widespread engine in the market as of this writing, has on-line support for all relevant game platforms, but no off-line support

[28] https://xiph.org/ogg/
[29] http://store.steampowered.com/steamos/

for *Linux*. That is, you can develop *Unity* games *for Linux*, but not *in Linux*. Something similar occurs with other off-line tools, like *3D Studio Max*[30]. Examples that go against this tendency include the already mentioned *Blender*, *Aseprite*, *OGRE*, *Godot*, *LMMS*, *MuseScore* and many others. It is possible to say that *Linux*-based development of games is completely feasible nowadays, even if there are still lingering discrepancies. Lastly, one very important fact in terms of technology compatibility when developing games is that the prevalent programming language in the industry is C++ [Gre14, Nys14].

3.3.4 Algorithms and Data Structures for Real-Time Audio in Games

In the early days of computer games, sound hardware was very limited. There was not enough memory to store big enough samples, so most sound cards provided support only for programmable synthesizers [KC08]. This caused incongruencies between the audio played for the same game in different machines, since each card had its own set of available timbres. On the other hand, the only way to insert music and sound effects into games was to write them as procedures – there was no way around procedural audio. Alas, the limitations upon the aesthetic and fidelity qualities of possible sounds drove the industry towards improving their hardware.

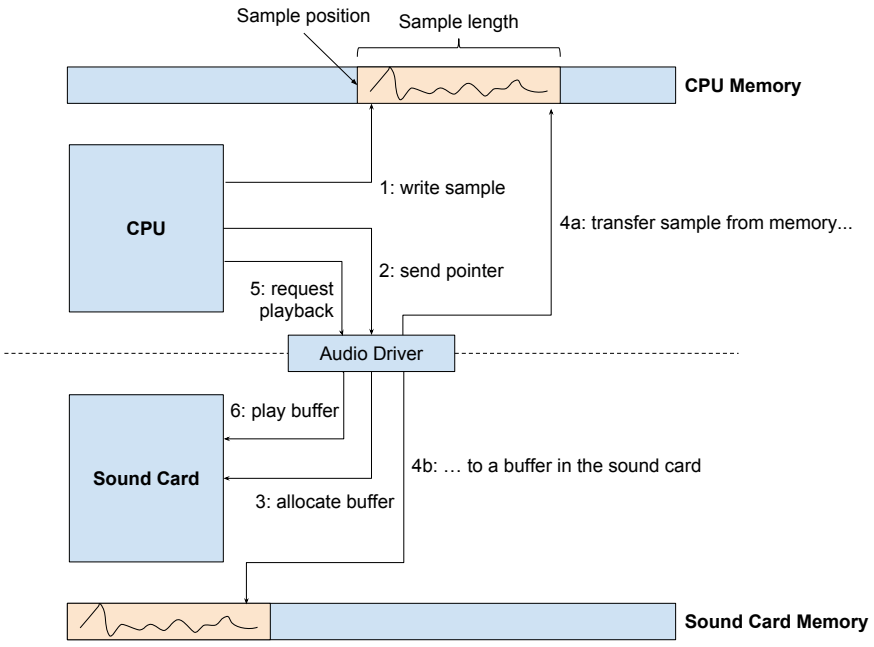


Figure 3.8: A typical process of playing a given audio sample in the sound card with the help of the corresponding driver.

Modern sound cards are now practically restricted to the playback of digital audio in sample form. With enough memory to store audio data, all sound can be produced via software (or even through the GPU) then sent to the sound card, whose sole function is to play it accordingly. As

[30] http://www.autodesk.com/products/3ds-max/overview

we explained in Section 3.1, this is done by specifying the sample format used, then passing the sample sequence itself. In more concrete terms, this means providing a pointer to the position in memory where the sample array we want to play is. Once sent, the sound driver is supposed to copy these bytes to *buffers* in the memory of the card. There, their playback can be further manipulated if the hardware has support for additional audio controls (generally only *volume*) as long as the driver or low-level programming library used exposes such features. Figure 3.8 illustrates this protocol. Using *OpenAL*, the actual code would look like Listing 3.1.

```
1  // let us skip OpenAL initialization for now
2  int16_t sample[1024];
3  loadSample(sample); // load from file or synthesize
4  // create a buffer on the sound card -- memory has not been allocated yet!
5  ALuint buffer;
6  alGenBuffers(1, &buffer);
7  // create a sound source -- this is OpenAL-specific
8  ALuint source;
9  alGenSources(1, &source);
10 // send sample to buffer and bind to source -- this allocates the buffer
11 alBufferData(
12   buffer,                              // which buffer
13   AL_FORMAT_MONO16,                    // sample format
14   static_cast<const ALvoid*>(sample), // pointer to sample
15   1024 * sizeof(int16_t),              // sample size (in bytes)
16   44100                                // sample rate (in Hertz)
17 );
18 // request playback
19 alSourcei(source, AL_BUFFER, buffer);
20 alSourcePlay(source);
```

Listing 3.1: Sending audio to the sound card.

As we discussed in Section 3.3.2, to achieve real-time responsiveness, games need to have a finer control of how often and how many samples are sent to the sound card. Now that we know how this data transfer really happens, we can better explain how to do that. Since the card works by allocating buffers and playing them on demand, there are two questions an audio engine must answer when striving for real-time sound:

1. **the size** of the buffers, that is, how many samples to send each time; and
2. **when** to send them, because, once sent, the samples can no longer be changed by what happens in the game, and sending them too late is not good either.

In Section 5.2.4, we present our own choices regarding this issue in the implementation of the audio engine for the middleware. It is important to note that it is possible to queue buffers to be played on the sound card once the current one is done. This buffer queue is very useful, since it dismisses precise synchronization with the playback of the samples. That is, we do not need to know exactly *when* a sample will end being played to then play the next. We can just *schedule* it to be played after the current one. One may even queue multiple buffers this way. The original

motivation for this was for streaming compressed audio, since decoding it is often slow, and with this mechanism the programmer could send chunks of audio constantly instead of having to wait until the entire file is decoded. But the buffer queue can also be used to help us deal with real-time control of the buffers being played on the sound card.

All of this works for a single sound. When we wish for multiple, concurrent sounds to be played, we require a way to send samples to independent *tracks* in the sound card. That means that, when we send a sample buffer, we must also specify which track it is bound to. Then we might even control, say, the volume of each track separately. In *OpenAL*, this is done through the concept of *sources*. The library models a 3D sound scene abstraction where there is a *listener point* and as many *source points* as the programmer demands. Then, previously sent buffers can be bound to a source point, which is itself the only possible target of playback requests in the *OpenAL* API. We showed this in Listing 3.1, but we repeat it in Listing 3.2 with the relevant parts commented.

```
1 ALuint source;
2 alGenSources(1, &source);
3 // *snip*
4 alSourcei(source, AL_BUFFER, buffer); // <-- bind the buffer to the source
5 alSourcePlay(source);                 // <-- request playback at source point
```

Listing 3.2: Using OpenAL Sources.

This multi-track sample playback is commonly referred to as **polyphony**. It comes with an important caveat: even if one is able to play hundreds of sounds at the same time, most humans are not able to discern that many. Thus, one of the challenges of polyphony is to regulate how many sounds are present in a given moment of gameplay, ensuring that the most relevant ones are properly audible to the player. This is one of the many reasons why audio tools need **mixers**.

```
1 // Everything is global for illustration sake
2 int16_t *music_track_explore_version;
3 int16_t *music_track_combat_version;
4 size_t  current_sample_pos;
5 double  mix = 1.0;
```

Listing 3.3: Declarating two variations of the same music track.

With that, we have all the basic tools to start audio programming for games. The next topic is how to use them to produce real-time soundtracks. In the code examples so far, we abstracted away how the sound signal was obtained (code lines 2 and 3 from Listing 3.1). The first step towards real-time sound is to support "nonlinear" signals. That is, to not follow an unchangeable specification of what should be written to the sound card buffers. In that sense, using prerecorded, sample-based audio is "linear" because the samples are predetermined. To be able to modify the samples and achieve variability, we require DSP. Let us simulate the effect in *Faster Than Light* as a very simple example. Suppose we have only one music track with two variations as in Listing 3.3. Then, we could implement `loadSample` as in Listing 3.4. Somewhere in the Game Loop,

we would have to update the value of `mix` across multiple frames whenever the player switches between exploration and combat, guaranteeing the cross-fade effect.

```
1  void loadSample(int16_t *sample) {
2    for (size_t t = current_sample_pos; t < current_sample_pos + 1024; ++t) {
3      sample[t] = music_track_explore_version[t]*mix +
4                  music_track_combat_version[t]*(1.0 - mix);
5    }
6    current_sample_pos += 1024;
7    // Let us skip verification of the track lengths
8  }
```

Listing 3.4: Loading sample as a mix of two samples.

This illustrates a hard-coded implementation of a hypothetical real-time soundtrack, but it shows well how a sample-based approach would work. The system needs to keep track of all relevant samples, their seek position[31], and a number of complementary data regarding how each one is supposed to be played. In the example, there was only one such data: the `mix` coefficient. We could have per-sample volume, pitch-shifting, band-pass filter parameters, etc. The other part of the implementation is making all such variables be affected by the game state and regulated by the appropriate time flow in the application.

If, on the other hand, we want to rely on procedural audio, the implementation becomes much more complex. After all, in order to support procedures that generate audio signals, we might as well implement a Turing-complete engine. In fact, that is what *MAX/MSP* and *Pure Data* do: they are programming languages specialized in real-time DSP applications. That is one of the main reasons why we consider these languages to be of great importance to our research, as we will discuss in Section 4.3. As for their design as a language, they follow a data flow paradigm, which closely resembles functional programming. The programmer works by instanciating logic blocks and connecting the outputs of one to the inputs of another, forming a directed graph where each node typically represents a DSP operation. Then, the audio signal "flows" through the graph, being transformed at each node, until it reaches special nodes that convert it into an analogue signal and play it through the computer speakers or earphones. Not all connections represent signals, though. They can also represent other data types and structures used to *control* how the DSP operations work. Additionally, many of the logic blocks available have an embedded clock counter to properly handle signal manipulation across the time domain. The general flow of the audio signal is then synchronized with the ticks from these clocks. How this is done depends on the tool itself – we will explain how *Pure Data* does it in Section 3.4.5.

As an example of a DSP-based soundtrack structure, let us consider a very simple scenario. We want a background music track that keeps playing a note at a fixed tempo but whose pitch varies according to the player character's health. We will use a sinusoidal wave for the timbre. Assume we have the following logic blocks available:

- **Sinusoidal Oscillator (DSP block):** Generates a sinusoidal signal according to the frequency it receives as input;

[31] In the example we used one for both samples, but in practice each one might require its own counter.

- **Square Pulse (DSP block):** Generates a single square pulse when triggered, outputs zero the rest of the time;
- **Multiplier (DSP block):** Multiplies its two input signals into the output;
- **Digital-Analogue Converter Output (DSP block):** Sends the input signal to the computer speakers or earphones;
- **Chronometer (Control block):** Sends a trigger command to its output at a fixed tempo;
- **Player Health Input (Control block):** Sends the player character's health as a numeric value whenever it changes; and
- **Transform to Frequency (Control block):** Transforms any number it receives as input into valid frequency values.

We can then achieve our intended soundtrack structure by following the layout shown in Figure 3.9. By combining the Square Pulse block with the Chronometer block, we are able to produce a rudimentary sequencer. Then, we use the Player Health Input block to obtain the relevant data from the game state and map it into note pitches using the Transform to Frequency block. The transformation it actually does could be something as trivial as using the player's health as MIDI values and directly converting them to frequency in Hertz, which is then used by the Sinusoidal Oscillator block to produce the desired sound wave. The resulting soundtrack is likely unpleasant at most, but serves the purpose of illustrating the elementary possibilities of DSP data flow structures.

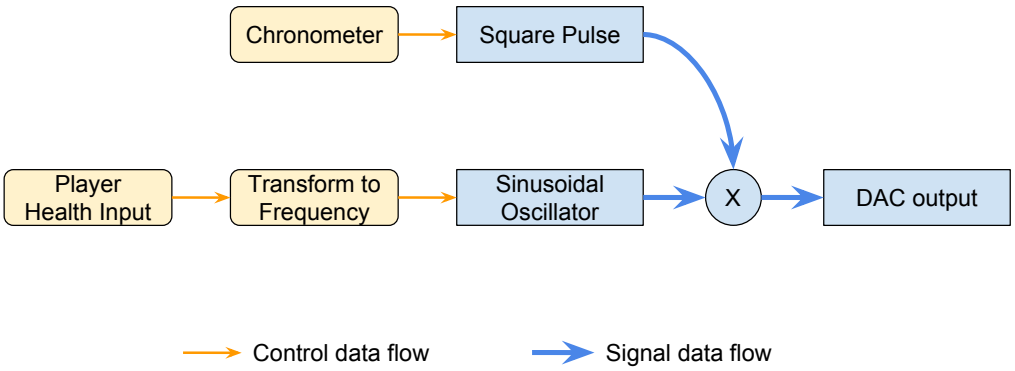


Figure 3.9: An example of a real-time soundtrack made using a hypothetical DSP data flow language.

Notwithstanding, asking the users to program everything they need from scratch is not a viable option. Even if a real-time soundtrack tool supports procedural audio through a full-fledged DSP language, not all sound designers can afford to understand the lower-level details of how digital audio and DSP works, even if what they do learn becomes a very valuable asset

in their skill set [Sco14]. As such, it is important to have higher-level interfaces for designing the soundtrack of a game. For example, if the music tracks were to be played by synthesizers, a possible structure to facilitate that would be a MIDI sequencer. It abstracts the generation of synthesized note pulses into musical events listed in a timeline, and could come with the ability to *jump* from one point of the music track to another, or even determine the notes themselves on the fly, composing the theme algorithmically or following some stochastic process.

This bring us to one of the dilemmas in this research area. Even when using procedural audio, we can still rely on samples to many degrees. That is good because sound designers can provide these samples utilizing the sample-based traditional tools they are used to. Besides, as we mentioned before, juggling a few samples is often enough to make very adequate real-time soundtracks. But when we do need to go beyond, specially with music tracks, it becomes harder and harder to avoid using some sort of symbolic representation, like MIDI. The problem is that either it relies on synthesized sounds (which is a severe aesthetic limitation) or instrument sample libraries, which could be quite expensive to acquire if one wishes for higher fidelity sounds. There is also no widespread tool for composing real-time MIDI music. Worse, there is no established file format for exporting and importing this kind of media, whereas there are many excellent format options for prerecorded samples.

3.4 Related Technologies

Having discussed all the basic concepts we need for our research, we dedicate this section to the exposition of currently available or historically important real-time audio middleware for digital games. Again, this is not supposed to be an exhaustive list. We will use this analysis in Chapter 4 as a referential basis for our own middleware design, and later in Chapter 5 when we base our implementation decisions on what has been done in similar technologies. The first tool we analyze is *iMuse*[32] [Luc94], one of the first middleware for real-time soundtrack in games, very different from all modern systems. Next, we analyze three modern commercial middleware systems: *Wwise*[33], *FMOD Studio*[34], and *Elias*[35]. The last tool we analyze is *Pure Data* itself, which is not a middleware but is a very important reference for our research, as will be discussed in Section 4.3. At the end of this chapter, in Section 3.4.6, we include a table comparing the five studied technologies.

3.4.1 iMuse

iMuse [Luc94] is a patented in-house software system developed by *LucasArts* in the early 1990s. It was "a computer-based music and sound effects system in which music and sound effects are composed dynamically in response to the action of a directing system". *LucasArts*' motivation for developing *iMuse* was the frustration that came out of the difficulties encountered during the production of the soundtrack of one of its game titles, *The Secret of Monkey Island* (*LucasArts*, 1990) [Moj08]. One of the studio's composer, Michael Land, took the lead in the project, which

[32] https://en.wikipedia.org/wiki/IMUSE
[33] https://www.audiokinetic.com/products/wwise/
[34] http://www.fmod.org/
[35] https://www.eliassoftware.com/

evidently guided the tool towards a musician-centric design. It is one of the first real-time game audio middleware in the industry, having been used in all subsequent adventure games from *LucasArts*. However, since the system was patented, *LucasArts*' games were the only ones to ever use *iMuse*.

Figure 3.10: A screen capture from *Monkey Island 2: LeChuck's Revenge* (*LucasArts*, 1991) [Luc91], the first game to ever use the *iMuse* system.

System Description

The software architecture of *iMuse* is divided in two parts: a **Sound Driver** and a **Directing System**. Besides these two components, *iMuse* also requires a **Composition Database** from which to draw the sonic contents of the game. All of it works thanks to a specific media format they invented, or rather, derived from the MIDI standard. General interaction between these components goes as follows. The Composition Database stores all composition sequences for the game using their extended MIDI format, while the Sound Driver is able to retrieve and process these sequences, forwarding the appropriate events to the available synthesizers (hardware-based ones at the time). The Directing System was responsible for sending commands to the driver according to the game context, and the driver, in turn, changes something about how it is playing the composition sequences from the Database. These changes include volume and instrument channel control, but the most important change was to jump between positions in the current composition sequence. Figure 3.11 illustrates this architecture in a diagram.

Essentially, they used two of the reserved "system exclusive" MIDI commands available in the protocol to provide time branching features to the format. They were named **Hook** and **Marker** commands. Both require an identification number parameter, which serves as a flag name. Whenever the Sound Driver process one such command, it checks if the flag name is marked as active inside the Directing System. If it is, the corresponding action happens. Otherwise, the commands are ignored altogether. Hook commands have several types, each of which causes a different effect. A few examples are changing the volume of a particular instrument or of the whole sequence, turning instrument channels on or off, transposing the pitch of a particular instrument or of the whole sequence, and jumping to a different location in the sequence. Markers, on the

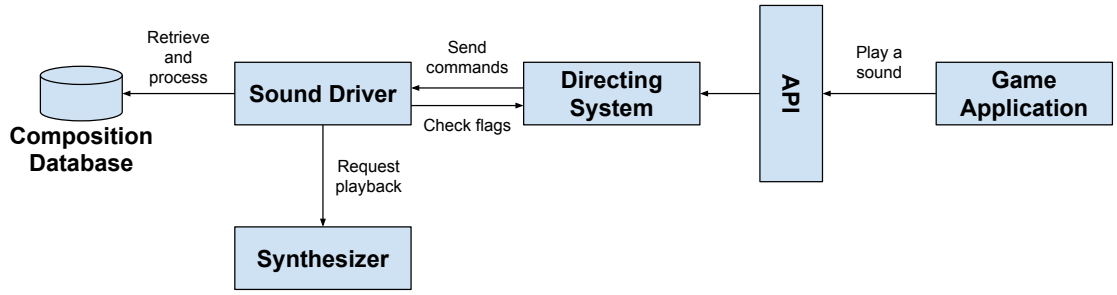


Figure 3.11: The architecture of the *iMuse* system.

other hand, have no type. Their flag name must be referenced in a different structure within the Directing System to make anything happen when the driver processes them. All the composition sequences must be stored in the Composition Database to be accessible to the Sound Driver and, thus, to the system as a whole.

The **Directing System** is the high-level module for controlling the soundtrack. It can issue commands to the Sound Driver, causing it to play the corresponding composition sequences at the right times, with the appropriate effects. It comes with an API through which the game programmer is able to request these operations. The Directing System is also responsible for keeping track of the currently activated flags, which can be manipulated through the API. Whenever the Sound Driver encounters a Hook or Marker command, it consults the Directing System to know how to proceed, as we explained above. One of the responsibilities of the Directing System is to also manage **Command Queues**. Once registered, each of these has a trigger mapped to a sound and a flag name, besides the queue of commands itself. The commands can be any of the already available ones in the API. As can be expected, this sequence of commands is executed whenever the sound driver reports a Marker command whose flag name corresponds to the one in the queue trigger and the sound sequence being played is also the one specified there. This mechanism works much like a callback function being scheduled for execution whenever a certain sound-flag pair is detected in the composition sequences, except it is more limited since only API commands can be used.

Critical Analysis

Being arguably the first of its kind, *iMuse* is an important milestone in the area of game soundtracks in general, and particularly revolutionary at the time for its effective approach to real-time sound and the way it instigated game development to properly face the sonic engagement of players. Nevertheless, it was designed to fit into the technological reality of the end of the last century. MIDI is no longer the preferred choice of game sound designers, and hardware-based synthesizers have no place in modern personal computers. The fact that the technology is patented obviously

poses as a severe limitation to the use of *iMuse* as well.

There are many design decisions we can learn from, though. Even if digital audio has almost entirely substituted MIDI in games, maybe the idea of using a symbolic representation to introduce real-time operations in composition sequences should not be entirely discarded. After all, the Hook and Marker commands used by *iMuse* came out of Michael Land's demand as a composer for more control over the dynamic behaviors of his creations inside the game. The Command Queues also point to Farnell's proposition on procedural audio, since they are essentially a data structure that stores a procedure to be executed under an audio-oriented instruction set, making them a more flexible and powerful solution than modern *ad hoc* implementations of sample-based real-time soundtrack effects.

As a reference work for our research, *iMuse* provides an example of an effective middleware architecture. It is a specially useful design to handle musical sequences, which comes naturally from using a MIDI-based format. But that is also a limitation, since making more abstractly structured compositions becomes more difficult. So is the dependency on synthesizers, which further restricts aesthetic possibilities. Even though we have seen that sample-based designs are too rigid, that does not mean that samples are to be abandoned altogether. It would be rather interesting to see how to conciliate *iMuse*'s ideas with the rich possibilities of sample audio we have in the present day.

3.4.2 Wwise

Figure 3.12: *AudioKinetic*'s *Wwise* logo [Aud16a].

Developed by *AudioKinetic*, *Wwise* is one of the main game audio middleware systems in the industry currently. Published for the first time in 2006, this tool can be embedded onto all modern game platforms, and many well known games in the market have used it in their development process [Aud15a]. A few examples are the *Assassin's Creed* (*Ubisoft*) and *Borderlands* (*Gearbox Software*) series, *League of Legends* (*Riot Games*, 2009), and *Overwatch* (*Blizzard*, 2016). *Wwise* provides an off-line digital content creation interface and an on-line SDK for the *Windows* and *Mac OS X* operational systems.

The digital content creation interface is an extensive framework for sound designers to work with. By relying on a number of sound object abstractions organized in a versatile hierarchy, its users acquire deep control over the game audio. Essentially, *Audiokinetic* has used an exhaustive interface design: for each possible need in game audio creation, real-time bound or not, there exists an object type in the *Wwise* hierarchy that provides the desired solution. We will describe what those object types are, and how they fit together to make the game soundtrack. The general division between digital content creation interface and SDK follows the architectural pattern we discussed in Section 1.1.2 and illustrated in Figure 1.5.

System Description

Wwise is a project-oriented tool: for every game the user wants to create a soundtrack for, a new *Wwise* project must be started and worked on. Inside a project, there are three great hierarchies: the **Actor-Mixer Hierarchy**, the **Interactive Music Hierarchy**, and the **Master-Mixer Hierarchy**. In each hierarchy, we create and manipulate logical trees whose nodes represent different abstractions in the game soundtrack, and the general flow of data starts at the roots, descends all the way to the leaves carrying control directives, then goes back to the roots bringing the audio itself. The **Actor-Mixer Hierarchy** allows us to create and manipulate Sound Objects, Motion Effects Objects, Containers, and Game Syncs. Sound Objects work like symbolic links to audio samples, so essentially they are the leaves in the hierarchy tree, being the source of the base sounds used to compose the soundtrack. Motion Effects Objects are mostly the same, except they provide signals to be used in motion feedback of game controllers instead of audio signal. Containers simply group many object nodes as a single sound, allowing localization of game titles across different languages. Game Syncs are the most interesting: they are the inner nodes of the tree, and define the behavior of their children. We will describe them in more detail soon. The **Interactive Music Hierarchy** works much like the Actor-Mixer Hierarchy. The main difference is that the leaf nodes are Music Track objects, which can be grouped as children of Music Segment nodes to make a "layered" composition. Aside from that, the Game Sync nodes are still there, being the most relevant structure for real-time effects.

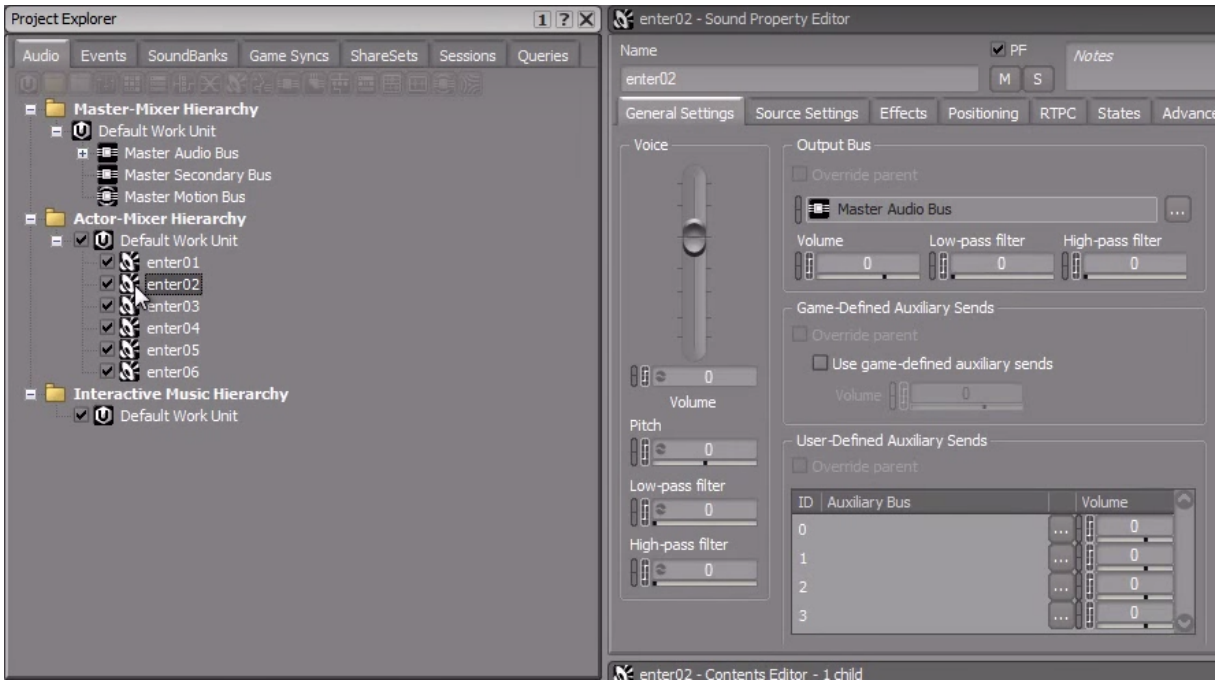


Figure 3.13: Screen capture of a piece of the *Wwise* interface. It shows the Hierarchy explorer, as well as a part of the property editor of the selected node, a Sound Object [Aud16b].

The **Master-Mixer Hierarchy**, on the other hand, is very different from the other two. It controls the flow of audio and motion signals that come from the other hierarchies, forwarding them up the tree until they reach the root – which is the audio master bus of the middleware. The roots from each of the other hierarchies are linked to leaf nodes in the Master-Mixer Hierarchy,

allowing the sound designer to specify how each sound is supposed to be played. Figure 3.14 illustrates an example of how all three hierarchies could be connected.

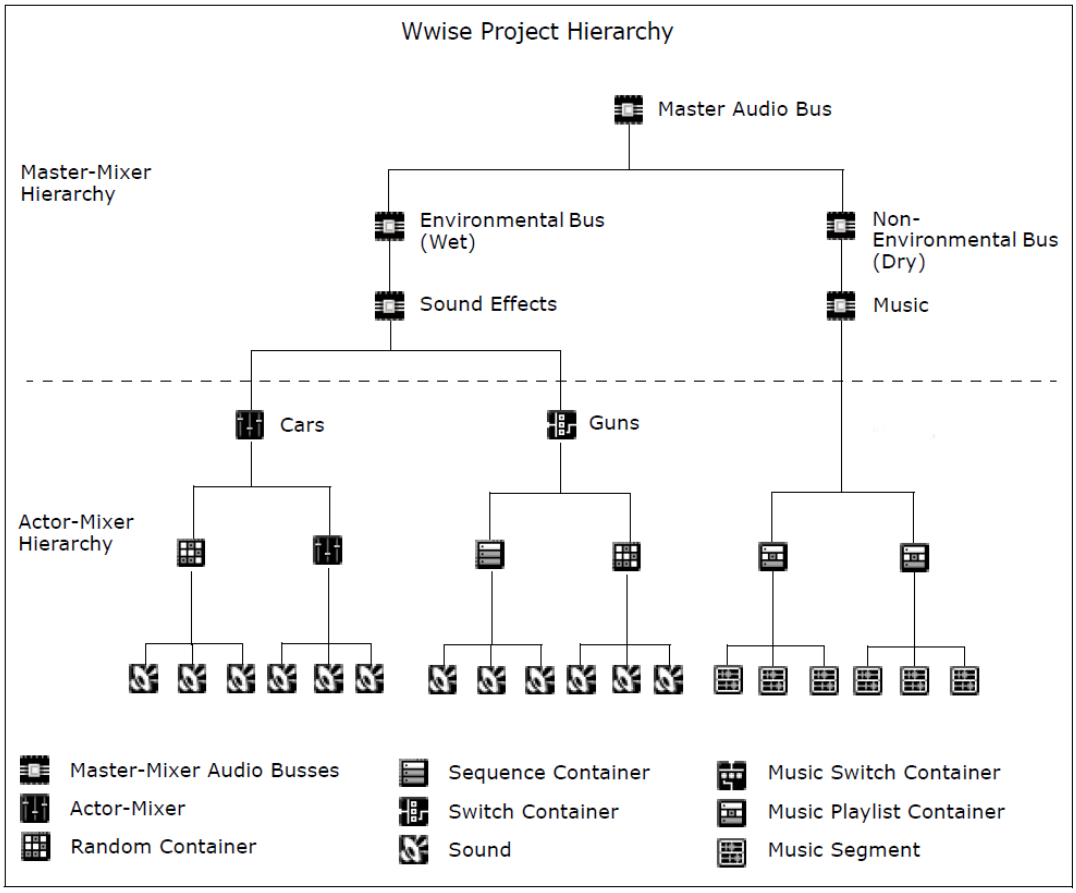


Figure 3.14: Example of object hierarchy in a *Wwise* project [Aud15b].

