



Maynooth University

– Department of Music –

**Integrating Csound into Unreal Engine for
Enhanced Game Audio**

Thesis Submitted in Partial Fulfilment for the Degree of
Master of Science

by

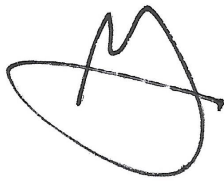
Albert Madrenys Planas

Student Number: 23251240

Supervisor : Victor Lazzarini

Declaration

I, Albert Madrenys Planas, declare that this thesis titled *Integrating Csound into Unreal Engine for Enhanced Game Audio* and the work presented within it are entirely my own and have been generated as the result of my original research. This work has not been submitted, in whole or in part, for consideration for any other degree or qualification at Maynooth University or any other institution. All references to the work of others or prior research have been appropriately acknowledged and cited within the text.

A handwritten signature in black ink, consisting of a large, stylized loop with a horizontal line crossing it, and a small 'M' shape above the loop.

Signed:

Date: 31/8/2024

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Victor Lazzarini, whose expert guidance, unwavering encouragement, and insightful support have been instrumental throughout this research. His ability to offer solutions and encouragement during challenging moments has been a source of great inspiration.

I am also profoundly thankful to my family and friends for their steadfast support and understanding while I pursued my studies abroad. Their encouragement has been a tremendous source of strength and motivation throughout this journey.

I wish to extend my appreciation to Maynooth University for providing the platform and resources that facilitated my research and academic growth.

A heartfelt thanks goes to the new people I have met during my year in Ireland. Despite the initial feeling of being alone, I have been warmly welcomed by a wonderful community. The friendships I have forged and the experiences I have gained are cherished memories that I will carry with me. This year has been filled with kindness and joy, and I will always remember it fondly.

Abstract

This thesis explores the development and integration of the MetaCsound plugin, designed to bridge the gap between *Csound*, a powerful sound synthesis language, and *Unreal Engine*'s MetaSounds system. The primary objective was to create a seamless interface that enables Csound's advanced audio processing capabilities within the MetaSounds environment, thereby enhancing the toolset available for interactive sound design in Unreal Engine.

The project began with a thorough review of the MetaSounds and Csound systems, focusing on the design and implementation of a plugin that facilitates the integration of these two platforms. Key components of the development process included the creation of a custom node architecture using C++ and Unreal Engine's API, the handling of Csound's files, and the implementation of the necessary features to support both audio and control signal processing.

Testing of the plugin was conducted through a series of MetaSound graphs and packaged game scenarios, demonstrating the plugin's robustness and reliability. Three test cases were used to evaluate different functionalities, including continuous audio effects, event-triggered sound changes, and stress testing of audio and control data handling. The results confirmed that the MetaCsound plugin performs effectively across various scenarios, maintaining its functionality and stability.

The discussion highlighted the achievement of the design goals, including novel integration, relevance, and ease of use, while also addressing challenges such as limited documentation, linker errors, and dynamic interface creation. Suggested improvements focus on expanding system compatibility, enhancing file management, and simplifying DLL deployment.

In conclusion, this thesis presents MetaCsound as a functional and promising tool for extending MetaSounds with Csound's capabilities. The findings indicate that the plugin meets its design objectives and offers a solid foundation for future development and enhancements.

Contents

1	Introduction	1
2	Literature Review	3
2.1	Sound Synthesis Environments	3
2.1.1	Csound	4
2.1.2	Pure Data	5
2.1.3	SuperCollider	7
2.2	Game Engines	7
2.2.1	Unity Engine	7
2.2.2	Unreal Engine	8
2.2.3	Godot Engine	8
2.3	Audio Middlewares	8
2.3.1	FMOD	9
2.3.2	Wwise	9
2.3.3	MetaSounds	10
2.4	Csound in game engines	10
2.4.1	FMOD or Wwise with Cabbage	10
2.4.2	CsoundUnity	11
2.4.3	Godot-csound	11
2.5	Summary	11
3	System Design and Implementation	12
3.1	Design Goals	12
3.2	Integrating Csound with MetaSounds	12
3.3	Understanding Csound for Development	13
3.3.1	Csound File Structure	13
3.3.2	Opcodes	14
3.3.3	Instruments	14
3.3.4	Signals	14
3.3.5	Loops and Variables	14
3.3.6	Buses	15
3.3.7	Audio Channels	16
3.3.8	Csound C++ API	16

3.4	Understanding MetaSounds for Development	17
3.4.1	MetaSounds Graph	17
3.4.2	Sample and Frame Difference	17
3.4.3	MetaSounds Pin Types	17
3.4.4	Communication between Blueprints and MetaSounds	18
3.4.5	Read and Write Data References	18
3.5	MetaCsound Node Design	19
3.5.1	Mapping MetaSounds Data References to Csound Variables	19
3.5.2	Node Vertex Interfaces	19
3.5.3	Similarities and Differences between Nodes	20
3.6	Integration Setup	21
3.6.1	Unreal Engine Architecture	21
3.6.2	Plugins	22
3.6.3	MetaCsound Plugin and Project	22
3.6.4	Plugin Modules and Dependencies	22
3.6.5	Dynamic Link Libraries	24
3.6.6	Unreal Build Tool	25
3.6.7	Importing and Linking Csound inside Unreal	25
3.6.8	Module Startup and Shutdown	26
3.6.9	Namespaces and API Macros	27
3.7	Text Localization	28
3.8	Curiously Recurring Template Pattern	28
3.8.1	CRTP in MetaCsound	29
3.8.2	Derived Operators	30
3.9	Factory Method	32
3.9.1	Smart Pointers	32
3.9.2	CsoundParams Namespace	32
3.9.3	Csound Operator Factory Method	34
3.10	Operator Constructor and Execute Method	36
3.11	Csound File Handling	36
3.11.1	File Directory	36
3.11.2	File Compilation	37
3.12	Audio Input and Output	37
3.12.1	Spout and Spin	38
3.12.2	Audio Output	38
3.12.3	Indexing the Spout	39
3.12.4	Audio Input	41
3.12.5	Indexing the Spin	41
3.13	Control Input and Output	43
3.14	Initializing Class Variables	45
3.15	Controlling Csound's Performance	47

3.16	Node States	47
3.16.1	Enumerating Operator States	47
3.16.2	FTrigger Data References	48
3.16.3	Play State	48
3.16.4	Stop State	49
3.16.5	On Finished Event	50
3.16.6	Clearing Channels	51
3.16.7	Error State	52
3.16.8	Node State Flux Diagram	53
3.17	Full Implementation of the Execute Method	54
3.18	Ensuring Node Visibility and Functionality in MetaSounds	56
3.18.1	GetNodeInfo Method	56
3.18.2	DeclareVertexInterface Method	57
3.18.3	BindInputs and BindOutputs Methods	58
3.18.4	Facade Class	59
3.18.5	Node Registration	60
3.19	Extending MetaCsound	60
3.20	Chapter Conclusions	61
4	Testing and Discussion	62
4.1	Examples	62
4.1.1	Wind Test	62
4.1.2	Bomb Test	65
4.1.3	Third Test	66
4.1.4	Testing with Packaged Game	68
4.1.5	Results of the Tests	68
4.2	Discussion	69
4.2.1	Evaluation of Final Results	69
4.2.2	Difficulties Encountered	70
4.2.3	Improvements	71
4.3	Chapter Conclusions	71
5	Conclusions	73
	Bibliography	74
A	Git Repository	77
B	Documentation	78
B.1	Installing	78
B.2	Using	78
B.3	Examples	79

B.4	Packaging your Game	80
C	License Information	81
C.1	Key Points of LGPL 2.1	81
C.2	Implications for this Project	81
C.3	Access to Code	81

1 Introduction

The gaming industry has experienced remarkable growth and innovation, continually pushing the boundaries of what is achievable in interactive entertainment. In recent years, there has been a heightened focus on enhancing audio experiences within games, particularly through realistic spatialization and adaptive audio. *Adaptive audio*, which refers to sound or music that dynamically adjusts in response to in-game events, is increasingly integral to creating immersive and engaging experiences.

Game engines, essential tools in game development, provide a comprehensive suite of functionalities including rendering, physics, sound, and graphical user interfaces, all optimized for real-time performance. While many engines come with built-in audio capabilities, developers often turn to specialized *middleware* solutions such as FMOD or Wwise for advanced audio features [15, 3]. These middleware options can be integrated into projects to offer enhanced audio functionality beyond the standard engine tools.

Csound, a powerful and flexible sound and music computing system, offers a wide range of features that are particularly suited for adaptive audio and music [18, 10]. While Csound has seen successful integration with modern game engines such as Unity and Godot, its integration with *Unreal Engine* remains limited and complex [26, 19]. This thesis focuses on the development of a tool designed to integrate Csound with Unreal Engine, a leading game engine known for its versatility and real-time capabilities. The primary objective of this work is to bridge the gap between Csound’s advanced audio features and Unreal Engine’s robust game development environment.

The design and implementation of this integration tool aimed to achieve several key goals:

- **Novel Integration:** Facilitate communication between Csound and Unreal Engine, two previously incompatible systems.
- **Relevance:** Enhance Unreal Engine’s audio capabilities with Csound’s advanced tools, providing a new set of possibilities for interactive audio experiences.
- **Ease of Use:** Ensure that the tool is intuitive and user-friendly, aligning with the logic of both Csound and Unreal Engine while simplifying the integration process.

The development process involved creating a *MetaSounds* plugin for Unreal Engine, which allows users to leverage Csound’s features directly within the engine [12, 11, 17]. This plugin was rigorously tested to ensure its reliability and effectiveness in various scenarios.

The results demonstrate that the plugin successfully integrates Csound with MetaSounds, providing a valuable tool for developers seeking to create dynamic and adaptive audio experiences in their games.

In summary, this work contributes to the field of game audio by enabling the use of Csound within Unreal Engine, offering a new dimension of flexibility and power for audio design in interactive environments. The integration of Csound into Unreal Engine not only enhances the capabilities of both systems but also opens up new possibilities for future developments in adaptive audio.

2 Literature Review

The development of *sound synthesis engines* has been pivotal in shaping the interactive and immersive experiences provided by modern *game engines*. With the increasing complexity of virtual environments, the demand for robust and flexible audio solutions has grown significantly. This chapter reviews the key literature and existing technologies that inform the integration of sound synthesis tools into game development environments.

The literature review begins by exploring the foundations of sound synthesis environments. These systems provide the basis for understanding how advanced sound synthesis can be achieved within a *digital signal processing* (DSP) context, which is essential for the integration efforts undertaken in this thesis.

Next, we examine game engines such as *Unreal Engine*, which has become a leading platform for both game development and real-time interactive simulations. The review highlights the role of sound engines and *middlewares*, such as *FMOD* and *Wwise*, in enhancing the audio capabilities of these engines, and discusses how these tools facilitate complex sound design and implementation within interactive environments.

Furthermore, the review addresses the methodologies and technologies for implementing and facilitating communication between sound synthesis environments and game engines. This includes an exploration of *Open Sound Control* (OSC) messages, as well as various tools and libraries such as *Cabbage*, *Csound API*, *libpd*, and *CsoundUnity*, which provide essential interfaces for integrating audio processing tools into game development workflows.

This chapter brings together the existing research and technologies related to integrating sound engines into game development. It aims to give a clear understanding of how these systems currently work, which sets the stage for the new contributions made in this thesis.

2.1 Sound Synthesis Environments

Sound synthesis environments, such as *Csound*, *SuperCollider*, and *Pure Data*, have played a crucial role in the development of *digital audio processing* and sound design. These platforms offer powerful tools for creating, manipulating, and controlling sound in a highly flexible and programmable way, making them essential for both artistic and technical applications. They continue to be relevant in modern audio design, especially in contexts where custom, real-time audio solutions are required.

