DESIGNING A LIBRARY FOR GENERATIVE AUDIO IN UNITY

Enrico Dorigatti and Stephen Pearse

School of Creative Technologies
University of Portsmouth
Portsmouth, UK
`enrico.dorigatti@port.ac.uk | stephen.pearse@port.ac.uk`

ABSTRACT

This paper overviews URALi, a library designed to add generative sound synthesis capabilities to Unity. This project, in particular, is directed towards audiovisual artists keen on working with algorithmic systems in Unity but can not find native solutions for procedural sound synthesis to pair with their visual and control ones. After overviewing the options available in Unity concerning audio, this paper reports on the functioning and architecture of the library, which is an ongoing project.

1. INTRODUCTION

Unity is a game development software used in a range of scenarios by a diverse and wide audience, from enthusiasts to researchers in academia. Besides its capabilities concerning the development of multi-platform games and software—Mac OS, Windows, iOS, and Android being the most popular ones—the wide choice of options and the flexibility it offers makes it the ideal, user-friendly environment for fast prototyping especially when it comes to VR (virtual reality), XR (extended reality), and the production of virtual and simulated environments. Examples of the widespread usage and diverse contexts of application can be found in [1] and [2] (VR); [3] (business); [4] (visualisation of biomedical data); [5] (scholarly research); and [6] (automotive). Furthermore, it is also used within the artistic context. Interestingly, there is a general lack of academic resources reporting on this usage of Unity; however, it is possible to trace it back by visiting the websites of some artists and reading their artistic statements or program notes—although, as the focus is usually on the artistic outcome or message conveyed by an artwork, the tools used in the creative process are often omitted. Some examples, however, are reported below, and, additionally, a dedicated page on the Unity website, reports and highlights the specific features the software offers to artists and designers[1].

Unity is composed of two main components. Firstly, a robust graphics and 3D engine allow one to quickly and easily draft complex scenes and environments using 3D models, lights and shadows, and materials with custom properties, as well as effects such as particle systems, filters, and custom shaders—either written in GLSL or created via the built-in node-based editor. Furthermore, the physics engine allows for the design of interactions between objects and, in general, the different surfaces of the environment, making it possible to create complex behaviours that either mimic real-world physics or introduce randomness and imaginary behaviours.

Within Unity, these two systems work seamlessly, and, together with the possibility to control the behaviour of almost any parameter via custom C# scripts—thus connecting graphic power to computation—makes Unity the ideal environment when it comes to the design of generative art systems based on algorithms, either contemplating human interaction or not. Some examples this way are given by the works of Danish artist Carl Emil Carlsen[2], some software developed by the composer and media artist Giovanni Albini, such as Memoriale[3], and Life[4], a generative artwork developed by the first author and based on the Life algorithm developed by J. H. Conway [7]. However, when it comes to audio, Unity does not natively offer the same level of flexibility one can find in its components dedicated to scripting and visuals, especially when it comes to algorithmically based projects.

Nowadays, the main competitor of Unity is Unreal Engine, a source-available proprietary software which includes Metasound[5], a low-level, sample-accurate node-based system that allows developers to create synthesis and music systems within the engine—one of the most notable additions to the fifth version of the software. However, despite the great potential of Unreal Engine, establishing itself as the leading and reference platform in a wide range of scenarios spacing from architecture and automotive rendering and visualisation to game development, it is a popular opinion (e.g. on dedicated websites[6] or online communities[7]), especially amongst enthusiasts and small or indie developers that it has a steeper learning curve and developing a project from scratch with it requires much more effort—although strategies such as the Blueprints system[8] have been implemented to ease it out. Unity, on the contrary, has established itself as the go-to platform for fast prototyping and creation, especially when it comes to mobile devices. In this perspective, the project presented in this paper could be seen as a rudimental version of Metasound, an attempt to fill the gap concerning procedural audio synthesis in Unity highlighted when developing Life. The goal of URALi is thus to offer user-friendly generative audio capabilities directly from within Unity.

