An Extensible and Flexible Middleware for Real-Time Soundtracks in Digital Games

Wilson Kazuo Mizutani and Fabio Kon

Department of Computer Science
University of São Paulo
{kazuo,kon}@ime.usp.br
http://compmus.ime.usp.br/en/opendynamicaudio

Abstract. Real-time soundtracks in games have long since faced design restrictions due to technological limitations. The predominant solutions of hoarding prerecorded audio assets and then assigning a tweak or two each time their playback is triggered from the game's code leaves away the potential of real-time symbolic representation manipulation and DSP audio synthesis. In this paper, we take a first step towards a more robust, generic, and flexible approach to game audio and musical composition in the form of a generic middleware based on the Pure Data programming language. We describe here the middleware architecture and implementation and its initial validation via two game experiments.

Keywords: game audio, game music, real-time soundtrack, computer music, middleware

1 Introduction

Digital games, as a form of audiovisual entertainment, have specific challenges regarding soundtrack composition and design. Since the player's experience is the game designer's main concern, as defended by Schell (2014), a game's soundtrack may compromise the final product's quality as much as its graphical performance. In that regard, there is one game soundtrack aspect that is indeed commonly neglected or oversimplified: its potential as a *real-time*, procedurally manipulated media, as pointed out by Farnell (2007).

1.1 Motivation

Even though there is a lot in common between game soundtracks and other more "traditional" audiovisual entertainment media soundtracks (such as Theater and Cinema), Collins (2008) argues that there are also unquestionable differences among them, of either technological, historical, or cultural origins. The one Collins points as the most important difference is the deeply nonlinear and interactive structure of games, which make them "*largely unpredictable in terms of the directions the player may take, and the timings involved*". Because of this, many game sounds and music tracks cannot be merely exposed through common

playback (as happens with prerecorded media): they also need some form of procedural control to be tightly knit together with the game's ongoing narrative. However, this is seldom fully explored. Except in a few remarkable cases, most game musical soundtracks tend to have little real-time connections between what is happening in the game and what is going on with the music, for instance.

The ways with which real-time soundtracks are typically dealt with are poor and do not scale well. Basically, the development team needs to find a common ground for musicians, sound designers, and programmers where they can reach an agreement on how to introduce real-time behaviour into the game source code modules related to audio reproduction and sound assets management. Farnell (2007) explains that the common approach is to produce as many assets as needed and then list all event hooks that must go into the game code to timely play the corresponding sounds and music (perhaps with one or two filters applied to the output). This is not only a waste of memory and a disproportional amount of effort, but also a very limited way of designing a game soundtrack. Farnell goes as far as to say that even advanced and automated proprietary middleware systems fail to provide a satisfactory solution, since they "*are presently audio delivery frameworks for prerecorded data rather than real 'sound engines' capable of computing sources from coherent models*".

1.2 Objective

In our research, we intend to provide an alternative to such excessively asset-driven solutions by empowering the musicians and sound designers with a tool to express procedurally how a game soundtrack is to be executed, and by embedding it into a cross-platform programming library that can be easily integrated into the development workflow of digital games. As such, we present, in this paper, Open Dynamic Audio (OpenDA), an open-source, extensible, and flexible middleware system for the development of real-time soundtracks in digital games as a first result of our ongoing research. The middleware implementation is available at https://github.com/open-dynamic-audio/liboda under the MPL 2.0 open source license.

2 Soundtrack Implementations in Digital Games

Matos (2014) states that soundtracks are essentially the union of all sound effects, voices, and music that are played along with a visual presentation. In the traditional asset-driven approach, each audio element from these soundtrack parts is implemented in the game code by specifying a playback trigger consisting mainly of (Farnell, 2007):

1. *Which* sample assets are to be played.
2. *How* they are to be played (that is, what filters should be applied).
3. *When* they are to be played.

As an example of this pattern, consider a gunshot sound effect. Using a prerecorded gunpowder explosion sound, one could produce different firearms sounds by applying multiple filters to it and then mapping the corresponding configuration whenever a weapon is shot. This way, when the player's character wields a specific type of pistol, its respective sound effect is triggered.

Being able to synchronize an audio element playback achieves an initial level of real-time behaviour on its own. It is often further expanded by allowing the other two parameters (the *which* and the *how*) to change according to the game state. In the game *Faster Than Light*[1], every musical piece in the soundtrack has two versions - one for exploration and one for combat - and they are cross-faded between each other whenever the game situation changes from exploration to combat and vice-versa. This consists of both a modification in the sample used (the *which*) and a real-time control over the filters, responsible for fading out the previous version of the music while fading in the next one (the *how*).