2.1.1 Csound

Csound is a specialized programming language and rendering system designed for audio processing, developed in *C* [9, 10, 18]. Originating from the *MUSIC-N* series created by Max Mathews, Csound was first released by Barry Vercoe in 1986 and has been integral to the computer music field since then. Currently, Csound supports a wide range of platforms, including desktop, mobile, embedded, server, and web environments, and is used globally for music software and composition.

Csound is open source and designed to be accessible to those with limited programming experience, making it straightforward to prototype tools and create sound designs. It benefits from active development by a large community.

2.1.1.1 Csound Application Program Interface (API)

Csound allows external applications to use it as a software synthesis engine through the *Csound Application Program Interface* (API) [7]. This capability enables Csound to be embedded in other programs as a library, providing access to all functionalities offered by the API.

The primary API is written in *C*, with a *C++* wrapper available that interfaces with the C API. Additionally, wrappers for other languages, such as Python, Java, and Lua, are also provided.

2.1.1.2 Csound Frontends

Csound, being a large library, requires frontends to interact with it via the API [10]. A frontend is the interface or application that allows users to interact with a system or software program. It serves as a bridge between the user and the underlying functionality or data of the system.

Csound Console Command The `csound` command is a basic frontend for Csound that allows users to generate sound output from Csound files with the `.csd` extension. It operates from the command-line interface.

Csound Web-IDE Csound Web-IDE provides an Integrated Development Environment (IDE) for Csound via a web interface [8]. It includes useful tools such as a MIDI keyboard and a spectrogram. The Web-IDE also offers online features for sharing and discovering work, making it accessible and easy to use.

CsoundQt CsoundQt is a cross-platform frontend for Csound that includes a code editor with syntax highlighting and autocomplete, interactive widgets, and integrated help [5]. It

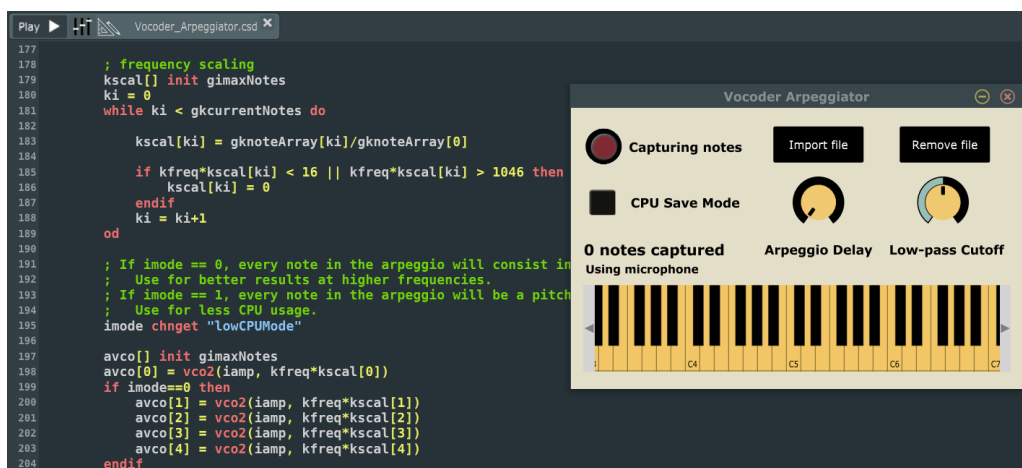


Figure 2.1: Example of a Cabbage project with Csound code and widgets.

provides a comprehensive development environment for Csound, featuring widgets for creating *graphical user interfaces* (GUIs) and visualizing real-time performance or controlling the performance itself.

2.1.1.3 Cabbage

Cabbage is a frontend environment for Csound that simplifies the creation of *graphical user interfaces* (GUIs) [27]. It provides a range of widgets, such as sliders and buttons, which allow users to control Csound instruments and effects (see Fig. 2.1). Cabbage includes numerous examples, making it accessible to users at all experience levels. Additionally, it supports exporting creations as standalone applications, *Unity* native plugins, *VST* plugins, and other formats, making it a versatile tool for both development and performance.

A *Unity* plugin is an extension for the *Unity* game engine, which will be explained in Section 2.2.1. A *Virtual Studio Technology* (VST) plugin is a software interface that integrates virtual synthesizers and audio effects into *digital audio workstations* (DAWs), emulating traditional studio hardware and enabling users to enhance their music production setups within a DAW.

Cabbage’s ability to export to various formats highlights Csound’s versatility, as it can be utilized across different platforms. This capability is particularly relevant to this thesis, which aims to develop a similar tool to integrate a sound engine with a game engine.

2.1.2 Pure Data

Pure Data (Pd) is an open-source visual programming language designed for creating interactive computer music and multimedia works [22]. Unlike textual programming envi-

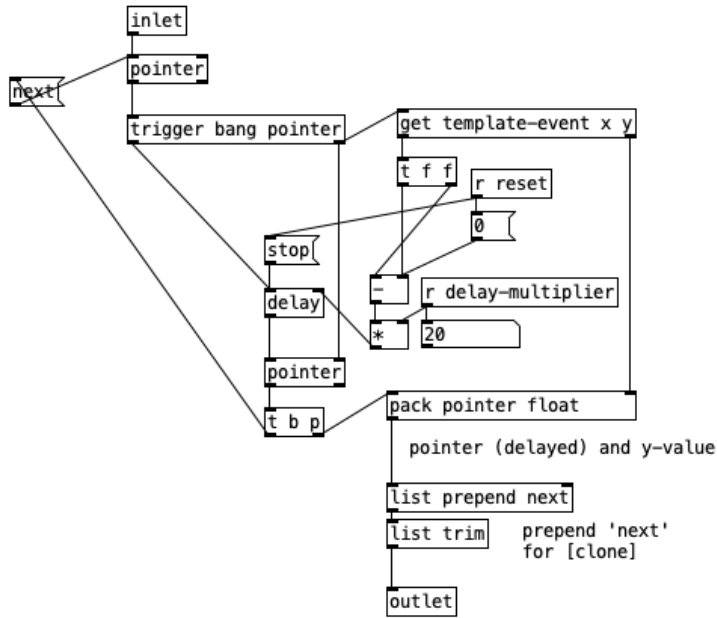


Figure 2.2: Example of a Pure Data patch.

ronments like Csound, Pure Data uses a graphical interface where users connect modular elements, known as "objects" or "nodes," to build audio and visual processes. This node-based system makes it accessible for users to design complex sound synthesis, audio processing, and multimedia applications without traditional coding. Figure. 2.2 shows an example of a patch, a composition created in Pd's visual interface.

2.1.2.1 Libpd

Libpd is a library that allows developers to embed Pure Data into other software environments, enabling the integration of Pd's powerful audio processing capabilities into various applications [4]. By using *libpd*, developers can run Pure Data patches within their software, such as games, mobile apps, or other interactive systems.

UnityPd, for instance, utilizes *libpd* to integrate Pure Data into the *Unity Engine* (see Section 2.2.1). With *UnityPd*, developers can embed Pure Data patches directly into Unity projects, facilitating dynamic and interactive sound design that responds to in-game events [21].

2.1.3 SuperCollider

SuperCollider is an open-source platform for audio synthesis, algorithmic composition, and real-time audio processing [24]. It consists of a programming language (*sclang*) and a real-time audio server (*scsynth*), which work together to enable users to create complex sound synthesis algorithms and process audio in real time.

SuperCollider excels in real-time performance and interaction, making it ideal for live coding and improvisation. Its environment allows for dynamic code changes while the program is running, which is particularly useful in live performances and experimental music. SuperCollider’s language is object-oriented, offering a flexible and modern approach to coding compared to older languages like Csound. This object-oriented design can be more intuitive for users familiar with such programming paradigms and allows for more modular and reusable code structures.

2.1.3.1 OSC Integration in SuperCollider

SuperCollider uses *Open Sound Control* (OSC) as a primary means of communication between its components (the language *sclang* and the sound server *scsynth*) and with external software. OSC is a protocol for networking sound synthesizers, computers, and other multimedia devices, allowing SuperCollider to interact with a wide range of other software, including DAWs, visual programming environments like *Max/MSP*, and other audio applications [6].

2.2 Game Engines

A game engine is a software framework designed to simplify the development of video games. It provides essential tools and features like rendering graphics, handling physics, managing game logic, and controlling audio. Game engines facilitate the creation of games across various platforms by offering scripting, animation and AI systems, allowing developers to focus on creating content rather than building these components from scratch.

2.2.1 Unity Engine

Unity Engine is a versatile and widely-used game engine renowned for its user-friendly interface and extensive support for 2D, 3D, augmented reality (AR), and virtual reality (VR) game development [25]. It features a powerful rendering engine, a robust asset management system, and the *C#* scripting language, which integrates with its visual development environment. Unity’s strengths lie in its cross-platform capabilities, allowing developers to build games for a wide array of devices including consoles, PCs, and mobile platforms from a single codebase. Its large community, extensive documentation, and asset store further enhance its accessibility and functionality.

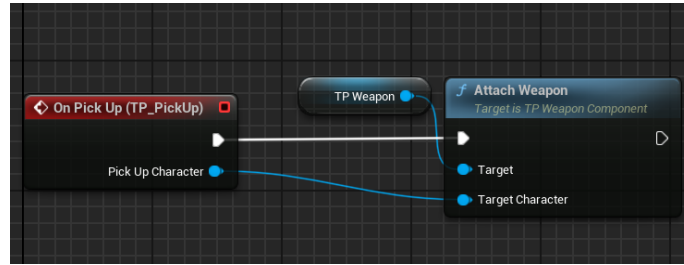


Figure 2.3: Example of a simple Blueprint in Unreal Engine.

2.2.2 Unreal Engine

Unreal Engine, developed by Epic Games, is acclaimed for its high-fidelity graphics and advanced toolset, making it a preferred choice for high-quality games and complex simulations [14]. It offers a sophisticated rendering engine with capabilities for photorealistic visuals, a built-in physics engine, and the *Blueprint* visual scripting system, which enables developers to create game logic without extensive coding (see Fig. 2.3). Unreal Engine also supports *C++* programming for deeper customization and performance optimization. It is particularly noted for its robust tools for 3D environments and real-time applications, including film production and architectural visualization.

2.2.3 Godot Engine

Godot Engine is an open-source game engine recognized for its flexibility and ease of use in both 2D and 3D game development [16]. It features a unique scene system that simplifies game design and organization, along with its own scripting language, *GDScript*, which is tailored for rapid development. Godot also supports *C#*, *VisualScript*, and *C++* for scripting, providing versatility in coding preferences. Its lightweight nature and efficient performance make it suitable for indie projects and smaller-scale games. The engine's open-source model encourages community-driven development and customization.

2.3 Audio Middlewares

Audio middleware refers to specialized software tools designed to enhance and manage audio within video games and interactive applications. While game engines provide basic audio functionalities, middleware solutions offer advanced capabilities that address the complex needs of modern audio design. Middleware typically includes features such as dynamic sound effects, real-time mixing, spatial audio, and interactive sound systems. These tools enable developers to create immersive and responsive audio experiences that might be difficult or inefficient to achieve with the base engine alone.



Figure 2.4: FMOD graphical user interface.

2.3.1 FMOD

FMOD is a widely-used audio middleware known for its flexibility and comprehensive feature set [15]. It supports advanced audio functionalities such as real-time mixing, spatial audio, and complex sound event management. FMOD integrates with various game engines and supports *VST* plugins, allowing developers to use external audio effects and instruments within the FMOD environment. Notably, FMOD is compatible with Unreal Engine, Unity, and Godot, making it a versatile choice for different development platforms. Its user-friendly interface and extensive documentation make it popular for both large and small-scale projects. Figure. 2.4 shows an example of an FMOD project.

2.3.2 Wwise

Wwise, developed by Audiokinetic, is another prominent audio middleware offering a robust suite of tools for interactive sound design [3]. It provides advanced features such as interactive music systems, dynamic audio mixing, and spatial audio. Wwise supports *VST* plugins, enabling the integration of external audio effects and processing tools. It is compatible with Unreal Engine and Unity, allowing seamless integration with these engines. However, Wwise does not natively support Godot, though there are community-driven solutions that can facilitate integration. Wwise is known for its detailed audio authoring tools and its ability to manage complex audio requirements across multiple platforms.