However, these hierarchies represent only the structure of the soundtrack. To actually execute them to obtain playback, the user needs to specify a series of behaviors, possibly with real-time constraints. For that, there are three main features *Wwise* supports: **Events**, **Game Objects**, and the already mentioned **Game Syncs**. **Events** are basically named hooks triggered by the programmers through the *Wwise* SDK from within the game code. On the sound designer's side, the user can specify two kinds of Events: Action Events or Dialogue Events. Action Events contain a list of commands much like the command queues from *iMuse*. Each command can play, stop or change a property in one or many sound objects across the project hierarchy. Figure 3.15 illustrates an Action Event being executed. Dialogue Events can only play sound objects, but on the other hand they provide a conditional branching mechanism. They map a series of state names to specific sound objects. When the Event is triggered by code, the actual value of the state in that context is checked to know which conditional path to follow in the Dialogue Event list. As the name suggests, this feature was originally intended to support dialogues with multiple voice samples from which to choose from, like when a character responds differently to the player according to a choice he or she made. Figure 3.16 illustrates how Dialogue Events could be used to implement a play-by-play voice in a sports game.

Game Objects are an abstraction within the *Wwise* digital content interface that the sound

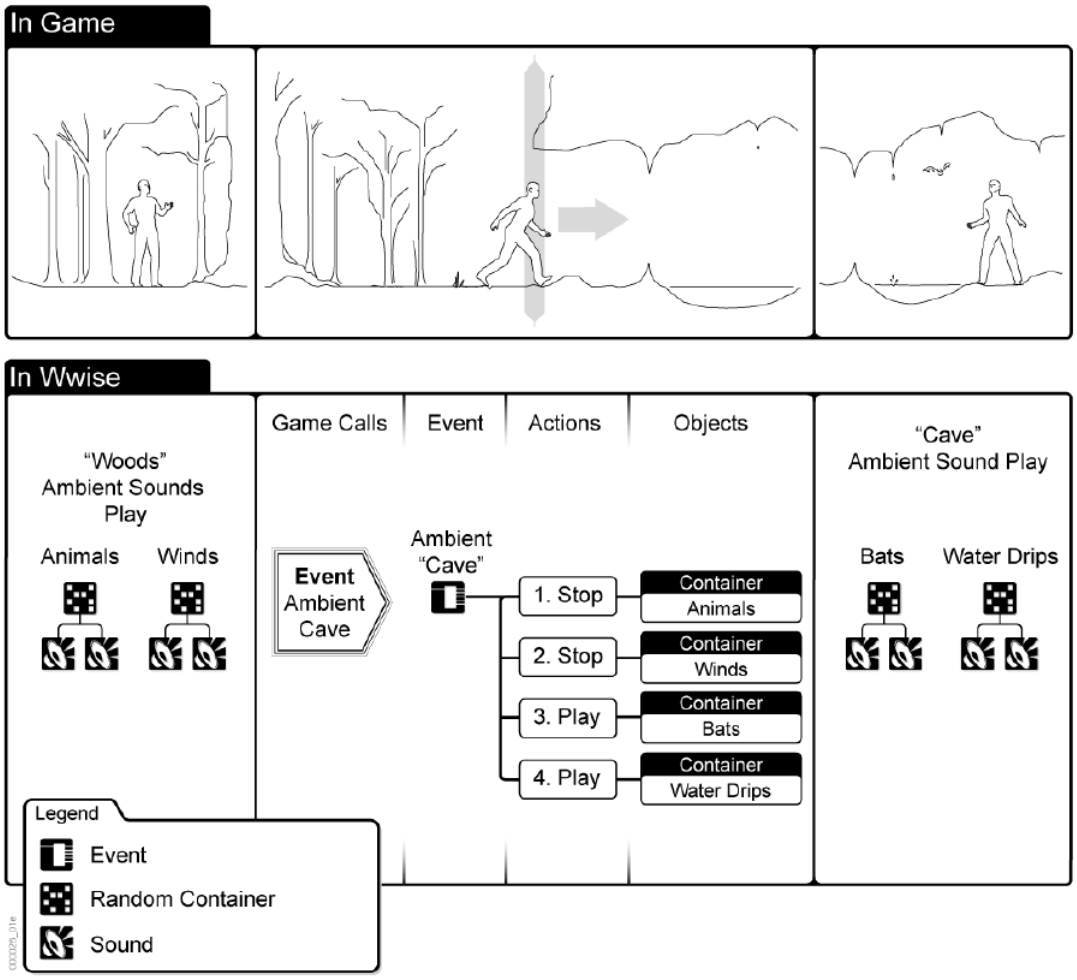


Figure 3.15: Example of an Action Event in *Wwise*. [Aud15b].

designers use to reference possible virtual actors from inside the game. They could be the characters themselves, objects that produce sounds, environment noises, or even User Interface elements with feedback sounds. **Game Objects** come with properties like volume, pitch, position, orientation, and Game Sync information. They must be registered, managed and unregistered by the game code. Many key API functions, like `AK::SoundEngine::PostEvent()`, require or optionally accept a Game Object as a parameter. This makes all the consequent effects of that function call be executed from the context of that Game Object. For instance, it makes any sounds played use binaural effects to simulate the 3D spatialization of the corresponding actor in-game.

Lastly, **Game Syncs** are intermediate nodes in the Actor-Mixer and Interactive Music Hierarchy trees. When "played", each of one them follows a specific, real-time behavior, using its child nodes as building blocks. When evoked with a Game Object as a parameter, they use the Game Sync information it carries as input to the behavior that will be used in that case. The available Game Sync nodes are the following.

- **States**. They can be assigned a number of property presets (volume, etc.), each with a specific name. The State can only have one activated preset at a time. By activating

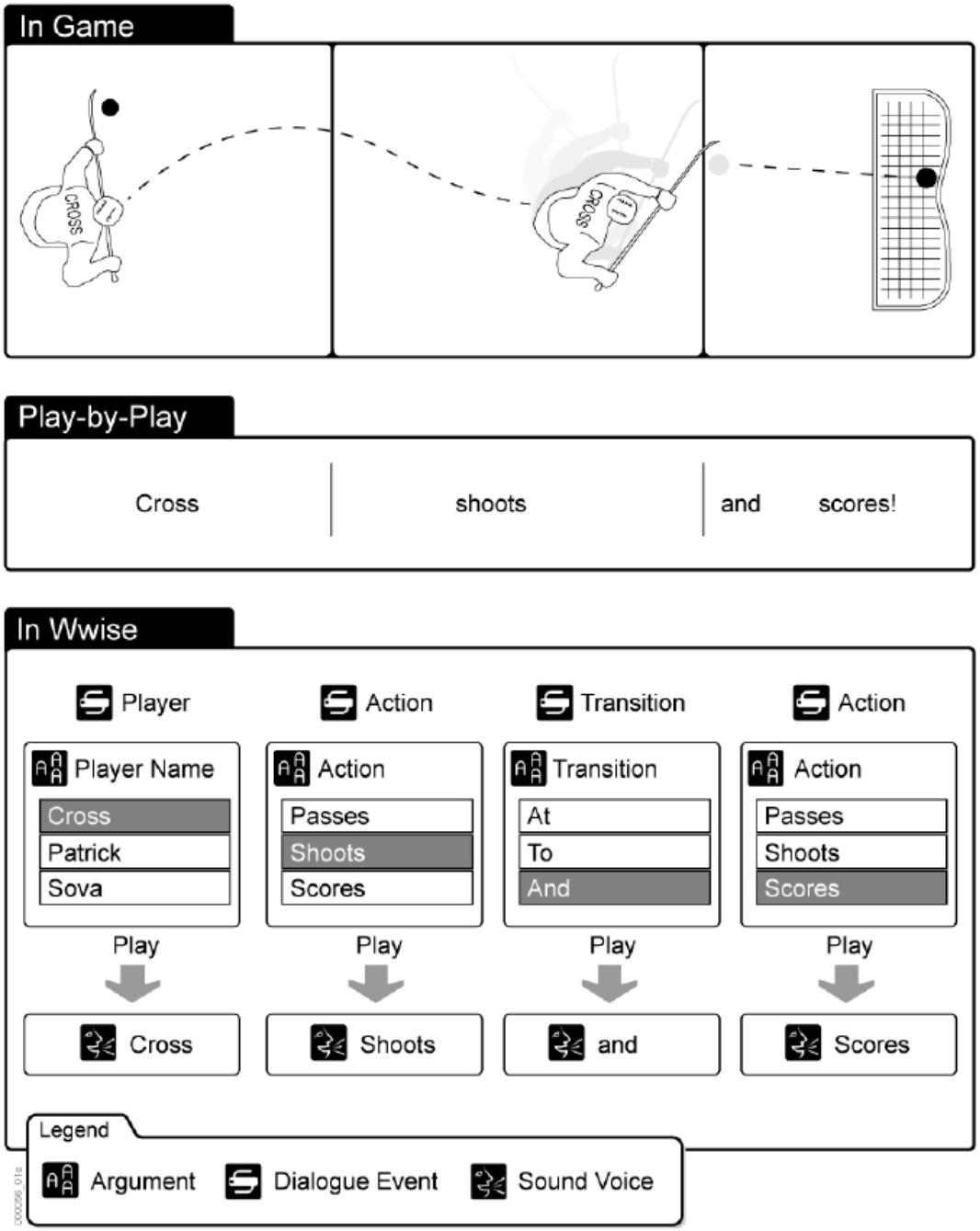


Figure 3.16: Example of a Dialogue Event in *Wwise*. [Aud15b].

one from code, all of its property values override the current values in the corresponding properties of child nodes.

- **Switches**. They are assigned a set of possible state values, each mapping to a different child node. When played, Switches check their current state and forward execution only to the matching child node.
- **Real-Time Parameter Controls (RTPCs)**. As their name implies, they provide a real-time mechanism to control property values in nodes from the hierarchy trees. They do so by being assigned a transformation curve, which is used to map input values obtained

from API calls into the actual property values. This could make a RTPC for the player's character health possibly correspond to a logarithmic increase in, say, the volume property of the affected nodes.

- **Triggers**. They are similar to Events in the sense that they must be posted from game code and require a Game Object to define their playback scope. The difference is that they belong in the Hierarchy tree, and the only action they do is play their immediate children, which can cascade into further behaviors and effects down the Actor-Mixer or Interactive Music Hierarchies.

Once all the desired objects are put into the project hierarchies, they can be exported as **Sound Banks** to different target platforms and languages. This Sound Banks are then loaded into the game through the API of the *Wwise SDK*. The programmers can reference the structures imported either through their name, using character strings, or through their auto-assigned identification numbers.

Critical Analysis

Wwise is a solid alternative for game audio middleware at present, and this is backed by the many respected titles in the industry that have relied on it. Its design approach is very different from usual DAWs musicians are used to. Notably, the notion of time is not clear at first when learning through the first steps of the tool. Even if the user can create a sound object and play with it as much as he or she likes, it is not evident how that is tied to the gameplay. At the same time, this tree-based architecture is powerful and versatile. As long as the user masters the interactions between the sound objects and the game, there are endless possibilities to the combinations and behaviors that can be achieved. Nonetheless, this middleware is ultimately sample-based. It does provide support for custom plug-ins, which allows for lower level DSP-generated audio, but its standard feature set still restricts the sound design of the game to samples and combinations thereof. Besides, it is a commercial product. Its license permits use free of charge for non-commercial projects and a limited use for commercial projects. Anything else beyond that requires payment.

In terms of what we can learn from this technology, the "building blocks" approach we saw in *Wwise* and its success in the industry might suggest that more abstract interfaces, in contrast to the traditional timeline based DAWs and composing tools, are also welcome by sound professionals. The types of Game Syncs and the Event system provide a good reference for what *AudioKinetic* considers important features for a game audio middleware after ten years of experience in this market. It is important that our own middleware considers the benefits of providing similar tools or, at least, the means for the user to achieve equivalent effects.

3.4.3 FMOD Studio

Another proprietary tool that has gained renown in current years is *FMOD Studio* from *Firelight Technologies*. Among the games that have used it, notable titles are *Diablo 3* (*Blizzard*, 2012), *Transistor* (*Supergiant Games*, 2014) [Fre], and *Guild Wars 2* (*ArenaNet*, 2012). Actually, *FMOD*

Figure 3.17: *Firelight Technologies*'s *FMOD Studio* logo [Aud16a].

Studio is the name of the off-line digital content creation interface, which is supported only in *Windows* and *Mac OS X* operational systems. There is another part, called *FMOD Studio Programmer's API*[36], used on-line by the game application to load and play the soundtrack the way the sound designer intended it to. This also follows the architecture depicted in Figure 1.5.

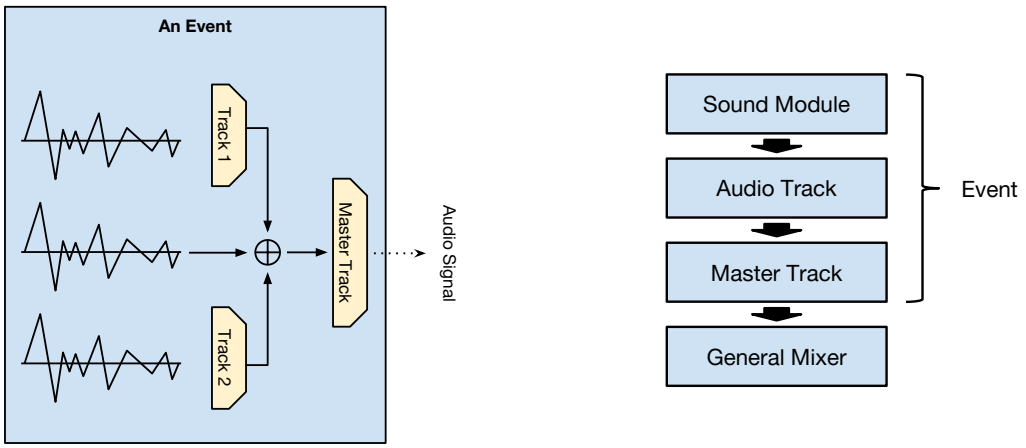
The design approach from *FMOD Studio* leans more towards DAW-like tools. As will be explained ahead, its main creation structure is based on a timeline interface where sounds can be positioned to be played following a clear sequence. It also provides a series of filters and effects whose "look and feel" closely resemble that of sliders and mixing tables present in physical devices sound designers are used to. The interface also focuses on cleaner layouts, darker shades and more colorful widgets.

System Description

FMOD Studio offers a single structure to work with, named **Event**[37]. The user workflow is centered on creating and manipulating Events through the *FMOD Studio* interface, then exporting them as **Event Banks**, a binary format the *FMOD Studio Programmer's API* will later load into the game. This format is platform independent, making it possible to export it in the sound designer's machine then import it in the game application running on the programmer's computer without any problems. Every Event possesses a timeline that can be layered in multiple tracks, one of which is the Master track and is always there by default. It is possible to place **Sound Modules** (composed of one or more sample audio files) on the tracks and, when we play the Event, each Sound Module is played when the time counter reaches them along the timeline. Since the tracks exist in parallel over the timeline, multiple sounds can be played at the same time. Besides, each track comes with an effects pipeline of its own, allowing individual effects to be selectively applied to them. In the end, the signal from all tracks is sent through the effects pipeline of the Master track, and the result is forwarded to the *FMOD* general mixer (which can be set up separately). This internal structure of Events is depicted in Figure 3.18a, while Figure 3.18b shows how that structure fits into the greater picture of the signal flow inside *FMOD Studio*.

So far, this interface design works much like traditional DAWs, indeed. As can be expected, individual Events can be evoked from game code through the *FMOD Studio Programmer's API*. That essentially brings us back to how sample-based audio works in games, except the export-import protocol is much more stable and user-friendly. Now we need to address how *FMOD* approaches real-time soundtracks. There are two complementary mechanisms it uses for this purpose: **Parameters** and the **Logic Track**. **Parameters** are the only entry point for real-time control given to the programmers besides evoking Events themselves. Each Event can be assigned any number of Parameters; each of them has a name, a value range, and an initial value.

[36] It is further divided into a separate *Low Level Programmer's API.* for specific use cases
[37] Similarities aside, it clearly has a different meaning than the Event from *Wwise*.



(a) The structure of an Event. **(b)** The standard flow of audio signal.

Figure 3.18: Workflow overview of *FMOD Studio*.

Then, from the API, programmers can change the current value of that parameter anytime from instanced Event references.

Inside the Event, every Parameter can influence its playback in two manners. The first is through the Automation feature. Any property of an Event can be automated by a Parameter, which means the value assigned to that property is determined by the current value of the Parameter, following a curve that can be constructed through the interface. The most common property is volume, but every effect present in the pipelines of each track count as automatable properties too, including low-pass filter cut-off frequencies, reverb settings, etc. As an example, we could implement *Faster Than Light*'s soundtrack transition using *FMOD Studio* as follows:

1. Create an Event for a given music theme;
2. Add two tracks to that Event – one for the main instruments, another for the drums and other sounds that only play during combat;
3. Place the corresponding samples in each track;
4. Add a Parameter "`Game-Mode`" ranging from zero to one and an initial value of zero;
5. Assume a zero value in the Parameter indicates "Exploration Mode", while a value of one means "Combat Mode";
6. Add Automation to the second track volume; and
7. Draw a curve to that Automation from the "`Game-Mode`" Parameter, making the volume start at silence and increase to its default level as the Parameter goes from zero to one.

Now, whenever the programmer changes the value of the "`Game-Mode`" Parameter in the game code from zero to one, the drums part starts playing, while doing the opposite causes it to become silent.

The other way with which Parameters can influence the soundtrack is related to how they interact with the **Logic Track** we mentioned before. As its names implies, one could treat the Logic Track as an additional, special track in the Event timeline. However, instead of storing Sound Modules, it stores logical markers that resemble the Hooks and Markers from the *iMuse* composition sequence format. The main available logical markers and their corresponding functions are as follows, with Figure 3.19 illustrating an example.

- **Marker.** The simplest of logical markers, used to literally mark or tag specific points in the Event timeline. Other logical markers may reference Markers to achieve their objectives.
- **Tempo Marker.** This logical marker indicates that from this point onward the Event timeline follows the given tempo. It divides the timeline in bars according to that tempo, making the placement of other logical markers and Sound Modules snap to the closest bar.
- **Loop Region.** A logical marker that makes a certain continuous sequence of the timeline be repeatedly played, unless an explicitly associated Parameter value is *outside* a user specified range when the time counter passes the end of the Loop Region.
- **Transition.** A logical marker of this type always references a Marker. When the time counter reaches this Transition, if an explicitly associated Parameter value is *inside* a user specified range, the time counter *jumps* to the referenced Marker.
- **Transition Region.** Works exactly like a Transition, except it may trigger the transition at any moment as long as the time counter is inside the specified region. Alternatively, the region may be divided in sub-regions, each of which checks the jump condition at its end.

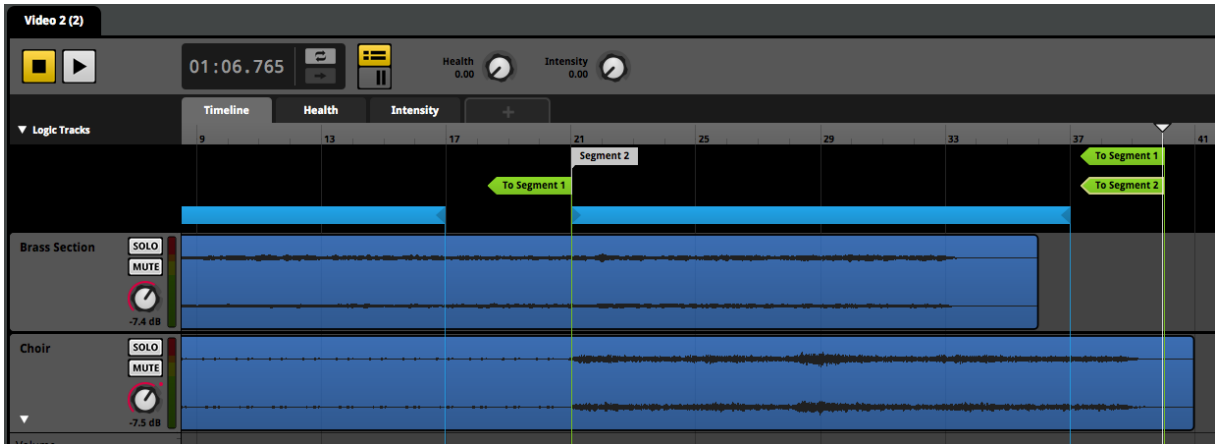


Figure 3.19: An example of an Event construction using the Logic Track [Tec16a]. Here, there are two Loop Regions (in blue), which can be left only when the value of the "Intensity" Parameter (seen as a tab) is within certain ranges. We can also see a Marker and three Transitions which help reinforce this structure.

Thus, each Parameter can be additionally used to control *which* parts of an Event are being played. A very simple example that relies on this feature set would be to design a game stage whose background music is divided in sequential parts, each corresponding to a region in that stage. Then, we could make the stage progression of the player character map to a Parameter

in *FMOD*. Using Loop Regions, Markers and Transitions, we could make the background music jump from and to each of its parts according to where the player character currently is in the stage. If the music was composed to account for such transitions, we would have achieved an adaptive music real-time soundtrack for that game. This might not be as simple as it seems, since unpredictable transitions can easily compromise the music integrity. *FMOD* helps mitigate this problem by providing a **Transition Timeline** feature. By double-clicking Transitions and Transition Regions, the sound designer is shown a special interface to create and populate a separate timeline that exists only between the transition point and its destination Marker. This gives the sound designer more control over how each transition is going to play out. Figure 3.20 shows an example of a Transition Timeline and how the *FMOD Studio* interface cleverly represents this in an intuitive and convenient way.

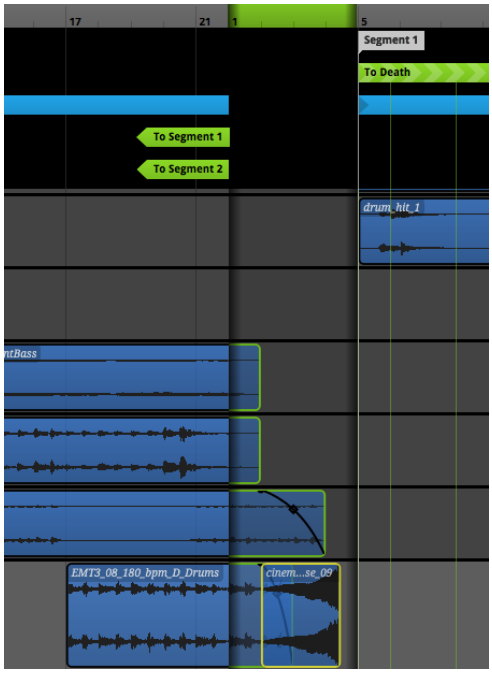


Figure 3.20: An example of Transition Timeline construction in the Logic Track [Tec16b]. Notice how the Transtion Timeline appears to be "hidden underneath" the main timeline, allowing the sound designer to place Sound Modules in it without polluting the rest of the Event with scattered structures.

Critical Analysis

FMOD Studio presents a design approach very different from *Wwise*, leaning more towards the design from *iMuse*. It tries to appeal to sound designers by using widgets familiar to those already skilled in conventional DAWs and by basing the structure of the soundtrack on timelines. This arguably makes it more user-friendly than *Wwise*. *FMOD*'s versatility, however, demands that the user keeps his or her Event timelines organized so that more complex soundtrack structures can be achieved in projects of larger scale. Using a linear design to produce nonlinear content can only go so far in terms of usability. This does not limit *FMOD*'s ability to produce very elaborate soundtracks though [Sch15a, Sch15b]. It does share a common issue with *Wwise*, the issue being that it is sample-based too. While its Logic Track brings the technology closer to conditional jumps in programming languages (as happened with *iMuse*), is still cannot help the

fact that its atomic building blocks cannot be divided further than whole sample sequences. Like with *Wwise*, the *FMOD Studio* license permits use free of charge for non-commercial projects and a limited use for commercial projects. Payment is required for anything else beyond that.

Regarding our research, this tool serves as an example that game audio middleware has many possibilities in terms of architecture design. It uses one single abstraction to provide all the features *Firelight Technologies* believes the users will need. Also, as we already said, it reinforces the idea of relying on sequencers we observed in *iMuse*. Then again, since it still focuses on a sample-based mindset, we are left with the challenge of projecting how the Event structure could work with synthesizers, for instance. Lastly, it is yet another case where the tool did not shy away from using programming features in its interface, essentially giving a "goto"-like tool to sound designers.

3.4.4 Elias

Figure 3.21: *Elias Software*'s logo [Sof16a].

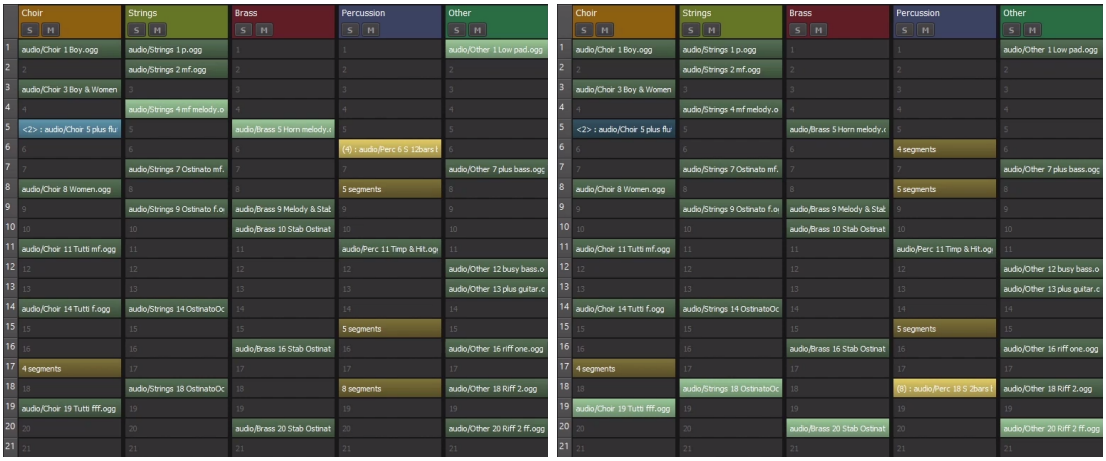
Elias – which stands for *Elastic Lightweight Integrated Audio System* – from *Elias Software* is a more recent addition to the game audio middleware industry. The corporation was founded in 2014, and released the first commercial version of *Elias* in 2015. Like its two predecessors, *Elias* comes in two parts: *Elias Studio* and *Elias SDK*. *Elias Studio* is the off-line audio content creation interface, and *Elias SDK* is the on-line API for embedding the technology into games. The *Elias* approach to game soundtracks is directed more exclusively at composers than at sound design in general, with adaptive game music being its core feature. Being a newer technology in the industry, there are fewer titles known for deploying it, the most relevant being the 2014 remake of *Gauntlet* (*Arrowhead* Game Studios, 2014).

System Description

Since *Elias* focuses on adaptive game music and composer users, its design is very straightforward and optimized for the pertinent use cases. As with *FMOD*, there is a single structure to work with, namely **Themes**. They represent a music composition that is to be played during the game, delimited by a looping sequence of bars, which are in turn dictated by a user-specified tempo. As with other modern middleware, it is sample-based. What will be actually played during each bar is determined in real-time, and the middleware uses the knowledge of their sizes and tempo to produce seamless transitions between each possible playback. Every *Elias* project can have any number of Themes, and each Theme has a set of **Loop Tracks** and **Stinger Tracks**. Both these sub-structures are responsible for describing the many adaptive music possibilities in the Theme, and are detailed further ahead. Themes also have a state value, called **Level**. It is represented by a single integer number, ranging from 1 to a per-Theme user defined maximum. This Level

abstracts the narrative "intensity" of a game at a given time, and allows the soundtrack to escalate accordingly. Most of the composer's work in *Elias Studio* is to adjust and fine-tune how Themes react to Level changes.

Loop Tracks behave like tracks in conventional DAWs with the caveats that they are enclosed in a looped, fixed length bar sequence and can be assigned content on a per-Level basis. For instance, assume a composer made three variations of the strings section in a particular music piece for the game. To work with *Elias*, each of these variations must be of similar length (or combinable into same-length sequences) and each of them must be placed in a corresponding Track Level. Let us say this particular theme has a Level range of twenty. The composer could assign the less intense variation to Level 1 of the strings Loop Track, the mildly intense variation to Level 8 of the same track, and the most intense variation to Level 14 of that track. Then, as the Theme Level varies during gameplay, the *Elias* music engine will cross-fade between the variations accordingly, and the overall loop of the Theme will retain its integrity. When a change to a Theme Level causes Loop Tracks to switch the sample being played, *Elias* recognizes and names it as a **Transition**. Figures 3.22a and 3.22b display the Loop Track interface in *Elias Studio* at two different Theme Levels. Notice how the currently active Track Levels are highlighted.



(a) Screen capture at Theme Level 6. **(b)** Screen capture at Theme Level 20

Figure 3.22: *Elias Studio* interface showing active Track Levels [Sof16b].

Stinger Tracks, at a first glance, look very similar to Loop Tracks. They can be assigned tempo-aligned samples at different Theme Levels. The difference is that Stingers must be explicitly triggered for playback. When a composer does so, he or she must specify which Stinger Track is to be played as well. The idea is that Stingers are music pieces that can be played at any time during gameplay, generally with the intention of matching a significant event in the narrative, like a new enemy appearing or the player falling into a trap. They are played over the current Loop Tracks using the sample assigned to the appropriate Theme Level. There are two ways in which a Stinger can be triggered: manually or through a Transition. The manual way, as can be expected, is achieved by pressing the "play" button in *Elias Studio* or calling a single routine in the *Elias SDK*. The other way – Triggering a Stinger though a Transition – is possible because *Elias* can register named **Transition Presets** in a Theme, each of which specifies a number of parameters indicating how Transitions between Theme Levels should occur and, in particular,

whether a Stinger should be played during that Transition.