However, since this method specifies only whole samples to be played, it excludes from the sound design space other forms of audio construction, notably symbolic representation (e.g., MIDI) and Digital Signal Processing (DSP) audio synthesis. The IMuse system (LucasArts 1994) was an example of how to use symbolic representation to allow music changes in real-time. Essentially, it provided if-then-jump commands among the other typical music score based messages, bringing symbolic representation closer to a programming language of its own. Regarding DSP audio synthesis, there are quite a few works on physically based real-time audio synthesis (James et al. 2006, Bonneel et al. 2008, Farnell 2010), which could contribute to many unexplored ways of dealing with sound effects in games using no sample files, but at a greater computational cost.

3 Real-Time Restrictions in Game Implementation

To be considered a real-time application, digital games rely on the *Game Loop* architectural pattern (Nystrom 2014). It guarantees that the time difference between the continuous input processing and output generation is so short that the user experiences it as being instantaneous. This is accomplished by dividing the program execution into very short steps between each input handling and output rendering and then finely controlling the process rate of these cycles inside an endless loop.

Nystrom (2014) shows how the Game Loop frequency is related to the game Frames Per Second ratio (FPS), i.e., the ratio of how many graphical frame buffers are fully rendered per second. Ideally a game's FPS is greater than or equal to the Game Loop frequency, which means that its visual output may change and adapt at least as often as changes are made to the game state. On the other hand, conventional asset-driven audio implementations in games provide a slower feedback mechanism, since they load as many audio samples as possible from the assets to the computer sound card (where they can no longer

[1] Subset Games, 2012

be promptly accessed) in each game cycle. The samples are transferred in chunks that are typically too big, causing effects applied to the game soundtrack to come with a perceptible delay, thus not satisfying the desirable real-time requirements. Essentially, it incurs in too much *latency*.

For instance, in the LÖVE game framework[2], the default audio stream buffer size contains 4096 samples, which, with an audio sample rate of 44100 Hz, leads to soundtrack changes being able to occur only every 93 milliseconds, approximately. The simple solution to this is to reduce the size of the audio buffers sent to the sound card, which actually means processing less audio in each game cycle. If a game is to be executed at 60 FPS and its audio is sampled at 44100 Hz, then each game cycle must provide only $44100/60 = 735$ audio samples. In the more general case, one cycle may actually have a variable time length. If we let f_{audio} be the audio sample rate (in Hertz) and Δt be the time difference (in seconds) between the current game cycle and the last, then the number N of maximum audio samples allowed for the current cycle would be:

$$N = f_{audio} \cdot \Delta t \tag{1}$$

As a matter of fact, the DSP graphical programming language *Pure Data*[3] has a standard cycle buffer size of merely 64 samples. That would be precise enough for a game running at approximately 689 FPS. The drawback of reducing the audio buffer size is that if the computations needed to fill it actually last long enough to make a perceptible time gap between each buffer update, then the sound might come out chopped by the sudden lack of samples to play. There is also an increased overhead in having a larger number of smaller data copies sent to the sound card. Thus, even though reducing the buffer size is important to decrease latency, a point of equilibrium must be found or the audio quality may be compromised. This depends on how much computation time the Game Loop has available for handling audio processing and on the technical capabilities of the sound hardware at our disposal.

4 Proposed Architecture

The main purpose of the OpenDA middleware is to bridge soundtracks and game engines. As such, we understand that its main user is the sound designer, although the programmers that bind the game code to the middleware must also be taken into consideration. This is commonplace for game audio middleware, requiring the division of the tool into two separate but complementary interfaces[4]. The first is a "soundtrack editor" - a visual application through which sound designers author audio content that can be later exported to the game.

[2] See http://love2d.org
[3] See https://puredata.info
[4] See Firelight's FMOD Studio (http://www.fmod.org) and Audiokinetic's Wwise (https://www.audiokinetic.com/products/wwise)

The second is a programming library exposed through an Application Programming Interface (API) that is capable of loading the media exported by the editor and playing it during the game execution (see Figure 1).

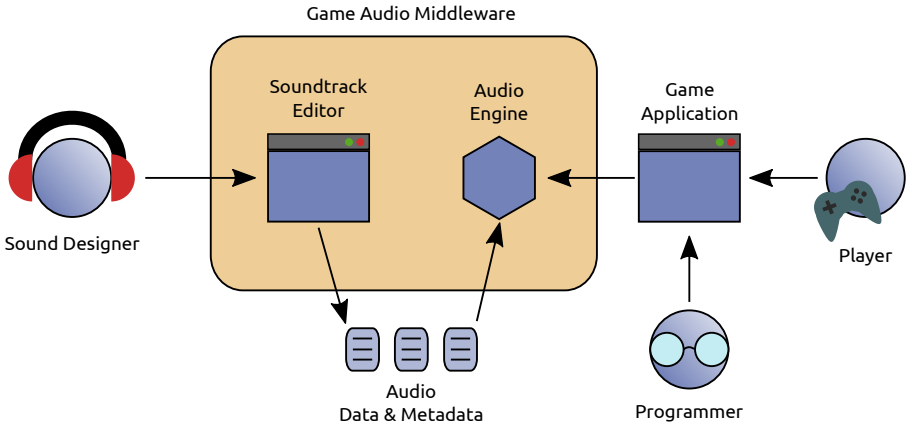

Fig. 1. Standard overview architecture for game audio middleware.