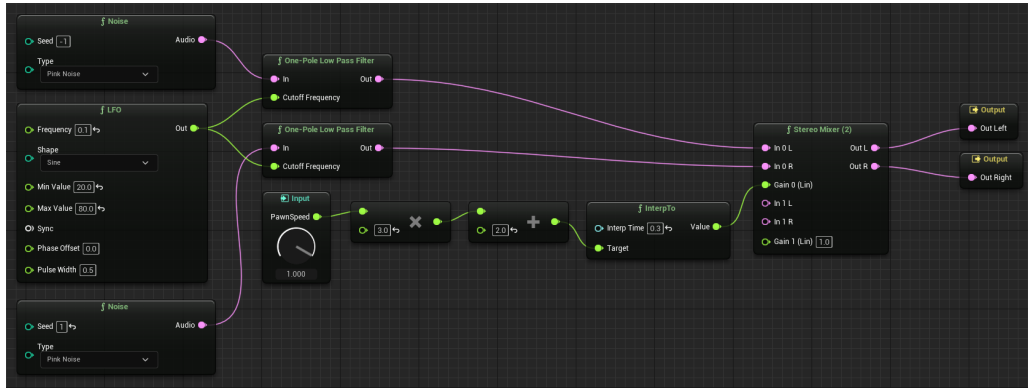


Figure 2.5: Example of a MetaSounds graph that simulates a wind effect.

2.3.3 MetaSounds

Metasounds is a relatively new audio middleware developed by Epic Games specifically for use within the Unreal Engine [12]. It features a modern, node-based interface for creating complex audio systems and effects, optimized for Unreal Engine (see Fig. 2.5). Although Metasounds is designed to be highly integrated with Unreal Engine, it is still evolving and lacks some advanced functionalities found in more established middleware solutions like FMOD and Wwise. As a newer environment, Metasounds shows promising potential but currently has limitations in features and community support.

Metasounds does not support *VST* plugins directly, which limits its ability to integrate with external systems like Cabbage that rely on *VST* functionality. Despite this, its tight integration with Unreal Engine ensures that it is highly optimized for the engine's environment, providing seamless and efficient audio management within Unreal projects.

2.4 Csound in game engines

Integrating Csound into game engines allows developers to leverage its powerful sound synthesis and processing capabilities within interactive applications. This integration can be achieved through various methods, including using *VST* plugins, or directly embedding via the Csound API.

2.4.1 FMOD or Wwise with Cabbage

FMOD and Wwise can both utilize *VST* plugins to integrate Csound's audio processing capabilities. By exporting Csound instruments or effects as *VST* plugins using Cabbage, developers can incorporate these *VSTs* into FMOD or Wwise (see Section 2.1.1.3). While this approach offers powerful capabilities, the integration process can be complex due to

the multiple tools involved. Properly configuring the pipeline to ensure seamless operation may require careful setup and troubleshooting.

2.4.2 CsoundUnity

CsoundUnity is a specialized integration tool that embeds Csound directly into Unity, providing a seamless way to use Csound’s sound synthesis and processing capabilities within the Unity Engine [26]. It allows developers to access Csound’s powerful audio features through Unity’s interface, enabling the creation of complex soundscapes and interactive audio experiences.

2.4.3 Godot-csound

Godot-csound is a relatively new integration that embeds Csound into the Godot game engine [19]. Similar to CsoundUnity, it allows developers to use Csound’s audio processing features within Godot’s development environment, providing a native solution for incorporating Csound into Godot projects.

2.5 Summary

This literature review has provided an overview of the foundational sound synthesis environments, such as Csound, SuperCollider and Pure Data, and examined their roles within the broader context of game development. Additionally, the review explored the capabilities of leading game engines, including Unreal Engine, and the integration of audio middlewares like FMOD and Wwise. The discussion also covered various tools and technologies that enable communication between sound synthesis environments and game engines, highlighting the existing solutions and their limitations.

The insights gained from this review underline the importance of flexible and powerful sound synthesis tools within interactive media. They also reveal opportunities for further innovation, particularly in integrating advanced audio processing systems into real-time environments like Unreal Engine.

In the following chapter, we will build upon this foundation by presenting a new approach to enhancing the sound design capabilities within Unreal Engine. This approach leverages the strengths of Csound and MetaSounds, aiming to address some of the limitations identified in the current literature.

3 System Design and Implementation

This chapter explores the design and implementation of a new tool for integrating sound synthesis environments into a game engine. Building on the foundational concepts and technologies discussed in previous chapters, we will detail the architecture, design, implementation, and key features of the integration tool.

3.1 Design Goals

The design and implementation of this tool are guided by the following goals:

- **Novel Integration:** The tool must enable communication between two systems that were previously incompatible.
- **Relevance:** The integration should provide meaningful benefits to both systems. If the systems share the same strengths and weaknesses, the integration may lack significance.
- **Ease of Use:** The tool should be designed to be intuitive and user-friendly. It should align with the logic of both systems, which may require omitting certain features from each environment if they do not contribute to a smooth integration.

3.2 Integrating Csound with MetaSounds

Currently, Csound can natively work with major game engines except Unreal. Unity has CsoundUnity, and Godot has godot-csound, but Unreal lacks an equivalent integration. To use Csound in Unreal, developers typically create a Cabbage VST plugin and import it into a middleware, which can be cumbersome.

MetaSounds, a middleware exclusive to Unreal and closely integrated with it, is a relatively new system. Although it has strong foundations and a promising future, it still lacks many features available in more established environments.

MetaSounds operates within a node-based interface. Developing a new family of nodes that internally communicate with the Csound API to execute a `.csd` file could be both feasible and beneficial. MetaSounds allows developers to create custom nodes using C++, enabling direct communication between these nodes and the Csound C++ API. Additionally, both MetaSounds and Csound have input and output channels for exchanging audio and other types of data between external environments.

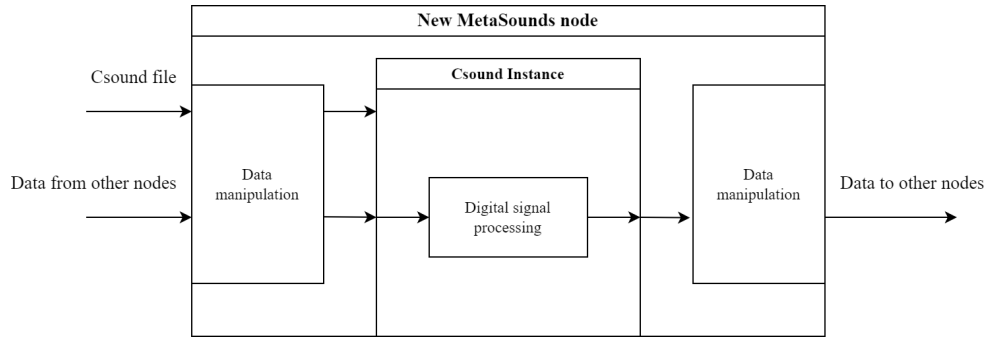


Figure 3.1: Design and data flow inside the new node.

This approach is promising and represents original work, as there is currently no direct communication between MetaSounds and Csound. Both environments would benefit from this integration: MetaSounds and Unreal would gain access to Csound’s extensive toolset, while Csound would have a direct link to one of the most widely used and impactful game engines today. Furthermore, a new set of nodes that interact with other MetaSounds nodes while internally running Csound offers an elegant and intuitive design.

Figure 3.1 illustrates a diagram showing how data from other nodes enters the new node, is manipulated and sent to Csound, which performs DSP operations. The processed data is then outputted from Csound, manipulated, and sent out of the node to be received by other nodes.

The tool has been named *MetaCsound* to reflect its role as a bridge between the two environments. The name is a play on words, combining *MetaSounds* and *Csound* to highlight the integration of these systems.

3.3 Understanding Csound for Development

Before fully exploring the C++ implementation, it is essential to gain a detailed understanding of how Csound operates. This section will focus on the aspects of the Csound environment most pertinent to this project [9, 10, 18].

3.3.1 Csound File Structure

Csound code is organized in files with the `.csd` extension. These are plain text files that use *XML* tags to structure the code. The primary sections are enclosed within the `<CsoundSynthesizer>` tags, and any content outside these tags is ignored.

- `<CsOptions>`: Options or flags that control Csound’s behavior, such as input and output devices and system parameters.

-
- **<CsInstruments>**: Csound code for DSP operations, including *instruments* and *opcodes*.
 - **<CsScore>**: Numeric score section. This section contains statements to trigger the instruments defined in the **<CsInstruments>** section. While this section is optional, as statements can also be received from other sources.

3.3.2 Opcodes

A typical Csound statement involves an *opcode* or operator, with inputs and outputs specified. Output variables are generally placed on the left side, while inputs are on the right.

```
output1, output2 opcode input1, input2, input3
```

3.3.3 Instruments

In Csound, the basic unit of code is the *instrument* block, where digital signal processing is defined. Without instruments, no signal processing occurs.

Instrument blocks are initiated with the **instr** keyword and terminated with the **endin** keyword. The **schedule** opcode is used to schedule or instantiate an instrument, taking arguments such as the instrument number, start time, duration, and any additional parameters required by the instrument.

Within an instrument block, signal processing operations are defined using variables and opcodes.

3.3.4 Signals

In sound programming, a signal is typically represented as an array of samples, with each sample being a numerical value that describes the sound wave at a specific point in time. These values are usually floating-point numbers within the range of $[-1, +1]$. Values outside this range can cause distortion.

In Csound, while it is possible to modify this range, the standard practice is to confine the maximum signal level to 1 to maintain this range. This is controlled by setting the environment variable **Odbfs** to 1.

3.3.5 Loops and Variables

In programming, a variable is a named storage location for holding data. Csound employs specific variable types depending on the nature of the data they manage. For the purposes of this thesis, we will focus on two key types: *audio-rate* and *control-rate* signals.

To differentiate between these types, it is essential to understand Csound's processing loop. Similar to other signal processing environments, Csound operates with a loop that executes periodically. In programming terms, this translates to a function or method called

at regular intervals to process chunks of audio samples, or buffers. Calling this process method for every individual sample would introduce significant overhead and inefficiency. Instead, the process is optimized by handling larger chunks of samples at once, as well as by computing variables that remain constant across samples only once per chunk.

In Csound, the buffer size is controlled by the **ksmps** variable. This setting determines that audio-rate variables will be represented as arrays or buffers containing **ksmps** samples. In contrast, control-rate variables have a single value for each control period, meaning they change only every **ksmps** samples. While audio-rate variables provide a detailed representation of the signal with frequent updates, control-rate variables are used for parameters that change less frequently, such as gain, frequency modulation, or filter resonance. These parameters do not require per-sample calculations, as rapid changes are imperceptible to human hearing. However, a larger **ksmps** value improves CPU efficiency but may affect audio quality.

Variable types in Csound are indicated by the first character of their name: for instance, **avar** denotes an audio-rate signal, while **kvar** indicates a control-rate signal.

```
instr 1
  kpitch expon 220, 5, 440
  // control-rate variable modifying pitch to an audio-rate variable
  asig vco2 0.6, kpitch
  outs asig, asig
endin
```

3.3.6 Buses

Buses are a fundamental mechanism for communication between Csound and other environments. They can be defined using the following opcodes:

```
chn_k Sname, imode
chn_a Sname, imode
```

Each bus is identified by a unique string (**Sname**). The opcodes specify the type of bus: audio-rate or control-rate. The **imode** parameter determines if the channel is used for reading, writing, or both. This mode serves as a hint for external tools interacting with Csound.