[1]https://unity.com/solutions/artist-designers

[2]https://cec.dk/

[3]https://play.google.com/store/apps/details?id=com.albinigiovanni.memoriale

[4]https://www.enricodorigatti.com/wp-content/uploads/2022/01/Life.mp4

[5]https://docs.unrealengine.com/5.0/en-US/metasounds-in-unreal-engine/

[6]https://gamedevacademy.org/unity-vs-unreal/

[7]https://www.quora.com/Is-Unity-easier-than-Unreal

[8]https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/

2. AUDIO IN UNITY

When it comes to generative audio, Unity does not natively offer solutions matching its possibilities in terms of graphics and physics. However, besides the stock audio filters providing common effects such as delay, echo, equalisation, and 3D spatialisation, different approaches can be followed to deal with sound when designing an application. Such approaches imply choosing different objects and workflows and are highly dependent on the type of project one is developing.

2.1. Audio Clips

The main way to deal with audio is based on the usage of pre-made clips loaded as assets of the application in development and which can be played when necessary—for example, when an action is performed in the case of a game or a button is pressed in the case of a user interface. Some methods of the AudioClip[9] object allow the programmer to retrieve information from the audio file—such as frequency, number of channels, and duration—and perform fairly advanced operations, such as setting the sample data that the clip contains. It is clear, however, that filling a clip with procedurally generated audio data is not a trivial operation and, on average, this method will mostly come in handy for shaping the amplitude of the original data, as shown in the example provided along with the documentation of the method[10].

Therefore, whilst audio clips are necessary in the average scenario—for example in the case of a game, where a finite set of defined actions and thus audio events is repeated over and over and possibly manipulated through the stock filters based on the properties of the action itself such as the composition of the ground for footstep—they are not the ideal solution in the case of generative and algorithmic systems, when the properties of the system itself 'evolves' unpredictably, based on the state of an algorithm. The most common scenario involving the usage of clips sees a scene prepared with an audio listener component—the 'ears' of the user—and different audio clips ready to be triggered when a condition occurs and possibly manipulated and filtered by the spatial properties of the space in which the scene takes place, as well as the distance of the player from the spot where the action happens. Employing AudioClip objects can be seen as the standard and most common way of dealing with audio in Unity and fits the vast majority of use cases—although it offers a limited amount of options and freedom.

2.2. Middleware

Another possible way to deal with audio in Unity is to employ audio middleware such as the popular Wwise[11] or FMOD[12]. Those tools, designed and developed with the video game industry in mind are meant to offer flexibility in this specific context, allowing studios to separate the development of the audio engine from the development of the game engine, keeping them, however, linked and eventually integrating them together. The functioning of middleware is mainly based on the audio clips a sound designer imports, which can however be triggered and manipulated in complex ways based on the calls and parameters the middleware receives from the game engine—an approach granting a high amount of freedom and flexibility.

The drawback of this approach to audio in Unity is that software such as Wwise and FMOD can be complex to master and not everybody, especially in the case of one-man-teams, is likely to learn them from scratch—on the contrary, in medium to big-sized game studios, there are professionals whose job is just to integrate the sounds provided by the sound designers, building all the pipeline and system necessary to make them work smoothly in accordance to what happens in the game engine, thus providing a smooth interaction between the two systems.

2.3. Third-party software

Furthermore, another and more experimental way to deal with audio in Unity contemplates the connection of third-party software providing specific environments, tools, and abstractions for sound and music—the most popular ones being SuperCollider, Pure Data, ChucK, and Max/MSP—through protocols such as the popular OSC [8]. These software are well known in the experimental music and sound design contexts for providing sandboxes in which it is possible to create algorithms for sound synthesis and manipulation and algorithmic music composition.

Offering probably the highest amount of flexibility, however, similar to what happens with middleware, they are different systems and languages from Unity, which means that, once again, one has to learn how to use them from scratch. On top of that, unlike Wwise or FMOD, they are not designed to natively integrate with Unity, as they are primarily conceived as standalone software. Thus, this means that one has to employ tools acting as bridges to port their functionalities within the software developed in Unity, which in turn leads to compatibility issues and the impossibility of using all the functions, classes, and objects available—especially when it comes to export a project for platforms such as Android. Projects like Chunity [9] have been developed to offer a fully-functioning connection between Unity and the ChucK audio programming language [10], allowing one to fully integrate Chuck code within Unity projects. However, this solution does not solve the issues of the different languages, which still forces an artist or developer to learn a different and specific one.