Elias also provides an automation feature called **Action Preset**. It is a very simple structure that allows the composer to specify certain soundtrack behaviors *a priori*. For instance, they can create an Action Preset called "Enter the fortress" which increases the current Theme Level to a certain value using a certain Transition Preset. That way, the tool dismisses the need for programmers to know exactly how each event in the game should affect the soundtrack. It is enough for them to know the names of the Action Presets instead, and properly invoke them from the API at the appropriate moments.

Lastly, contrary to what *Wwise* and *FMOD* do, the *Elias SDK* is not capable of sound playback. It merely provides the resulting signal array to callback routines registered by the programmer. The host application is the one responsible for doing whatever it needs to make this data produce the corresponding sound.

Critical Analysis

Elias brings a very different middleware proposal to the table. The developers chose to focus on a single demand instead of embracing all of soundtrack creation facets. This allowed them to produce a very optimized workflow that solves that one problem very well. As long as the composer is working on a musical theme driven by the intensity of the game narrative, *Elias* is, indeed, the most proficient tool one can use right now to achieve that. However, for game music in general, it still imposes many restrictions. This is related to the discussion in Section 1.1.3 about how every technology comes with some sort of aesthetic restriction, one way or another. In the case of *Elias*, this is reflected in many of its design choices, like using sample-based composition, structuring Themes as loops and forcing adaptive behavior control down to a single parameter (the Theme Level). Nonetheless, that does not compromise the validity of the tool. If there is a present demand for this specific kind of game music composition, then *Elias* is a technology the industry needs if it desires to improve real-time soundtracks in games.

In that regard, there are a number of ideas our middleware project could draw from here. For instance, sometimes the most practical choice is to embrace loop music in a game, like in puzzle games where usually the player might spend dozens of minutes in the same stage. In theses cases, having a loop engine ready to swap in and out different variations of the theme can save the development team a considerable amount of time and effort. Tempo-aligned stingers is also a promising idea. Additionally, *Elias* shows us that there are aesthetic effects one cannot achieve but by relying on well crafted and composed samples. As much as we understand the benefits of procedural audio, this is yet another evidence that there is no reason to neglect sample-based audio entirely.

3.4.5 *Pure Data*

As we have mentioned, *Pure Data* is not a game audio middleware, although it has been used for game audio [Pau03]. It is a visual programming language which follows the data flow paradigm and is intended for developing real-time DSP and multimedia applications. It was developed by Miller Puckette in the 1990s as an open source version of the *Max* programming language,

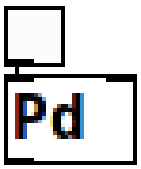


Figure 3.23: *Pure Data*'s logo [Puc16].

which was later derived into the *Max/MSP* language. *Pure Data* has many uses, among which interactive music stands out. It has an active community of programmers and musicians alike, even though its uses are usually more common in experimental music and academic research than in popular commercial music. Since Puckette's original release, there have been a number of different distributions of *Pure Data*. The versions distributed by Puckette himself are normally referred to as *Pure Data* "vanilla". Examples of other distributions are *Pure Data Extended*[38] and *Pd-L2Ork*[39]. In this thesis, we will always refer to *Pure Data* "vanilla".

Pure Data is provided as a stand-alone application whose interface allows the creation of DSP programs called **Patches**. These Patches are interpreted by the *Pure Data* application itself, being possibly executed in real-time and on-the-fly as the programmer or sound designer "writes" the Patch. Since this is the intended workflow of the tool, it was not developed to be embedded in other applications, and thus its use on games is rather limited. Thanks to `libpd`[40], there are a few less known titles in the market that do use *Pure Data* for the game soundtrack. An example of that is the iOS game *Wave Trip* (*Lucky Frame*, 2013) [Lib16]. Here we give a very brief description of the core *Pure Data* features along with some others that are particularly pertinent to our research. There are, however, other more adequate reference materials for those who wish to deepen their knowledge in the language. The *Pure Data* manual is one such alternative, but we also recommend a selection of chapters from Farnell's *Designing Sound* [Far10, Chapters 8-14].

System Description

Every Patch in *Pure Data* consists of a bi-dimensional **Canvas** where **Objects** can be placed at any integer position and then **Connections** can be traced between them. Objects are represented by a filled rectangle (with slight visual variations) and can be treated as programming functions. Each Object has a name, a list of creation arguments, a set of zero or more **Inlets** and a set of zero or more **Outlets**, all of which are visible in the rectangle representation. The Object name determines the Object type, which in turn determines how it uses its creation arguments and how many Inlets and Outlets it has. **Inlets** stay on the top edge of the Object, and work as the input gate for the Object function. **Outlets** stay on the bottom edge and work as the output gate. **Connections** can only be made from an Outlet to an Inlet, and the receiving Inlet must be compatible with the **Connection Type**. There are two Connection Types: **Message**

[38] https://puredata.info/downloads/pd-extended
[39] https://puredata.info/downloads/Pd-L2Ork
[40] http://libpd.cc/

Connections (thin lines) and **Signal Connections** (thick lines)[41]. The Outlet determines the Connection Type, since the user always starts a Connection by clicking on the Outlet of an Object, then dragging the other Connection end to an adequate Inlet. Figure 3.24 illustrates a classic "Hello World" patch in *Pure Data*, where a sinusoidal wave is played.

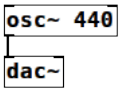


Figure 3.24: A simple *Pure Data* Patch that connects the sinusoidal oscillator object `[osc~]` to a digital-to-analog converter `[dac~]` through a Signal Connection. This effectively produces the sound of a sine wave at 440 Hz.

The Connection Types determine how and when data flows between the Objects in a *Pure Data* patch. **Message Connections** must be triggered by the Object whose Outlet it connects. When this happens, the results of the parent object are sent through the Connection to the corresponding Inlet in the child object. This might, in turn, trigger other messages to be sent through the Connections in the child Object. This process behaves in a depth-first manner. Some Objects can trigger their Message Connections "spontaneously", by being clicked by the user (like the `[bng]` Object), by receiving a packet from the network, etc. There is also a special kind of Object called **Message**. It has a single Outlet, no Inlets, and can be assigned a character string. Whenever it is pressed, a Message Object sends its content to its Outlet, possibly starting a whole chain of Messages down the Patch. Figure 3.25 shows an example of a Message Object and a Message Connection. A special kind of Message, simply containing the string "bang", is often used to trigger effects in child Objects without really sending any real information.

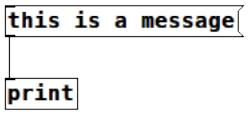


Figure 3.25: A Message Object connected to the `[print]` Object. Clicking the Message sends the string "`This is a message`" to the child Object, which prints it in the *Pure Data* console.

All basic arithmetic and logic operations are already provided as native Objects in *Pure Data*, as well as an extensive collection of DSP Objects such as `[noise~]` (white noise generator), `[lop~]` (low-pass filter), `[hip~]` (high-pass filter), `[bp~]` (band-pass filter), among many others. Users can define additional Objects with three different methods, described as follows.

- **Abstractions.** When an Object name does not correspond to any known type, *Pure Data* treats it as an Abstraction. It searches a number of predetermined paths for a `*.pd` file whose name matches the one written in the Object. Creation arguments are accessible inside it, and the `[inlet]`, `[inlet~]`, `[outlet]`, and `[outlet~]` Objects can be used to add Inlets and Outlets to the Sub-Patch. Abstractions can be instanced multiple times. To differentiate each, a special symbol `$0` can be used – it expands to a unique identifier as long as it is within the right Patch. We call this the Zero Dollar of the Patch.

[41] The *Pure Data* manual calls them "Audio Connections", but we believe that "Signal Connections" is more appropriate since not every signal represents a sound.

- **Sub-Patches.** When a `[pd <name>]` Object is created, a new Canvas is associated to it, and the user can populate it as he or she sees fit. It then loads that Patch as the Object. Creation arguments, Inlets and Outlets can be added the same way as with Abstractions. However, Sub-Patches share the same Zero Dollar as their parent Patch.

- **Externals.** These work like Abstractions, except the loaded files must be a dynamically linked library containing the Object behavior implemented in `C`. This is used to allow more efficient implementations of heavy algorithms and data structures.

Another important type of Object is the **Array** type. It comes with a single Inlet that can be sent a few key Messages, but most of Array manipulation is actually done with the help of auxiliary Objects such as `[array get]` and `[array set]`. Every Array has a unique name, and the user must provide it to the auxiliary Objects for them to work. Arrays can be used to store signals, **being the most direct method of storing and playing samples in *Pure Data***.

Pure Data, being designed for real-time DSP, provides per-Patch settings that determines the sample rate used and how many samples it processes per **Tick**. The default Tick size for *Pure Data* is 64 samples long. Every Tick, the DSP engine iterates over all Signal Connections following a deterministic order, and transfers the corresponding amount of samples for that Tick, unless an Object outputs a different number of samples. This means even silent Signal Connections – that is, Connections that are carrying constant-value signals – still demand processing, costing CPU time for the Patch execution. Note how this differs from Message Connections, which are triggered only per demand.

Lastly, as we mentioned in the beginning of this section, `libpd` provides a way to embed *Pure Data* Patches into other applications. Essentially, this library links the host application to the *Pure Data* engine and exposes its key features in a `C` API (with bindings to a number of other programming languages also available). Besides opening and loading Patches, `libpd` can also send and receive Messages to and from the loaded Patches, read and write to available Arrays and, most importantly, request DSP ticks and inspect the resulting output signals. That means it does not provide sound playback like *Pure Data* does. The host application is responsible for forwarding the obtained audio signal to the appropriate low-level routines that ultimately lead it to the sound card.

Critical Analysis

Being a full-fledged programming language, *Pure Data* provides endless possibilities for real-time soundtracks. Both sample-based and procedural audio approaches are possible. One can even emulate old-school MIDI synthesizers and program algorithmic or stochastic melodies. Besides, the tools has a community with a long history, which means an active support for eventual issues the user may face. The downside of all this is the very fact that it *is* a programming language. Programming is not easy, and sound designers might even take it as an abuse if it is demanded of them. Among all the tools presented here, *Pure Data* has definitely the steepest learning curve.

Another problem is that, being a free software mostly maintained by a single developer, its usability issues stand out as much as its potential, specially when compared to the other com-

mercial, high-budget projects we have discussed. The situation is even more aggravating for its use in games, since it was not designed to be used as an embedded library, but as a stand-alone application. Even though `libpd` circumvents that restriction, the internal implementation of *Pure Data* still leaks a number of difficulties through the wrapper `libpd` provides. For example, all of its internal states are kept in global scope. This immediately poses a hurdle for the development of games, since it demands some sort of mediatory code to allow safe multi-thread programming.

Nonetheless, it is the technology that comes closest to our research interests, for many reasons. It is the only free software among the studied tools, which aligns with our own middleware license. *Pure Data* is specifically designed for real-time DSP, thus consisting of a solid implementation reference. It also has a respectable user base, which indicates us that, despite the problems we have listed, *Pure Data* is rather good at what it does, or it is the only reasonable alternative available at the moment. Lastly, the data flow paradigm present in the language offers us yet another way to achieve a procedural mechanism for the user, where the other tools have preferred approaches closer to structured programming.

3.4.6 Comparison

Here we finish this chapter by presenting Table 3.1, where we compare the main aspects of the studied middleware systems. We focused the comparison mainly on how the tool abstracts the soundtrack design process to the user – if it uses sample-based or procedural audio, for instance – and how the technology is licensed for use. It is easy to see how *Pure Data* stands out, followed by *iMuse* (which is understandable since it was developed in a very different context from the other systems). It is also interesting to see that *FMOD Studio* actually resembles *iMuse* the most.

Tecnology	License	Sound Representation	Work Abstraction
iMuse	Patented	MIDI-based	Sequences + conditional jumps
Wwise	Commercial	Sample-based	Tree structure
FMOD Studio	Commercial	Sample-based	Time-lines + conditional jumps
Elias	Commercial	Sample-based	Layered track loops
Pure Data	Modified BSD	Procedural	Data flow programming

Table 3.1: Comparison between the studied sound technologies.

Chapter 4

Proposed solution

In Chapter 1, we stated the problem we want to solve in our research in Section 1.1, the problem being the inherent difficulties in making real-time soundtracks in games due to their interactive and unpredictable nature, and we focused on the roles technology plays to aggravate or improve that situation. Then, in Section 1.2, we presented how we intend to solve that problem, which is through a real-time game audio middleware of our own. Chapters 2 and 3 built the knowledge base we need to achieve that. Now, this chapter takes the first concrete step towards our research objective. Here, we will discuss what we have learned and bring forth the solution we reached for the stated problem. We named the proposed middleware ***VORPAL***, a recursive acronym for *Vorpal Open Real-time Procedural Audio Layer*[1]. Chapter 5 will then describe in more detail how that solution was implemented, and Chapter 6 will discuss how that meets our validation requirements.

Section 4.1 will present the methodology we used to build our proposed solution, using the intermediate objectives from Section 1.2.1 and the validation requirements from Section 1.2.3 as a starting point. Next, Section 4.2, as part of the methodology process, gathers system requirements for the proposed game audio middleware from both the literature and interviewed professionals in the area. In Section 4.3, we address one of the core design decisions of the middleware: how to digitally represent real-time soundtracks. Based on that, Section 4.4 finishes this chapter by presenting the full software architecture we designed for the proposed game audio middleware.

4.1 Methodology

Every (useful) technology serves a human need. In this thesis, we have shown that in digital games development there is a need for giving sound designers more control over the real-time soundtracks they create. We have described the ways with which game studios can deal with that need with the techniques and technologies currently available in the industry and the community. Through all this research, we have come to highlight a number of particular aspects of the problem at hand and, sometimes, how others have been able to solve them. Thus, to reach our own objective of designing and developing a game audio middleware to satisfy that need, we will explicitly list all of these individual aspects of the problem, then propose a middleware architecture that is

[1] "Vorpal" is a word invented by Lewis Carrol in "Through the Looking-Glass", and is often used in games to refer to particularly deadly blades, as in "vorpal sword".

intended to satisfy each and every one of those aspects. We refer to these aspects as the **System Requirements** of our middleware. Our design decisions are based on them in the sense that the resulting technology should enable the users to overcome each listed aspect of the problem. For instance, if a System Requirement is "the tool should be able to provide sample playback during game execution", and if there is at least one way for the user to achieve that through the present features of our middleware, then we consider that this Requirement has been met.

This method allows us to clearly validate our research with the first two criteria established back in Section 1.2.3, namely the Basic Feature Support and the Advanced Feature Support validations. The first can be validated by including an initial subset of System Requirements that essentially demand that the middleware be compatible with an actual game. The second validation, on the other hand, involves gathering a more comprehensive list of features from both literature and professionals working with game soundtracks. To keep the scope of this process manageable, we limited ourselves to Collins' work for the former and three interviewed professionals for the latter. The choice of author is due to her work being the only one in our bibliographic spectrum that has systematically enumerated the possibilities of real-time soundtracks in games, thus providing a solid foundation for more complex requirements. As for the sound professionals, the three we elected were the only ones that agreed to an interview among studios we managed to reach out to in our local community.

The first ten months of our research were spent contacting other research groups and local game studios, as well as reading through the gathered bibliographic material to understand and build our list of System Requirements. At the same time, we experimented with the available technologies and developed prototypes to verify how each of them could help us achieve the intended feature set of our middleware. From the sixth month of research, we started development of the system itself. We based our implementation process on agile software development principles, which prioritize having a minimal, usable product as early as possible [BGM+01, BA04]. For that, we developed a small test suite where we could promptly verify whether the System Requirements were being met. Besides, starting from the second year of our research, we joined efforts with one of the interviewed professionals to develop a proof of concept game together as a way to explicitly achieve the Basic Feature Support validation. The resulting title was *Sound Wanderer*, which we describe in greater detail in Section 6.1. A beta version of the game was released six months later [MVK16].

4.2 System Requirements

As explained in Section 4.1, here we gathered the System Requirements for our game audio middleware as part of our research methodology. We used Collins' list of dynamic music variabilities from Section 2.1 as a starting point, then we analyzed the interviews we made with active sound designers in the industry to complete the list as needed. This was particularly convenient since Collins' list is mostly limited to game music, lacking feature descriptions for sound effects and voice, a gap the sound designers' experience promptly filled in. We will not present Collins' list here again to avoid being repetitive, but we will show the full System Requirements list in Section 4.2.4 when we have finished gathering all the remaining requirements. For that, Sections

4.2.1, 4.2.2, and 4.2.3 will address each of the interviewed professionals, highlighting the System Requirements we can derive from the report on their experiences **in bold**.

4.2.1 Rodolfo Santana

Rodolfo Santana is a sound designer from *Tapps Games*[2] studio in São Paulo, Brazil. He has worked there since 2014, but his experiences with game sound predates that. Until very recently, the *Tapps Games*' development pipeline was very aggressive and "waterfall" oriented [Roy70]. That allowed them to produce over two hundred titles in only a few years (although a number of them are merely *reskins*[3]). With Santana being one of the few sound designers at the studio (when we interviewed him there was only one more person helping him), that can attest to how much experience he accumulated over his career at the studio.

Tapps Games works exclusively for the mobile market, and this reflects on the daily challenges Santana faces. For instance, there is an enforced limit to how much memory space the sound assets he delivers can occupy. "*I have [...] a megabyte and a half for music, and there is about three megabytes for [sound] effects*", he says. He believes that real-time sound synthesis would be a good solution, specially because the main game engine they use (*Corona*[4]) has some issues with sample playback synchronization: "*[...] it is not that easy to synchronize things, because since it [Corona] has to read the sample, [...] if the device takes a little longer to load one of them, on that run it will be played outside the tempo. [...] With synthesis, this would not happen*". Much of the difficulties he deals with when working with samples coincide with Farnell's criticism [Far07]. He says that "*It is very toilsome to build transition cells*" when working on music transitions, and that when he needs small variations of the same song, he has no choice but to deliver multiple sample files, each processed with a different effect, instead of just doing so from inside the game code. That leads us to our first interview-derived System Requirement: **real-time sound synthesis as an alternative to sample-based sounds**, because it consumes much less memory space and it enables many real-time manipulations that would otherwise be impossible to achieve – "*That is, from what I imagine, the maximum dynamism one can achieve*", says Santana.

Santana also mentions many of the music variabilities on Collins' list, like variable volume, mix, tempo, pitch, and DSP (specifically filters in this case). An additional feature he defends is **sound physics**, such as 3D spatialization, through attenuation and binaural effects, and reverberations based on the virtual space geometry. He states that this enhances music as much as it enhances character voice and sound effects. We discussed a few available solutions to sound physics back in Section 2.3 which we can use as reference.

Lastly, Santana had some previous experience with *FMOD Studio* and we asked him about his impressions on game audio middleware. It was a technology he could not use at *Tapps Games* due to the lack of compatibility with *Corona*. He found its licensing price unaffordable, and its sample-based approach limiting, even though he admits that there had been people who managed to circumvent that by using small sets of very short samples and building the soundtrack by combining them instead of recording more samples. Notwithstanding these issues, he ultimately

[2] http://tappsgames.com/
[3] A game title with the same code base as another but with different assets, mainly graphics related ones.
[4] https://coronalabs.com/corona-sdk/

"*sees it more as a way for the sound designer not to depend on the programmer*", which he thinks is an excellent quality of game audio middleware. As such, we will add one last System Requirement from this interview: **the less the sound designer depends on programmers, the better**.

4.2.2 Kaue Lemos

Kaue Lemos is the audio director at *Insane Games*[5] studio and *7Sounds Game Audio Solutions* company, both from São Paulo, Brazil too. The studio has a wider spectrum of target platforms, including desktop computers, social networks, and mobile. They are currently developing a *Massive Multiplayer On-line Role Playing Game* and have at least one title published on *Steam*[6] in early access. In his work, Lemos uses *FMOD Studio* extensively, and shared his experience with game audio middleware during the interview.

When asked about what he believes are key features this kind of technology should offer, he highlighted **sample playback randomization** and **per-sample polyphony control**, among other requirements we have already discussed such as pitch and volume variability. In the case of polyphony, he uses an example to explain the motivation behind it, and defends that the main purpose of the feature is to "*limit the number of times this [spaceship shooting] sound can be played simultaneously*". He mentions that, in *FMOD*, there are two polyphony policies: Quietest and Oldest. Whenever there are too many copies of the same sound being played, the first policy drops the copy that has the least volume, while the second drops the ones which have been playing longer.

A concern he showed when describing his workflow with *FMOD* was guaranteeing there would not be any leaking resources in the soundtrack. That is, ensuring unused or mostly silent samples and Events had been freed from memory, recovering not only memory space but also CPU time to work on more relevant parts of the soundtrack. The polyphony control was one of the ways he pointed to solve this. One other way was to explicitly bring the volume level of specific sounds down to zero when they were no longer needed, since *FMOD* employs an optimization mechanism to clear these resources from memory in this case. With this, we have the System Requirement of allowing **resource usage optimization** by the sound designer. Besides this, Lemos also stated that "*everything I have said can be done without a middleware – the purpose of the middleware above all else is to facilitate all this*". He did not have problems working with sample-based audio, but agreed that he expected the technology to give him more independence to shape the soundtrack as he thought was best.

4.2.3 Dino Vicente De Lucca

Dino Vicente De Lucca does not work in the game industry. He is a musician and sound designer with his own studio, DVMúsica[7] (also from São Paulo), who works in a variety of sound projects. He has composed soundtracks for movies, commercials, and documentaries alike, but has also

[5] https://www.insanegames.com.br/
[6] http://store.steampowered.com/
[7] http://dvmusica.wixsite.com/dvmusicaapresentacao

designed and presented a number of interactive performances. One of them was a gamified exposition, and another used the language of games and a live boxing performance to manipulate an experimental music show in real-time using *Max/MSP*[8]. He specializes on sound synthesis, with a personal inclination towards old-school analogue synthesizers. De Lucca believes that most multimedia productions neglect sound and he has been a witness (and "victim") to the traditional process of leaving the soundtrack for last in audiovisual projects, as we discussed in Section 3.2.1. He defends that sound should be designed since the beginning of development, because it is as important (or even more important) than the visual part of a production. De Lucca is also our partner in the *Sound Wanderer* project, the proof of concept game we developed to validate *VORPAL*.

One of the features De Lucca asked for the most was **binaural audio**. He defends that having an appropriate distribution of sound among the output devices (earphones, speaker sets, etc.) is one of the key factors in creating immersive experiences. This means not only being able to send multi-channel signals to the sound card, but also being able to map spatialization and sound physics effects to the corresponding channels. On that matter, De Lucca also wished for **stereo sample playback** support, which would normally be a very reasonable feature request, but we found that it is actually not very clear how to make it coexist naturally with spatialized binaural audio. The reason is that spatialized sounds work like in real life: people hear only a single sound coming from each source, even if it is stereo sound. That is why the cinema, live performances, and home theaters use multiple speakers in *different positions* to provide multi-channel sound. Thus, to play a stereo sample in a spatialized binaural environment, we could either simulate two "virtual earphones" on both sides of the listener, or use a separate environment exclusively for stereo samples.

When working on some real-time music themes for *Sound Wanderer*, De Lucca suggested we based our soundtrack on a **sequencer-like structure**. He defends that linear structures like sequencers and timelines are easier to work with, even if a tree-based structure (like the one used by *Wwise*) might be more powerful. The sequencer we used essentially consisted of a tempo matched chronometer over a two bar loop, in which we could assign sounds to each beat. This is marginally similar to using a timeline in a DAW or on *FMOD*, except you cannot really place sounds in *any* moment of the timeline, only on beat positions. Next, to produce the timbres we wanted for the music notes, we decided to use synthesized sound. De Lucca asked for an **interface that mimicked analogue synthesizers**, since that way he would be able to work with the tools he is already used to. This imposes certain limitations on the range of possible timbres, but, on the other hand, it eases the workflow for the sound designer by bringing the features closer to a language he or she definitely understands. Such approach reflects one of De Lucca's opinions regarding audio tools in general: more powerful technology may give the sound designers more control, but at the same time if *shifts* the responsibility and burden of more technical aspects from programmers to sound designers as well. This is aggravated by the fact that such increasing skill demands are not always properly converted into an appropriate remuneration increase (if any). This is related to the discussion brought by Scott's work [Sco14] we presented in Section 2.1: at the same time that game sound requires new technological horizons to achieve better

[8] https://www.youtube.com/watch?v=uicFemu89T0

soundtracks, the professionals behind these soundtracks need to invest more and more resources to catch up with the new production paradigms, which is not always a rewarding task.

4.2.4 Final List of System Requirements

Having taken into consideration the experience and reports of professional sound designers, each with his own context and needs, we can now merge Collins' variabilities list with the System Requirements we have highlighted from the interviews. We also use this opportunity to formulate a minimal list that meets our Basic Feature Support validation. However, many of the System Requirements we have seen can be joined into more concise features. In this section, to do that, we will make a step-by-step description of how we massaged the list until it reached a more objective set of System Requirements. First, let us consider Collins' requirements:

1. Variable Tempo
2. Variable Pitch
3. Variable Rhythm
4. Variable Volume
5. Variable DSP (frequency filters)
6. Variable Melody
7. Variable Harmony
8. Variable Mix
9. Variable (Open) Form
10. Variable (Branching Parameter-Based Music) Form

In terms of implementation, Requirements 4 and 8 are very much alike: both are done by controlling the volume of different audio channels – the difference is that Requirement 4 does so only to a single channel. We can merge them into the same Requirement. Also, Requirements 6, 7, and 9 are all about real-time changes to the sequencing of a music, be it in its melody or be it in its harmony. We shall also join them. Requirement 10 is more about transitions than the sequencing itself, and as such we will keep it a separate feature. The resulting list is:

1. Variable Tempo
2. Variable Pitch
3. Variable Rhythm
4. Variable Mix (was 4 and 8)
5. Variable Frequency Filter (was 5)
6. Variable Music Sequence (was 6, 7, and 9)

7. Music Transitions (was 10)

Now, let us consider the System Requirements needed to achieve the Basic Feature Support validation, which states that our middleware should be compatible with actual game applications. Since the validation of this Requirement is very straightforward (we need only develop a reasonable game that works with the middleware) we will also keep it simple here. Even though we call it a Basic Feature, it really is the minimum expected if our middleware is to be considered a viable technology, and for that we consider it a fundamental Requirement. As such, we will number it the zeroth System Requirement:

0. Compatibility with Game Applications

Next, we add the Requirements gathered from the interviews.

8. Sound Synthesis (Santana)
9. Sound Physics (Santana)
10. Independence from Programmers (Santana)
11. Playback Randomization (Lemos)
12. Polyphony Control (Lemos)
13. Resource Usage Optimization (Lemos)
14. Binaural Audio (De Lucca)
15. Stereo Sample Playback (De Lucca)
16. Sequencer Structure (De Lucca)
17. Industry Standard Synthesizer Interface (De Lucca)

There are many Requirements that work together or complement each other now. With the intention of keeping the final list lean, we will merge a few more features. Requirement 8 is very generic, but it becomes manageable if we put it together with Requirement 17. Requirement 2 is usually implemented as a synthesizer parameter, so we will include it together with 8 and 17 too. Sound Physics, on the other hand, embraces too many possibilities, like the works of James *et al.*, Bonneel *et al.*, Taylor *et al.*, and Raghuvanshi *et al.* we discussed in Section 2.3 [JBP06, BDT+08, TCAM09, RSM+10]. For the purposes of demonstrating the potential of real-time audio physics, we will bundle Requirement 9 together with Requirements 14 and 15 as "3D Audio Spatialization". Since rhythm and tempo are both parameters of music sequencers, we can now join Requirements 1, 3, 6, and 16 as "Real-Time Controlled Music Sequencing". Thus, the resulting list of System Requirements our middleware will support is:

0. Compatibility with Game Applications
1. Independence from Programmers

2. Real-Time Controlled Music Sequencing
3. Real-Time Controlled Synthesis
4. Music Transitions
5. Variable Mix
6. Variable Frequency Filter
7. 3D Audio Spatialization
8. Playback Randomization
9. Polyphony Control
10. Resource Usage Optimization

Having established these System Requirements, there are now a number of key design decisions we need to address and base the middleware architecture and implementation on. That is what the rest of this chapter will do. The most important of these decisions – given the need for data-driven design in games and what we learned from other game audio middleware – is what kind of data format we will use to represent a real-time soundtrack. This will be discussed in Section 4.3. Then, we must build the architecture and its components around that format, given its central role in enabling our proposed technology. This is done in Section 4.4.

4.3 Digital Representation of Real-Time Soundtracks

If we go back to Figure 1.5 and inspect the middleware systems analyzed in Section 3.4 through it, it is easy to see why the data format used to represent real-time soundtracks is a fundamental design decision in this kind of tool. Game audio middleware always comes with two interacting software components: a digital content creation interface and a real-time audio engine. The soundtrack production pipeline consists of using the digital content creation interface to make the sound elements of the game, *then exporting them as a set of files* which the real-time audio engine is able to process and reproduce the originally intended soundtrack from. The gap between sound designers and programmers is closed by the employment of an *intermediate representation*, which is neither the exact playback the sound designer may have heard or the precise code the programmer would write to derive the necessary interactive and dynamic behaviors. It is, instead, a format that has a little from both worlds. Among the studied commercial middleware systems, the only one we had the opportunity to inspect more closely was the MIDI-based format used in *iMuse*, but it is not hard to extrapolate what kind of information the exported files from *Wwise*, *FMOD Studio*, and *Elias Studio* carry. In this section, we will describe the main possibilities we have found, compare their advantages and disadvantages, and elect the one we find most appropriate based on the System Requirements raised in Section 4.2.4.