Within an instrument, the following opcodes can be used to read from and write to buses:

```
kval chnget Sname
aval chnget Sname

chnset kval, Sname
chnset aval, Sname
```

These opcodes correspond to different variable types. The **Sname** parameter identifies the channel by its string name. In the read opcodes (**chnget**), the result is stored in the output

variable, while in the write opcodes (`chnset`), the first input is the value to be written to the channel.

It is worth noting that while these features are termed *buses* in Csound, the opcodes use the term *channels*. This terminology can be confusing, but we will use *bus* to distinguish these from the audio channels discussed in the next section. In the Csound API, functions for interacting with buses will also use the term *channel*.

3.3.7 Audio Channels

While audio-rate buses can be used for input and output in Csound, audio channels are the more commonly used method. Audio channels handle the stream of audio-rate data during each control-rate (or k-rate) call and are similar to channels found in audio files (e.g., mono, stereo, quad).

The number of input and output audio channels can be configured using the following Csound environment variables:

```
nchnls_i = iarg
nchnls = iarg
```

Here, `nchnls_i` defines the number of input audio channels, and `nchnls` defines the number of output channels. The `nchnls_i` variable is optional; if not set, it defaults to the value of `nchnls`.

Audio channels can be read from and written to using the following opcodes:

```
ar1 in
aarray in
ar1, ar2 ins
ins
out asig1[, asig2,...]
outs asig1, asig2
```

These opcodes manage the audio data flow in Csound. The `in` and `ins` opcodes are used for reading audio input, while the `out` and `outs` opcodes handle audio output.

3.3.8 Csound C++ API

The Csound C++ API centers around the `Csound` class, which is defined in the `csound.hpp` header file. This class represents an instance of Csound and provides a range of functionalities including compiling scores, managing data with audio channels and buses, sending events, and controlling performance playback. By creating an instance of the `Csound` class, developers can interact programmatically with Csound, leveraging its extensive capabilities for audio processing and performance management.

3.4 Understanding MetaSounds for Development

Before delving into the original work, it is crucial to thoroughly understand how MetaSounds operates. Gaining this insight will enable us to better comprehend the environment and make informed decisions to ensure an effective design [12, 17].

3.4.1 MetaSounds Graph

In a typical MetaSounds graph, input nodes are positioned on the left side. These inputs can be graph variables written by other *Blueprints*, which facilitates mapping them to in-game events such as player movements, new enemy appearances, or passing vehicles. The central part of the graph is reserved for DSP (Digital Signal Processing) nodes, which are interconnected with each other, the graph inputs, and variables. Outputs are located on the right side of the graph. Generally, a single audio output channel is used for mono audio, while two output channels are used for stereo audio. An example of a MetaSounds graph is shown in Figure 2.5.

3.4.2 Sample and Frame Difference

A *sample* refers to a single value from one audio channel at a specific point in time. In contrast, a *frame* represents the complete set of samples across all channels at that same moment. For instance, in a stereo setup, each frame consists of two samples: one from the left channel and one from the right channel. Thus, while a sample pertains to an individual channel, a frame encompasses all samples from all channels, providing a snapshot of the entire multichannel audio data at a given time.

3.4.3 MetaSounds Pin Types

Each MetaSounds node features its own *vertex interface*, which includes various input and output *node parameters*. These node parameters come in different *pin types* and can only be connected to pins of the same type. The pin types most relevant for this project are:

Audio Buffer: Represents an audio buffer. When executing the node, they will appear as a vector of samples. Float: Represents a floating-point number. When executing the node, they will represent a single value float. String: Represents a string. Trigger: Sample-rate accurate trigger. This means the signal will be able to distinguish the exact frame on where the trigger has been triggered, in relation to audio buffers.

- **Audio Buffer:** Represents an audio buffer as a vector of samples. During execution, it handles a chunk of audio data.
- **Float:** Represents a floating-point number. During execution, it deals with single value floats.

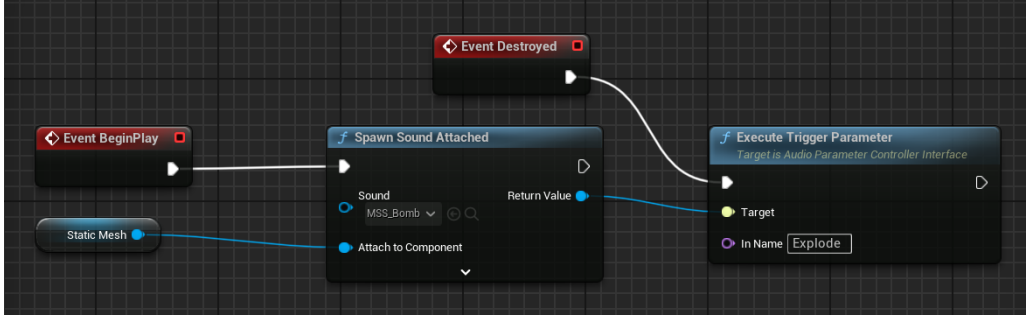


Figure 3.2: Blueprint interacting with a MetaSounds graph.

- **String:** Represents a text string.
- **Trigger:** A sample-rate accurate trigger that can identify the exact frame where the trigger occurs in relation to the audio buffer.

In the MetaSounds graph, each pin type is color-coded for clarity. Additionally, the C++ API allows developers to define custom pin types as needed. Figure 2.5 illustrates how nodes with various vertex interfaces are interconnected through node parameters of matching pin types.

3.4.4 Communication between Blueprints and MetaSounds

As discussed previously, graph inputs and variables can be modified by external Blueprints. This is achieved by accessing the running MetaSounds graph and using specific Blueprint nodes to alter the value of a named variable. Various Blueprint nodes are available, each tailored for different variable and input types. Figure 3.2 illustrates how a Blueprint triggers the MetaSounds graph input named *Explode*.

3.4.5 Read and Write Data References

All of the pin types previously mentioned are translated into data references when programming in C++. A data reference is a template type and can serve as either an input or output. For instance, `FTriggerReadReference` is a type representing a data read reference for the `FTrigger` type, which is equivalent to using the template specification `TDataReadReference<FTrigger>`.

In addition to the type of data these references contain, they can be categorized as read or write. Read references are input parameters intended to be consulted by the node, while write references are output parameters, which the node uses to populate with output data.

```
// This two variables types are equivalent
FTriggerReadRef PlayTrigger;
```

```
TDataReadReference<FTrigger> StopTrigger;

// This represents a string input parameter
FStringReadRef CsoundFile;

// This represents an audio buffer output parameter
FAudioBufferWriteRef AudioOut;

// Both variables contain floats, but one is for reading the input and
// the other one is for writing the output.
FFloatReadRef ControlIn;
FFloatWriteRef ControlOut;
```

3.5 MetaCsound Node Design

With a solid understanding of both Csound and MetaSounds, we can now proceed to design a new set of MetaSounds nodes that will run a Csound instance internally. This section will focus on combining these environments, using their strengths to create a smooth connection between MetaSounds' node-based interface and Csound's audio processing functions.

3.5.1 Mapping MetaSounds Data References to Csound Variables

In MetaSounds, an audio buffer data references in node execution consists of a chunk of audio samples, an audio buffer, or an audio array. These are analogous to Csound's audio-rate (a-rate) variables. On the other hand, float data references, which contain a single value per execution call, correspond to Csound's control-rate (k-rate) variables.

The new nodes will bridge the two frameworks (Csound and MetaSounds) by converting audio buffer data references into Csound a-rate variables, and float data references into Csound k-rate variables. After Csound processes these variables, the data will be converted back: a-rate variables to audio buffer data references and k-rate variables to float data references.

The size of a MetaSounds audio buffer is determined by the `BlockRate` and `NumFramesPerBlock` parameters in the `FOperatorSettings` class, which will be discussed in detail later in the Section 3.9.3. This value must be consistent across all audio buffer references in a MetaSounds graph and corresponds to the `ksmps` environment variable in Csound.

3.5.2 Node Vertex Interfaces

Because we are designing a new family of nodes, we will create multiple vertex interfaces with various parameters. Despite these variations, the interfaces between the nodes will remain highly consistent.

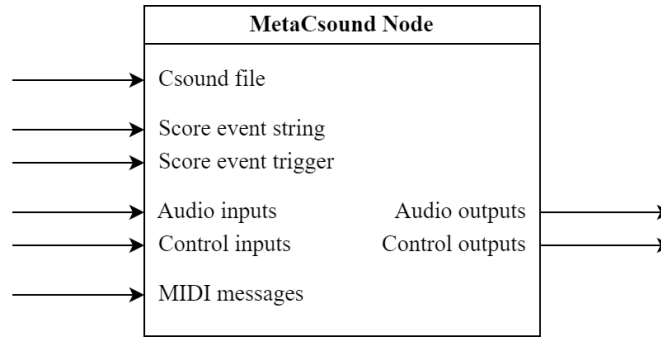


Figure 3.3: Layout of the new Csound node’s vertex interface.

The primary pins will be the *audio buffer* pins and *float* pins. The input pins will carry data from other nodes or variables to be processed by Csound. The output pins will deliver data processed by Csound for use by other nodes. Their equivalents in Csound have been explained in the previous section.

In addition to these, several other pins will be necessary. One pin will specify the name of the *Csound file* to be executed by the node. *Play* and *stop* pins will control the execution: the play trigger input pin will start the compilation and execution of the Csound file, while the stop trigger will halt the performance. The play input can be triggered multiple times to restart the score. A *finished* trigger output pin will be included to signal when the node has completed its performance, either because the Csound score has ended or the stop input has been activated.

This setup ensures that, although the node can begin executing, the *.csd* file will only start playing once the play trigger is activated. This design, with play, stop, and finished triggers, is reminiscent of other MetaSounds built-in nodes, such as the WavePlayer. Figure 3.3 illustrates the layout of the new nodes.

This design layout also includes *MIDI messages*, as they offer another potential method of communication between the two environments, given that both Csound and MetaSounds support MIDI.

3.5.3 Similarities and Differences between Nodes

Each node in this new family will include the Csound file reference, play and stop input triggers, and an output finished trigger. All nodes will have 8 float inputs and 4 float outputs. This design choice accommodates the need for multiple float inputs for complex Csound operations, while the output ports are fewer, reflecting the typical use case where the primary purpose of these nodes is to offload complex DSP operations to Csound rather than performing them within MetaSounds.

The only variation among the nodes will be in the number of audio buffers they handle.

There will be three types of nodes to accommodate different audio configurations: one for *stereo*, one for *quadraphonic*, and one for *octophonic* audio. Consequently, these nodes will support 2, 4, and 8 input and output audio buffers, respectively.

3.6 Integration Setup

With the design of the new nodes clarified, we can now explore the programming aspect within Unreal Engine. We will begin by establishing the structure to house the new tool. Unreal Engine supports two primary methods for game development: *Blueprints* and *C++ programming* [13, 17].

3.6.1 Unreal Engine Architecture

Blueprints, as discussed earlier, is a visual scripting feature in Unreal Engine that allows developers to create game logic using a node-based interface. Nodes with basic functionalities can be linked together to build more complex features. An example of a Blueprint is shown in Figure 2.3.

Blueprints offer a user-friendly approach, making game development accessible to those without extensive programming experience. However, as game logic becomes more complex, Blueprints can become cumbersome and difficult to manage. To address this, developers often implement complex operations in custom nodes programmed in C++.

Unreal Engine is fundamentally built in C++, and its source code is publicly available in a Git repository. While it is possible to modify the engine's source code, it is generally recommended to extend the engine by creating custom Blueprint nodes within the developer's own game modules rather than altering the engine itself.

Unreal Engine uses a modular architecture, where functionalities are organized into *modules*. These modules are essential for both game and plugin development. When developing a game or plugin, developers create new modules that are compiled as if they were part of the engine. This modular approach helps manage the engine's functionality and keeps the codebase organized.

Building a packaged game versus building the editor and project involves compiling different *targets*. Some modules may contain editor-specific code, but the core functionality remains consistent. To streamline development, Unreal Engine supports live coding, which allows developers to modify project-specific modules without restarting the editor. This feature enhances productivity by enabling real-time code changes during development.