2.4. OnAudioFilterRead

There is one last option to deal with audio in Unity natively by taking advance of the possibility offered by implementing a custom OnAudioFilterRead[13] function. According to the documentation, it fits the audio DSP chain as a custom filter—a filter is each effect within the chain, such as an echo effect or a low-pass filter—manipulating the data flowing from the preceding one. However, if OnAudioFilterRead is the first or sole filter in the chain, it can be used to procedurally generate the audio data. The purpose of OnAudioFilterRead is, to summarise, to provide an 'access window' on the audio data passing through. Beyond that, it is the programmer's choice to decide where to place it and, therefore, whether to use it to generate the data or only access and manipulate the existing ones—as an example, multiplying them by 0.5 will approximately halve the level output level.

Similar to what has been previously said about the AudioClip object, the challenge resides in that generating audio procedurally can be challenging. However, the difference lies in the way the two

[9]https://docs.unity3d.com/ScriptReference/AudioClip.html

[10]https://docs.unity3d.com/ScriptReference/AudioClip.SetData.html

[11]https://www.fmod.com/unity

[12]https://www.fmod.com/unity

[13]https://docs.unity3d.com/ScriptReference/MonoBehaviour.OnAudioFilterRead.html

systems work. Whilst audio clips are especially useful when dealing with fixed sounds that do not change and need to be played only when necessary (e.g. a footstep occurring only when the player moves a step), OnAudioFilterRead, on the contrary, gives the possibility to create, manipulate and, in general, work on constant streams of data in real-time. This, in turn, makes possible the realisation of a flexible framework for the algorithmic generation of the samples—that is, the procedural generation of sounds.

3. URALI

Taking advantage of the OnAudioFilterRead function, URALi is a library designed and developed to provide an easy-to-use framework for procedural audio in Unity without the need for external software such as Wwise or FMOD nor specialised programming languages such as SuperCollider or Pure Data and related bridging software when the connection is not natively possible. URALi is the acronym for Unity Real-Time Audio Library, and, as the name suggests, it offers a collection of functions and classes meant to work as the building blocks for audio algorithms. The functioning of the library recalls the fashion of the aforementioned audio programming languages, where the programmer can use, connecting them together, a set of objects and functions which eventually will compose an algorithm for audio synthesis or manipulation. As it happens in SuperCollider, such objects are defined by strings instead of being represented by visual nodes, characteristics of languages such as Pure Data or Max/MSP.

The development of URALi started with different motivations, as it originally was an exploratory attempt to investigate possible solutions for generative audio in Unity for the artwork described in [7]. However, as the system developed proved worthy, and, most importantly, with potential yet to explore, it was reworked with the goal of building a solid, flexible, and scalable framework designed to offer easy and efficient access and utilisation of the functions and objects of the library, avoiding the drawback of the other audio solutions previously explored [11].

3.1. Design of the Library

The first consideration done while developing the library was that relying on the thread running OnAudioFilterRead for the calculation of the audio samples would lead—especially in the case of complex synthesis algorithms with many different generators, and especially when running on older systems—to slowdowns and data starvation. The official OnAudioFilterRead documentation indeed states that it needs to process chunks of audio data at fixed intervals to provide a smooth stream; should it not be able to do so—for example, due to too many calculations to perform—the data stream would break and, perceptually, this would result in glitches. For this reason, OnAudioFilterRead runs on a separate thread, which incidentally means that many Unity functions can not be used. This considered, as every operation performed within the dedicated thread would reduce the headroom for further processing and increase the risk of occurring in data starvation, it was decided to split the calculation of the data from their retrieval.

The solution implemented consisted of another separated thread continuously computing new audio samples without timing or constraints. The results are stored in a circular buffer accessible both by this dedicated thread and OnAudioFilterRead, and the safety of the data concerning overwriting is guaranteed by a system of flags which pauses the calculation of new samples if there is no more space available for storing them. Whilst paused, through busy waiting [12], the thread periodically checks if some buffer space has been marked as free by the flag system and possibly resumes its activity. The free space is flagged as that by OnAudioFilterRead, which, when necessary, fetches chunks of data from the buffer and, by leveraging the flag system, marks the corresponding slots as overwritable. Retrieved data are then sent through the pipeline, ready to be outputted or modified by any filters natively available in Unity, should they be stacked in the DSP chain.