OpenDA follows this general architecture. However, instead of focusing the soundtrack editor in audio assets management, we chose to develop a procedure-oriented tool. We designed a collection of high-level abstractions in the Pure Data programming language that sound designers can use to produce Pure Data patches as part of a game soundtrack specification. The game engine can then link to OpenDA's audio engine (a programming library) to load and *execute* the soundtrack patches authored this way. Our intention is to focus on giving sound designers full control over the sonic behaviour instead of just rigid sonic scheduling, since they can *program the soundtrack themselves* with an accessible, community-approved language such as Pure Data.

4.1 Pure Data Integration

Pure Data originally comes as a stand-alone application capable of creating, editing, and executing patches by itself, promptly serving as our soundtrack editor. However, even though it is capable of communicating with other applications through sockets or MIDI channels, ideally one would not want to have multiple applications launched when playing a digital game. The solution would be to run Pure Data's DSP *from inside the game*. There is a community developed programming library called `libpd`[5] that provides access to Pure Data's core implementation, allowing its host program to load and execute Pure Data patches without the Pure Data application.

However, when using `libpd`, there is no audio output handling. The host application is only given the processed signal and is responsible for sending it to the sound card or doing whatever it wants with it. Additionally, the processed

[5] See http://libpd.cc

signal is provided in blocks of 64 stereo samples, as mentioned before. Our audio engine must synchronize the retrieval of these blocks with the game run time flow as in Equation (1). For that, the engine API routines demand precise timing information from the game execution *and* properly timed evocation, since time management is controlled by the game code.

The communication between the sound designer patches and our audio engine consists of two main types of data transfer. The first, which we just described, results from `libpd`'s audio signal computation happening every frame. The other transfer occurs when our middleware must inform the designer's patch of a relevant change within the game state. Based on this communication, the patch may do whatever it deems necessary for the soundtrack to follow up the game narrative. The overall architecture of our engine can be seen in Figure 2.

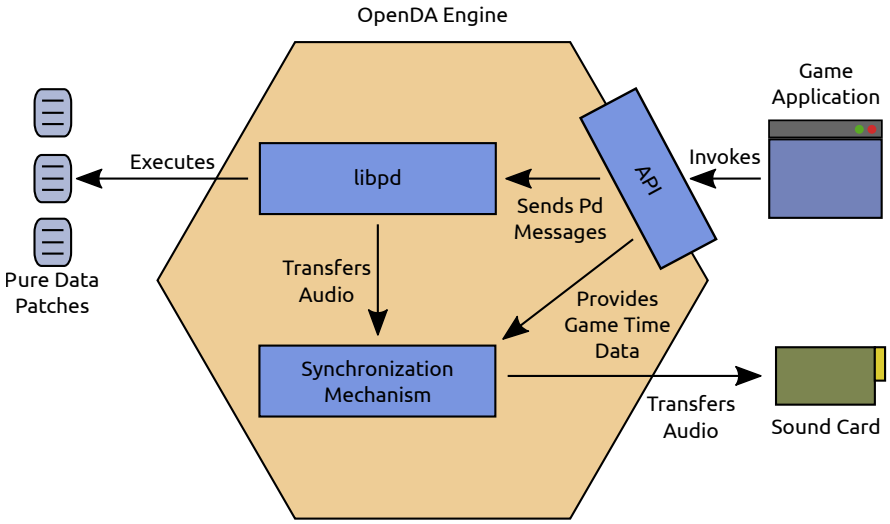

Fig. 2. OpenDA Engine's architecture.

5 Implementation

For the sake of game engine compatibility and performance (Nystrom 2014), we chose to develop our middleware in `C++`, except for the parts that must be made in Pure Data. The audio engine uses OpenAL[6] for cross-platform open-source-friendly access to the sound card, enabling us to produce proper playback of the desired audio.

5.1 Audio Engine

To satisfy the real-time restrictions described in Section 3, our middleware audio engine strongly relies on OpenAL's buffer queueing mechanism. It enables the

[6] See https://www.openal.org and http://kcat.strangesoft.net/openal.html

allocation of multiple buffers whose purpose is to send audio data to the sound card in FIFO order. Each buffer can have arbitrary sizes, but we fit them to Pure Data's cycle block size (64 samples). Then, OpenAL automatically switches the current buffer to the next one when it has finished playing. That way, even when the game cycles do not match Pure Data's cycles, we can schedule the next block. Doing so increases latency, but since the block is very small the difference is minimal and allows us to reduce the previously discussed overhead.