A module in Unreal consists of predefined elements, primarily the `build.cs` file, written in C#, and a C++ class that implements `IModuleInterface`. We will explore this further in Sections 3.6.6 and 3.6.8.

3.6.2 Plugins

A plugin in Unreal Engine is a package that can contain one or more modules, as well as other resources such as assets, configuration files, and data. Plugins offer an efficient way to distribute and reuse code and assets, enabling the extension of the engine's functionality with new features or tools.

Plugins are defined by a `.uplugin` file, which is a plain text file using JSON format. This file specifies the modules included in the plugin, their dependencies, and other relevant metadata, allowing the engine to manage and integrate the plugin properly.

Similarly, the `.uproject` file is used to contain data related to the project (or game) rather than a plugin.

3.6.3 MetaCsound Plugin and Project

To ensure portability and facilitate easy integration with other projects, the developed tool will be packaged as its own plugin. This plugin will contain a single module encompassing the entire codebase. Although the design and structure will adhere to Unreal's module and plugin architecture, the final result will be delivered as a plugin within an Unreal project. This approach ensures that MetaCsound remains decoupled from both the game and the engine, allowing it to be easily imported into other Unreal projects. The project itself will serve as both a development and testing environment, as well as a demonstration platform to showcase the feature and create examples.

One of the primary objectives of this thesis is to provide developers with a simple and straightforward method to integrate Csound with Unreal. The goal is to make the process as easy and intuitive as possible.

3.6.4 Plugin Modules and Dependencies

The project will rely on two key components: MetaSounds, the official plugin developed by Epic Games, and MetaCsound, the custom plugin being developed. MetaSounds can be easily added by searching for it in the *Plugin Manager* window. In contrast, MetaCsound will need to be created from scratch using the "New" feature within the Plugin Manager.

The project's dependencies, as described in the `.uproject` file, will include references to both MetaSounds and MetaCsound. Below is an example of how these dependencies appear in the `.uproject` file:

```
{
  "FileVersion": 3,
  "EngineAssociation": "5.3",
  "Category": "",
  "Description": "",
  "Modules": [
    {
      "Name": "MyProject2",
```

```

        "Type": "Runtime",
        "LoadingPhase": "Default"
    }
],
"Plugins": [
    {
        "Name": "Metasound",
        "Type": "Runtime",
        "Enabled": true
    },
    {
        "Name": "MetaCsound",
        "Type": "Runtime",
        "Enabled": true
    },
    (...)
]
}

```

Similarly, since MetaCsound will depend on MetaSounds, this dependency must be specified in the `.uplugin` file for MetaCsound, much like in the `.uproject` file. The loading phase of the MetaCsound module is strategically set to occur after the MetaSounds plugin by using the *PostDefault* option. This ensures that MetaSounds is fully loaded and available before MetaCsound initializes, avoiding potential dependency issues.

```

{
    "FileVersion": 3,
    "Version": 1,
    "VersionName": "1.0",
    "FriendlyName": "MetaCsound",
    (...)
    "Modules": [
        {
            "Name": "MetaCsound",
            "Type": "Runtime",
            "LoadingPhase": "PostDefault",
            "PlatformAllowList": [
                "Win64"
            ],
            "PlatformDenyList": [
                "IOS",
                "Linux",
                "Android"
            ]
        }
    ],
    "Plugins": [
        {
            "Name": "Metasound",
            "Type": "Runtime",

```

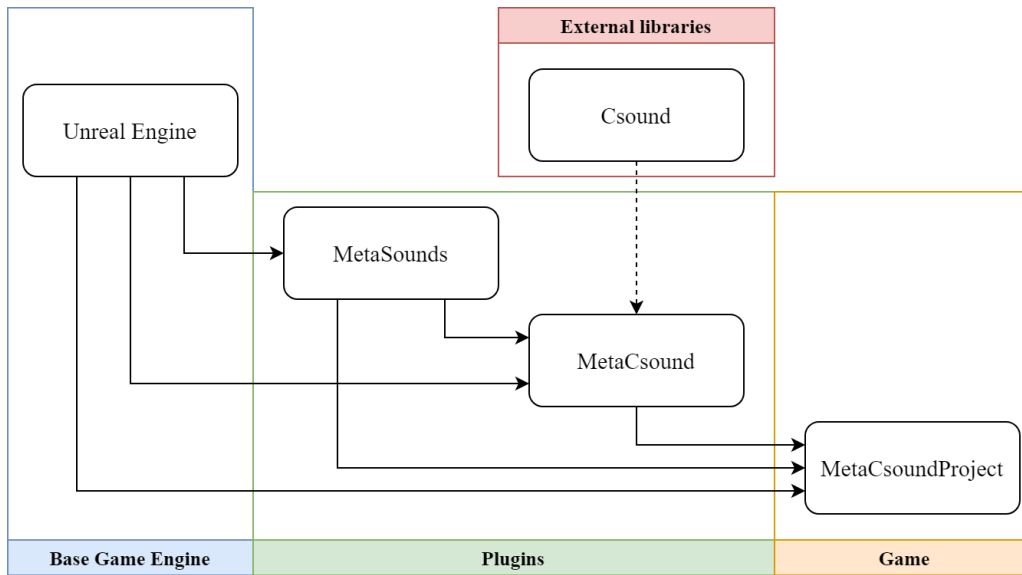


Figure 3.4: Graph of all components and their dependencies.

```

    "Enabled": true
  }
]
}

```

Figure 3.4 illustrates the relationships between all components and their dependencies.

3.6.5 Dynamic Link Libraries

A *Dynamic Link Library* (DLL) is a file format used to store executable functions or data that can be accessed by multiple programs simultaneously [23]. It is "dynamic" because it is loaded into memory only when needed, which helps conserve resources and facilitates code reuse. DLLs reduce the size of individual executables and simplify updates, as modifying the DLL updates all programs that use it. They are stored in `.dll` files.

In contrast, a *static library* contains code that is linked directly into an executable at compile time. This results in larger executables but eliminates the need for separate library files at runtime.

A `.lib` file is a static library that provides the necessary code and data to link with a DLL. Each DLL typically has a corresponding `.lib` file, which helps the linker resolve references to the DLL's functions during the build process. This static library ensures that the executable can call the DLL's functions correctly when it runs.

When embedding Csound into external applications, it is linked as a dynamic library.

3.6.6 Unreal Build Tool

Unreal Build Tool is a specialized tool designed to manage the compilation of Unreal Engine source code across different build configurations. Unreal Engine is organized into numerous modules, each with a `.build.cs` file that defines how the module is built. This file, written in C#, outlines dependencies, additional libraries, include paths, and other settings, while the modules themselves are written in C++. Typically, these modules are compiled into *DLLs* and loaded by a central executable.

3.6.7 Importing and Linking Csound inside Unreal

The MetaCsound plugin will consist of a single module, named accordingly. Consequently, only one `build.cs` file is required. This file will specify the module's dependencies and include instructions to properly integrate and link Csound within the module.

Currently, this module is designed to function exclusively on the Windows operating system. Future development will focus on enhancing cross-platform compatibility by addressing the requirements for linking dynamic libraries on other operating systems.

The MetaCsound plugin will consist of a single module, named accordingly. Consequently, only one `.build.cs` file is required. This file will specify the module's dependencies and include instructions to properly integrate and link Csound within the module.

Currently, this module is designed to work exclusively on the Windows operating system. Each operating system has its own requirements for linking dynamic libraries, which have to be addressed to ensure cross-platform compatibility. The following code shows the content of the `.build.cs` file.

```
using UnrealBuildTool;

public class MetaCsound : ModuleRules
{
    public MetaCsound(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = ModuleRules.PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] {
            "Core",
            "MetasoundGraphCore",
            "MetasoundFrontend",
        });

        PrivateDependencyModuleNames.AddRange new string[] {
            "CoreUObject",
            "Engine",
        });

        PublicIncludePaths.AddRange(new string[] {
            "C:/Program Files/Csound6_x64/include/csound/",
        });
    }
}
```

```

});

// Include the import library
PublicAdditionalLibraries.Add(
    "C:/Program Files/Csound6_x64/lib/csound64.lib");

// Load library
PublicDelayLoadDLLs.Add("csound64.dll");
}
}

```

By adding a public include path, we enable the preprocessor to locate and include all header files of the Csound library. This allows the use of the `#include "csound.hpp"` directive to access the Csound library as needed.

Typically, when packaging a game, the DLL file would be deployed using the following instruction:

```

RuntimeDependencies.Add(
    "C:/Program Files/Csound6/x64/bin/csound64.dll");

```

However, in this case, we do not want to deploy `csound64.dll` with the game. Instead, users will need to install Csound manually, allowing them to choose the version that best suits their needs. If necessary, this instruction can be added for games that require DLL deployment.

The static part of the library is managed using the `PublicAdditionalLibraries` command, while the dynamic part is handled by `PublicDelayLoadDLLs`. Note that when using `PublicDelayLoadDLLs`, specifying the full path is not required.

3.6.8 Module Startup and Shutdown

Each module implements the `IModuleInterface`, which requires overriding two key methods: one for initializing the module and one for shutting it down. In our implementation, the startup method will log a message to the Unreal Editor console indicating whether the Csound DLL was successfully found, which is useful for debugging. The `IMPLEMENT_MODULE` macro is used to make the module available to the rest of the engine.

```

class FMetaCsoundModule : public IModuleInterface
{
public:

    /** IModuleInterface implementation */
    virtual void StartupModule() override
    {
        (...)
        const FString LibExamplePath =
            TEXT("C:/Program Files/Csound6_x64/bin/csound64.dll");
        void* DynamicLibExampleHandle =
            FPlatformProcess::GetDllHandle(*LibExamplePath);
    }
}

```

```

    if (DynamicLibExampleHandle != nullptr)
    {
        UE_LOG(LogTemp, Log,
            TEXT("csound64.dll loaded successfully!"));
    }
    else
    {
        UE_LOG(LogTemp, Fatal,
            TEXT("csound64.dll failed to load!"));
    }
}

virtual void ShutdownModule() override { }
};

IMPLEMENT_MODULE(FMetaCsoundModule, MetaCsound)

```

The `UE_LOG` macro is used to log messages to the Unreal Editor console. The `TEXT` macro will be explained in Section 3.7.

3.6.9 Namespaces and API Macros

In Unreal Engine, code that is not defined as `UCLASS` or `USTRUCT` should be placed within a namespace to avoid name clashes.

When UBT builds the project, it generates a definition header file for each module. These definition files contain various macros that facilitate programming within the module, including API macros.

API macros manage symbol visibility when working with dynamic libraries (DLLs) in C++ projects. They ensure that classes, functions, and other symbols are correctly exported and imported when building and using dynamic libraries.

On Windows, these macros are defined as `__declspec(dllexport)` when exporting symbols from a DLL and `__declspec(dllimport)` when importing symbols from a DLL [20]. This differentiation allows the compiler to handle code linkage between different modules or applications properly. In Unreal Engine, API macros extend DLL import macros for each module dependency listed in the `.build.cs` file, while the module being implemented will use DLL export macros.

For example:

- When building the MetaCsound plugin, the macro `METACSOUND_API` expands to `__declspec(dllexport)`, instructing the compiler to export the class or function for use by other modules.
- When using the MetaCsound plugin in another module, `METACSOUND_API` expands to `__declspec(dllimport)`, indicating to the compiler that the class or function is located in an external DLL and should be linked accordingly.

This mechanism is crucial for ensuring that code within a DLL can be accessed by other parts of the project and helps Unreal Engine manage modularity across different plugins. These macros are typically used in class definitions, following the `class` keyword. They will be encountered when showing code in the future sections.

3.7 Text Localization

In Unreal Engine, text can be localized into different languages and cultures using the `FText` type in C++. This type includes three components: a key for identification, a namespace to avoid key clashes, and a source string that serves as the default translation.