User side, URALi is designed as a sandbox in which the programmer can work with all the classes and methods contained in the library and beyond. This means that it is possible to integrate and expand an audio chain built with the objects available from within URALi with custom processes and logic, as well as data, from other parts of the program—for example, to control the frequency of an oscillator based on the speed of the player, which is calculated in a separate script. URALi is indeed written in C#, the scripting language used in Unity, and this makes it possible to have a seamless stream of data to and from other scripts and to use functions defined elsewhere.

Concretely, URALi needs the programmer to define a synthesis function in which to build the actual synthesis chain. This function has to be passed to the class implementing the audio engine, and this latter has to be started. Once done, the dedicated audio thread of the library initiates its task, and audio data become available from the circular buffer as they get calculated. As a last step, to fetch these data and have them outputted, the programmer needs to implement a custom OnAudioFilterRead function from which to access, through a specific call to the library, the circular buffer, to retrieve the next chunk of audio data.

Outside the code, in the inspector, URALi, which within the environment is represented by OnAudioFilterRead, is visualised by the default visualisation of the latter, namely a VU meter showing the output level and the processing time it takes. However, due to the library design, this number should always stay in the green zone—provided that no additional operations are performed in OnAudioFilterRead—as all the calculations are done in the dedicated thread. As the OnAudioFilterRead representing the library is a node of the audio chain, it is possible to stack it with all the audio effects shipped with Unity, such as echo, delay, and filters. The possibility to seamlessly integrate URALi within the existing audio ecosystem of Unity makes it possible to streamline its development, avoiding the necessity of implementing, and thus duplicating, audio effects already available natively.

4. CONCLUSIONS

URALi is a project forked from specific technical and artistic research; thus, its advancement in terms of maintenance, optimisation, and implementation of new functions relies solely on the efforts and free time of the first author. This is why, although the project started in 2017 and was initially presented in 2019 [13], an initial version has not been released yet, and the documentation has not been compiled. At the current time, URALi provides access to an envelope generator, granular [14], frequency modulation (FM) [15], and additive synthesisers, waveshaper, LFO (low-frequency oscillator), lookup oscillator with built-in tables and the possibility to load new ones, panners with different pan laws, classic waveform oscillators (sawtooth, triangular, square, sinusoidal), and white noise generator. While implementing these nodes, the problem of aliasing [16] was faced and subsequently addressed by implementing the technique described by Välimäki et al. [17].

Other functionalities are already listed for implementation and have been selected by also taking inspiration from the palette of well-established software such as Max/MSP[14]. However, unlike this language, it was chosen to not include any object dealing with logic as, given the nature of URALi and the language in which it is written, and based on what has been discussed previously, it would be easier for a programmer to implement their custom conditions and logic to fit their peculiar case rather than understanding and adapting any generic object provided by the library. This is diametrally different from what happens in Max/MSP, where the possibility to use stock objects offering even the most basic logic functions, controls, and tasks improves the readability of the code and prevents the programmer from building large networks of control objects or even being forced to deal with scripting. In general, however, the confrontation with other musicians and artists would be beneficial for the development of URALi as it would help to understand what is missing and what should be prioritised.

Currently, some demos of URALi, built for Windows-based machines, are available and aim to demonstrate some key features of the library: the procedural generation of audio, the seamless integration within the Unity audio ecosystem and the broader environment, and the possibility to control real-time the parameters of the synthesis algorithm. Each demo is a standalone application and showcases one or more of these features. Specifically, each of them employs a different audio generator unit (e.g. waveshaper, noise, FM synthesizer) and the movement of the mouse on both X and Y axes to control relevant parameters. Furthermore, some employ stock Unity audio effects (e.g. a low-pass filter, an echo). The code of these demos was written by linking the library to the Unity project as a .dll file and follows the structure mentioned earlier, as it is composed of the implementation of the synthesis function, the instructions to set up the audio engine, and the code to control the particle system in the middle—which has a merely aesthetic role although shares the data of the mouse position with the audio chain to loosely determine the direction of the particles. The demos will be made open source at release time as example projects; currently, however, the executable files can be sent upon request to the first author, and an audiovisual recording showcasing the functioning of some of them is available as well[15].

5. ACKNOWLEDGMENTS

The first author acknowledges the University of Portsmouth - Faculty of Creative and Cultural Industries for the PhD Scholarship supporting this research.