4.3.1 Considered Formats and Comparison

The first format we will consider is the one used by *iMuse*. Since it was based on MIDI, it is very interesting for score-based music compositions. On the other hand, it completely leaves out the possibility of using more complex samples, which was actually part of its design since the technology at the time did not support sample-based audio well enough. This makes it all the more evident that the format used by *iMuse* was developed in a very different context and with very different needs than modern middleware systems. As such, we find that this kind of format is not flexible enough to handle soundtrack quality demands currently valued in the game industry. In this sense, one could say that the format used by *FMOD* is an evolution of *Lucas Arts*' breakthrough. The timeline present in the Event structure is no different to timestamped MIDI commands, with the exception that it supports sample insertions along with the Logic Track, making it as dynamic as the original *iMuse* system, yet more compatible with sample-based audio. The problem is that it represents a new extreme in and of itself, since it is not possible anymore to synthesize real-time sounds like *iMuse* did – not without some unnatural sample juggling at least. A good balance between these two formats might be a promising compromise though, specially if we could use the Command Queue from *iMuse*.

Which brings us to *Wwise* and its Event structures. Both them and the Command Queues allow the sound designer to specify a routine to be executed over the soundtrack. The main difference is that Events in *Wwise* are triggered directly by game code, while Command Queues in *iMuse* are executed when a corresponding Marker is processed in a composition sequence. The important aspect both have, though, is that they give the sound designer control over a programming-like interface, and this is what allows them to dismiss the intervention of a programmer to create a desired real-time behavior in the game soundtrack, as dictates System Requirement 1. *Wwise*'s tree structure is also less linear than the structures of *iMuse* and *FMOD*, allowing it to avoid using a logic jump mechanism like them since its soundtrack layout naturally provides a branching function. In terms of how that reflects in the format, there are a number of approaches one could use to represent a tree. The downside of the design of *Wwise* also lies in its tree-based structure: it makes *Wwise* the only middleware with an interface layout and workflow completely different from traditional DAWs and other composing software. Even if we strive for real-time soundtracks, which are mostly nonlinear, it still feels more natural for a sound designer to have some linearity to work on top of, as we saw from De Lucca's report in Section 4.2.3.

As for *Elias*, as much as it provides a very efficient path for producing adaptive music in games, it is not enough to make the whole soundtrack. We can, however, learn from their success in building an intuitive workflow for that specific purpose, specially if we consider System Requirement 2, "Real-time Controlled Music Sequencing". Besides that, the format used by *Elias* is mostly concerned with storing the matrix of Track Levels, something we ought to consider if we face a similar need in our own format.

Lastly, there is *Pure Data*. Even though it does not follow the pattern in Figure 1.5, it does have a very important interaction between the format it uses and how the language works. Since a *Pure Data* Patch is nothing but a directed graph written in a plain-text format, we were able to explicitly investigate it. It is a very simple representation: every Object is listed with their corresponding creation arguments, then every Connection is listed indicating which Object

Outlet and Object Inlet it connects. Since *Pure Data* is actually a programming language, this format is enough to express the structures needed to write any Patch because it is the Object themselves that provide the relevant features used to program the Patch behavior. In fact, *it is even possible to replicate most of the capabilities from the other middleware systems*, with the benefits that

1. *Pure Data* comes with dozens of useful DSP functions;
2. Signal data is naturally treated as procedural audio;
3. The sound designers can effectively *program* the soundtrack on their own (Requirement 1);
4. There is an active community of both artists and researchers around the language; and
5. It is free software.

Given our time constraints and the challenging scale of developing a game audio middleware [Gre14, Chapter 13], we see here an opportunity to take the most out of a well established technology by having *VORPAL* built on top of it.

4.3.2 Chosen Format

By using the *Pure Data* language format we can provide an intermediate API, in the form of Abstractions and possibly Externals, through which the sound designer can author the game soundtrack. Since mastering *Pure Data* as a whole might not be practical, this API will serve as a dialect, a subset of the language made more accessible and efficient for end users to work with. Then, thanks to a community effort such as `libpd`, we can promptly embed *Pure Data* Patches as soundtrack units in a digital game application. For the purposes of advocating procedural audio and opening potential future paths for real-time soundtrack research in games, we find the advantages of using *Pure Data* Patches as our base format very appealing, even with the caveats we discussed in Section 3.4.5 (such as *Pure Data* not being designed for embedded use). **As such, our real-time soundtrack format will consist of a *Pure Data* dialect tuned to satisfy the System Requirements** exposed previously in this chapter. We analyze the benefits and issues this design decision brought to our research in Section 6.4.

4.4 Architecture

Being a game audio middleware, *VORPAL* abides by the standard and time tested general architecture we described in Figure 1.5, centered around the soundtrack format we chose in Section 4.3. There are two main components composing the system: the **Audio Engine** and the **Soundtrack Creation Kit**, much like the *FMOD Studio Programmer's API* and *FMOD Studio* pair, respectively. In our case, the Audio Engine is implemented as a programming library, presented in Section 4.4.1, and the Soundtrack Creation Kit consists of the *Pure Data* application itself plus a set of soundtrack creation Patches, as we will discuss in Section 4.4.2. Section 4.4.3 closes this chapter by describing how our architecture integrates this two components into the

middleware as a whole. In these Sections we only explain the general idea behind each part of the architecture, since the details of how they are actually implemented and used in practice are discussed at length in Chapter 5.

4.4.1 Audio Engine

This is the on-line part of the *VORPAL* middleware: a programming library written in `C++` (because of Requirement 0) implementing the Audio Engine the game application needs to play the real-time soundtrack Patches made by sound designers with the Soundtrack Creation Kit. The library comes with an API the programmer can use to load, play, and interact with the imported Patches. Since every game engine has its own time management peculiarities, we use an idle-frame synchronized service interface (see Section 3.3.2) through which the programmer can keep the Audio Engine and the Game Loop correctly synchronized. Patch embedding is achieved with `libpd`, and *Pure Data* Messages are used to control the loaded Patches. The resulting audio signal must be properly synchronized with the time data obtained from the game application, and then finally sent to the sound card for playback. This internal organization of the *VORPAL* Audio Engine is illustrated by Figure 4.1.

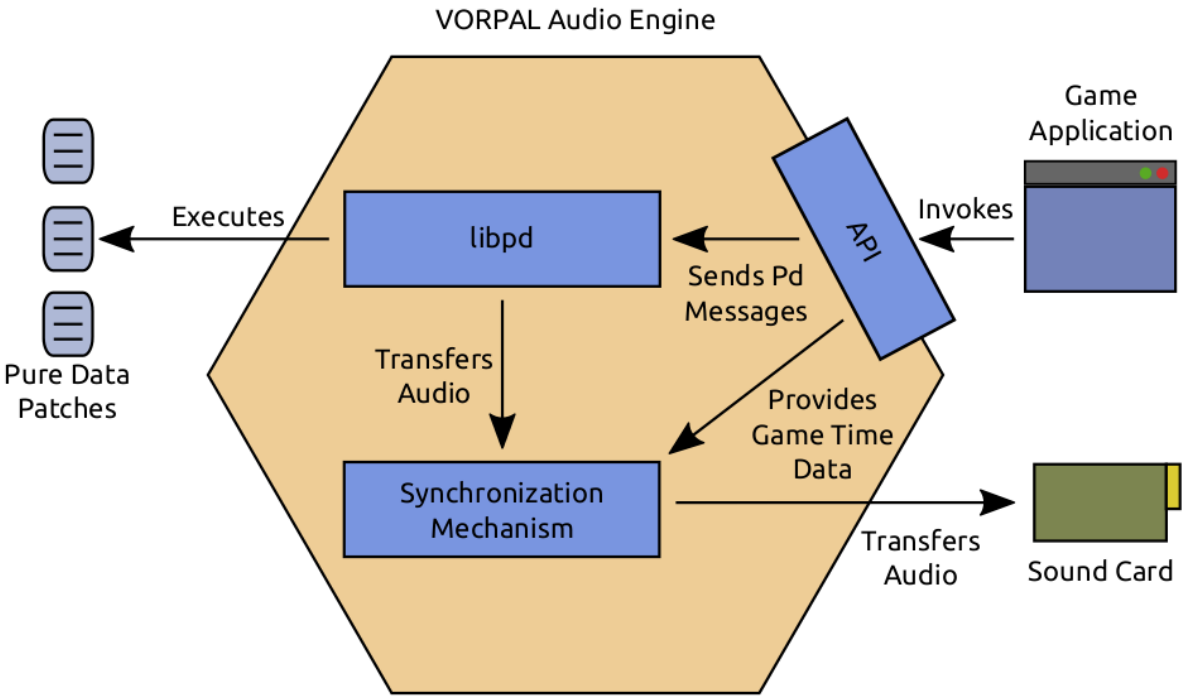


Figure 4.1: An overview of the main components and their interactions inside the *VORPAL* Audio Engine at game execution time.

The Audio Engine API essentially exposes two main `C++` classes to the programmer: the `vorpal::Engine` class and the `vorpal::SoundtrackEvent` class. `vorpal::Engine` is a singleton that serves both as an entry point for the API and as a game sub-system which must be serviced every game frame using idle-frame synchronization. `vorpal::SoundtrackEvent`

is the API representation of a loaded *Pure Data* Patch, assigned to a 3D position (for Requirement 7) and to a corresponding audio channel. We say it represents a **Soundtrack Event** inside the *VORPAL* middleware. This design uses an Object-Oriented Programming approach to the soundtrack, treating Patches as objects that can receive **Commands** (method invocation) through *Pure Data* Messages. The available Commands in each `vorpal::SoundtrackEvent` instance mostly depend on the Patch the sound designer made. Figure 4.2 contains a UML class diagram of this two classes. We describe the actual API, along with a number of implementation details, in Section 5.2.

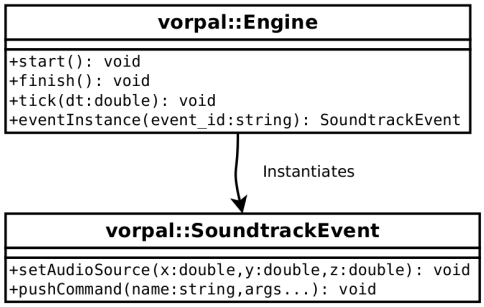


Figure 4.2: The two main classes in the *VORPAL* Audio Engine API exposed to the programmer.

4.4.2 Soundtrack Creation Kit

Since we chose *Pure Data* Patches as the base for our soundtrack file format, we need no more than the *Pure Data* application itself as a soundtrack content creation interface. Through it, the sound designer can create Objects from a set of Abstraction and Externals we provide and build the intended soundtrack for the game. In programming terms, this component of the middleware works as a "game soundtrack SDK" for *Pure Data*. This means that part of the *VORPAL* package consists of a number of *Pure Data* Patches and binary Externals which need to be installed in the user's machine together with *Pure Data* itself. The Objects available in this Soundtrack Creation Kit are designed to cover most of the System Requirements we gathered in Section 4.2; the ones they do not are directly supported by the Audio Engine in some other way (like System Requirement 7, "3D Audio Spatialization").

One of the advantages of using *Pure Data* like this is that System Requirement 1, "Independence from Programmers", is promptly achieved (albeit at the cost of a steeper learning curve), while Requirements 5, 6, 8, and 9 are indirectly satisfied through basic *Pure Data* constructions. For instance, Requirement 8, "Playback Randomization", can be achieved by simply using the `[rand]` Object, which comes with *Pure Data* by default. Requirements 2 and 3, on the other hand, require more specialized Objects: a **Sequencer** and a **Synthesizer**, respectively. These two Abstractions are the main elements of the Soundtrack Creation Kit. The specific way with

which they are combined in a soundtrack is able to achieve System Requirement 3, "Real-time Controlled Synthesis", a fact we properly demonstrate in our proof of concept game, *Sound Wanderer*, in Chapter 6. Lastly, this kit also comes with a custom sound output Object type called **Output Bus** to better organize audio channels and the mixing thereof, improving support on Requirement 5, "Variable Mix". Bus Objects, together with **Command** Objects, are important in the integration between the Soundtrack Creation Kit and the Audio Engine, as we will explain in Section 4.4.3. Lastly, the Kit also comes with **Sample** Objects to load sample-based audio. The specifically supported formats depend on some implementation aspects of the Kit, so we will address them later when we discuss the implementation details of the *VORPAL* Soundtrack Creation Kit in Section 5.3, where we also show in detail how these Objects look like. Usage examples are shown Section 5.4.5 too. Thus, the full list of Abstractions and Externals that compose the Kit consists of the following Object types:

- Sequencers;
- Synthesizers;
- Output Buses;
- Commands; and
- Samples.

4.4.3 Components Integration

As explained in the description of the Audio Engine in Section 4.4.1, *VORPAL* fits *Pure Data* Patches made through the Soundtrack Creation Kit into the Object-Oriented Programming paradigm, by expressing them through the `vorpal::SoundtrackEvent` class in the Audio Engine. To do this, the Kit provides the Command Objects. They have a creation argument indicating the name of the Command. When the programmer pushes a Command to a `vorpal::SoundtrackEvent` instance, he or she must specify the Command name and parameters. The Engine then maps the *Pure Data* Message to the Command Object carrying the same name in the associated Patch and sends the given parameters to it. For instance, a soundtrack Patch could have a Command Object with creation argument "play", and when the programmers pushes the "play" Command with no parameters to the corresponding `vorpal::SoundtrackEvent` instance, the Command Object outputs a simple "bang" message (since no parameters were given) in the Patch, and this could be used to actually start playing something in it. Likewise, the Output Bus Object provided in the Kit goes the other way around. It receives audio signals generated inside the Patch (be it from samples or synthesis) and sends them back to the matching `vorpal::SoundtrackEvent` in the Engine. Then, when the Audio Engine is serviced during the Game Loop, all audio signal captured by instances of `vorpal::SoundtrackEvent` are properly transferred to the sound card for playback. Since *Pure Data* works in Ticks of 64 samples, the Engine can control how much signal is processed every frame to guarantee that the soundtrack is played inside real-time constraints.

Again, note that all of these concepts explained here are detailed in Chapter 5, where we discuss the implementation and usage of each of the architecture parts.

Chapter 5

Implementation

Chapter 4 defined the architecture of the *VORPAL* middleware and the soundtrack format it uses based on the System Requirements raised and the concepts explained in previous chapters. The next step is to delve into the implementation of the core aspects of the middleware, which is the role of this chapter. Here, we discuss *how* we achieve the proposed architecture and which technologies, data structures, algorithms, and conventions we utilized to do so. We also address the resulting workflow of our tool, that is, how users should proceed to actually make use of it. As stated in Section 4.1, we developed the system following agile principles. One of the consequences of this is that, before arriving on the current design of the middleware, we had already iterated over it a number of times. Here, we omit all the intermediate steps the development process took, but we will briefly discuss one of the prototypes we built, since it was the first to present relevant results in terms of real-time soundtrack support.

From the middleware specifications in Sections 4.3 and 4.4, we know that *VORPAL* is implemented using a mixture of `C++` and *Pure Data*. *Pure Data* has its own means of producing sound playback, but the same could not be said of the parts written in `C++` – namely, the *VORPAL* Audio Engine. For that purpose, we chose to use *OpenAL*, which is a cross-platform free software programming library for sound playback. We have already shown some of it in the code examples of Section 3.3.4. Besides the reasons we just mentioned, we also decided to use *OpenAL* because it supports 3D spatialized audio out-of-the-box, which brings us closer to meeting System Requirement 7 ("3D Audio Spatialization"). Since we preferred to avoid too many dependencies on our system, there are no other libraries *VORPAL* needs aside from *OpenAL* and `libpd`. It does require support for the `C++11` standard, since most modern compilers provide it and it relieves us of further dependencies for multi-threading and time synchronization features. Outside of the system itself, it is worth noting that we use *Git* for source code version control (hosted at https://github.com/vorpal-project) and *CMake*[1] as our building framework. These are dependencies only developers need, not the end users. Besides, all `C++` code written for the middleware, including the examples and excerpts found in this thesis, follow a slight variation of the *Google `C++` Style Guide*[2].

Section 5.1 presents the first real-time soundtrack prototype we developed using a more

[1] https://cmake.org/
[2] https://google.github.io/styleguide/cppguide.html

unstable integration with *Pure Data*. Next, Sections 5.2 and 5.3 address the implementations of the *VORPAL* Audio Engine and Soundtrack Creation Kit, respectively. In Section 5.4, we describe in details: the workflow programmers and sound designers must follow to use our middleware; game engine support for *VORPAL*, demonstrating how it works by exposing implementations of this support made for two free software engines; and two usage examples that show the *VORPAL* middleware in action.

5.1 Prototype

In the early process of designing *VORPAL*, when we were pondering the choice of soundtrack format, we decided to develop a quick prototype to validate whether *Pure Data* could work as a language that expresses real-time soundtracks. To avoid spending time in the development of a brand new game for the sole purpose of this prototype, we searched for an open source title we could modify and insert the new soundtrack into. Our middleware did not have a single line of code at this moment, and we wanted to be able to show people what our research was about without having to actually implement the whole system beforehand. Since our methodology involved gathering feedback from musicians and sound designers, having a demonstration of what we meant by "real-time soundtrack in games" was paramount.

For the game itself, we decided on modifying *Mari0* (*Stabyourself*, 2012)[3], distributed under the *Creative Commons BY-NC-SA 3.0*[4] license. It is a parody game that reconstructs the classic *Super Mario Bros.* (*Nintendo*, 1985) but mixes in the portal mechanics from *Portal* (*Valve*, 2007). The fact that it resembles a relatively famous game helps get the point across when showcasing the prototype, but that was not the only reason that made us choose *Mari0*. The other reason is that it was developed using the *LÖVE* framework, which we have extensive experience working with due to previous projects we participated in at *USPGameDev*[5], a student group from University of São Paulo that researches game development. That saved us the time of learning a new development tool just to change a few dozen lines of code to add the new soundtrack. Figure 5.1 shows a screen capture of the game.

The next step was to determine how we would embed a *Pure Data* Patch into the game. We were aware of `libpd`, but since we wanted the fastest way to achieve a real-time time soundtrack, we took a different approach. We wanted an alternative where we did not have to implement sound playback, since that required time and more software dependencies (in fact, we did add the *OpenAL* dependency for this part later). Thus, inspired by one of the features from *FMOD* where the sound designer could tweak the soundtrack while playing the game at the same time, we chose to play everything from the *Pure Data* application itself, while receiving real-time data from the game through a direct socket connection. Since both programs would be running on the same machine, we could assume there would be no issues such as packet loss, significant delays, etc. But, to do that, we needed to define a communication protocol. Again, as to not waste too much time on a relatively lesser detail of a prototype, we searched for something we could use out-of-the-box. The sufficiently good alternative we found was the *Open Sound Control*

[3] http://stabyourself.net/mari0/
[4] https://creativecommons.org/licenses/by-nc-sa/3.0/
[5] https://uspgamedev.org/

Figure 5.1: An in-game screenshot of *Mari0* (*Stabyourself*, 2012) [Sta12]. In this parody game, Mario can shoot portals and use them to traverse the classic challenges of the original title from *Nintendo*.

(OSC) protocol[6], for which we promptly arrived at an accessible and lightweight implementation, `liblo`[7]. On the *Pure Data* side, we ended up temporarily moving to *Pure Data Extended* (refer to Section 3.4.5) to find Externals that could decode OSC messages. After that, all we had to do was write a *Lua*[8] binding for `liblo` (another task we already had experience with [Miz]) and use it in *Mari0* to send messages containing relevant real-time data from gameplay, which, in turn, we used to control the soundtrack inside *Pure Data*.

For the actual soundtrack, we focused on the first stage of the game and started by simply playing a shortened version of its original theme, "*Overworld Theme*" by Koji Kondo. We implemented it in *Pure Data* using simple synthesized timbres and a custom sequencer. Then, we added two real-time effects to it. The first was inspired by the battle theme variations from *Faster Than Light*: the percussion track of the theme was removed, and we added a new one with hits that played according to the quantity of enemies nearby. The more enemies there were around Mario, the more hits the percussion track would play. There was a total of four levels of percussion intensity. Notice how we would have needed at least five different sound samples (the melody plus the four percussion variations) if we had not implemented it through a procedural sequencer. The other real-time effect we added to the soundtrack was a modulation in the melody. When the player managed to grab the *Fire Flower* item in the game (which makes Mario much stronger than normal), the theme key would rise a whole tone. If he got hurt and lost the *Fire Flower* bonus, the melody would go back to normal. Again, this would have been much harder to do with a sample-based implementation without incurring in a time-stretch of the sample (or using a second sample). Thus, both these effects served the purpose of not only illustrating what we envisioned as a real-time soundtrack, but also explicitly highlighting the advantages of procedural audio over sample-based audio.

The first "released" version of the prototype was finished around the middle of 2015[9]. Later, after the *VORPAL* middleware achieved its own initial release, we ported most of the prototype to the new system, dropping OSC and `liblo` in the process. The revised prototype was used

[6] http://opensoundcontrol.org/
[7] http://liblo.sourceforge.net/
[8] *Lua* is a Brazillian scripting language used worldwide by the community of game developers, being the language of choice of many game engines, such as *LÖVE*.
[9] https://github.com/vorpal-project/mari0/releases/tag/prototype/zero

to verify the other side of the coin: we composed our own arrangement of "Overworld Theme" using the *LMMS* tool and inserted it into the game using a sample-based approach. Due to the restrictions of this choice, we reduced the real-time effects in the new version of the soundtrack to a variable mixing (System Requirement 5) effect. Now, the percussion track merely increases in volume when enemies are close, and the bass track increases in volume according to Mario's size (that is, how many "power ups" he currently has in effect).

The original prototype served as a solid indication of the possibilities brought by procedural audio and confirmed that using *Pure Data* as a soundtrack creation interface was a viable option. It was an important first step in our research on top of which we based the current middleware implementation. At the same time, we learned that using *Pure Data Extended* could become a burden to the sound designer, since it came with a considerable number of dependencies and had actually not been updated for a while. The agile approach allowed us to detect this issue and move back to *Pure Data "Vanilla"*, at the same time that it showed us that supporting real-time soundtracks in games was an achievable and fruitful goal.

The following sections present a more in-depth description of the implementation details and usage protocols of the middleware. Readers more interested in a broader view of this work may skip directly to Chapter 6.

5.2 Audio Engine

The *VORPAL* Audio Engine is the part of the middleware responsible for processing the Patches provided by the sound designer, and producing the correct and synchronized playback that was intended for them. It is also the part the game programmer has contact with. These interactions have already been presented in Figure 4.1. As such, there are four main topics of interest we need to address to properly explain the workings of the Engine. The first is discussed in Section 5.2.1: the main classes and interfaces exposed to the programmer that enable the use of the Engine features. Section 5.2.2 presents the second topic, which explains how we manage the loading and processing of *Pure Data* Patches through `libpd`. In Section 5.2.3, we discuss the method we used to produce sound playback by integrating *OpenAL* into the Engine, including its binaural audio support. Then, Section 5.2.4 addresses how we achieve real-time synchronization with the Game Loop of the host game application, the fourth and last topic.

5.2.1 High-Level API

Following the good principles of encapsulation and cohesion from Object-Oriented Programming, the *VORPAL* Audio Engine exposes only the strictly required operations to the programmers, divided in classes so that each has only a single responsibility. This led us to implement two such classes: `vorpal::Engine` and `vorpal::SoundtrackEvent`, which we mentioned in Section 4.4.1. While `vorpal::Engine` represents the Audio Engine as a whole, `vorpal::SoundtrackEvent` represents individual Soundtrack Events. `vorpal::Engine` manages `vorpal::SoundtrackEvent` instances and provides the programmer with methods to load a single Soundtrack Event or synchronize all current Soundtrack Events, while the `vorpal::SoundtrackEvent` abstracts Soundtrack Events as 3D sound sources with real-time

behavior that respond to Commands much like objects respond to method invocations in Object-Oriented Programming. Their position is always relative to the virtual listener, or rather, the simulated listener is always at the origin of this 3D space.

System Requirement 10, "Resource Usage Optimization", among other aspects, demands that no longer used resources in the *VORPAL* middleware be freed from memory and process time. Since `vorpal::Engine` is a singleton, the only class the game programmer could leak around is `vorpal::SoundtrackEvent`, which might have numerous instances during game execution, many of which might not be needed all the time. The simplest way to solve this is using some form of garbage collecting mechanism: unused references to Soundtrack Events are automatically deleted. To do this in `C++`, *smart pointers* are typically used [GHJV94]. In particular, the `C++11` standard comes with the `std::shared_ptr` template class which promptly gives us a reference-counting garbage collection feature[10] by wrapping the actual pointer of an object but allowing the programmer to use the wrapper as if it was the pointer itself (this is the main characteristic of smart pointers in `C++`). Additionally, we also used the Null Object design pattern [Woo98] to relieve the programmer from having to check for null `vorpal::SoundtrackEvent` pointers all the time.

That brings us to a sensitive matter in terms of API design. When the programmer requests the Audio Engine to load a Soundtrack Event, there must be a well defined behavior for when the Engine fails to do so (maybe because the corresponding *Pure Data* Patch file does not exist, for instance). `C++` exceptions would be the first option, since they can be used to accurately pinpoint error causes and they usually avoid backtracking code when errors cascade through the call stack. Notwithstanding, they are not a viable choice in practice, mainly because using exceptions forces the host application to use them too. There are many reasonable motivations for not using exceptions [Mey05], and we should not impose our own programming preferences on our users (specially if we want them to really choose our solution over others). The next simplest alternative would be to just return a Null Object instance and let the programmer check it, but that prevents us from providing further information regarding the cause of the error. The same is true if we just return a boolean value. Thus, we decided to use the following convention. Methods in the API that should return a meaningful object but may cause errors are to return an instance from a special class we implemented for this specific purpose, the `vorpal::Status` class. When no error occurs, the object is to be returned through an output parameter, that is, a parameter that receives a valid pointer to the object type, writing the actual result to it instead of returning it. When an error does occur, nothing is written to the output parameter, and a `vorpal::Status` instance is returned containing information regarding the error. With that in mind, we now describe the API of `vorpal::Engine` and `vorpal::SoundtrackEvent` classes.

`vorpal::Engine` Class

As explained before, the `vorpal::Engine` uses the Singleton design pattern [GHJV94]. However, it does so in a very unorthodox way, thanks to the fact that *Pure Data* (and consequently `libpd`) stores all of its state globally. To avoid making unnecessary dynamic allocations while

[10] http://en.cppreference.com/w/cpp/memory/shared_ptr

```
vorpal::Engine engine;
engine.start();
```

Listing 5.1: Initializing the Engine.

not degenerating to a static-only class, we implemented the `vorpal::Engine` as follows: at any moment in code, the programmer can declare an instance of `vorpal::Engine` and it will always refer to the same global and unique engine state. That means that even though the class can be instanced multiple times, all instances are actually the same, thus conforming to the Singleton pattern. For instance, to globally initialize the engine, we could write as in Listing 5.1, and then, when we want to load a Soundtrack Event later, we do as in Listing 5.2. Both instances are the same and only engine. In the rest of this section, we briefly describe the main parts of the `vorpal::Engine` API. All function signatures are exposed as if they were inside the `vorpal` namespace.

```
vorpal::Engine engine;
std::shared_ptr<vorpal::SoundtrackEvent> event;
engine.instanceEvent("event-name", &event);
```

Listing 5.2: Instancing an Event.

- `Engine::start`

  ```
  Status Engine::start(const std::vector<std::string>& event_paths = {});
  ```

  This method is responsible for initializing the Audio Engine as a whole. The returned `vorpal::Status` instance contains detailed information in case an error occurs, and the name of the sound device found for playback otherwise. The optional parameter `event_paths` is a list of paths where the engine should look for *Pure Data* patches to load as Soundtrack Events. See also Section 5.2.2.

- `Engine::started`

  ```
  bool Engine::started() const;
  ```

  A simple method that checks whether the global engine state has already been initialized.

- `Engine::registerPath`

  ```
  void Engine::registerPath(const std::string& path);
  ```

  Registers a path to look for events in. See also Section 5.2.2.

- `Engine::finish`

  ```
  void Engine::finish();
  ```

Clears the global engine state, freeing all used resources. There cannot be any Soundtrack Event references left in the application before invoking this method. Their destuctors will try to free their resources, which will already have been released by this method.

- `Engine::tick`

```
void Engine::tick(double dt);
```

This is the real-time servicing method through which the Audio Engine simulates the soundtrack. It uses idle-frame synchronization, as was justified in Section 3.3.2. That means that the parameter it receives should contain the amount of time that has passed in the last frame of the Game Loop. In our case, this parameter is expected to be in seconds.

- `Engine::eventInstance`

```
Status Engine::eventInstance(const std::string& event_name,
                             std::shared_ptr<SoundtrackEvent> *event_out);
```

Lastly, this method loads the specified Soundtrack Event and writes it to the provided smart pointer. There is no need for a method that unloads Soundtrack Events, since the `std::shared_ptr` class already handles reference-counted garbage collection for us. The code example we used earlier in this section illustrates how to use this method, but more complete examples will be presented in Section 5.4.5.

`vorpal::SoundtrackEvent` Class

Every *Pure Data* Patch the sound designer makes for the soundtrack of the game is loaded by the Audio Engine as a `vorpal::SoundtrackEvent` instance. It represents a sound object – with its own individual audio channel – inside a 3D virtual space (which can be treated as a 2D space too). This means that, internally, Soundtrack Events bridge two parts of the middleware: *Pure Data* Patches and *OpenAL* sources. This is done by two separate classes that we do not expose to the programmer: `vorpal::DSPUnit` and `vorpal::AudioUnit`, respectively. Such division allows us to connect a single Patch to multiple sound sources, or many Patches to the same sound source, among other possibilities that optimize resource usage. The rest of this section describes the main methods the programmers uses from the `vorpal::SoundtrackEvent` API, still assuming that the signatures lie within the scope of the `vorpal` namespace.

- `SoundtrackEvent::setAudioSource`

```
void SoundtrackEvent::setAudioSource(float x, float y, float z);
```

This method sets the 3D position of the sound source assigned to this Soundtrack Event. Remember that all positions are relative to the virtual listener, who is always at the origin of space.