When creating localizable text using literals, two macros are available:

- **NSLOCTEXT**: This macro creates a localized text by specifying the namespace, key, and source string. It's ideal for cases where the namespace is used infrequently within a file.
- **LOCTEXT**: This macro generates localized text by defining only the key and source string. It relies on a predefined namespace set by the `LOCTEXT_NAMESPACE` macro. Before the file ends, the namespace must be undefined. This is useful for creating multiple localized texts within the same file under the same namespace.

For non-localizable strings, the `TEXT` macro should be used for string literals. The corresponding type for non-localizable strings in Unreal is `FString`, which is Unreal's equivalent of C++'s `std::string`.

3.8 Curiously Recurring Template Pattern

The *Curiously Recurring Template Pattern* (CRTP) is a design pattern in which a class `Derived` inherits from a template class `Base`, with the template argument for `Base` being the derived class itself. This creates a situation where `Derived` is both the subclass and the type passed to its base class template [1]. CRTP is utilized in the design of MetaSounds. Below is a basic example:

```
template <typename T>
class Base {
public:
    void interface() {
        static_cast<T*>(this)->implementation();
    }

    // Optional, if we want a default implementation
    void implementation() {
        // Default behavior
    }
}
```

```
};

class Derived : public Base<Derived> {
public:
    void implementation() {
        // Derived-specific behavior
    }
};
```

Advantages of CRTP include:

- **Compile-Time Polymorphism:** Unlike traditional polymorphism, where virtual functions introduce runtime overhead, CRTP allows for polymorphism that the compiler can resolve entirely at compile time. This makes it more efficient in terms of performance.
- **Static Interface Enforcement:** CRTP can be used to enforce an interface that derived classes must implement, without requiring virtual functions. This allows the base class to call methods on the derived class as if they were virtual, but without the associated runtime cost.
- **Code Reuse and Extension:** CRTP enables code reuse and extension in a way that avoids the drawbacks of multiple inheritance or the need for runtime checks. This pattern is particularly useful in cases like implementing static polymorphism, mixins, or compile-time configuration.

3.8.1 CRTP in MetaCsound

CRTP is used in MetaSounds to implement static polymorphism. When defining a new operator node, the corresponding operator class inherits from the template `TExecutableOperator`, with the first template argument being the derived class itself.

```
class FCsoundOperator :
    public TExecutableOperator<FCsoundOperator> { }
```

This is the most straightforward use of the `TExecutableOperator` template. However, as we'll explore later, creating different versions of the same Csound node, each with a distinct vertex interface, requires polymorphism with static methods. Given that CRTP is already in use, the pattern is extended to accommodate multiple specializations of the base Csound operator class. The following code example and Figure 3.5 illustrate the derived operators with different vertex interfaces implementing CRTP for handling stereo, quadraphonic, and octophonic audio, respectively.

```
template<typename DerivedOperator>
class TCsoundOperator :
    public TExecutableOperator<DerivedOperator> { }
```

```

class FCsoundOperator2 :
    public TCsoundOperator<FCsoundOperator2> { }

class FCsoundOperator4 :
    public TCsoundOperator<FCsoundOperator4> { }

class FCsoundOperator8 :
    public TCsoundOperator<FCsoundOperator8> { }

```

When discussing the implementation of new nodes in MetaSounds, it will be understood that the code examples primarily involve the `TCsoundOperator` template. Specific implementations within derived operators will be explicitly noted as needed.

3.8.2 Derived Operators

The core functionality for the new nodes will be implemented in the base class `TCsoundOperator`. Variations in the nodes will be based on differences in the number of input and output audio buffers and float parameters. Each derived class will have distinct names, display names, and descriptions to ensure clear differentiation in the MetaSounds graph.

```

class METAC SOUND_API FCsoundOperator2 :
    public TCsoundOperator<FCsoundOperator2>
{
public:
    static const FNodeClassName GetClassName()
    {
        return { TEXT("MetaCsound"),
            TEXT("Csound8"), TEXT("Audio") };
    }
    static const FText GetDisplayName()
    {
        return LOCTEXT("NodeDisplayName", "Csound (2)");
    }
    static const FText GetDescription()
    {
        return LOCTEXT(
            "NodeDesc",
            "Csound with stereo input and output"
        );
    }
}

// Two Audio channels in and out because
// is the stereo operator
static constexpr int32 NumAudioChannelsIn = 2;
static constexpr int32 NumAudioChannelsOut = 2;
static constexpr int32 NumControlChannelsIn = 8;
static constexpr int32 NumControlChannelsOut = 4;
};

```

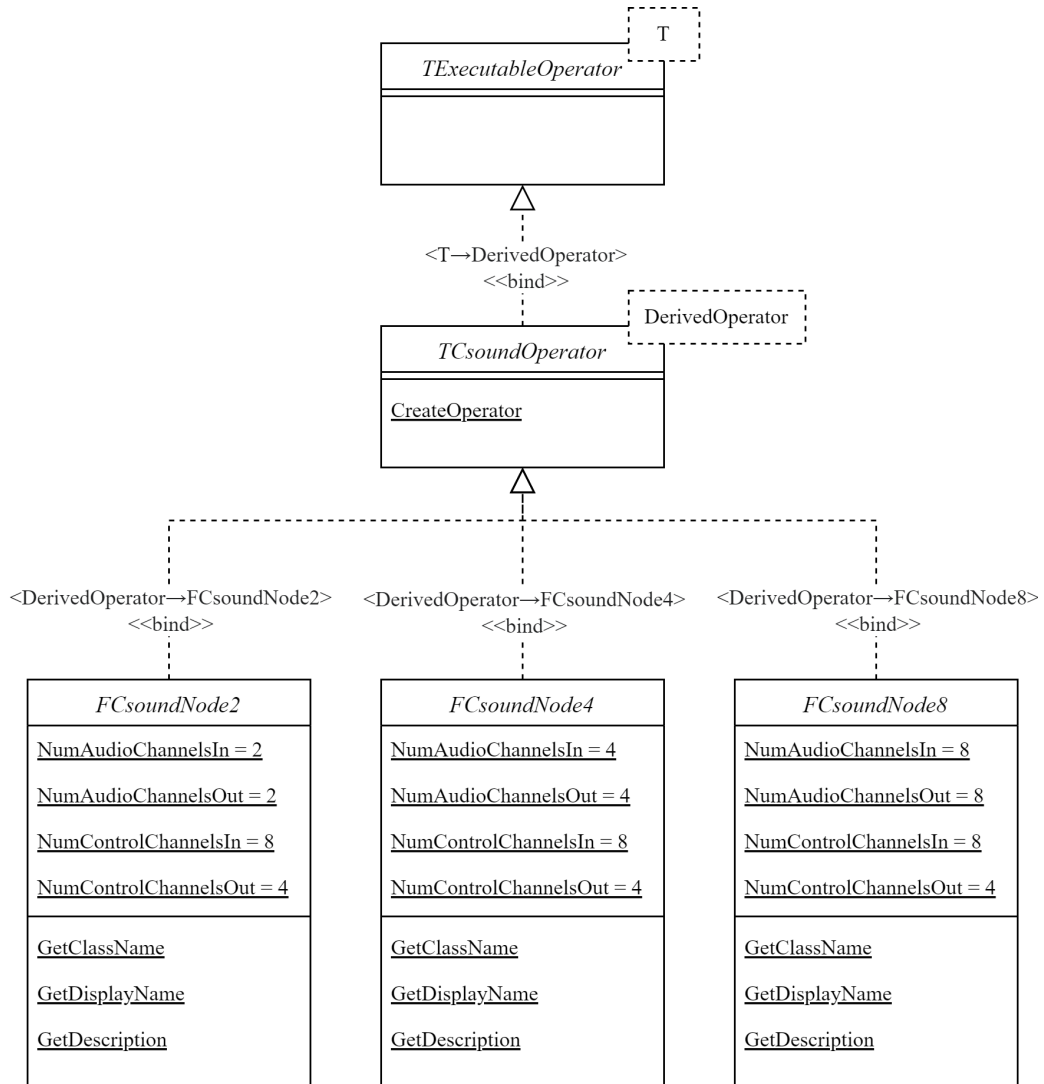


Figure 3.5: Class diagram showing CRTP.

3.9 Factory Method

A *static factory method* is a static method in a class that returns an instance of that class. This method abstracts the instantiation process, often returning a *unique smart pointer* for better memory management and clear transfer of ownership. The method is part of the class it creates instances of, providing an alternative to directly calling the constructor [2].

Before examining the particular implementation of static factory methods in this project, it's important to introduce and understand some concepts:

3.9.1 Smart Pointers

A *smart pointer* is an object in C++ that manages the lifetime of a dynamically allocated object, ensuring that the object is automatically destroyed when the smart pointer goes out of scope [23]. This helps to prevent memory leaks and dangling pointers, which are common issues with manual memory management.

Unique pointers (`TUniquePtr` in Unreal) are a type of smart pointer that ensures exclusive ownership of the object. Only one `TUniquePtr` can manage a given object at a time, and it automatically deletes the object when the pointer is destroyed or reassigned. This is an example of creating a unique pointer that points to an integer with value 25.

```
TUniquePtr<int> PtrA = MakeUnique<int>(25);
```

Because this pointer is unique, the following expression is forbidden. Only one pointer can have the ownership of the data pointed at.

```
TUniquePtr<int> PtrA = MakeUnique<int>(25);  
TUniquePtr<int> PtrB = PtrA; // Not allowed
```

If the ownership of a unique pointer needs to be transferred, the function `MoveTemp` can be used. This will make `PtrB` point at the same data that previously was pointing `PtrA`, but at the same time `PtrA` will be set null.

```
TUniquePtr<int> PtrA = MakeUnique<int>(25);  
  
// Ownership transferred to PtrB  
TUniquePtr<int> PtrB = MoveTemp(PtrA);
```

3.9.2 CsoundParams Namespace

The macro helper family, such as `METASOUND_GET_PARAM`, used extensively in various methods, requires further explanation. These macros serve as convenience tools to retrieve the name and tooltip description of node parameters. By defining these references in a single place at the top of the header file, rather than retyping them throughout the code, it enhances maintainability and consistency [17]. This approach minimizes errors and ensures that any changes to parameter names or descriptions are applied uniformly across the codebase.

```

namespace MetaCsound::NodeParams
{
    METASOUND_PARAM(PlayTrig, "Play",
        "Compiles the Csound file and starts performing.");
    METASOUND_PARAM(StopTrig, "Stop",
        "Stops the Csound performance.");
    METASOUND_PARAM(CsoundFile, "File",
        "The name of the `.csd` file to be performed.
        This file must be placed in the CsoundFiles folder.");
    METASOUND_PARAM(FinTrig, "On Finished",
        "Triggered when the performance stops, either due
        to a Stop trigger or the Csound performance ending.");

    METASOUND_PARAM(InA, "In Audio {0}",
        "Input audio channel {0}. Can be accessed in Csound
        using opcodes like `in` or `ins`.");
    METASOUND_PARAM(OutA, "Out Audio {0}",
        "Output audio channel {0}. Can be accessed in Csound
        using opcodes like `out` or `outs`.");
    METASOUND_PARAM(InK, "In Control {0}",
        "Input control channel {0}. Corresponds to the Csound
        control bus with the same name.");
    METASOUND_PARAM(OutK, "Out Control {0}",
        "Output control channel {0}. Corresponds to the Csound
        control bus with the same name.");
}

```

When a method needs to access parameter information, including the using namespace NodeParams directive inside the method body allows for easy retrieval of this data. Some parameters use the {0} placeholder in their string literals for indexed parameter definitions. This placeholder enables the creation of multiple parameters with similar names but different indices. This is particularly useful for parameters like audio inputs and outputs, where nodes may have varying numbers (e.g., 2 inputs vs. 4 inputs). The {0} placeholder is replaced with the specific index number during formatting, ensuring that each parameter is correctly named and described.