5.2 Low-Level Messages

Having completed a minimal framework that can integrate `libpd` with the Game Loop pattern, the current stage of the project also supports low level patch-engine communication. The audio transfer from the patch to the engine uses the Pure Data array API instead of its standard output since this simplifies the use of multiple simultaneous audio buses. The engine recognizes the audio buses thanks to a naming convention, but a convenient Pure Data abstraction is provided to wrap and hide this and other implementation details needed to properly synchronize the filling of the arrays with the engine cycles.

On the engine side, we implemented a very simple API for the game code to notify its state changes. It relies on Pure Data's messaging mechanism, accessed via `libpd`. With a single routine call, a message is sent to the sound designer's soundtrack patch containing as much information as desired, so long as it can be represented by a list of numbers and character strings (Pure Data symbols).

5.3 Game Audio Experiments and Prototypes

To validate our proposal, we are using our middleware to create soundtracks for two very different games. First, we forked the open-source game Mari0[7] and replaced its default soundtrack for one entirely produced by our middleware. The game is a parody of two other famous games: Super Mario Bros. (Nintendo 1985) and Portal (Valve 2007). The focus was the music track, and we experimented only with Koji Kondo's "Overworld Main Theme", the music track for the first stage of the game. By dividing the music in its three voices – melody, bass and percussion – and the score bars into a few sections, we were able to add the following real-time behaviours to the game soundtrack:

1. The music advances through the bars as the player progresses through the stage, so that each of its parts has a particular music segment associated.
2. Mario's size directly influences the bass voice. The stronger he gets, the louder the bass line becomes.
3. The quantity of enemies nearby also increases the percussion intensity, in an attempt to suggest that the situation became more action-intensive.

The second validation is a completely new soundtrack being composed in partnership with a professional sound designer. It follows the bullet hell genre,

[7] See http://stabyourself.net/mari0

where the player moves an avatar that must dodge dozens of falling bullets in a vertical shooter format. The idea is to find a way to synchronize the bullets' choreography to the soundtrack music, making the gameplay itself a form of composition. To achieve that, we are working in partnership with volunteer sound designers, while also evaluating the tool's usage by potential users. The source code of both games is available under an open source license at http://compmus.ime.usp.br/en/opendynamicaudio/games.

6 Conclusion and Ongoing Work

The current priority of our ongoing research is to shape a higher level abstraction model for how a game real-time soundtrack should be authored. Finishing the bullet hell prototype will help validate this model. There are interesting challenges waiting ahead in the project roadmap. First, as widespread as Pure Data is, it lacks a simple user interface that would be appealing to sound designers and composers with no programming experience. Second, with a new soundtrack composition workflow comes a whole new skill set for sound designers and composers to reach out to – an unavoidable cost of breaking well established paradigms (Scott 2014). Our expectations are that, with this work, we will at least open the way for new methods of producing soundtracks for digital games that actually try to exploit the medium dynamic and interactive nature, avoiding further waste of this game design space.

References

Bonneel, N., Drettakis, G., Tsingos, N., Viaud-Delmon, I., James, D.: Fast Modal Sounds with Scalable Frequency-Domain Synthesis. ACM Trans. Graph. volume 27, number 3 (2008)

Collins, K.: Game Sound: An Introduction to the History, Theory, and Practice of Video Game Music and Sound Design. The MIT Press (2008)

Farnell, A.: An introduction to procedural audio and its application in computer games. http://cs.au.dk/ dsound/DigitalAudio.dir/Papers/proceduralAudio.pdf (2007)

Farnell, A.: Designing Sound. The MIT Press (2010)

James, D. L., Barbic J., Pai, D. K.: Precomputed Acoustic Transfer: Output-Sensitive, Accurate Sound Generation for Geometrically Complex Vibration Sources. Proceedings of ACM Special Interest Group on Graphics and Interactive Techniques, Volume 25 Issue 3, Pages 987-995 (2006)

LucasArts: Method and Apparatus for Dynamically Composing Music and Sound Effects Using a Computer Entertainment System. United States Patent 5315057 (1994)

Matos, E.: A Arte de Compor Música para o Cinema. Senac (2014)

Nystrom, R.: Game Programming Patterns. Genever Benning (2014)

Schell, J.: The Art of Game Design: a Book of Lenses, Second Edition. A. K. Peters, CRC Press (2014)

Scott, N.: Music to Middleware: The Growing Challenges of the Game Music Composer Proceedings of the Conference on Interactive Entertainment, Pages 1-3 (2014)