- `SoundtrackEvent::pushCommand`

```
void SoundtrackEvent::pushCommand(const std::string &name,
                                  const std::vector<Parameter> &parameters);
template <typename... Args>
void SoundtrackEvent::pushCommand(const std::string &name, Args... args);
```

This method has two available signatures. Both do the same, but each has its own conveniences. Their role is to invoke a method in the Soundtrack Event as if it were an object within the Object-Oriented Programming paradigm, except we call it a Command as explained in Section 4.4.1. The Command name and parameters are passed as arguments to `pushCommand`. We use a special type `vorpal::Parameter` to wrap parameter values because they can be either numbers (`float`) or *Pure Data* symbols (`std::string`). In both signatures, the name is just a string. However, while in the first signature the parameters are passed in an array, in the second they can be passed directly – they do not even need to be wrapped in a `vorpal::Parameter` instance, thanks to the *variadic templates* feature in the `C++11` standard[11]. Again, examples are provided in Section 5.4.5 to better illustrate how to use this API.

5.2.2 *Pure Data* Patch Management

Since we decided to use *Pure Data* Patches as the soundtrack format for the *VORPAL* middleware (see Section 4.3), we require a means to embed such Patches into the game application. `libpd` does just that. It has many available API bindings for different languages, the core one being its `C` API. We used, however, its `C++` API, since it is compatible with the `C++` *Template Standard Library*, thus saving us the effort of adapting the API ourselves. But, independently of which language binding we use, the same feature set is provided by `libpd`: DSP Tick requests, Patch loading, etc. In the specific case of `C++`, most of these are laid out in a class called `pd::PdBase`. It can initialize and clear the global *Pure Data* state, load and unload Patches, enable and process DSP Ticks, send and receive Messages (containing only numbers and symbols), read from and write to Arrays and register Patch search paths. Patches are loaded as instances of `pd::Patch`, with its most important feature being that it can tell what the Zero Dollar of that Patch instance is. DSP Ticks are applied to all currently loaded Patches, and may receive arrays as input and output signals, corresponding to the `[adc~]` and `[dac~]` Objects respectively.

The fact that we can specify how many Ticks we want processed at a time is of great convenience to our implementation, specially because the *Pure Data* default Tick size is of 64 samples. This is small enough for acceptable latency but not small enough to make a bottleneck out of the data transfer to the sound card. The problem is that, since we want per-Patch 3D spatialized audio later on, we cannot have all Patches output their signal into the same universal audio bus (which is the output signal provided in the DSP Tick methods of `pd::PdBase`), because we would lose the capability of treating their spatial effects separately. To circumvent this issue, we decided to use *Pure Data* Arrays instead: every soundtrack Patch the sound designers

[11] http://en.cppreference.com/w/cpp/language/parameter_pack

make should send its resulting sound to a specific Array, not the standard signal output (unless the sound designer wishes to hear a playback preview from *Pure Data* itself, which is perfectly acceptable). Then, to find these Arrays, we just tag them with the corresponding Dollar Zero from the Patch it lies in. With this, we know which sound each Patch produces, and we can assign them to their own virtual spatial sources. Sending Messages as Commands to them is done the same way: Command Objects are tagged with the appropriate Dollar Zero, allowing the middleware to know where to send the Commands to. We also need to register search paths in `pd::PdBase` for it to know where to look for the Patches.

All this we implemented in two internal *VORPAL* classes called `vorpal::DSPServer` and `vorpal::DSPUnit`. `vorpal::DSPServer` is also a Singleton class like `vorpal::Engine`, responsible for handling DSP Ticks and interacting with loaded Patches, thus called the **DSP Server** of the middleware. `vorpal::DSPUnit` is a reference-counted class that abstracts individual *Pure Data* Patches following the Object-Oriented Programming flavor of providing a Command interface as if each Patch was an object that responds to method invocations – and we call them **DSP Units**. Every DSP Unit has its own audio bus, and may transfer its content to an Audio Unit, which we explain in Section 5.2.3. Each *Pure Data* Tick the DSP Server executes and fills the corresponding audio buses in every DSP Unit instance. The rest of this section describes the main methods from both these classes. It is worth remembering that they are internal to the middleware, and the programmer should use them directly only at their own risk. The proper "public" API of the system is limited to the `vorpal::Engine` and `vorpal::SoundtrackEvent` classes. Like in Section 5.2.1, all signatures are assumed to be inside the `vorpal` namespace.

`vorpal::DSPServer` Class

- `DSPServer::start`

```
Status DSPServer::start(const std::vector<std::string> &search_paths);
```

  This method initializes the global *Pure Data* state, and is automatically called by the method `Engine::start`. The search paths are fed to `pd:PdBase` so it can properly locate and load *Pure Data* Patches.

- `DSPServer::finish`

```
void DSPServer::finish();
```

  Clears all global *Pure Data* state. Called by `Engine::finish`.

- `DSPServer::loadUnit`

```
std::shared_ptr<DSPUnit> DSPServer::loadUnit(const std::string &name);
```

  Returns a reference-counted DSP Unit instance of the associated Patch with the given name, or a Null Object implementation of DSP Unit in case no such Patch exists in the registered paths.

- `DSPServer::handleCommands`

```
void DSPServer::handleCommands();
```

A method responsible for flushing Soundtrack Event Commands to the corresponding Messages in loaded *Pure Data* Patches. It uses the Dollar Zero obtained from `pd::Patch` to do this.

- `DSPServer::processTick`

```
void DSPServer::processTick();
```

This method executes exactly one *Pure Data* DSP Tick, which computes the next 64 samples of real-time audio coming from all loaded Patches and storing those signals in the buses of the corresponding DSP Units. This mapping from Patches to DSP Units is possible due to the tagging of the Patches using their own Dollar Zeroes.

- `DSPServer::cleanUp`

```
void DSPServer::cleanUp();
```

Calling this method unloads all Patches that have no more DSP Unit referencing them. It is called every `Engine::tick`.

`vorpal::DSPUnit` Class

- `DSPUnit::transferSignal`

```
Status DSPUnit::transferSignal(std::shared_ptr<AudioUnit> audio_unit));
```

Responsible for transferring the audio signal stored in the bus of a DSP Unit to an Audio Unit. This is what later causes the playback of the sound produced by the *Pure Data* Patch associated with this DSP Unit.

- `DSPServer::pushCommand`

```
void DSPUnit::pushCommand(const std::string &name,
                          const std::vector<Parameter> &parameters);
```

Lastly, this method is used to send Commands to the loaded Patches as if they were method invocations in objects from Object-Oriented Programming. As can be inferred from the similar signatures, this is called by `SoundtrackEvent::pushCommand`. The difference is that here there is only the `std::vector`-compatible version of the signature, since the variadic template version was provided only for usability reasons, and this class should not be used directly by the programmer.

5.2.3 Sound Playback

As we explained in the beginning of this chapter, the *VORPAL* middleware uses *OpenAL* for a number of reasons. Besides providing the system with the indispensable sound playback support, it is free software, it is capable of streaming real-time audio, and it implements 3D sound spatialization for us. The *OpenAL* API was designed to follow the *OpenGL* standard, which means it is provided as a `C` API where all routines have an `al` prefix and mostly work as a state machine, aside from some other naming conventions to specify parameter types. *OpenAL* lets us open **Devices**, which correspond to sound card drivers available in the platform, then create **Contexts** to each Device. Every Context applies only to a single Device. Each Context has a 3D virtual space with exactly one **Listener** and zero or more **Sources**, all of which have a 3D position. Each Device has a hardware-bound pool of **Buffers** available for storing audio data too. Buffers can be fed to Sources, causing the stored sound to be played in the Device to which the Context of that Source belongs (recall Figure 3.8), and the adequate 3D spatialization effects will be applied. There are two ways in which a Buffer can be fed to a Source: statically or through streaming. The first method is commonly used when the programmer has the whole sound signal ready to be played. The second, as expected, is used when the whole signal is not immediately available, but is otherwise obtained in sequential chunks of data from time to time. This second method uses a per-Source **Buffer Queue**, to which we can feed Buffers at different times, and it is guaranteed that their playback will happen following a First-In-First-Out order. The API offers a routine that checks for which Buffers in a Queue have already been processed and can be unqueued for other uses.

The DSP Server and DSP Units we discussed in Section 5.2.2 are responsible for keeping a set of real-time processed audio buses synchronized with their *Pure Data* counterparts. To actually play those sounds, the middleware needs to systematically forward the content of those buses to *OpenAL*. From what we have just explained, this means transferring the contents from these buses to Buffers, then assigning them to properly spatialized Sources. For that, we need to instantiate both a Device and a Context through the *OpenAL* API, then keep track of Buffers and Sources to make sure every sound is played in its proper place and time. We use the Buffer Queue since it fits perfectly our real-time restraints, even if its original purpose was to deal with encoded audio streaming.

As with DSP management, we implemented sound playback by defining two abstractions: the **Audio Server** and the **Audio Unit**. The first, corresponding to the `vorpal::AudioServer` class, is another singleton, this time responsible for interacting with the *OpenAL* API as a whole. The second, implemented by `vorpal::AudioUnit`, is used to encapsulate the behavior of a single Source inside the virtual 3D space of the *OpenAL* Context used inside the Audio Server, much like a DSP Unit individualizes a *Pure Data* Patch inside the DSP Server. Each Audio Unit is reference-counted and has two main features: streaming a sample array (of arbitrary length) to be played at the corresponding Source, and changing the actual 3D point in space where the corresponding Source is currently placed at. The role of the Audio Server, on the other hand, is to manage available resources in the sound card – Sources and Buffers, specifically –, to map each Audio Unit to the associated Source, and to keep the Buffer Queues of each Source in check – which means unqueueing Buffers that have already been processed so they can be used again

in future playback requests. As before, the rest of this section is dedicated to address the main methods of the classes discussed, assuming that all signatures are declared within the `vorpal` namespace.

`vorpal::AudioServer` Class

- `AudioServer::loadUnit`

```
std::shared_ptr<AudioUnit> AudioServer::loadUnit();
```

When invoked, this method provides a reference-counted Audio Unit instance. If no more Sources are available, it returns a Null Object implementation of Audio Unit.

- `AudioServer::availableBuffers`

```
size_t AudioServer::availableBuffers() const;
```

This method tells how many Buffers are currently available in the Audio Server. We explain in Section 5.2.4 how we use this to implement the real-time soundtrack synchronization of the *VORPAL* Audio Engine.

- `AudioServer::update`

```
void AudioServer::update() const;
```

Invoking this method causes the Audio Server to check the Buffer Queues of all Sources, unqueueing already processed Buffers back to the Buffer pool.

`vorpal::AudioUnit` Class

- `AudioUnit::stream`

```
void AudioUnit::stream(const std::vector<float> &signal);
```

Transfers the given sample array to a Buffer (if there is any available), then queues it in the corresponding Source for playback. This is called by `DSPUnit::transferSignal` whenever the assigned DSP Unit (see Section 5.2.4) receives its signal from its *Pure Data* Patch.

- `AudioUnit::setPosition`

```
void AudioUnit::setPosition(float x, float y, float z);
```

This method effectively changes the 3D spatial position of the associated Source inside the *OpenAL* Context. Called by `SoundtrackEvent::setPosition`.

5.2.4 Real-Time Soundtrack Processing

In Sections 5.2.1, 5.2.2, and 5.2.3, we introduced all the relevant classes inside the implementation of the *VORPAL* Audio Engine. Here, we will discuss how they are all tied together to produce the main subject of our interest in this research: real-time soundtracks. It is very clear that we have divided the Engine internal organization in two big parts: the DSP related classes and the audio related classes. The DSP classes capture the results of *Pure Data* Patches and keeps them in DSP Units. The audio classes must eventually receive these results in the Audio Units, distributing the obtained signals to the corresponding *OpenAL* Sources, thus achieving playback of the sound originally produced by the sound designer in the *VORPAL* Soundtrack Creation Kit. Since each Patch is abstracted as a Soundtrack Event to the programmer in the most external interface of the Engine (even though inside we know that is the role of a DSP Unit), it means that `vorpal::SoundtrackEvent` essentially *bridges* DSP Units to Audio Units. In fact, it is practically a thin interface that holds an instance of each, and maps its method invocations directly to the corresponding methods in each Unit. This is evident from the signatures exposed in the previous sections. `vorpal::SoundtrackEvent` is implemented as a convenient wrapper for the game programmer, but its abstraction as a soundtrack "object" still holds. The most important connection between DSP and Audio Units is the `DSPUnit::transferSignal` method, which, as we have explained, is responsible for bringing the sound from the *Pure Data* domain to the *OpenAL* domain through the DSP-Audio Unit pair.

However, being able to transfer the signal from one side to the other is far from enough. The key problems in developing a real-time soundtrack engine are: knowing *how much* signal to play and *when* to play it. We pointed this out back in Section 3.3.4, and we now must address this design decision. Given that *Pure Data* processes signal in Ticks of 64 samples and that *OpenAL* can send Buffers of arbitrary length to the sound card, our limiting factor is *Pure Data*. We established that the standard frequency of the system is 44100 Hz, which means that a *Pure Data* Tick stands for 1.45 milliseconds of audio, approximately. That is short enough to avoid a perceptible latency (in fact, *Pure Data* would lose most of its usefulness if it did not support immediate sound feedback). After some empiric tests, we also noted that 64 samples is long enough to not overload the data transfer to the sound card. Thus, flushing all DSP Units to their respective Audio Units every 1.45 milliseconds is our first attempt at real-time synchronization in the Audio Engine.

Notwithstanding, there is another problem we must deal with. As we explained a number of times before, the sound card has only a limited memory. There can only be so many Buffers allocated at a given time. When there are no Buffers available, we cannot process any more audio. In fact, if there are fewer Buffers than Soundtrack Events, then there are not enough resources for a full soundtrack Tick to be processed, as every Soundtrack Event would require exactly one Buffer to be queued in its Audio Unit to produce enough sound for the current 1.45 milliseconds of soundtrack time. Consequently, our soundtrack synchronization mechanism must keep track of available Buffers (which is done through the `AudioServer::availableBuffers` method), and hold back soundtrack playback if necessary.

Having discussed all that, we now present the final algorithm used to synchronize the game soundtrack, that is, the implementation of the `Engine::tick` method. Since it is an idle-frame

synchronized sub-system service for the host application, it receives a parameter containing the amount of time elapsed since the last frame in the Game Loop. It is also called at potentially irregular intervals, but that are somewhat guaranteed to be as often as possible by the game application. Since we only process the soundtrack following *Pure Data* Ticks of 64 samples, we cannot execute these Ticks every time the `Engine::tick` is called. Not only that, but if it is called after too long an interval, we need to compensate for that by processing multiple Ticks. For that, we keep track of how much time has passed since the last Tick, a value we call the **lag** of the Audio Engine. If it is smaller than the Tick period (1.45 milliseconds), then we should not process any Ticks at the moment. If it is larger, then we have to process as many Ticks as possible such that the sum of the time intervals they correspond to comes as close as possible to the accumulated lag. At the same time, we should cancel Ticks if there are no more Buffers available. The resulting code is shown in Listing 5.3 in a simplified version that omits some C++ nuances and implementation details such as logging.

```
// A Tick size in seconds, approximately 1.45 milliseconds
const double TICK = 64.0/44100.0;
void Engine::tick(double dt) {
  // Variables ending in __ are static
  lag__ += dt;
  // Update Audio Server to unqueue finished buffers from sources
  audioserver.update();
  // Close unreferenced Patches
  dspserver.cleanUp();
  // Flush Command Messages to Patches
  dspserver.handleCommands();
  // Process Ticks according to current lag and Buffer availability
  while (lag__ >= TICK && audioserver.availableBuffers() >= events__.size()) {
    // Process one Tick
    dspserver.processTick();
    // Transfer audio from DSP Unit buses to Audio Unit Buffers
    for (shared_ptr<SoundtrackEvent> event : events__)
      event->processAudio();
    // Update lag
    lag__ -= TICK;
  }
}
```

Listing 5.3: Simplified Engine Tick implementation.

5.3 Soundtrack Creation Kit

While the Audio Engine is responsible for the real-time playback of soundtracks from inside the game application, it is the role of the Soundtrack Creation Kit to provide an authoring interface for the sound designer to make the soundtrack in the first place. In Section 4.4.2, we explained that this Kit is provided as a set of *Pure Data* Abstractions and Externals that build a soundtrack control interface on top of the *Pure Data* language and application. In this section, we describe

in more detail how each of those Abstractions and Externals work, and how they are integrated into the middleware as a whole.

5.3.1 Output Bus

As discussed in Section 5.2.3, we cannot use the standard *Pure Data* audio output because it sums all incoming signals, making it practically impossible for us to apply a 3D spatialization effect later in the Audio Engine when we do have access to game object positions. Instead, the adopted solution was to use *Pure Data* Arrays as per-Patch individual output buses. As such, the first and most important Abstraction we provide in the Soundtrack Creation Kit is the **Output Bus**, in the form of the `[vorpal_bus]` Object. It should be used just like a `[dac~]` Object, except it only accepts one Signal Connection (mono sound) instead of two (stereo) sound. This is due to one of the problems we addressed on De Lucca's interview back in Section 4.2.3. You cannot have – or rather, there is no point in having – stereo playback together with spatialized audio, because the binaural effect will just play both stereo channels as if they came from the same source position. We chose to offer the alternative of using two separate Audio Units to achieve the effect of stereo audio instead.

Section 5.2.3 also explained that Arrays are tagged with the Dollar Zero of the Patch that contains them, and the `vorpal::DSPServer` in the Audio Engine uses the `pd::Patch` class to obtain the Zero Dollar of loaded Patches, thus allowing it to track the Output Buses of the Patches made by the sound designer. More specifically, given a loaded Patch instance, the `vorpal::DSPServer` searches it for an Array with name `"vorpal-bus-X"`, where `X` must be the Dollar Zero of that specific Patch instance. It is important to remember that different instances loaded from the same Patch have different Dollar Zeroes, which makes their Output Buses completely independent from one another. For instance, if there is a certain monster species in a game, and its specimens all like to gnarl and roar, the sound designer would only need to make exactly one Patch for their sound, and the programmer would instantiate it for every monster specimen, giving each of them their own voice to threaten the player with.

Since each Output Bus has its own Array, we made use of this to also align its implementation with System Requirement 5, "Variable Mix". Every `[vorpal_bus]` has two Inlets: one for the output signal that Patch wants to send to the Audio Engine, and one for setting the volume of that Output Bus. It receives a number from zero to one, which simply directly multiplies the output signal. When creating a `[vorpal_bus]` Object, the sound designer **must** pass the Dollar Zero as the only creation argument. That is, it should always be created as `[vorpal_bus $0]`. Internally, the Dollar Zero value is captured by the `$1` directive, and is used to name the Array where the received signal is stored. The Array itself is created with a capacity for 64 samples, being completely overwritten every *Pure Data* Tick (thanks to the `[bang~]` Object). Additionally, the Abstraction has a feature that allows the sound designer to hear a preview of the resulting sound of that Patch. We explain this in Section 5.4.3. Figure 5.2 shows the Output Bus Abstraction implementation. The blue part is how the `[vorpal_bus]` looks when created in a parent Patch.

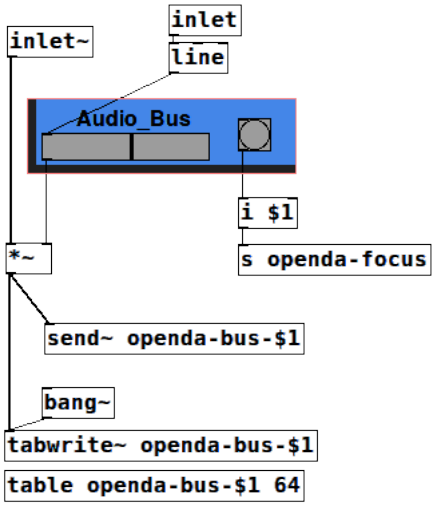


Figure 5.2: The implementation of the Output Bus Abstraction in the Soundtrack Creation Kit.

5.3.2 Commands

Following our abstraction that soundtrack Patches should behave like objects from Object-Oriented Programming, we explained in Section 4.4.3 that Command Objects are provided in the Soundtrack Creation Kit for the purpose of receiving the "method invocations" that come from the game application through the Audio Engine. On that side, the programmer calls the `vorpal::SoundtrackEvent::pushCommand` method, passing the Command name and an arbitrary quantity of numbers and symbols as parameters. On this side, the sound designer's Patch must create a Command Object with the exact same name, and after that, whenever the game pushes the specified Command, a Message will be sent out from the Command Object containing the list of given parameters.

Again, the *VORPAL* middleware relies on the Zero Dollar feature of *Pure Data* to map `C++` class instances to *Pure Data* Objects. Command Objects, implemented as the Abstractions `[vorpal_command]`, require two creation arguments: the Zero Dollar of the soundtrack Patch (like with Output Buses), and the Command name. The first argument is used to create a `[receive]` Object, which is the *Pure Data* Object for receiving Messages from anywhere, including through the `libpd` API. Next, we use the `[route]` Object to filter the incoming Messages, letting only the ones starting with the given Command name pass. Lastly, we simply forward the rest of the Message – which contains the parameters provided by the game code – and send them through the Outlet of the Command object. This implementation can be seen in Figure 5.3. The gray part is what the Abstraction exposes to the sound designer.

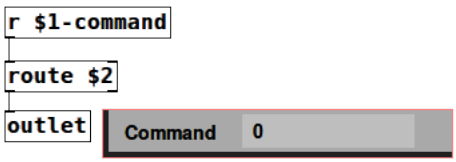


Figure 5.3: The implementation of the Command Abstraction in the Soundtrack Creation Kit. The place with a zero changes to the Command name when it is created on a sound designer's Patch.

5.3.3 Music Sequencing

As explained in Section 4.4.2, some key Abstractions distributed in the *VORPAL* Soundtrack Creation Kit are associated with a **Sequencer** feature, with System Requirement 2 in mind. There are many possible ways one could sequence a music or some other sonic piece. By suggestion from the interviewed sound designers, we opted for something simple as a starting point. The Sequencer in *VORPAL* is designed to trigger events following a looping and fixed 16-slot grid with variable Tempo. That is, when the sound designer creates a Sequencer in the soundtrack, he or she must also specify an initial BPM rate, then possibly assign behavior triggers to each slot among the 16 available ones. A behavior trigger could range from playing a single synthesized note to cross-fading a whole set of ambiance samples organized in a greater loop sequence. Additionally, each individual slot can be either activated or deactivated, muting the corresponding trigger.

Differently from previous Soundtrack Creation Kit Abstractions, this one comes as a set of two Objects: `[vorpal_metro]` and `[vorpal_seq16]`. The first implements a metronome with variable BPM and time signature, and can be turned on and off at any time. It simply outputs a number Message from zero to the time signature count minus one at double the given tempo. By itself, the `[vorpal_metro]` Object is already capable of providing a sequencing mechanism: the sound designer needs only match the beat numbers to the corresponding sonic behaviors. The `[vorpal_seq16]` is where the timed numbers from the metronome are treated and organized in a fixed 16-slot grid structure, and also where the slots may be blocked or activated for a finer control of what is supposed to be played in the sequence. It has two Outlets: one for the current slot position in the Sequencer (as sent by the metronome), and one that just sends a "bang" whenever a full loop cycle has been played. To combine these two Abstractions together, the Outlet from the metronome (with time signature set to 16) must be connected to the left Inlet of the `[vorpal_seq16]`, then the actual slots must be toggled (through the right Inlet) to enable their corresponding outputs. The actual implementation of `[vorpal_seq16]` looks rather confusing, but it is only a repetition of sixteen `[spigot]` Objects, which basically control whether the matching number from the metronome should pass or not. Both Abstractions are shown in figures 5.4 and 5.5, respectively, with the gray parts being the interface displayed to the sound designer.

5.3.4 Samples

There is still much that can be done with sample-based audio, and as such the *VORPAL* Soundtrack Creation Kit also supports it through its own **Sample** Abstractions. There is one Abstraction for each type of sound encoding, which are actually only two: WAV and OGG. WAV is the most straightforward format for audio, and *Pure Data* provides native Objects to handle it. OGG is an open format for compressed audio, which made it possible for us to find a community implementation for OGG decoding in *Pure Data*. Thus, the two Sample Abstractions are `[vorpal_sample]` and `[vorpal_ogg]`. The first makes use of the native `[soundfiler]` Object to read WAV files and store it in an Array, which we then read on demand. The `[vorpal_ogg]` is implemented similarly, except it uses the `[oggread~]` External. For the sake of brevity, we only illustrate `[vorpal_sample]` in Figure 5.6. Again, the blue part is the

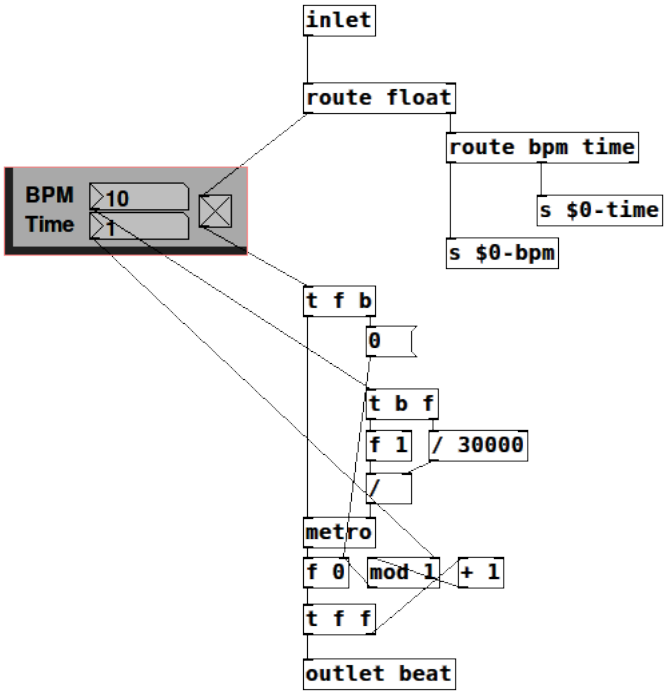


Figure 5.4: The implementation of the metronome Abstraction in the Soundtrack Creation Kit, part of the Sequencer feature.

interface exposed to the sound designer.

5.3.5 Sound Synthesis

From Chapter 1 to Chapter 4 we have advocated the use of procedural audio over sample-based audio. The only way to obtain a sound signal without reading it from a pre-established sample is to synthesize it at run-time. There are many kinds of sounds that can be synthesized this way, be it sound effects, voices, or music. *Pure Data* opens many possibilities in that regard. However, we found it necessary to provide some higher-level tools for sound synthesis in the Soundtrack Creation Kit, specially due to Santana and De Luccas's statements (Sections 4.2.1 and 4.2.3, respectively). Thus, we chose to implement some form of simple synthesis designed with music sequencing in mind. That is, the **Sound Synthesis** feature of the *VORPAL* Soundtrack Creation Kit is intended to provide a straightforward way of playing synthesized melodic notes to be composed together in greater musical structures, possibly with the help of the Sequencer feature. This is also in accordance with System Requirement 3 ("Real-Time Controlled Synthesis").

Still, there are many reasonable synthesizer implementations. We followed a suggestion from the interviewed sound designers and based our synthesizer on a commercial analogue synthesizer, more specifically the *Minimoog* model from *Moog Music*[12]. That is, we implemented our own synthesizer using the *Minimoog* features and design as reference. The choice of model itself is arbitrary, since the point is only to provide *some* way of meeting System Requirement 3. *Minimoog* is a synthesizer that looks like the keyboard instrument, with the sound played by its keys being synthesized according to its settings. It has three oscillators, a noise generator, a

[12] https://www.moogmusic.com/products/Minimoog

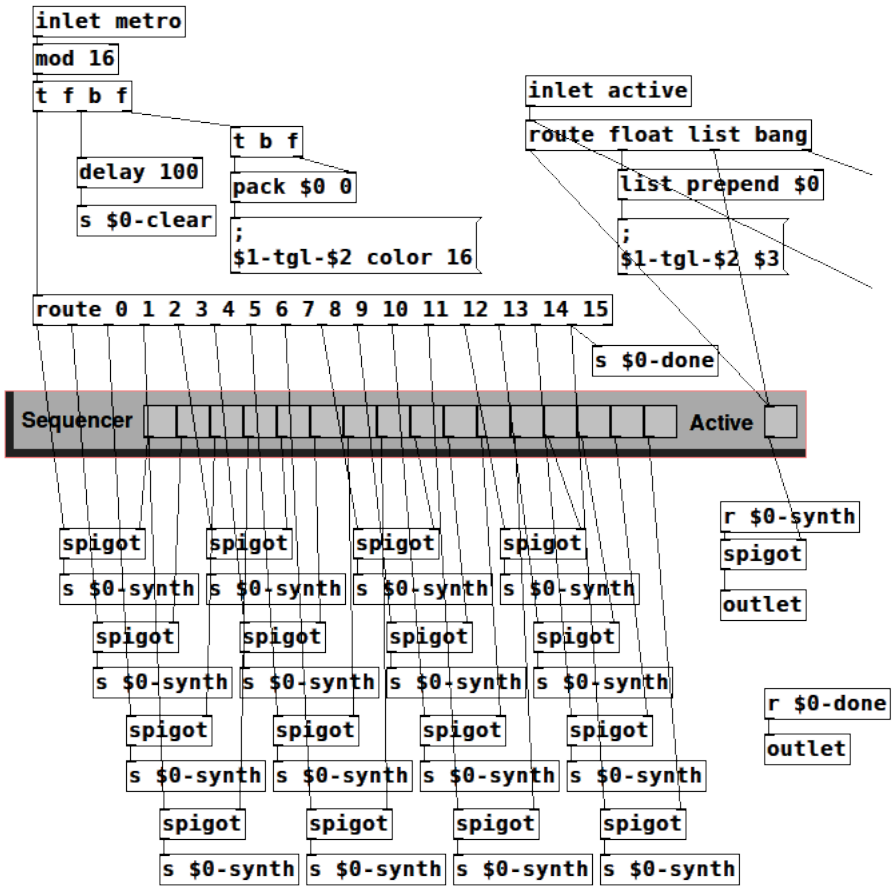


Figure 5.5: The (partial) implementation of the 16-slot grid Abstraction in the Soundtrack Creation Kit, part of the Sequencer feature. The omitted part is just for cleaning up the grid.

frequency filter, two envelope controllers (one for the oscillators and one for the filter contour), and a number of switches that route the signal across the device, specially with the purpose of using an oscillator to modulate some other control. Figure 5.7 presents a photograph of the *Minimoog* synthesizer.