In many of the methods discussed, macros are used to retrieve node parameter data. Examples of these macros are provided below, demonstrating both indexed and non-indexed parameter definitions:

```

METASOUND_GET_PARAM_NAME_AND_METADATA(PlayTrig);
METASOUND_GET_PARAM_NAME_WITH_INDEX_AND_METADATA(
    InA, index);

METASOUND_GET_PARAM_NAME(PlayTrig, PlayTrigger);
METASOUND_GET_PARAM_NAME_WITH_INDEX(
    InA, index), AudioInRefs[index])

```

Note that in the indexed version of these macros, the index number is used as a parameter to format the strings.

3.9.3 Csound Operator Factory Method

MetaSound node operators use a *static factory method* instead of directly calling the constructor. This method creates input data references and other necessary variables to be passed to the constructor. It returns a *unique pointer* containing a newly created instance of the class [2].

The factory method takes two parameters: the first is a class that includes node information, and the second is a non-const reference used for error reporting. The first parameter, which contains the `OperatorSettings` class, is crucial as it specifies the size of the audio buffers.

As previously discussed, this method handles the creation of input data references. The `Start`, `Stop`, and `CsoundFile` inputs are passed to the constructor by reference, while buffer and float data references are sent in an array, accommodating variable numbers based on the vertex interface. Due to the static polymorphism enabled by *CRTP*, we can determine the number of input and output audio and control channels from the derived operator's static values. The constructor parameters will also include the number of output audio and control channels along with the `OperatorSettings` class.

The implementation of the factory method is as follows:

```
TUniquePtr<Metasound::IOperator> CreateOperator(
    const FCreateOperatorParams& InParams,
    TArray<TUniquePtr<IOperatorBuildError>>& OutErrors)
{
    using namespace NodeParams;

    const FDataReferenceCollection& InputCollection =
        InParams.InputDataReferences;
    const FInputVertexInterface& InputInterface =
        DeclareVertexInterface().GetInputInterface();

    // Play Trigger
    TDataReadReference<FTrigger> PlayTrigger =
        InputCollection.
        GetDataReadReferenceOrConstructWithVertexDefault<FTrigger>
            (InputInterface,
             METASOUND_GET_PARAM_NAME(PlayTrig),
             InParams.OperatorSettings);

    // Stop Trigger
    TDataReadReference<FTrigger> StopTrigger =
        InputCollection.
        GetDataReadReferenceOrConstructWithVertexDefault<FTrigger>
            (InputInterface,
             METASOUND_GET_PARAM_NAME(StopTrig),
             InParams.OperatorSettings);

    // Csound File
```

```

TDataReadReference<FString> CsoundFileString =
    InputCollection.
    GetDataReadReferenceOrConstructWithVertexDefault<FString>
        (InputInterface,
         METASOUND_GET_PARAM_NAME(CsoundFile),
         InParams.OperatorSettings);

// Input audio buffers
TArray<TDataReadReference<FAudioBuffer>> AudioInArray;
AudioInArray.Empty(DerivedOperator::NumAudioChannelsIn);
for (int32 i = 0; i < DerivedOperator::NumAudioChannelsIn; i++)
{
    AudioInArray.Add(TDataReadReference<FAudioBuffer>(
        InputCollection.
        GetDataReadReferenceOrConstructWithVertexDefault<FAudioBuffer>
            (InputInterface,
             METASOUND_GET_PARAM_NAME_WITH_INDEX(InA, i),
             InParams.OperatorSettings)
    ));
}

// Input floats (k-rate)
TArray<TDataReadReference<float>> ControlInArray;
ControlInArray.Empty(DerivedOperator::NumControlChannelsIn);
for (int32 i = 0; i < DerivedOperator::NumControlChannelsIn; i++)
{
    ControlInArray.Add(TDataReadReference<float>(
        InputCollection.
        GetDataReadReferenceOrConstructWithVertexDefault<float>
            (InputInterface,
             METASOUND_GET_PARAM_NAME_WITH_INDEX(InK, i),
             InParams.OperatorSettings)
    ));
}

// Unique pointer and call to the constructor
return MakeUnique<DerivedOperator>(
    InParams.OperatorSettings,
    PlayTrigger, StopTrigger,
    CsoundFileString,
    AudioInArray, DerivedOperator::NumAudioChannelsOut,
    ControlInArray, DerivedOperator::NumControlChannelsOut
);
}

```

The factory method requires the input and output interface classes, which are obtained through the vertex interface created by the `DeclareVertexInterface` method. This method will be explained in detail later in Section 3.18.2.

The `InputCollection` class facilitates obtaining data read reference values by retrieving the parameter connection from another node or variable in the graph. If no connection

exists, it initializes the reference with the default value of that parameter.

3.10 Operator Constructor and Execute Method

MetaSounds node operators rely on two fundamental methods: the constructor and the **Execute** method.

The constructor is responsible for creating and initializing new instances of the operator class. Like in any class, this method is invoked when a new object is instantiated, including when a smart pointer to a new operator is created within the factory method. For the MetaCsound template operator we are developing, the constructor will handle the initialization of the variables, including the instance of Csound.

The **Execute** method is called periodically while the MetaSound graph is running. This method reads the input data, performs the necessary processing, and writes the results to the outputs [17]. It is designed to fill the entire length of the output audio buffers with the processed audio data.

3.11 Csound File Handling

As previously mentioned, the `.csd` file that describes the Csound instance's operations is input into the node using a string data reference.

3.11.1 File Directory

All `.csd` files must be placed in a folder named **CsoundFiles** within the Unreal Project's **Content** folder. For organizational purposes, these files can be stored in subfolders within the **CsoundFiles** directory. When specifying the string variable in the MetaSound graph, it is essential to include the subfolder's name, followed by the file name, separated by a / forward slash symbol.

It is important to note that, unfortunately, `.csd` files will not be recognized by the Unreal Editor's Content Browser. However, this does not mean that the files are missing; using the default operating system file explorer will reveal their presence in the specified directory.

When packaging the game (which in Unreal involves creating a standalone executable), certain project folders will not be included in the final package. This is logical since not all files are necessary for the final executable. However, if the **CsoundFiles** folder is not explicitly included, it will be omitted from the package. To prevent this, developers must add an exception in the project settings:

Navigate to:

Project Settings -> Packaging -> Additional Non-Asset Directories to Copy.

This setting option is an array. Developers should add a new element to the array with the value set to the folder's name, in this case, **CsoundFiles**.

3.11.2 File Compilation

When the play trigger is activated, the `Play` method is invoked from within the `Execute` method. This `Play` method reads the name of the `.csd` file, compiles it in the `Csound` instance, and updates the variables associated with the `Csound` performance. Below is the code responsible for compiling the `.csd` file.

```
const FString FullPath = FPaths::ProjectContentDir() +
    TEXT("CsoundFiles/") + *CsoundFile.Get() + TEXT(".csd");
const char* FullPathANSI = StringCast<ANSICHAR>(*FullPath).Get();

const FString SrOption = "--sample-rate=" +
    FString::FromInt((int)OpSettings.GetSampleRate());
const char* SrOptionANSI = StringCast<ANSICHAR>(*SrOption).Get();

const int32 ErrorCode =
    CsoundInstance.Compile(FullPathANSI, SrOptionANSI, "-n");
```

To retrieve the full path of the `.csd` file as an `FString`, `FPaths::ProjectContentDir()` can be used. This function provides the absolute path to the `Content` folder, ensuring compatibility regardless of the project's location, the operating system, or whether the code is running in the editor or as a standalone packaged application [13].

The `CsoundFile` variable is an `FString` data read reference. To convert the `FullPath` `FString` into a C-style string that the `Csound` API can understand, we use a `StringCast<ANSICHAR>` object.

The full path is then passed to the `CsoundInstance.Compile` method. This method attempts to compile the `.csd` file, returning an error code if it fails. Compilation could fail for several reasons, such as a syntax error within the `.csd` file or an inability to locate the file.

Several options are passed as parameters to the `Compile` method. The `-n` option instructs `Csound` not to use its default input and output audio devices, which is necessary when the developer intends to manage input and output data manually, as in this case [10].

Another crucial option is the sample rate. To ensure accurate and consistent communication between `Csound` and `MetaSounds`, both environments must operate at the same sample rate. We achieve this by overriding the sample rate specified in the `.csd` file using the `--sample-rate` flag. The flag is appended with the sample rate from the `OpSettings` object, and this result is converted into a C-style string for use in the `Compile` method.

3.12 Audio Input and Output

In this section, we will explore the manipulation of audio buffers to facilitate the sending and receiving of audio-rate data between `MetaSounds` and `Csound`.

3.12.1 Spout and Spin

The output of a Csound performance can be directed to the sound card, saved to a file, or retrieved by other applications using the *spout* [7]. The spout in Csound is a constant pointer to the data that, during a performance, contains the result of the last processing loop. It can be treated as an array of doubles, with each double representing an audio sample that must not exceed the `0dBFS` environment variable.

If the Csound is performed with only one output audio channel, as defined by the `nchnls` environment variable, the spout length will match the `ksmps` value, which is the number of audio samples per control period. If there are multiple output channels, the channel data will be interleaved. This means that the first sample of the first channel appears first, followed by the first sample of the second channel, and so on until all channels have their first sample. The second sample of each channel then follows in the same order. Consequently, the length of the spout will be `ksmps * nchnls`.

Similarly, the input that Csound reads from during each performance loop is handled by the *spin*. spin is a pointer to the data that, during a performance, contains the input information read by opcodes like `in`. Unlike spout, spin is not constant because it needs to be modified externally so that Csound can read it. The spin data is structured similarly to spout, with interleaved channels, but it is based on the input audio channels, as defined by the `nchnls_i` environment variable. The length of the spin will be `ksmps * nchnls_i`.

The following code illustrates a simple Csound instrument that reads from the spin, applies a filter, and outputs the result to the spout:

```
nchnls = 2
nchnls_i = 2
0dbfs = 1

instr 1
    // read from spin
    ain1, ain2 ins

    alp1 butterlp ain1, 440
    alp2 butterlp ain2, 2000

    // write to spout
    outs alp1, alp2
endin
```

3.12.2 Audio Output

This section will cover how to read the output from Csound via the spout and translate it into the audio buffer data references used by the MetaSounds node. As previously explained, this translation occurs during each `Execute` call, with each call responsible for populating the entire output audio buffers. The following code snippet demonstrates this process within the `Execute` method:

```

for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
{
    (...)

    for (int32 i = 0; i < MinAudioOut; i++)
    {
        BuffersOut[i][f] = (float)Spout[SpIndex * CsoundNchnlsOut + i];
    }

    if (++SpIndex >= CsoundKsmps)
    {
        (...)
        // Set SpIndex to 0
        CsoundPerformKsmps(f);
        (...)
    }
}

```

The first `for` loop iterates over all the frames of the audio buffers. It starts at zero and ends at the number of frames per block specified by the `OpSettings`.

`BuffersOut` is a two-dimensional array where each row represents a different output audio buffer of the node, and each column represents a frame. The second `for` loop iterates over each functional output audio buffer. The number of functional audio channels is determined by comparing the output channels in Csound with those in the MetaSounds node and selecting the minimum of the two.

3.12.3 Indexing the Spout

When accessing the spout, it's important to introduce the concept of the `SpIndex` variable. `SpIndex` is used to track the current frame index being read from the spout. It's crucial not to use the loop variable `f` from the first loop because the buffer sizes of MetaSounds and Csound may differ, leading to desynchronization between the two systems. While synchronizing Csound's buffer size with MetaSounds is one option, it is not always necessary. Instead, we use `SpIndex` (a member variable of the object) to keep track of the frame index between `Execute` calls. This ensures that `SpIndex` maintains its value across calls, allowing consistent access to the spout.

The `SpIndex` value increments with each frame loop iteration. Eventually, `SpIndex` may exceed the buffer size of Csound, known as `ksmps`. When this occurs, it signals that a new Csound performance call is required to refresh the data in the spout. This is achieved by invoking the `CsoundPerformKsmps` method. This method not only performs a new Csound performance cycle but also resets the `SpIndex` to zero, ensuring that the indexing starts from the beginning for the new performance cycle. This method will be discussed further on Section 3.15.