6. REFERENCES

[1] J. Jerald, P. Giokaris, D. Woodall, A. Hartholt, A. Chandak, and S. Kuntz, “Developing virtual reality applications with unity,” in *2014 IEEE Virtual Reality (VR)*, 2014, pp. 1–3.

[2] S. Wang, Z. Mao, C. Zeng, H. Gong, S. Li, and B. Chen, “A new method of virtual reality based on unity3d,” in *2010 18th International Conference on Geoinformatics*, 2010, pp. 1–5.

[3] S. Patil, G. Gaikwad, S. Hiran, A. Ikhar, and H. Jadhav, “metaar – ar/xr shopping app using unity,” in *2023 International Conference for Advancement in Technology (ICONAT)*, 2023, pp. 1–11.

[4] N. H. Khalifa, Q. V. Nguyen, S. Simoff, and D. Catchpoole, “A visualization system for analyzing biomedical and genomic data sets using unity3d platform,” in *Proc. 8th Australasian Workshop on Health Informatics and Knowledge Management*, 2015, pp. 47–53.

[5] F. Fontana, R. Paisa, R. Ranon, and S. Serafin, “Multisensory plucked instrument modeling in unity3d: From keytar to accurate string prototyping,” *Applied Sciences*, vol. 10, no. 4, pp. 1452, Feb 2020.

[6] R. Schroeter and M. A. Gerber, “A low-cost vr-based automated driving simulator for rapid automotive ui prototyping,” in *Adjunct Proceedings of the 10th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, New York, NY, USA, 2018, AutomotiveUI ’18, p. 248–251, Association for Computing Machinery.

[7] E. Dorigatti, “Automating art: A case-study of cellular automata in generative multimedia art,” in *Proc. Intl. Computer Music Conf.(ICMC)*, Limerick, Ireland, Jul. 3-9 2022, pp. 175–181.

[8] M. Wright, “Open sound control: an enabling technology for musical networking,” *Organised Sound*, vol. 10, no. 3, pp. 193–200, 2005.

[9] J. Atherton and G. Wang, “Chunity: Integrated audiovisual programming in unity.,” in *Proc. Intl. Conf. New Interfaces for Musical Expression (NIME)*, Genova, Italy, Jun. 5-7 2018, pp. 102–107.

[10] G. Wang and P. R. Cook, “Chuck: A concurrent, on-the-fly, audio programming language,” in *Proc. Intl. Computer Music Conf. (ICMC)*, Singapore, Sep. 29-Oct. 4 2003.

[11] E. Dorigatti, “Interacting with audio in unity [manuscript submitted for publication],” in *Proc. Conf. Dictionary for Multidisciplinary Music Integration (DiMMI)*, Trento, Italy, Nov. 25-26 2022.

[12] F. Corradini, G. Ferrari, and M. Pistore, “Eager, busy-waiting and lazy actions timed computation,” *Electronic Notes in Theoretical Computer Science*, vol. 7, pp. 96–114, 1997, EXPRESS’97.

[13] E. Dorigatti, “Urali: a proposal of approach to real-time audio synthesis in unity,” in *Proceedings of the 16th Sound & Music Computing Conference*, I. Barbancho, L. J. Tardón, A. Peinado, and A. M. Barbancho, Eds., Malaga, Spain, May 28–31 2019, pp. 86–87.

[14] C. Roads, “Introduction to granular synthesis,” *Computer Music Journal*, vol. 12, no. 2, pp. 11–13, 1988.

[15] J. M. Chowning, “The synthesis of complex audio spectra by means of frequency modulation,” *journal of the audio engineering society*, vol. 21, no. 7, pp. 526–534, september 1973.

[16] J. Schimmel, “Audible aliasing distortion in digital audio synthesis.,” *Radioengineering*, vol. 21, no. 1, 2012.

[17] V. Välimäki, J. Pekonen, and J. Nam, “Perceptually informed synthesis of bandlimited classical waveforms using integrated polynomial interpolation,” *The Journal of the Acoustical Society of America*, vol. 131, no. 1, pp. 974–986, 2012.

[14]https://docs.cycling74.com/max8/vignettes/max_alphabetical

[15]https://www.enricodorigatti.com/wp-content/uploads/2021/12/URALi.mp4