Following the design of the *Minimoog*, we broke down the Sound Synthesis feature of the Soundtrack Creation Kit in three Abstractions: an oscillator (`[vorpal_osc]`), a frequency filter (`[vorpal_filter]`) and an envelope controller (`[vorpal_env]`). The idea is that one may reproduce the effects of the *Minimoog* by properly combining Objects of these Abstractions, but the sound designer is free to connect them in different ways too. The `[vorpal_osc]` Object supports five different waveform types: sinusoidal waves, phasor waves, triangle waves, square waves, and short rectangular waves. Besides the waveform, the sound designer can also set the tonal range, the pitch shift, and the amplitude of the generated signal. The left Inlet may also be used to modulate the frequency of the oscillator. The `[vorpal_filter]` has two main controllers, one for the cutoff frequency, and one for the emphasis factor. It operates as a *Voltage Controlled Filter* (VCF), for which there already is a *Pure Data* primitive object, `[vcf~]`. Our Abstraction simply provides a more user-friendly interface to it, in addition to Inlets for frequency modulation and contour envelope control. Lastly, the `[vorpal_env]` Object

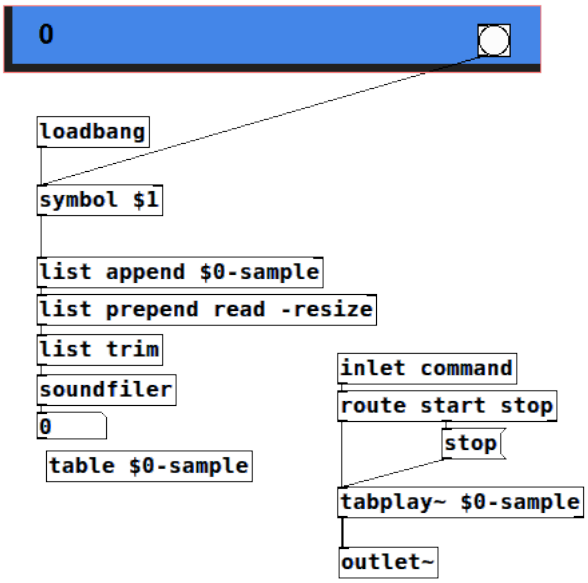


Figure 5.6: The implementation of the WAV Sample Abstraction in the Soundtrack Creation Kit. The zero is replaced by the path to the sample file when the sound designer creates an Object from this Abstraction.

Figure 5.7: The *Minimoog* synthesizer [Har09].

implements a typical *Attack-Delay-Sustain-Release* (ADSR) envelope, which can be toggled on or off to emulate pressing, holding, and releasing a keyboard key. Its Outlet outputs the envelope signal to be used with `[vorpal_osc]` and `[vorpal_filter]` Objects. Figure 5.8 presents the interfaces of these Objects when created in the sound designer's Patch.

5.4 Middleware Usage

Having explained the main aspects of the *VORPAL* middleware inner workings, we now discuss what the users actually see and use when they interact with the system. This includes everything from the process of obtaining a copy of the middleware and setting it up in a development environment (Section 5.4.1), to actually implementing Audio Engine integration in the game code (Section 5.4.2) and creating Soundtrack Events with the Soundtrack Creation Kit (Section

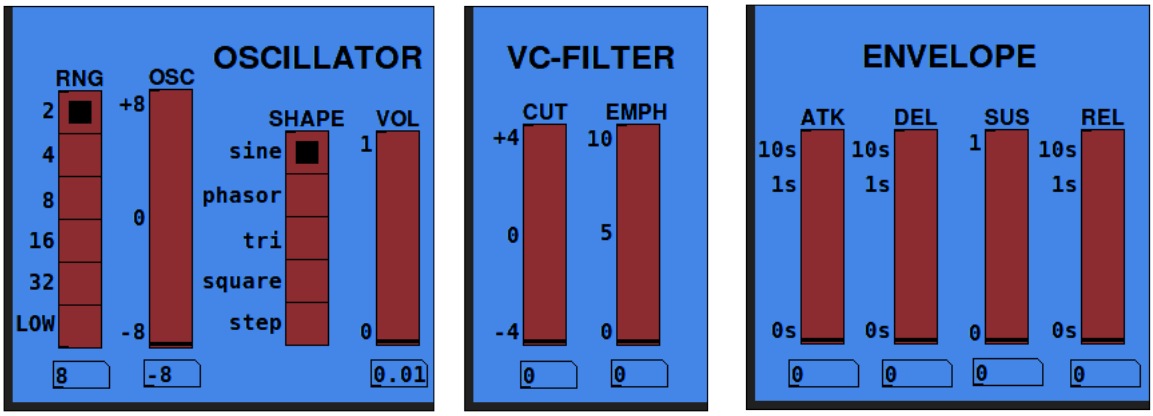


Figure 5.8: The interfaces of the Sound Synthesis Objects from the Soundtrack Creation Kit. From left to right: the oscillator (`[vorpal_osc]`), the VCF (`[vorpal_filter]`), and the ADSR envelope (`[vorpal_env]`).

5.4.3). However, more often than not developers rely on a game engine to make their games, and thus an important issue to address with game audio middleware is how to integrate it with third party systems like game engines of the most diverse types (Section 5.4.4). Lastly, we find that presenting some concrete examples of the middleware usage *VORPAL* was designed for is paramount to understanding its contributions, possibilities, and limitations (Section 5.4.5).

5.4.1 Distribution

As we mentioned in Section 1.2, the *VORPAL* middleware is available as free, open source software (Mozilla Public License 2.0) at a repository hosted in https://github.com/vorpal-project. The current means of distribution consists of publishing two downloadable archives, both based on a specific stable version of the repository. One of the archives contains the source and build files needed to compile the Audio Engine, while the other carries the *Pure Data* Patches the Soundtrack Creation Kit comprises. We do not provide any pre-compiled binary versions of the Audio Engine for now, but since the system has very few dependencies (practically *OpenAL* only), compiling from scratch is very straightforward. All that is necessary is a minimally recent version of *CMake* and a `C++11` conforming compiler. On the other hand, the sound designer only needs the Soundtrack Creation Kit and *Pure Data* itself. The two following sections explain how programmers and sound designers make use of the middleware upon acquiring each of these archives, respectively.

5.4.2 Programmer's Workflow

The distributed Audio Engine archive bundles all that is needed to compile the *VORPAL* Audio Engine, except for *OpenAL*, which must be installed in the user's system separately. We bundle the `libpd` source code together with the Audio Engine as a *Git Submodule*[13], so there is no need to install it separately. After building the Audio Engine with *CMake*, the programmer may choose to install it wherever he or she prefers. Then, it is only a matter of including the headers

[13] https://git-scm.com/docs/git-submodule

and linking the game code to the Audio Engine, since it presents itself as a usual programming library does. The only header that needs to be included is shown in Listing 5.4.

```
#include <vorpal/vorpal.h>
```

Listing 5.4: Including middleware headers.

Once that is done, the next step is to initialize the `vorpal::Engine` Singleton, then create instances of `vorpal::SoundtrackEvent`. To do that, it is important to register the paths where the Patches are to be loaded from (using `vorpal::Engine::registerPath`) – this includes the Patches from the Soundtrack Creation Kit! Everything should be put into folders the game application has access to. Since `libpd` uses its own code to open and read those files, this might not be that simple on certain platforms (like mobile phones). Aside from that, `vorpal::SoundtrackEvent` instances should be kept to ensure that their reference-counting garbage collection does not prematurely free them from memory. For instance, each game entity (players, monsters, items, etc.) could carry its own Soundtrack Event object with them. Entities of the same type would use instances diverged from the same Soundtrack Event, so that every dragon sounds about the same (and yet each with its own independent sound instance) but a fireball sounds very differently from a singing bard. A simple way to do this is to name the Soundtrack Events after the type of entity they represent. The resulting code could be something to the effect of what is presented on Listing 5.5.

```
vorpal::Engine engine;
vorpal::Status status;
// Try to initialize the engine
if (!(status = engine.start()).ok()) {
  std::cout << "Error: " << status.description() << std::endl;
  exit(-1);
}
// Register path to Patches
engine.registerPath("path/to/patches");
// Load an Event for each game entity. Assume a fictional entity list.
for (GameEntity *entity : entities_list()) {
  std::shared_ptr<vorpal::SoundtrackEvent> event;
  if (!(status = engine.eventInstance(entity->get_type(), &event)).ok()) {
    std::cout << "Error: " << status.description() << std::endl;
    exit(-1);
  }
  entity->set_soundtrack_event(event);
}
```

Listing 5.5: Mapping Events to game entities.

Once the Soundtrack Events are instantiated, all that is left is to keep the Audio Engine synchronized with the Game Loop by invoking `vorpal::Engine::tick` at the appropriate times and to update information about the game state through the corresponding invocation of `vorpal::SoundtrackEvent::pushCommand`. The Game Loop synchronization could be

done through a code roughly like Listing 5.6[14].

```
vorpal::Engine engine;
double previous = getCurrentTime(); // in milliseconds
double lag = 0.0;
while (true) // Game Loop
{
  double current = getCurrentTime();
  double elapsed = current - previous;
  previous = current;
  lag += elapsed;

  processInput();

  while (lag >= MS_PER_UPDATE)
  {
    update(); // game simulation step
    lag -= MS_PER_UPDATE;
  }

  render(lag / MS_PER_UPDATE);
  engine.tick(elapsed / 1000.0); // convert to seconds
}
```

Listing 5.6: Possible Game Loop.

To keep game state information up-to-date inside the Audio Engine, the programmer needs to report any relevant changes and triggers inside the simulation to the appropriate instances of `vorpal::SoundtrackEvent`. Following our example where every game entity has one Soundtrack Event, assume now that we want to inform the Audio Engine of their 3D positions and health status. The code could be like Listing 5.7.

```
void GameEntity::set_position(Vector3 new_position) {
  this->soundtrack_event.setAudioSource(new_position.x, new_position.y,
                                        new_position.z);
  this->position = new_position;
}
void GameEntity::takeDamage(uint32_t amount) {
  this->health = max(this->health - amount, 0u);
  this->soundtrack_event.pushCommand("health", this->health);
}
```

Listing 5.7: Possible game entity methods.

In summary, the game programmer must find all soundtrack sensitive parts in the code and push the adequate Commands, which are the only way in which the soundtrack (that is, the set of Patches provided by the sound designer) can know what, when and how everything is supposed to be played.

[14] This Game Loop code is based on the suggested implementation from Nystrom [Nys14].

5.4.3 Sound Designer's Workflow

For the sound designer, the first step is to install *Pure Data*, preferably the latest version (also remember that we use *Pure Data* "Vanilla"). Next, he or she should acquire the *VORPAL* Soundtrack Creation Kit from the corresponding archive, extracting the Patches into a folder pertaining to the search paths of *Pure Data* (which can be set up through the application interface). Then, the sound designer must define a directory where all soundtrack Patches of the game will be stored in together. This directory is what should be delivered to the programming team so they can embed the Patches into the game using the Audio Engine. If there are any other Abstractions or Externals the soundtrack demands, they should be placed in that directory too[15]. Lastly, it is important to take notice of the Patch names. When loading them as Soundtrack Events inside the game, the programmer will refer to them by name, which is the part of the Patch file name that comes before its extension (`.pd`). Thus, the naming convention used should be agreed upon between the sound and programming teams.

Once the working environment is set up, the sound designer may begin creating the Patches that the game soundtrack will comprise. First of all, every Soundtrack Event Patch requires at least one Output Bus Object to work with the middleware. To correctly create an Output Bus Object, the Patch Dollar Zero must be provided, as in `[vorpal_bus $0]`. It is worth remembering that this Abstraction has two Inlets. The left one receives the output signal from that Patch, and the right one receives a volume control value to set to that bus. To listen to what that output sounds like, we developed a special, optional Patch in the Soundtrack Creation Kit called the **Panel**. Differently from the rest of the Soundtrack Creation Kit, it must be opened as a separate Patch together with the Patches the sound designer is working on. Then, they must activate the Output Bus by clicking on the empty circle present in its interface (refer to Figure 5.2). Now, when DSP is turned on in *Pure Data*, the signal going through that Output Bus will be sent to the Panel, where it will be played for the user to hear. However, if the sound designer prefers, he or she may just use a `[dac~]` Object to directly capture and play the sound that goes into the Output Bus Object. That is why the Panel is only optional. Figure 5.9 shows a very simple example of how to use the Output Bus Abstraction.

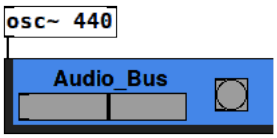


Figure 5.9: A very simple Patch using the Output Bus Abstraction. In this example, a sinusoidal soundwave is sent to the Output Bus at a frequency of 440 Hz.

Having the Output Bus Object set up allows sound to be sent from the Patch to the Audio Engine. To do the opposite – receive information *from* the Audio Engine – the sound designer uses the Command Abstraction. Each `[vorpal_command]` Object, created with the right Dollar Zero and Command name, will output a Message containing the Command parameters whenever the method `vorpal::SoundtrackEvent::pushCommand` is invoked from the game code.

[15] Depending on the case, the programmer might need different versions of the Externals, since their format is platform dependent.

This behavior can be easily simulated using native *Pure Data* Messages Objects. Figure 5.10 shows how the parameters sent with a Command "hero-jump" could be used inside a Patch.

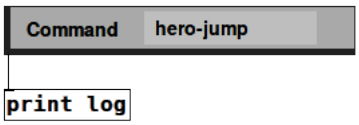


Figure 5.10: A very simple Patch using the Command Abstraction. The parameters from Command "hero-jump" are sent to the `[print]` Object.

By combining these two main Abstractions with the other tools provided in the Soundtrack Creation Kit (Sequencer, Synthesizer, etc.), the sound designer can compose the game soundtrack with all the control he or she wants. Commands could trigger from individual notes to whole automated mixing movements, transitioning from different ambiances or narrative moments. Both sample-based and procedural audio can work together to achieve yet unexplored aesthetic possibilities in games.

5.4.4 Game Engine Integration

Game developers do not always have access to the native `C++` code underlying the application, specially when using data-driven game engines. In this case, the usual method of including third party systems is through some sort of *plug-in mechanism*, where small or medium scale pieces of software can be easily linked (either statically or dynamically) with the game engine code. This mechanism is responsible for detecting which plug-ins are currently bound to the engine, then registering them so that the game code and content may refer to the third party features later. For instance, a game engine may receive a plug-in that enables Virtual Reality input-output support, or maybe some non-standard advanced rendering technique for ultra-realistic visual effects, etc. Essentially, game engine plug-ins are necessary to extend the basic feature set provided.

As such, the *VORPAL* middleware must follow the same protocol to be used in games powered by game engines, a context in which our system clearly would be considered a third party software. However, we do not need to port every part of the middleware for it to become fully compatible with an engine. All we need is to write a plug-in encapsulating the Audio Engine. The Soundtrack Creation Kit is only necessary for the sound designers, who should work directly on *Pure Data* even if the game is developed on a data-driven game engine. The users must still deliver the Patches from the Soundtrack Creation Kit together with the soundtrack Patches, though. Once they are appropriately placed into a directory the game application can reach, the Audio Engine, now embedded into the *game* engine, will find and load them accordingly.

We illustrate this process by describing how we implemented such plug-ins for two separate free software game engines: *LÖVE* and *Godot*. The first is not a data-driven engine, while the second is. This allows us to show not only that the middleware is compatible with both kinds of engines, but also how the corresponding plug-in should be implemented in each case. It should not be difficult to understand how to generalize this process for other engines, even commercial ones. Thus, the rest of this section is dedicated to discussing the implementation we provide of *VORPAL* plug-ins for the *LÖVE* and *Godot* game engines.

LÖVE Integration

Figure 5.11: *LÖVE*'s logo [Tea16].

LÖVE is a 2D focused game engine in which everything is done by programming in a *Lua* environment. A comprehensive API is provided with arguably everything a 2D game might need, and there is no software architecture imposed – although the default implementation comes with a simple idle-synchronized Game Loop. The engine itself is programmed in `C++`, then wrapped as *Lua* modules. This means that extending *LÖVE* is as simple as adding a new *Lua* module to the game project folder. Thus, to implement a plug-in for this game engine we basically need to write *Lua* bindings for the *VORPAL* Audio Engine. There are many sources available on how to write *Lua* bindings, including some of our previous work [Miz].

Essentially, a *Lua* binding (like with many other script languages) is done by compiling a dynamic library containing some `C` routines with specific signatures and naming convention, which the host program looks for using some platform-specific API (for instance, `dlopen` in *Linux* systems). Even though these special routines must be compiled as "pure" `C`, the library as a whole needs not, which allows us to easily bind `C++` code. Since the code necessary to bind the Audio Engine was very small and straightforward, we simply added an optional build target to the *VORPAL* repository, which basically compiles `src/lua/vorpal_lua.cxx` in our source tree as a *Lua*-compatible dynamic library. In that file, we provide bindings for the following classes and methods:

- `vorpal::Engine::start`
- `vorpal::Engine::finish`
- `vorpal::Engine::registerPath`
- `vorpal::Engine::tick`
- `vorpal::Engine::eventInstance`
- `vorpal::SoundtrackEvent::pushCommand`
- `vorpal::SoundtrackEvent::setAudioSource`

Since `vorpal::Engine` is a Singleton, its *Lua* binding treats it as a module (the *Lua* equivalent, roughly). On the other hand, `vorpal::SoundtrackEvent` is bound as a proper class with instances and methods. In Section 5.4.5 we show a concrete example of how to use these bindings.

Godot Integration

Godot has a scripting language of its own, called *GDScript*. It is a strictly Object-Oriented language with a syntax that closely resembles *Python*. To make *Godot* plug-ins that extend that

Figure 5.12: *Godot*'s logo [Stu16].

language by adding third party features, *Okam Studio* has developed a modular mechanism in the build system of the engine. In short, by placing the appropriate files in a special folder inside the *Godot* repository, its build system (written in *Scons*[16]) automatically compiles the code in that folder into the engine. Compared to *LÖVE*, it involves a more advanced understanding of large-scale `C++` application building. The files required for a *Godot* module like this are:

- The source code of the third party feature;
- Some glue code that binds the features as *GDScript* classes; and
- Two *Scons* configuration files with compilation flags and the like.

Since this module mechanism works by simply placing a folder inside the *Godot* build tree, we preferred to implement our *Godot* bindings in a separate repository, which is to be cloned directly inside the *Godot* repository. It currently supports only *Godot* 2.1, which, as of this writing, is the latest stable version of the engine. Our bindings do not include the *VORPAL* Audio Engine. Instead, we assume it is installed in the programmer's environment. Here we took a different approach from *LÖVE* when implementing the Audio Engine bindings. We wrapped both `vorpal::Engine` and `vorpal::SoundtrackEvent` in a single *GDScript* class called `VorpalModule`, which contains a list of all created Soundtrack Events and operates on them by providing and using a unique identifier integer for each Event – essentially their index on the list. We chose this shortcut since this binding was part of the *Sound Wanderer* project, and we had other, more important priorities to focus on in that game. In fact, the *Sound Wanderer* code base is the best example currently available of how to use the *VORPAL Godot* bindings.

5.4.5 Examples

We now discuss two usage examples of the *VORPAL* middleware. The first example is a rather artificial implementation in pure `C++` that serves to illustrate how the API should be used. The second is more illustrative, comprising a micro-game – more of a "toy" actually – where you can control a simple amplitude modulation by moving a circle with the keyboard directional keys. Both examples are available at a separate *GitHub* repository: https://github.com/vorpal-project/examples.

Example 1: Native API

This example is located at the `src/basic` directory in the repository tree. It simulates how a Game Loop behaves by sleeping a random amount of time every frame, then using that amount

[16] http://scons.org/

to tick the Audio Engine. There is only a single soundtrack Patch loaded during the example execution, which by default is a Patch called "basic" in the patches folder. It plays a sinusoidal wave whose frequency changes as times passes in the "game". The resulting soundtrack is not quite interactive, but our main concern here is how the game code uses the Audio Engine API. The source file src/basic/main.cxx contains all the code for this example. We now explain its main parts.

```
1  int main (int argc, char** argv) {
2    string event_name = "basic";
3    if (argc >= 2) {
4      event_name = argv[1];
5    }
6    srand(time(0));
```

Listing 5.8: File src/basic/main.cxx, lines 27-32.

The main code starts by checking whether the user specified a Patch name (Listing 5.8). If he or she has not, it defaults to "basic" as explained before. Then we initialize the standard random number generator with a seed, since we use randomly generated numbers to simulate the variability of frame duration in a Game Loop. Next, we initialize the Audio Engine while checking for errors. Notice how we use the parameter in the vorpal::Engine::start method to provide search paths for loading Patches in line 3 of Listing 5.9. The first one is a relative path to the patches folder in the repository tree, while the second is a macro containing the path to the Soundtrack Creation Kit Patches installed in the system. As explained in Section 5.4.3, another way to do this would be to place the Soundtrack Creation Kit Patches in the same directory as the soundtrack Patches of the game. Listing 5.10 presents the part that loads the Soundtrack Event. As with the Audio Engine initialization, it checks for errors. If there is any, it clears all used resources through vorpal::Engine::finish at line 6 before exiting the application.

```
1    vorpal::Engine engine;
2    {
3      vorpal::Status status = engine.start({"../patches", VORPAL_PATCHES_PATH});
4      if (!status.ok()) {
5        cout << "Error: " << status.description() << endl;
6        return -1;
7      }
8      cout << "Opened device " << status.description() << endl;
9    }
```

Listing 5.9: File src/basic/main.cxx, lines 34-42.

Listing 5.11 contains the simulated Game Loop. It forcefully runs for only ten seconds before terminating the game execution. This is done by accumulating passed time in the seconds variable. We calculate the frame duration by keeping track of the moment the last frame started in the variable last (lines 2 and 5) and computing the difference between it and the current frame (line 4). We use the Clock type, which is an alias we made for a system clock class in

```
1    shared_ptr<vorpal::SoundtrackEvent> ev;
2    {
3      vorpal::Status status = engine.eventInstance(event_name, &ev);
4      if (!status.ok()) {
5        cout << "Error: " << status.description() << endl;
6        engine.finish();
7        return -1;
8      }
9    }
```

Listing 5.10: File `src/basic/main.cxx`, lines 45-53.

the `C++11` standard. Since it uses an internal implementation-dependent representation of time, we just cast it to `double` and divide by the known duration of a whole second. Line 6 calls `gameTick`, which we discuss ahead and represents a frame of the Game Loop. It provides the Soundtrack Event by parameter so it can send Commands to it. Then we tick the Audio Engine at line 7 and finish the frame by updating the total time of execution in seconds. Lastly, we define the `gameTick` routine in Listing 5.12. Line 3 calculates a random amount of time the simulated game frame will "last", choosing from 16 to 32 milliseconds (somewhere between 60 FPS and 30 FPS, respectively). Lines 4 accumulates the `delta` values to send in discretized step values as a Command to the soundtrack Patch – that is, it sends how many whole seconds have passed by truncating the floating point value. Lastly, the `gameTick` routine sleeps for the chosen amount of time in line 6.

```
1    double seconds = 0.0;
2    auto last = Clock::now();
3    while (seconds < 10.0) {
4      double delta = 1.0*(Clock::now() - last).count()/ONE_SECOND.count();
5      last = Clock::now();
6      gameTick(delta, ev);
7      engine.tick(delta);
8      seconds += delta;
9    }
```

Listing 5.11: File `src/basic/main.cxx`, lines 55-63.

The "`basic`" Patch can be seen in Figure 5.13. It uses the Output Bus and Command Abstractions from the *VORPAL* Soundtrack Creation Kit to synthesize and output a sinusoidal sound wave at varying frequencies, calculated from the `step` Command parameter sent by `gameTick`. Essentially, the "time steps" from the "game" are controlling the sound pitch in real-time, albeit not interactively. It does react properly to the random sleep periods, evidencing the robustness of the Audio Engine.

Example 2: Integrating with *LÖVE*

This example is located at the `src/demo` directory in the repository tree. It contains a simple interactive application (still not quite a game) written using the *LÖVE* framework. When it runs, the user acquires control over a circle avatar in front of a black background inside a 2D space.

```
1  void gameTick (double delta, shared_ptr<vorpal::SoundtrackEvent> ev) {
2    static double total = 0.0;
3    auto sleep_time = Clock::duration(milliseconds(16 + rand()%16));
4    total += delta;
5    ev->pushCommand("step", static_cast<int>(total));
6    sleep_for(sleep_time);
7  }
```

Listing 5.12: File `src/basic/main.cxx`, lines 70-76.

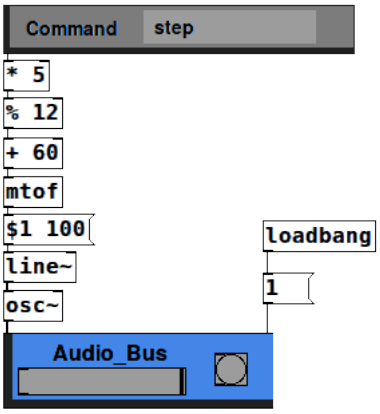


Figure 5.13: The "`basic`" Patch from the first example. It plays a sinusoidal wave with a frequency determined by the parameters sent through the `step` Command.

Using the directional keys in the keyboard, the user can move the circle around. The position of the circle is then fed to the Audio Engine both as a Command parameter and as the Soundtrack Event position. On the soundtrack Patch side, the `x` value of the position determines the pitch of a triangular wave from 60 up to 84 in the MIDI scale, while the `y` value determines the pitch of a rectangular wave from 84 down to 60 in the MIDI scale. Both waves are multiplied before being sent to the Output Bus of the Patch. As with the previous example, we now explain each of the main parts of the code and finish this chapter by presenting the Patch used. The example source code is mostly written in `src/demo/main.lua`, but there are two other auxiliary files which we do not cover here.

```
1  local vorpal = require "vorpal"
```

Listing 5.13: File `src/demo/main.lua`, line 4.

The code starts by `require`ing the *Lua* module that binds Audio Engine features, as shown in Listing 5.13. Modules in *Lua* often behave like "objects", so we can store them in variables and access their fields. Listing 5.14 presents a *LÖVE*-specific function that is one of the first called when the game runs. It provides the programmer a place to write initialization code. In this case, we initialize the *VORPAL* Audio Engine in line 2, register the same paths as the first example in lines 3 and 4, then load the Soundtrack Event for this game in line 5. The other initializations are omitted.

LÖVE provides the function depicted in Listing 5.15 for the programmer to implement what the Game Loop should do (using idle-frame synchronization by default). In this example, it merely

```
1  function love.load ()
2    vorpal.start()
3    vorpal.registerPath("../patches")
4    vorpal.registerPath(VORPAL_PATCHES_PATH)
5    ev = vorpal.eventInstance "demo"
6    -- *snip*
7  end
```

Listing 5.14: File `src/demo/main.lua`, lines 12-20.

```
1  function love.update (dt)
2    -- *snip* (update avatar position)
3    ev:pushCommand("pos", avatar.x/W, avatar.y/H)
4    ev:setAudioSource(5*(avatar.x - W/2)/W, 5*(avatar.y - H/2)/H, 0)
5    vorpal.tick(dt)
6  end
```

Listing 5.15: File `src/demo/main.lua`, lines 35-46.

updates the user's avatar position according to what keyboard keys he or she is pressing. Then, at the end of the frame, it sends the "`pos`" Command to the Soundtrack Event passing the `x` and `y` values of the avatar's position as parameters; updates the Soundtrack Event position in the virtual space of the Audio Engine; then finally ticks the Engine using the `dt` parameter provided by the Game Loop of the *LÖVE* framework. Although *Lua* has its own garbage collecting mechanism, we still need to make sure that instances of `vorpal::SoundtrackEvent` are freed before the Audio Engine is finished as a whole. That is why we added the code from Listing 5.16 implementing the `love.quit` callback, which is called when a *LÖVE* game is closed normally.

```
1  function love.quit ()
2    ev = nil
3    vorpal.finish()
4  end
```

Listing 5.16: File `src/demo/main.lua`, lines 48-51.

The Patch comprising the soundtrack of this example if shown in Figure 5.14. It is slightly more complex than the one from the first example. Here we preferred to use the Oscillator from the Soundtrack Creation Kit to synthesize triangular (left) and rectangular (right) waves. The triangular wave has its frequency controlled by the `x` parameter of the "`pos`" Command, increasing in pitch the more the avatar goes right. The rectangular wave is controlled by the `y` parameter, increasing in pitch as the avatar goes up. Both waves are multiplied by each other before being sent to the Output Bus. Since the rectangular wave uses lower frequencies, the resulting sound produces perceptible vibration effects.

We started this Chapter by presenting a prototype game we used to validate the proposal that *Pure Data* works as a real-time soundtrack representation. Based on that, we developed the *VORPAL* middleware, which we detailed throughtout Sections 5.2 and 5.3. Then we demonstrated how both kinds of user (programmers and sound designers) can make use of the system. In the next Chapter, we will discuss the results observed from the actual use of the middleware, which we complement by applying the validation criteria defined in 1.2.3.

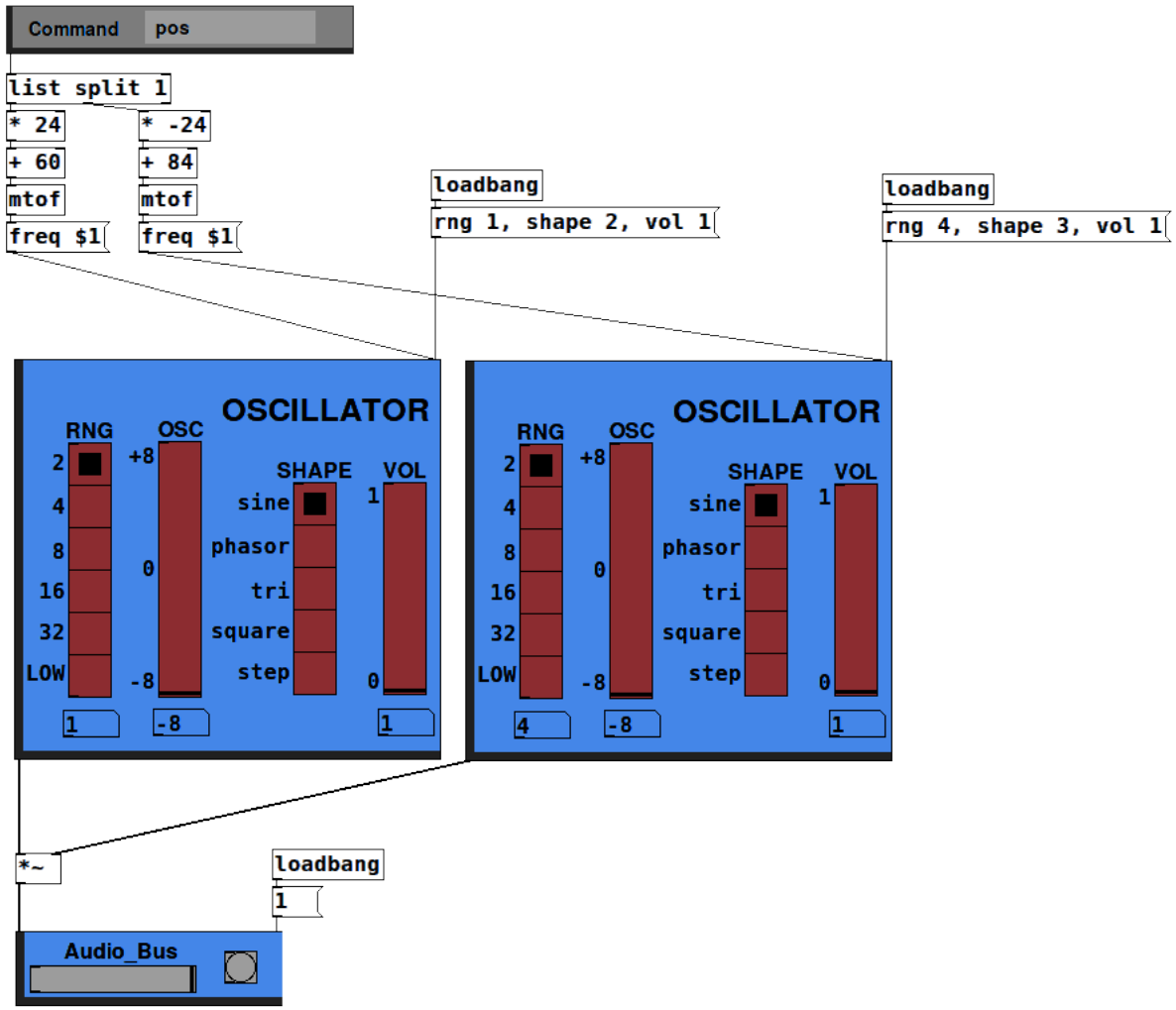


Figure 5.14: The "demo" Patch from the second example. It plays a signal resulting from the multiplication of two synthesized waves with frequencies derived from the user's avatar position in the game.

Chapter 6

Results

This chapter discusses the main results of our research and how they validate our middleware solution following the criteria from Section 1.2.3. Each of the following sections directly corresponds to one of the criteria, except for the last one. Section 6.1 fulfills our first validation, Basic Feature Support, by presenting an actual game we developed that uses our middleware to produce its soundtrack. Section 6.2 meets the second validation, Advanced Feature Support, which involved gathering feature requests from the literature and professional sound designers to guide the development of *VORPAL*, then verify whether it really supports each of them. In Section 6.3, we address the third validation, Usability, which was included in this thesis with the intention of identifying how our tool could put all of its features to a better use by providing better interfaces and workflows for its users, programmers and sound designers alike. Then, Section 6.4 makes a deeper analysis of the *VORPAL* middleware by inspecting the consequences of our decision to use *Pure Data* as both a digital content creation interface and as a real-time soundtrack representation format.

6.1 Sound Wanderer

Sound Wanderer is a proof of concept game we developed in a partnership with Dino Vicente De Lucca (refer to Section 4.2.3), who contributed as both game designer and sound designer to the application. The game was not developed after the conclusion of the middleware, but rather side by side with it. It served as our guiding line much as it served to showcase the features of the system. Thus, developing this game was an essential part of our research, specially due to our Basic Feature Support validation, which specified that our middleware be tested using an actual game, and not only code examples (see Section 5.4.5). It was never intended to become a commercial product, although we did aim for a pleasant and complete game experience, albeit not exactly a challenging one. To avoid the nontrivial task of writing a game from scratch, *Sound Wanderer* was developed using *Godot* and the *VORPAL* plug-in discussed in Section 5.4.4. It was only tested on *Linux* platforms. An unedited gameplay video of the released version discussed here is available at https://www.youtube.com/watch?v=oCw719VmIn8.

Sound Wanderer is divided in two parts. During the first of them, the game uses a first-person view to allow the player to explore a dark building where he or she must solve three puzzles to

Figure 6.1: One of the puzzle rooms in *Sound Wanderer*. This particular room has a *Tetris* statue which plays "*Theme A*" from that game.

open the exit. There is no explicit plot, but an extra-diegetic voice guides the player's actions. Starting from a small corridor that leads to a larger hall, the player finds three rooms and one closed gate, the exit. Each room contains one of the puzzles, which involves a real-time music composition we explain ahead. After solving the three puzzles and passing through the exit gate, the player "falls" to the second part of the game, which changes the perspective into a 2D top-view of the player's avatar – a triangle – inside a psychedelic amorphous dimension. There, he or she must find yet another exit, but there is none. Most of this part is about experiencing the soundtrack, which responds to the player's movement and exploration. After a certain amount of time passes, a black-hole appears to suck in the player's avatar and end the game with an inexorable "Game Over".

In the first part of *Sound Wanderer*, each puzzle room contains a statue resembling a famous game and playing a certain music theme. Upon closing in on the statue, the music stops, and the player is prompted to try out three different music themes and point out which one belongs to the statue. The player may try as many times as needed, and when the right theme is selected, he or she is teleported back to the central hall. The right theme should be the same as the one that was playing as he or she entered the room, and the appearance of the statue should be a strong hint, specially for veteran players. However, the melody of each theme is stochastic, that is, it is randomly determined as it plays, which might make it slightly harder to know which song belongs to each statue. The stochastic behavior of these themes follows a Markov chain derived from one of the original themes from the corresponding game title depicted by the statue. The process used to create this Markov chains essentially analyzes the frequency with which each melody step occurs after the corresponding previous step, thus capturing melodic patterns of the original score and playing a random theme which often sounds like the original but is seldom

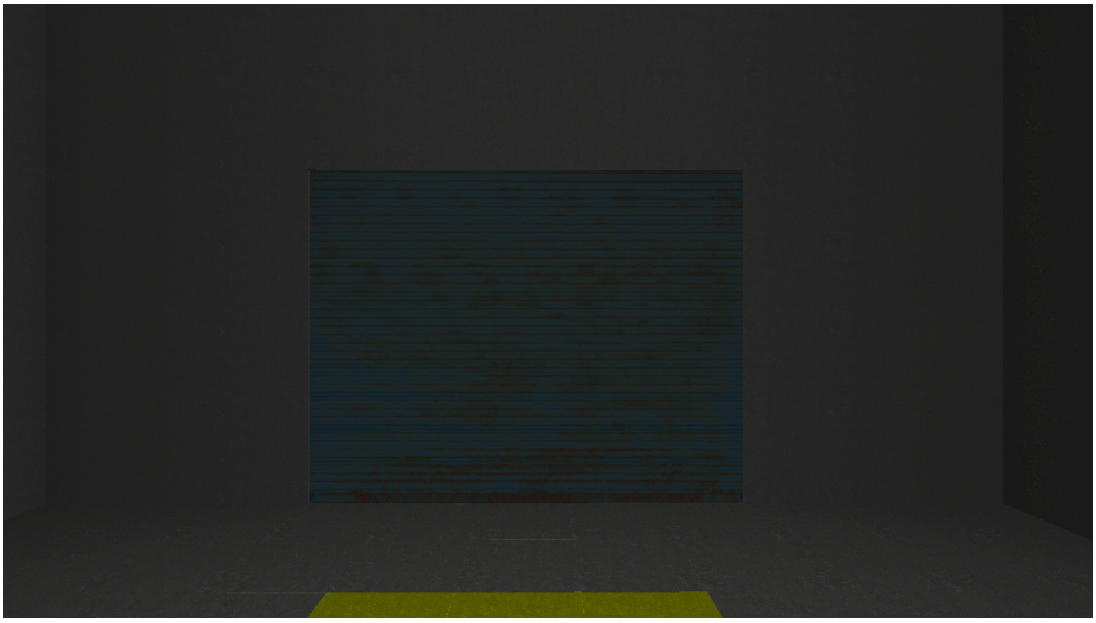


Figure 6.2: The exit gate from the puzzle rooms sequence. It constantly emits a pulsing sound and opens after all three puzzles are solved.

really the same. We do not describe the exact Markov chain extraction algorithm here since it is outside the scope of this thesis. The important part is that each chain is fed to a *Pure Data* Patch which plays the stochastic melody at run-time, that is, each and every note is decided in real-time. This would be simply impractical to do with sample-based audio, making this part of *Sound Wanderer* one of the strongest examples of the contributions provided by the *VORPAL* middleware. Figure 6.1 shows one of the puzzle rooms in *Sound Wanderer*. The games and songs used in the statues are:

- *Super Mario Bros.* (*Nintendo*, 1985) – "*Overworld Theme*" by Koji Kondo;
- *Tetris* (*GameBoy* version, *Nintendo*, 1989) – "*Type A*" arrangement by Hirokasu Tanaka of the Russian folk song, "*Korobeiniki*"; and
- *Undertale* (Toby Fox, 2015) – "*Spiders Dance*" by Toby Fox.

The puzzles are not the only soundtrack elements of the first part though. The player's footsteps also have sound effects, which vary according to whether the avatar has just started walking, has been walking for more than one step, has just stopped walking or is turning around. All puzzle rooms have a door that is closed when the player enters, causing a loud and echoing bang sound throughout the room. The exit gate is also continuously emitting a pulsing sound to enhance its mystery. There is the guiding voice, played from recorded samples of voice actor Yuri Koster. Lastly, when the player leaves the first part by falling through the exit gate, a Sheppard tone [She64] is played to add to the illusion of a free fall. Figure 6.2 depicts the exit gate of the first part when it is still closed. All the sound effects are spatialized with binaural effects.

After falling through the first exit, the player arrives at the second part of *Sound Wanderer*, which we call the "Damnation Room". It is intended as a merely exploratory environment with

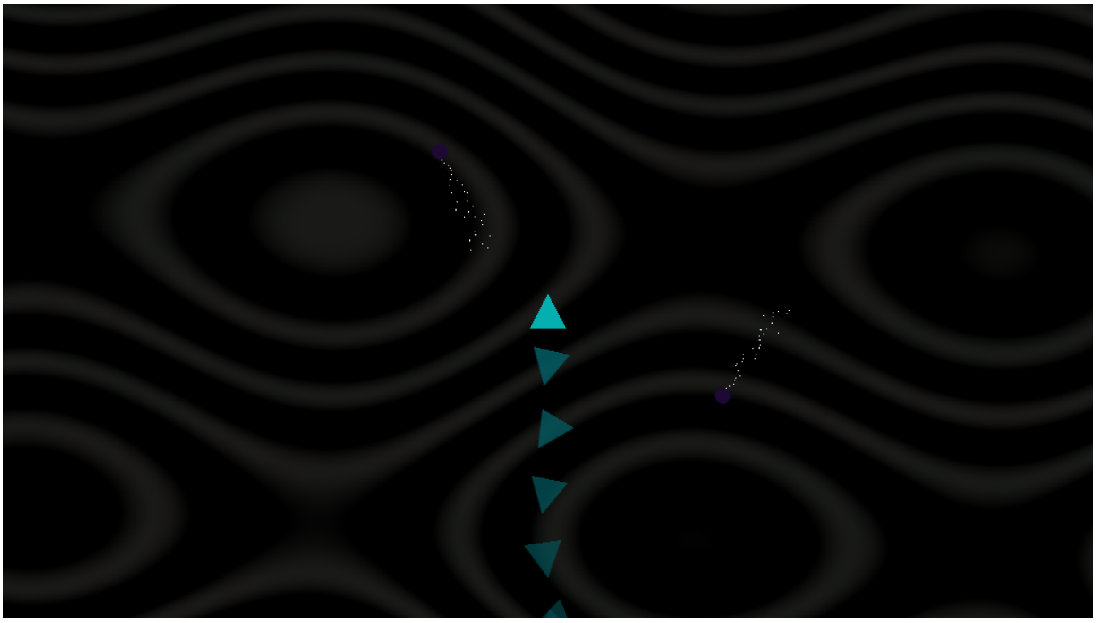


Figure 6.3: During the first part, an electronic music plays according to whether the player is moving or not. The little moving objects are just for decoration.

no clear purpose but that showcases a number of real-time effects in its soundtrack. It is also further divided into two parts. In the first, there is a predominantly gray background, and some electronic music grows in the soundtrack. There is also an introduction to the "Damnation Room" by the guiding voice, now distorted to a much lower pitch. The main real-time effect here is in the electronic music. It is composed of a bass, a hi-hat, and two melody voices. Except for one of the melody voices, all other parts of the music come from timbres synthesized in *Pure Data* using the Synthesizer and Sequencer features of the *VORPAL* Soundtrack Creation Kit. The hi-hat is synthesized from a noise generator, and has its pitch very subtly modulated by a sinusoidal wave. The bass and the sequenced melody vary which notes are to be played according to whether the player is moving or not. This is done by activating the slots of their corresponding Sequencers at the right moments. The other melody is a looped sample theme composed directly by De Lucca. Figure 6.3 shows how this part of the game looks like.

The second part of the "Damnation Room" has a colorful background pattern accompanied by a drum sequence. This sequence is entirely made of samples owned by De Lucca. Here we intended to show that *VORPAL* can also handle sample-based audio just as its commercial alternatives. The drum sequence has five variations, one for each cardinal direction and one for when the player is standing still. We use the player movement direction to determine which of the first four sequences to play, and synchronize the sample so that they always play seamlessly from one to the other. Aside from the drums, this part of the "Damnation Room" features random polygonal obstacles appearing from time to time near the player. Each of these obstacles talks using the guiding voice (still acted by Yuri Koster) when the player comes close to them. The voices use the 3D spatialization effect, so that when the player just passes by the side of one of the obstacles, he or she can hear the voice walking past them. The actual speech each obstacle

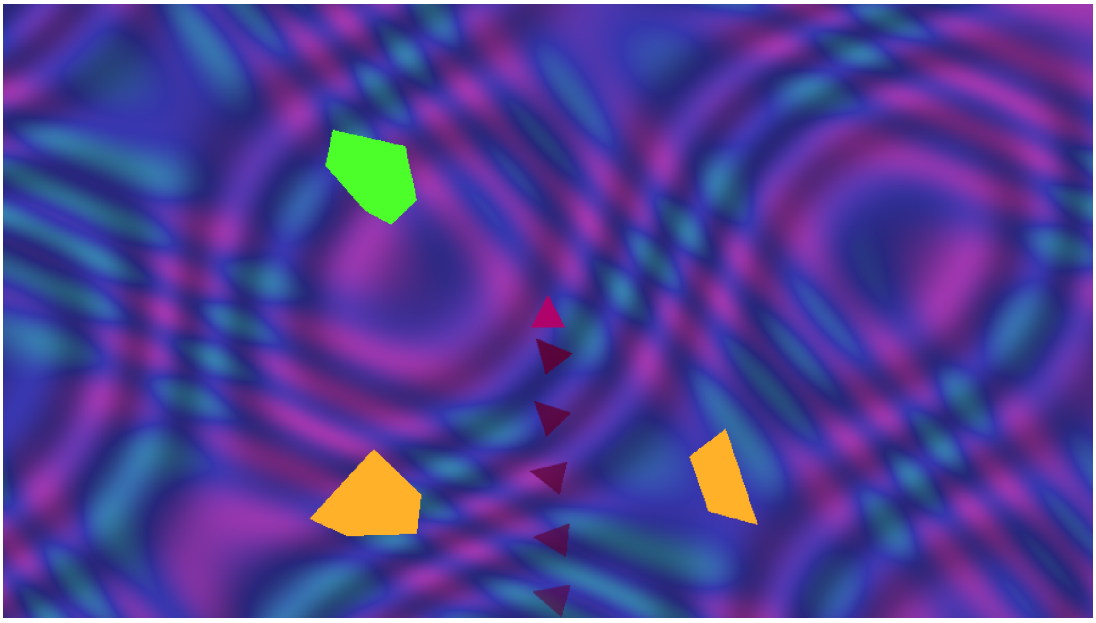


Figure 6.4: In the second part, the drum sequence varies according to the cardinal direction the player is moving to, while the obstacles talk nonsense when approached.

says is randomly selected from a pool of hand picked phrases from the *Oblique Strategies* deck[1], to which a random effect is applied – such as reverberation, ring modulation, etc. Figure 6.4 shows how this part of the game looks.

An initial release of the game was submitted as a demonstration to the 12th International Symposium on Computer Music Multidisciplinary Research and accepted [MVK16]. There, we let conference participants play our game and provide feedback. We also had a short paper accepted in the same conference [MK16], which served to further explain the technology behind *Sound Wanderer* to the players. Figures 6.5a and 6.5b show pictures of the demonstration. After this release, we continued our partnership with De Lucca mainly to improve the "Damnation Room" based on the feedback we obtained at the conference.

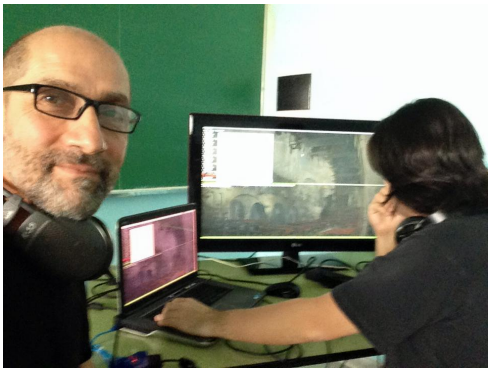
6.2 Advanced Features

In Section 1.2.3, we defined the second validation criterion of our research as a feature list derived from both the literature and interviews with professional sound designers. The list was formalized in Section 4.2.4 under the name of System Requirements. Here we verify, one by one, whether the *VORPAL* middleware satisfies these Requirements.

0. Compatibility with Game Applications

This System Requirement is met by the proof of concept game *Sound Wanderer* described in Section 6.1, since it not only demonstrates that the middleware is compatible with an actual

[1] https://en.wikipedia.org/wiki/Oblique_Strategies



(a) Some developers of *Sound Wanderer*. De Lucca (left), game and sound designer, and Wilson Kazuo Mizutani (right), game designer and programmer.

(b) Some conference participants playing *Sound Wanderer*. The game was designed to be played using *PlayStation*-compatible gamepads.

Figure 6.5: Pictures taken by De Lucca during the 12th International Symposium on Computer Music Multidisciplinary Research held in São Paulo, June 2016.

game application, but that it can be used together with industry-tested game engines (*Godot* in our case).

1. Independence from Programmers

Since the Soundtrack Creation Kit is but a set of *Pure Data* Abstractions and Externals, and the sound designer is free to do anything he or she is capable of in the visual programming language of *Pure Data*, this means that they have control over all the aspects of the game soundtrack. The Markov chain implementations in the puzzle rooms of *Sound Wanderer* are a good example of how much liberty and control the sound designer has when producing the soundtrack. The only part they cannot control are *when* Commands (along with their parameters) are sent and *where* the Soundtrack Events are positioned inside the virtual sound space of the Audio Engine, but this is practically inevitable. Only the game application knows about the game state (which carries the timings and the game entity whereabouts inside the virtual world simulation). Even if the Soundtrack Creation Kit had direct access to the game state, it would still require that the host application provided it in some way in the first place, so the dependency would still be there on way or the other.

2. Real-Time Controlled Music Sequencing

The Sequencer feature of the Soundtrack Creation Kit implements support for this System Requirement by offering a relatively simple mechanism for sequencing soundtrack behavior. We demonstrated its capabilities in the background music of the first part of the "Damnation Room" in *Sound Wanderer*, when the electronic music responds to whether the player is moving by adding or removing notes from one of the melody sequences. Additionally, the Sequencer is not the only way the Soundtrack Creation Kit can achieve real-time controlled sequencing. The Markov chains in the puzzle rooms have their own sequencing mechanism, which manipulates the melody in real-time note by note. This means that even if the Sequencer is too simple for

advanced sequencing implementations, it can still be done by programming it directly in *Pure Data*.

3. Real-Time Controlled Synthesis

This System Requirement is satisfied by the Synthesizer feature of the Soundtrack Creation Kit, which allows at least as many timbres as the *Minimoog* synthesizer, and has been successfully demonstrated in the "Damnation Room" of *Sound Wanderer* too. The second example from Section 5.4.5 also shows how it is possible to change a timbre at its wave shape level in a real-time interaction with the game application. Besides, the possibility still remains of synthesizing timbres using *Pure Data* directly, like in the first example from Section 5.4.5.

4. Music Transitions

There is no particular feature in the Soundtrack Creation Kit that directly meets this System Requirement, but it is nevertheless supported and *Sound Wanderer* clearly demonstrates that. During the second part of the "Damnation Room", the game soundtrack transitions between five different looped drum samples. All transitions there are properly synchronized, displaying no perceptible artifacts when the middleware changes from one sample to the next. The Sample feature from the Soundtrack Creation Kit helps in the case of sample-based transitions, but in the more general case transitions are feasible through a direct implementation thanks the procedural audio approach of the middleware.

5. Variable Mix

The Output Bus Abstraction in the Soundtrack Creation Kit, which is the main audio output of each Soundtrack Event, comes with a volume control which allows for real-time variable mix between Soundtrack Event instances. For a variable mix within the same Patch, a direct *Pure Data* implementation is required, which is demonstrated by the latest version of our *Mari0* extension prototype, with its soundtrack being composed of a variable mix between the drums, the bass and the melody. Thus, this System Requirement is also met.

6. Variable Frequency Filter

The `[vorpal_filter]` from the Synthesizer feature in the Soundtrack Creation Kit supports this System Requirement by emulating the filter of the *Minimoog* synthesizer. This is demonstrated by the hi-hat in the electronic music of the first part of the "Damnation Room" in *Sound Wanderer*, since it essentially plays a noise signal band-pass filtered at a frequency modulated by a sinusoidal wave. As with other Requirements, it could also have been done directly in *Pure Data*.

7. 3D Audio Spatialization

By enabling the 3D audio spatialization feature of the *OpenAL* programming library in the Audio Engine API – more specifically in `vorpal::SoundtrackEvent::setAudioSource`

– the *VORPAL* middleware successfully meets this System Requirement. This is clearly shown by the second example of Section 5.4.5, but is also present throughout most of the *Sound Wanderer* soundtrack, specially with the talking obstacles in the second part of the "Damnation Room". However, it is thanks to this that all the Soundtrack Events can only output mono sounds, as discussed in Section 4.2.3.

8. Playback Randomization

We have demonstrated support for this System Requirement both at the Markov chains from the puzzle rooms and the talking obstacles in the "Damnation Room" of *Sound Wanderer*. In the Markov chains, the game manages to randomize music themes on the granularity level of its score, choosing the next note to be played stochastically. In the "Damnation Room", a recorded speech is chosen from a pool of samples to be played when the player comes near the obstacles. There is also a randomization in the effect applied to the sample chosen.

9. Polyphony Control

Polyphony control could be implemented at two different levels inside the *VORPAL* middleware. The first is to allow Patches developed with the Soundtrack Creation Kit to control polyphony inside themselves, and this is indeed supported – it is only a matter of using multiple Synthesizers and Samples and regulating how many can be playing at the same time. The second level of polyphony control would be between Soundtrack Events, and this is not supported, because it has a slight conflict with the next System Requirement. While we do consider that this Requirement is met in the current implementation, we intend to improve its support to include polyphony control between Soundtrack Events in future versions of the middleware.

10. Resource Usage Optimization

We support this feature by using `C++` smart pointers that self-manage their own allocation process through reference-counted garbage collection. This essentially means that as long as the game code holds a reference to a Soundtrack Event instance, it will be kept in memory, playing its sounds. Unused Events are interrupted and removed from memory, thus optimizing this resource usage. However, this also means that a Soundtrack Event that is referenced in game code cannot be prematurely removed by, say, a polyphony control mechanism. To do that, the Audio Engine would either have to invalidate an Event instance already referenced by game code, or allow unreferenced instances to persist in memory until they expire or their polyphony threshold demands so.

6.3 Usage Feedback

In this section, we discuss the third and last validation of the *VORPAL* middleware. It is worth remembering that the motivation behind this validation is to identify how much the features guaranteed by the System Requirements are actually accessible to the users of our system. Properly addressing this issue involves other research areas beyond computer music and game

programming, and as such we take this validation more as a future guide for how to actually make our contribution usable in real world projects, not only prototypes and proof of concept games. In that sense, there are two very different types of users to our technology: programmers and sound designers, both from a digital games development context.

The only actual programmer who used the technology was the author of this thesis when developing the *Mari0* prototype and *Sound Wanderer*. So we strove to have a strictly critical analysis of how much the Audio Engine – which is the part of the middleware programmers interact with – was really easy to understand and work with. The fact that its API is essentially composed of two classes with few methods makes it undoubtedly quick to grasp. Yet, there are at least two details that might not seem obvious at a first glance. The first is that, since Soundtrack Events are not sequential sound pieces, there is no "play" or "stop" operation. As soon as a Soundtrack Event is instanced, it is running and potentially playing something. Any kind of playback control has to be implemented as Commands the sound designer uses to control the Event. The second issue is with the time parameter of `vorpal::Engine:tick`. It should contain the value of how many seconds have passed since the last call to that method, and is supposed to be obtained from the synchronization mechanisms of the Game Loop of the host application. When we integrate the middleware with other game engines, this data is promptly available. However, in the first example of Section 5.4.5, when we emulate a Game Loop, we found that actually measuring the time difference between frames might not be that simple, and even the slightest deviations immediately incur in audio artifacts in the Audio Engine. Thus, it could be improved by either measuring this time by itself or by being more robust to errors in the parameter (by using a mean, for instance). Aside from the API, one other issue with the Audio Engine is that it requires the developer to compile `libpd` to avoid the extra dependency, and while we had no problems with that on *Linux* platforms, our few attempts at compiling in *Windows* failed due to errors inside the *Pure Data* source code[2].

For the sound designer perspective, we discuss here how both De Lucca and us worked on the soundtrack of *Sound Wanderer*. The most noteworthy point in reviewing that process is that practically all real-time effects De Lucca proposed for the game were successfully implemented save for time constraints, which sustains the claims on the power of procedural audio. There are, however, two caveats to this statement. The first is that, being a sound designer not used to working with games, De Lucca was still experimenting with the possibilities and might not have reached for more complex effects, thus meaning that we likely could have pushed the middleware further (and eventually met one or more design restrictions). The second caveat is that, since De Lucca had no previous experience with *Pure Data* (only *Max/MSP*), all the effects he wanted to produce had to be implemented together with us – he was, after all, learning two new languages at the same time (*Pure Data* and the Soundtrack Creation Kit) while designing a game for the first time. This comes as no surprise given what we discussed back in Section 2.1 about Scott's work on the challenges of forcing ever new technologies on game musicians for the sake of better soundtracks or more efficient workflows [Sco14]. In fact, given that *Pure Data* is a full-fledged programming language, expecting a sound designer to feel comfortable with it means demanding

[2] There was likely some missing (or exceeding) compilation directives, which we decided not to spend any more time looking into given our research deadlines.

that they learn how to program at least to a certain degree. In other words, we have essentially inverted one of the original problems with game audio: where the programmers had to understand sound and music to implement what the sound designers could not, now it is the latter who has to comprehend programming concepts to produce the soundtrack. On the other hand, providing a general purpose procedural audio tool through a completely programming-free environment is borderline utopic. There are two possible extremes here with a full spectrum of combinations in-between: to specialize the interface and embrace the narrowing of aesthetics, or close in on Turing-completeness to widen its capabilities. The *VORPAL* middleware leans more towards the latter at the cost of a steeper learning curve, but with the benefits of using a language arguably widespread in the computer music community.

6.4 Middleware Limitations

Pure Data was a fundamental design decision in the architecture of *VORPAL*. It not only supports real-time procedural audio, but it sped up our development process by relieving us of implementing an interpreter and an editor to the real-time soundtrack file format used by the middleware. Besides, being a tool reasonably widespread in the computer music community, we found no lack of material and references in the literature or the Internet. Most important of all, it ultimately works. *Pure Data* successfully achieves real-time soundtracks in games. That being said, there are severe limitations to both its implementation and usability.

On the implementation side, as we point out a number of times throughout Chapter 5, the main issue with *Pure Data* is that all of its state is coded into the global scope of the application. This forced our own implementation to behave as if it was on a global scope, even if we managed to mostly hide that from the user thanks to design patterns such as the Singleton pattern. The problem is that the global state will also propagate to the user's application one way or another, imposing a likely undesired restriction on their code. Globally scoped state complicates multi-threaded programming, and prevents proper encapsulation of concerns since every part of the code base can now access shared data - making it harder to trace the origin of errors when the software escalates. Another problem with the *Pure Data* implementation (and `libpd`) is that it can only load Patches from file paths, meaning its code forces the programmer to let *Pure Data* open and handle files by itself. While this may seem like a burden off the programmer's work at first, it actually complicates things when the file system of the target platform is not compatible to what *Pure Data* expects, for instance on a mobile system. In this case, it would be better if the host application opened the files then fed *Pure Data* with its raw data.

In terms of usability, the *Pure Data* application and its Patch editor have a number of minor quirks which make the development of Patches cumbersome at times. For instance, linking Connections to Objects can only be done if the user puts the end point of the Connection almost exactly on top of the Inlet of the Object. This requires precise control of the mouse and can slow down the process of writing complex Patches with a high number of Connections. However, these relatively small peculiarities are still overshadowed by the fact that Abstractions cannot have their state saved along the parent Patch. The *Pure Data* format only stores the creation arguments of any Object created in a Patch. If that Object is a slider, the slider value is also

a creation argument and its state is properly saved. But when an Object is an Abstraction, the state of Objects *inside* the Abstraction are not part of its creation arguments, and are thus not saved in the file of the parent Patch. There are a few contrived ways of circumventing this[3], but it pushes even more complex operations to the sound designer. This makes the use of our Synthesizer feature, for instance, very uncomfortable since the sound designer has to provide its state through Message Objects in the parent Patch instead of just tweaking the sliders and having they stay that way when the Patch is reopened later (see examples back in Section 5.4.5).

[3] http://forum.pdpatchrepo.info/topic/8803/state-saving-abstractions

Chapter 7

Conclusion

We started this thesis by exposing how making soundtracks for games, specially real-time controlled ones, is currently an endeavor entirely dependent on the technologies at the disposal of the developers. We then moved on to propose the *VORPAL* middleware as a free software solution based on procedural audio. This decision was based in great part on the works of Farnell, who advocates the use of procedural audio over sample-based audio, and Collins, which discusses the history of sound in games and what are the possibilities of real-time soundtracks. After explaining all the concepts and tools needed to understand and implement a game audio middleware, we formulate our methodology and the final software architecture of our proposed middleware solution. The actual implementation of this solution was discussed at length, exposing the two main components of the system: the Audio Engine and the Soundtrack Creation Kit.

This chapter, which concludes this thesis, addresses our conclusions regarding our research and its contributions in Section 7.1, including the technological challenges involved and what are our expectations on real-time soundtracks in games in general. Lastly, Section 7.2 addresses future work that would bring our research ever closer to better solutions to the process of creating and delivering entertaining real-time soundtracks in digital games.

7.1 Final Considerations

Games with real-time soundtracks have existed since the birth of the game industry itself. Arcade games had no choice but to deploy procedural audio for both their sound effects and music. Not long after, by the end of the last century, *iMuse* had already developed features for dynamic music scores, which was then used for game titles that earned their place in game history. Then, for the last two decades, computers and consoles started having more and more process power and memory capacity, game budgets skyrocketed and development teams grew more diverse and experienced. Now we have soundtracks with full orchestras and a handful of bleeding edge audio middleware systems. Yet, this research has argued that this is not enough.

First, from a developer's perspective, we explained how the lack of a dedicated game audio middleware burdens both programmers and sound designers, since programmers have to do part of the sound designers' work, while the sound designers depend on both the programmers and the underlying technology used to shape the soundtrack. This is solved by the aforementioned

middleware tools, possibly at a financial cost. Second, there are practically no games exploring the possibilities of procedural audio, which means there are many potentially interesting experiences being entirely neglected by the whole game development community, despite all the benefits defended by Farnell [Far07]. With this research, we have gathered a number of approaches to real-time soundtracks and studied the industry needs to develop our own game audio middleware with the intention of at least opening the way for this still barely explored design space.

Our proposal is not free of its own shortcomings. While aesthetic limitations on synthesized sounds are being constantly reduced by continuous technical advances, real-time procedural composition of music, for instance, has still much to be developed. As long as sequential music – in the sense that its contents follow a predetermined and unchangeable sequence of events – provide better results, relying on sample-based audio remains the best option for this soundtrack role (preferably with the support of a dedicated middleware), both in terms of the resulting sound aesthetic and of the technical and technological skills required of the sound designers involved. On the other hand, by bringing a viable and accessible alternative to the table, we promote further and more effective investment and research exactly on approaches of yet experimental nature such as real-time procedural music in games. The problem is that, if *VORPAL* remains a strictly feature-oriented tool, and not exactly easy to use, this objective will stay a far fetched reality. As we discuss in Section 7.2, there is a number of ways in which we and other interested parties can take real-time soundtracks in games a step closer to being a reliable design choice. Until then, the industry standard will likely remain on the sample-based side of sonic composition, because anything besides a licensed technology such as *Wwise* or *FMOD Studio* involves too much technical and artistic risk if the game is not particularly intended to sell on an innovative soundtrack experience.

One possibility our middleware opens is to provide a framework for adaptive music structures such as the ones proposed by Livingstone *et al.* and Eladhari *et al.* (discussed in Section 2.2). Both the CMERS and the Humor Matrix these authors propose can be trivially implemented with the Command Abstraction of the Soundtrack Creation Kit. For instance, let us say we want to implement the Humor Matrix, which is a five-by-five matrix mapping pairs of mood values (inner and outer moods of game characters) to twenty-five possible samples containing different versions of a music theme. It is but a matter of defining two Commands – `inner-mood` and `outer-mood` – which must be triggered from the game code in the appropriate narrative conditions (or using the Mind Module proposed by Eladhari *et al.*), then mapped in a *Pure Data* Patch to the adequate samples. This could be as simple as transforming the pairs of received values in a number ranging from 1 to 25, then using the `[route]` Object to forward the control to play (or just increase the volume) of the right sample.

Soundtracks in digital games is one of those intrinsically multidisciplinary research contexts where there is no such thing as knowing too much of other areas involved. Which is why we strove to work alongside professionals and researchers which could not only complement our experience in computer science and game design, but also properly direct us at what were the core issues, the real problems that needed solving in game soundtracks. All of these contributions came out of these people's good will and free time, and if not for them we would be left to work on and develop a tool that would likely serve no actual purpose. Although the *VORPAL* middleware would have

certainly benefited from having ourselves know more about and have more experience with sound and music. That being said, the fact that soundtracks in games (and movies for that matter) are often approached as an afterthought – as confirmed by both the literature [Mat14] and the interviewed sound designers (see Section 4.2.3) – remained a cornerstone in the motivation of this research, leading us to where we are now.

7.2 Future Work

In the last section of this thesis, as we consider the future of real-time soundtracks in digital games and the roles of game audio middleware systems in that context, we present a list of punctual improvements to the *VORPAL* middleware and future research topics of interest. Sections 7.2.1 and 7.2.2 discuss what could be improved in the Audio Engine and the Soundtrack Creation Kit, respectively. Then, in Section 7.2.3, we consider the implications of no longer using *Pure Data* in the *VORPAL* middleware and if it would be worth the effort. Section 7.2.4 addresses our future plans for supporting mobile and web platforms. Lastly, in Section 7.2.5, we discuss what research topics are left to be tackled as possible continuations to this thesis.

7.2.1 Audio Engine Improvements

The following is a list of relevant improvements that could be added to the *VORPAL* Audio Engine.

Dynamic Buffer Allocation

As of the latest implementation of the Audio Engine, the number of created *OpenAL* buffers is fixed at initialization time. This is a waste of resources when the number of Soundtrack Events sending their audio to the sound card is not that high. One simple way to improve this is to use a dynamic amortized allocation where the buffers are created on demand, then freed when audio transfers are reduced.

Patch-Controlled 3D Spatialization

In our 3D spatialization feature, we assumed that the only motivation a sound designer would have to use binaural effects would be to simulate the positions of game entities as actual sound sources. Thus, we designed our API to only allow control of these positions from the source code side of the game. This becomes a restriction when the sound designer wants to use a binaural effect for any different purpose. As such, another improvement in the Audio Engine would be to implement a protocol for capturing Messages from loaded Patches specifying the 3D virtual position of the corresponding Soundtrack Event.

Robust Time Handling

As discussed in Section 6.3, slight oscillations in the time parameter required by the method `vorpal::Engine::tick` are prone to cause sound artifacts. This can be avoided by using a more robust control of how time passes inside the Engine, like using the mean of the last $N > 1$

time values instead. Another very different approach would be to decouple the Audio Engine from the Game Loop of the host application by running it in a separate thread, managing its own synchronization mechanisms. This requires a much more careful implementation, but would make the programming usability of the middleware much better, as long as the underlying platform supports multithreading.

7.2.2 Soundtrack Creation Kit Improvements

The following is a list of relevant improvements that could be added to the *VORPAL* Soundtrack Creation Kit.

Documentation and Examples

In *Pure Data*, most Objects have a documentation Patch attached to them, which can be viewed by right-clicking them and selecting the "Help" option. All native Objects from *Pure Data* come with at least one such Patch containing an example. This is specially important for the user to know what are the roles of the Inlets and Outlets of Objects. Thus, it would be a significant improvement to the usability of the Soundtrack Creation Kit if proper documentation Patches for every Abstraction were written and distributed along the rest of the middleware package.

Tool Bar Patch

One of the first obstacles to using *Pure Data* for non-programmers is that Objects are created by typing their names. There is no tool bar with a pallet of all available Objects, since it would be impossible to fit them all anyway. However, inside the scope of the Soundtrack Creation Kit, the actual number of Objects available is quite manageable. As such, one possible improvement to the Soundtrack Creation Kit would be to write a "tool bar" Patch from where the sound designer could pick Object types to place in his or her own Patch. This is possible thanks to how the *Pure Data* format works, which allows for *dynamic patching*, that is, using Messages to create and connect Objects in a Patch procedurally (the *Pure Data* equivalent of metaprogramming).

Bank of Procedural Sound Effects

Following the implementations presented by Farnell [Far10], for instance, we could include dozens of built-in sound effect Abstractions in the Soundtrack Creation Kit. This would save the sound designer's time when creating many soundtrack elements of a game, even if only as temporary placeholder sound until better quality samples as available. Unless, of course, one implemented state-of-the-art algorithms for physics-based sound effects such as the ones described in Section 2.3.

7.2.3 Beyond *Pure Data*

As discussed in Section 6.4, at the same time that *Pure Data* was what made the *VORPAL* middleware possible, it also has its own limitations that stop the system from being a more attractive alternative to commercial tools. Here, we discuss what would be necessary to emancipate

VORPAL from *Pure Data*, and what would be the benefits and setbacks of doing so. One way or another, migrating to an independent soundtrack format would most certainly be the next important step in the development of *VORPAL*.

The simplest solution to this would be to base the new format on *Pure Data*, but taking care to handle the very same problems we are trying to avoid. For instance, just by providing an alternate implementation of the *Pure Data* virtual machine that does not leave its state in global scope would already be a great improvement. The next step after that would be to extend the Patch file format to allow storing the state of Objects inside Abstractions. This would likely require major changes to the format as a whole, but just by making these two modifications most of the internal problems of the Soundtrack Creation Kit would be gone. The biggest improvement and challenge, however, would be to re-implement the *Pure Data* Patch editor interface. Designing a more fluid and intuitive interface would come a long way in terms of usability, but writing code for graphical user interfaces is always a large and error-prone enterprise.

Even if we did manage to develop our own format and the corresponding editor and interpreter, we would still be left with a problem. One of the greatest advantages of using *Pure Data* is that it has a history and a community. This means two things. The first is that many contributions accumulated over time, and now even the core set of native Objects in *Pure Data* is just too much to replicate without a dedicated development team. That is, our new Soundtrack Creation Kit would not have all the features it has now, making it much less useful. The other problem is that *Pure Data*, when compared to an arbitrary new language, is a rather well established standard in the computer music community when it comes to free software visual programming languages. Making a new one demands that users learn it, where the current implementation of *VORPAL* starts from day zero with a relatively large pool of potential users already comfortable with its language. All this means that actually implementing our own format and editor is a task that should be tackled with care, since it would not only be an investment of our time, but also the users' time. If it is not worth it, we are probably better staying with *Pure Data*.

7.2.4 Other Platforms

The current stable version of the *VORPAL* middleware supports only desktop *Linux*, but with enough time to fine-tune the *CMake* build scripts, support for *Windows* and *Mac OS* is also possible. Mobile might be a more challenging target, due to the restrictions on file system access and the way *Pure Data* handles it (see Section 6.4), but it would definitely be the next platform compatibility priority, since procedural audio allows mobile games to carry much more sonic content at a memory cost orders of magnitude lower (see also Section 4.2.1). The most challenging, however, is the Web platform, since everything there has to ultimately run in *JavaScript*. This would mean using a Web implementation of *Pure Data*[1], as well as porting the Audio Engine and adapting it to work with the *Web Audio* API[2].

[1] https://github.com/sebpiq/WebPd
[2] https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

7.2.5 Research Perspectives

The multidisciplinary aspects of this research area allow our contribution to open paths to a number of novel research topics that can further improve the soundtrack of digital games, specially now that an accessible technology is available to explore the design space of real-time soundtracks. In game design, if one considers the possibilities of procedural audio, we could analyze how it supports player interaction with the interface and controls, that is, how feedback sound effects can be automated, and how that improves the user experience. There are also many possible research paths in the general direction of using real-time soundtracks to construct unique narratives, where every little action the player does changes something in the music or ambience, from a single note to whole transitions or improvised arrangements. In a more mechanics and semiotic oriented approach, one could study how real-time sound effects and voices balance action games with intensive output feedback[3], identifying a sonic language that can keep the players informed of the things even the best HUD interface cannot.

In computer music, one of the most promising topics is the simulation of the physics of sound, such as the works seen in Section 2.3. Of course, most real-time synthesis technique studies are relevant to achieving interesting timbres efficiently too, specially if they can balance memory and processing time with sound quality and fidelity, like with dynamic LOAD. Researching for generic and flexible sound effects synthesis techniques would be a welcome contribution to sandbox games like *Minecraft* (*Mojang*, 2011) where players construct their own worlds and machinery, since it could potentially provide infinite sound signatures for the players' creations. There are also the already mentioned music automation techniques from Section 2.2. One could research into specific aesthetics and design real-time music composition frameworks that can generate or manipulate the playback of music tracks in games with little or no effort from both programmers and sound designers. For example, we could study the use of Jazz themes in Shoot'em Up games, establishing a standard protocol for programmers to feed genre-specific data to the middleware and for the sound designer to compose the themes around with. Another possible approach to this could be to use machine learning to derive music tracks from a large database of themes pertaining to the targeted game or music genre, as long as *Pure Data* is able to handle the demanded level of computation complexity; or one could use on-line machine learning that judges the player's acceptance of the presented soundtrack as he or she plays the game, tuning it to his or her taste or performance.

For game programming in a more general sense, it might be necessary to reconsider how sound is approached from game code if the real-time behaviors of the soundtrack become more frequent and time-intensive. There might be other design patterns and algorithms, beyond those discussed in Chapter 5, which are more appropriate for such extreme conditions. Besides, when the CPU is no longer enough for real-time computation of a game soundtrack, there are many possible approaches using the GPU instead, specially with the recent release of *Vulkan*[4]. This new technology has yet to be fully explored by both the industry and the academia. There are certainly a number of new graphics and audio pipeline architectures to be discovered.

[3] In the recent title *Overwatch* (*Blizzard*, 2016), for instance, avatars shout out when they are going to unleash a powerful ability, and the fact that it helps player react to it is an intrinsic part of how the game is balanced.

[4] https://www.khronos.org/vulkan/

One specific topic we believe would be the next greatest step in this research, specially if we opt to dismiss *Pure Data*, is how to computationally represent real-time soundtracks. That is, finding data structures, protocols, and eventually higher level abstractions that form an effective, efficient, and convenient representation of what real-time soundtracks are, what they can do, how they interact with the game, etc. For instance, with 3D models, due to the sheer amount of geometry data involved, most animation formats are designed around key frames, using interpolation to determine intermediate movements and to mix animations together. Using it as a metaphor, we could wonder what would be the "vertices" and "faces" of sound, and how they could be "animated" in real-time. Finding something of this sort while trying to keep aesthetic impositions to a minimum would be a very adequate follow up research topic to this thesis.

Games would be nowhere as entertaining without the sonic engagement of players, be it to reinforce the suspension of disbelief by giving texture to the game narrative, or be it to tell what the image cannot explain by itself. That is why, just as the actions of the player leads to both epic adventures and wrenching tragedies, so should they take part in the construction of the music and sound of that experience. By sticking to the stagnating sample-based approach that permeates the industry, we are actively taking the safer, but longer path towards that end. With this research and the proposed middleware technology, we provide one of the many missing pieces of the puzzle that effectively bridge programmers and sound designers in this joint quest for interactive, dynamic, adaptive and real-time soundtracks that will make our games ever better.

Bibliography

[Aqu13] Aquegg. 4-bit-linear-pcm.svg. http://creativecommons.org/licenses/by-sa/3.0, 2013. Creative Commons BY-SA 3.0 License, last access July 14, 2015. 25

[Atl08] Atlus. https://nintendo-okie.com/2010/09/22/shin-megami-tensei-persona-4-review/, 2008. Last access August 15, 2016. 29

[Aud15a] AudioKinetic. Audiokinetic customers. https://www.audiokinetic.com/community/customers/, 2015. Last access September 22, 2016. 50

[Aud15b] AudioKinetic. Audiokinetic documentation. https://www.audiokinetic.com/library/, 2015. Last access September 22, 2016. 52, 53, 54

[Aud16a] AudioKinetic. Audiokinetic wwise logo. https://www.audiokinetic.com/products/wwise/, 2016. Last access September 22, 2016. 50, 56

[Aud16b] AudioKinetic. Video tutorials. https://www.audiokinetic.com/resources/videos/, 2016. Last access September 29, 2016. 51

[BA04] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2Nd Edition)*. Addison-Wesley Professional, 2004. 68

[Bas15] Sérgio Basbaum. Interação música-imagem. Lecture Notes for Sound Design Course, 2015. 28, 29

[BDT+08] Nicolas Bonneel, George Drettakis, Nicolas Tsingos, Isabelle Viaud-Delmon and Doug James. Fast modal sounds with scalable frequency-domain synthesis. *ACM Trans. Graph.*, 27(3):24:1–24:9, Agosto 2008. 21, 73

[BGM+01] Kent Beck, James Grenning, Robert C. Martin, Mike Beedle, Jim Highsmith, Steve Mellor, Arie van Bennekum, Andrew Hunt, Ken Schwaber, Alistair Cockburn, Ron Jeffries, Jeff Sutherland, Ward Cunningham, Jon Kern, Dave Thomas, Martin Fowler and Brian Marick. Manifesto for agile software development. http://agilemanifesto.org/, 2001. 33, 68

[DPS+15] Gabriel Durr, Lys Peixoto, Marcelo Souza, Raisa Tanoue and Joshua D. Reiss. Implementation and evaluation of dynamic level of audio detail. In *Audio Engineering Society Conference: 56th International Conference: Audio for Games*, Feb 2015. 10

[DT07] Christopher DeCoro and Natalya Tatarchuk. Real-time mesh simplification using the gpu. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 161–166, New York, NY, USA, 2007. ACM. 10

[E01] Thomas Engel and Factor 5. A technique to instantaneously reuse voices in a sample-based synthesizer. In Mark DeLoura, editor, *Game Programming Gems II*, pages 521–524. Charles River Media, 2001. 9

[ENF06] Mirjam Eladhari, Rik Nieuwdorp and Mikael Fridenfalk. The soundtrack of your mind: Mind music - adaptive audio for game characters. In *Proceedings of the 2006 ACM SIGCHI International Conference on Advances in Computer Entertainment Technology*, ACE '06, New York, NY, USA, 2006. ACM. 19, 20

[Ent14] Blizzard Entertainment. http://hearthstone.gamepedia.com/File:Main_menu.jpg, 2014. Last access August 15, 2016. 29

[Ext] ExtraCreditz. Video game music. https://www.youtube.com/watch?v=CKgHrz_Wv6o. Last access July 27, 2016. 18

[Far07] Andy Farnell. An introduction to procedural audio and its application in computer games. 2007. 4, 8, 9, 11, 16, 21, 69, 126

[Far10] Andy Farnell. *Designing Sound*. The MIT Press, 2010. 4, 9, 16, 21, 63, 128

[FMM08] Fabio Furlanete, Jônatas Manzolli and Kenji Mase. Ludo: A collective sound sculpting game over the network. 2008. 22

[Fre] Will Freeman. Sound in transit. Develop, July 2014 issue, p. 19. 55

[Gam12] Subset Games. https://en.wikipedia.org/wiki/File:FTL_Faster_Than_Light_Screenshot.jpg, 2012. Last access July 25, 2016. 7

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. 85

[Gre14] Jason Gregory. *Game Engine Architecture*. A. K. Peters/CRC Press, 2014. 2, 6, 32, 34, 35, 37, 39, 42, 76

[Har09] Alex Harden. https://upload.wikimedia.org/wikipedia/commons/a/a2/Minimoog_(Buffalo_Museum_of_Science).jpg, 2009. Last access October 21, 2016. 100

[HLZ04] Robin Hunicke, Marc Leblanc and Robert Zubek. Mda: A formal approach to game design and game research. In *In Proceedings of the Challenges in Games AI Workshop, Nineteenth National Conference of Artificial Intelligence*, pages 1–5. Press, 2004. 33

[JBP06] Doug L. James, Jernej Barbič and Dinesh K. Pai. Precomputed acoustic transfer: Output-sensitive, accurate sound generation for geometrically complex vibration sources. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, pages 987–995, New York, NY, USA, 2006. ACM. 21, 73

[Joh97] Ralph E. Johnson. Components, frameworks, patterns. In *Proceedings of the 1997 Symposium on Software Reusability*, SSR '97, pages 10–17, New York, NY, USA, 1997. ACM. 39

[KC08] Karen Karen Collins. *Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design*. The MIT Press, 2008. 1, 3, 17, 25, 28, 29, 42

[Lib16] Libpd. Libpd showcase. http://libpd.cc/portfolio/showcase/, 2016. Last access September 29, 2016. 63

[LK04] Nelson Posse Lago and Fabio Kon. The quest for low latency. In *Proceedings of the International Computer Music Conference*, pages 33–36, 2004. 38

[LMBT10] Steven R. Livingstone, Ralf Muhlberger, Andrew R. Brown and William F. Thompson. Changing musical emotion: A computational rule system for modifying score and performance. *Computer Music Journal*, 34(1):41–64, Março 2010. 19, 20, 27

[Luc91] LucasArts. https://upload.wikimedia.org/wikipedia/en/8/82/Monkey_island_2_prison.png, 1991. Last access September 19, 2016. 48

[Luc94] LucasArts. Method and apparatus for dynamically composing music and sound effects using a computer entertainment system. United States Patent 5315057, 1994. 47

[Mat14] Eugênio Matos. *A Arte de Compor Música para o Cinema*. Senac, Brasília, DF, Brasil, 2014. 1, 4, 19, 21, 28, 29, 30, 31, 127

[Men13] Lucas Correia Meneguette. Situações Sonoras e Jogos Digitais. *Simpório Brasileiro de Games*, pages 30–33, 2013. 1, 16

[Mey05] Scott Meyers. *Effective C++, Third Edition*. Addison-Wesley, 2005. 85

[Miz] 83, 106

[MK16] Wilson K. Mizutani and Fabio Kon. An extensible and flexible middleware for real-time soundtracks in digital games. In *Proceedings of the 12th International Symposium on CMMR*, pages 175–182. The Laboratory of Mechanics and Acoustics, 2016. 37, 117

[Moj08] The International House of Mojo. Lucasarts' secret history: Monkey island 2: Lechuck's revenge. http://mixnmojo.com/features/sitefeatures/LucasArts-Secret-History-Monkey-Island-2-LeChucks-Revenge/, 2008. Last access September 19, 2016. 47

[MP11] Lucas Correia Meneguette and Pontifícia Universidade Católica De São Paulo. Áudio Dinâmico Para Games : Conceitos Fundamentais E Procedimentos De Composição Adaptativa. *Simpório Brasileiro de Games*, pages 1–10, 2011. 29

[MVK16] Wilson K. Mizutani, Dino Vicente and Fabio Kon. Sound wanderer: An experimental game exploring real-time soundtrack with openda. Demonstration at the 12th International Symposium on Computer Music Multidisciplinary Research, 2016, São Paulo, 2016. 68, 117

[Nin85] Nintendo. https://en.wikipedia.org/wiki/File:NES_Super_Mario_Bros.png, 1985. Last access July 22, 2016. 3

[NW07] Chris Nelson and Burkhard C. Wünsche. Game/music interaction: An aural interface for immersive interactive environments. In *Proceedings of the Eight Australasian Conference on User Interface - Volume 64*, AUIC '07, pages 23–26, Darlinghurst, Australia, Australia, 2007. Australian Computer Society, Inc. 22

[Nys14] Robert Nystrom. *Game Programming Patterns*. Genever Benning, 2014. 35, 37, 42, 103

[Pau03] Leonard Paul. Audio prototyping with pure data. http://www.gamasutra.com/view/feature/131258/audio_prototyping_with_pure_data.php, 2003. Last access November 16, 2016. 62

[PH07] J. R. Parker and J. Heerema. Musical interaction in computer games. In *Proceedings of the 2007 Conference on Future Play*, Future Play '07, pages 217–220, New York, NY, USA, 2007. ACM. 22

[PH08] J. R. Parker and John Heerema. Audio interaction in computer mediated games. *Int. J. Comput. Games Technol.*, 2008:1:1–1:8, Janeiro 2008. 22

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation, Second Edition.* Morgan Kaufmann, 2010. 21

[Puc16] Miller Puckette. Pure data logo. https://puredata.info/downloads/pure-data/logo, 2016. Last access September 29, 2016. 63

[Rab00] Steve Rabin. The magic of data-driven design. In Mark DeLoura, editor, *Game Programming Gems*, pages 3–7. Charles River Media, 2000. 5, 6

[Roa96] C. Roads. *The Computer Music Tutorial.* MIT Press, 1996. 24

[Roy70] Winston Royce. Managing the development of large software systems. In *In Proceedings of IEEE WESCON*, pages 1–9, 1970. 33, 69

[RSM+10] Nikunj Raghuvanshi, John Snyder, Ravish Mehra, Ming Lin and Naga Govindaraju. Precomputed wave simulation for real-time sound propagation of dynamic sources in complex scenes. *ACM Trans. Graph.*, 29(4):68:1–68:11, Julho 2010. 21, 73

[Sch14] Jesse Schell. *The Art of Game Design: A Book of Lenses, Second Edition.* A. K. Peters/CRC Press, 2014. 1, 2, 28, 33

[Sch15a] Stephan Schutze. The music system 15 years in the making. http://gamasutra.com/blogs/StephanSchutze/20150105/233385/The_music_system_15_years_in_the_making.php, 2015. Last access April 29, 2015. 59

[Sch15b] Stephan Schutze. The music system 15 years in the making part 2. http://gamasutra.com/blogs/StephanSchutze/20150113/234015/The_music_system_15_years_in_the_making_Part_2.php, 2015. Last access April 29, 2015. 59

[Sco14] Nathan Scott. Music to middleware: The growing challenges of the game music composer. In *Proceedings of the 2014 Conference on Interactive Entertainment*, IE2014, pages 34:1–34:3, New York, NY, USA, 2014. ACM. 15, 47, 71, 121

[She64] Roger N. Shepard. Circularity in judgements of relative pitch. *Journal of the Acoustical Society of America*, 36:2346–53, 1964. 115

[Sof16a] Elias Software. Elias logo. https://www.eliassoftware.com/licensing/, 2016. Last access September 28, 2016. 60

[Sof16b] Elias Software. Tutorials. https://www.eliassoftware.com/about/tutorials/, 2016. Last access September 28, 2016. 61

[Sta12] Stabyourself. http://stabyourself.net/images/screenshots/mari0-1.png, 2012. Last access October 13, 2016. 83

[Stu99] Ensemble Studios. https://en.wikipedia.org/wiki/File:Age_ii_feudal_age_celts.jpg, 1999. Last access July 22, 2016. 4

[Stu16] Okam Studio. Godot logo. https://commons.wikimedia.org/wiki/File:Godot_(game_engine)_logo.svg, 2016. Last access October 26, 2016. 107

[TCAM09] Micah T. Taylor, Anish Chandak, Lakulish Antani and Dinesh Manocha. Resound: Interactive sound rendering for dynamic virtual environments. In *Proceedings of the 17th ACM International Conference on Multimedia*, MM '09, pages 271–280, New York, NY, USA, 2009. ACM. 21, 73

[Tea16] LÖVE Development Team. LÖve logo. https://love2d.org/wiki/Löve_Logo_Graphics, 2016. Last access October 26, 2016. 106

[Tec16a] Firelight Technologies. Adaptive music in fmod studio: Transition markers and logic. http://www.fmod.org/adaptive-music-fmod-studio-transition-markers-logic/, 2016. Last access September 26, 2016. 58

[Tec16b] Firelight Technologies. Adaptive music in fmod studio: Transition timelines and submixes. http://www.fmod.org/adaptive-music-fmod-studio-transition-timelines-submixes/, 2016. Last access September 26, 2016. 59

[USP10] USPGameDev. https://uspgamedev.org/horus-eye/, 2010. Last access July 25, 2016. 5

[VN14] Bruno Santos Viana and Ricardo Nakamura. Immersive interactive narratives in augmented reality games. In *Proceedings of the Third International Conference on Design, User Experience, and Usability. User Experience Design for Diverse Interaction Platforms and Environments - Volume 8518*, pages 773–781, New York, NY, USA, 2014. Springer-Verlag New York, Inc. 1

[Wik] The Free Encyclopedia Wikipedia. Lua-scripted video games. https://en.wikipedia.org/wiki/Category:Lua-scripted_video_games. Last access July 25, 2016. 6

[WL01] Keith Weiner and DiamondWare Ltd. Interactive processing pipeline for digital audio. In Mark DeLoura, editor, *Game Programming Gems II*, pages 529–538. Charles River Media, 2001. 4, 37

[Woo98] Bobby Woolf. *Pattern Languages of Program Design 3 - Null Object*. Addison-Wesley, 1998. 85