For a given frame, the data for the first channel can be accessed directly by multiplying the `SpIndex` (current frame) by `CsoundNchnlsOut` (total number of channels). To access

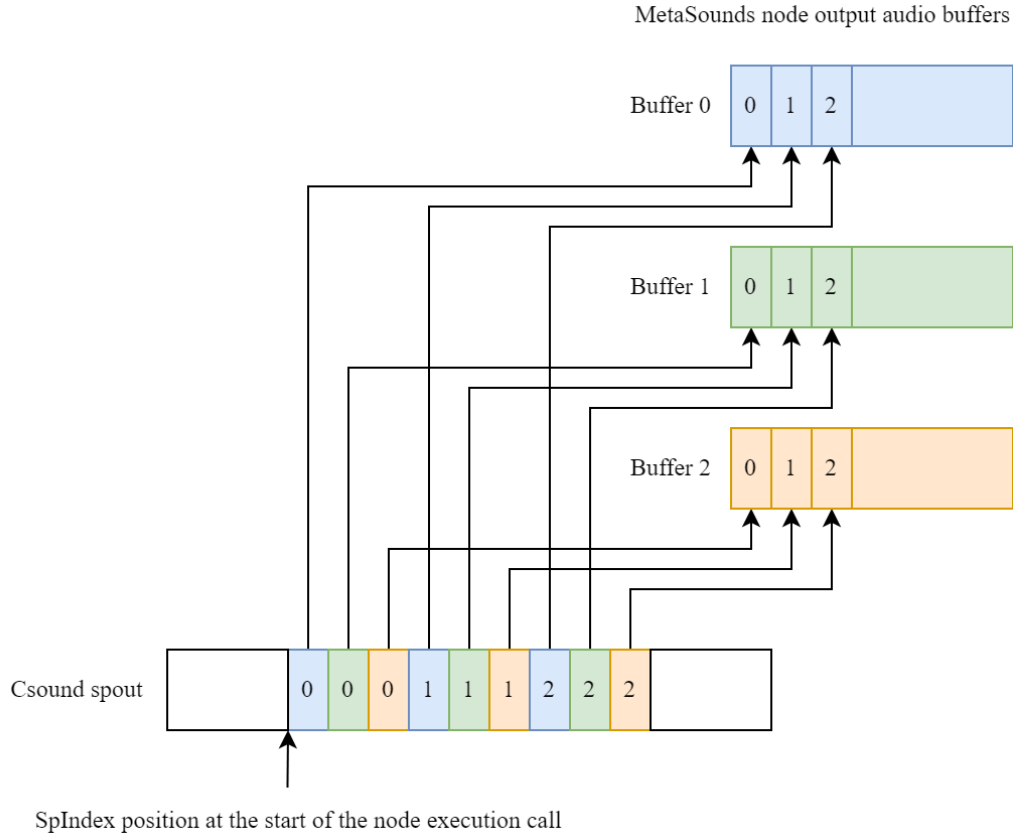


Figure 3.6: Translating the interleaved spout into single-channel output audio buffers.

data for other channels, we must add the *i* variable (current channel) to this result.

For instance, if we need to access the sample for frame *SpIndex* on channel *i*, we can calculate the position in the spout buffer as follows:

$$Position = (SpIndex * CsoundNchnlsOut) + i$$

This approach works because the samples for each channel are interleaved in the buffer. The frame index determines the starting point of a frame's samples, and the channel index offsets this point to access samples for different channels.

In simpler terms, to iterate over all samples of a specific channel for each frame, we need to jump through the spout buffer in steps equal to the number of channels. This ensures that we access the correct sample for each frame and channel combination. The frame index dictates the starting position of these jumps, while the channel index adjusts for different channels within the same frame.

3.12.4 Audio Input

This section covers the process of reading data from the node's audio buffer inputs and translating this information into Csound's input, known as the spin. Handling audio input is similar to managing audio output. Each call to the **Execute** method is responsible for reading the entirety of the input audio buffers and transferring this data into the spin.

```
for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
{
    (...)

    for (int32 i = 0; i < MinAudioIn; i++)
    {
        Spin[SpIndex * CsoundNchnlsIn + i] = (double)(BuffersIn[i][f]);
    }

    (...)

    if (++SpIndex >= CsoundKsmps)
    {
        (...)
        // Set SpIndex to 0
        CsoundPerformKsmps(f);
        (...)
    }
}
```

The code shown above mirrors the **for** loop from the Section 3.12.2. It iterates over all frames of the audio buffers, starting at zero and ending at the number of frames per block specified by **OpSettings**. This ensures that the entire length of the audio buffers is processed.

BuffersIn, like **BuffersOut**, is a two-dimensional array where each row represents a different node input audio buffer and each column represents a frame. The second **for** loop begins at zero and terminates when the number of functional audio channels is reached. This number is determined by comparing the input channels in Csound with the input channels of the MetaSounds node and taking the minimum of the two.

3.12.5 Indexing the Spin

Accessing the spin is similar to accessing the spout. The same **SpIndex** variable can be used to track the current frame index for writing to the spin. As explained in Section 3.12.3, **SpIndex** is incremented with each frame iteration. This synchronization is possible because both the spin and spout are iterated based on the same frame index, even if they have different numbers of channels. This consistency is ensured by Csound's **ksmps** environment variable, which sets the audio buffer size uniformly across input, output, and any a-rate variables.

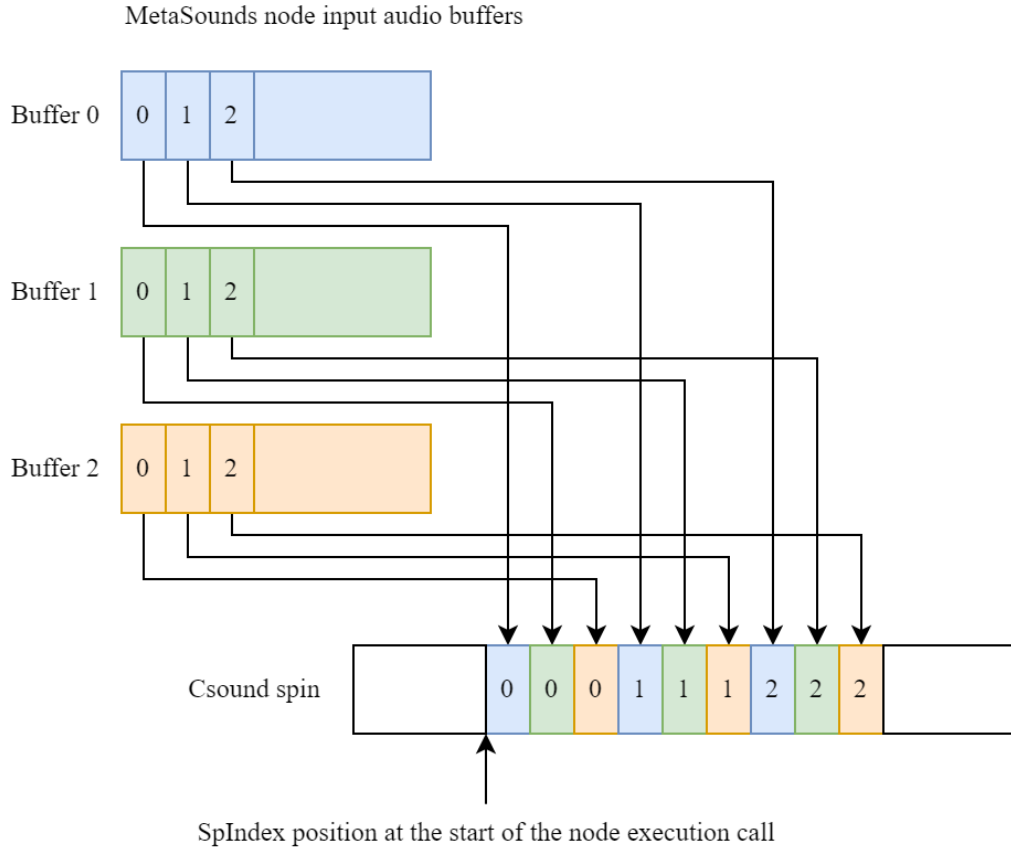


Figure 3.7: Translating the audio input buffers into a spin interleaved structure.

The `SpIndex` value increases with each frame loop iteration. When `SpIndex` exceeds the buffer size of Csound (`ksmps`), it indicates that the spin has been filled with new data and the spout has been read completely. At this point, a new Csound performance call must be executed to process the spin data before it is reset. This is achieved by calling `CsoundPerformKsmps`. This method will be discussed further on Section 3.15.

The only difference in accessing the spin compared to the spout is that instead of using the `CsoundNchnlsOut` variable, which represents the number of Csound output audio channels, `CsoundNchnlsIn` is used, which represents the number of input channels. The updated mathematical expression to access the spin is:

$$Position = (SpIndex * CsoundNchnlsIn) + i$$

3.13 Control Input and Output

To send and receive single floats to and from Csound, control buses are utilized. Although the term *bus* may be somewhat misleading (since the opcodes in *.csd* files are named *chnget* and *chnset*, and the Csound API also refers to *channels*) the term *bus* is used to differentiate these from audio channels.

As discussed in Section 3.3.6, control buses are identified by a name, which in the context of the Csound C++ API is a C-style string [7]. In MetaCsound, the control buses are named *In Control {0}* and *Out Control {0}*, where {0} represents the bus number, starting from 0 up to the maximum number allowed by the specific derived operator. These names are stored in two *FString* arrays: *ControlInNames* for input control buses and *ControlOutNames* for output control buses. Correspondingly, *ControlInRefs* and *ControlOutRefs* hold the float read and write data references for the node.

To write to a Csound control bus from an external application, the *SetControlChannel* method of the Csound object is used. The first parameter of this method is the name of the bus, and the second parameter is the value to be set. To read a control bus, *GetControlChannel* is used, which takes the bus name as its parameter and returns the current value of the channel.

To ensure accuracy between systems that may be running at different audio buffer sizes, Csound input control buses are updated just before executing a new performance call. MetaSounds write float data references are updated immediately after the Csound performance call (see Section 3.15). This process involves two for loops to iterate through all input and output references. The following piece of code from the *Execute* method illustrates this.

```
void Execute()
{
    (...)
    for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
    {
        // In and out audio channels
        (...)

        if (++SpIndex >= CsoundKsmmps)
        {
            SetInputControlChannels();
            CsoundPerformKsmmps(f);
            if (OpState == EOpState::Playing)
            {
                GetOutputControlChannels();
            }
        }
    }
}

void SetInputControlChannels()
```

```

{
    for (int32 i = 0; i < ControlInRefs.Num(); i++)
    {
        CsoundInstance.SetControlChannel(StringCast<ANSICHAR>(
            *ControlInNames[i]).Get(), (double)*(ControlInRefs[i]));
    }
}

void GetOutputControlChannels()
{
    for (int32 i = 0; i < ControlOutRefs.Num(); i++)
    {
        *(ControlOutRefs[i]) = (float)CsoundInstance.GetControlChannel(
            StringCast<ANSICHAR>(*ControlOutNames[i]).Get());
    }
}

```

The control bus names are `FStrings`. To convert an `FString` into a C-style string compatible with the Csound API, the use of the `StringCast<ANSICHAR>` object is needed.

When reading the output control buses from Csound, the loop will execute only if Csound is still performing, as indicated by the expression `OpState == EOpState::Playing`. This comparison is detailed in Section 3.16.1.

The following code illustrates a simple Csound instrument that reads from two input control buses, applies an arithmetic operation, and outputs the result to the output control buses:

```

0dbfs = 1

// it is required to use these names
chn_k "In Control 0", 1
chn_k "In Control 1", 1
chn_k "Out Control 0", 2
chn_k "Out Control 1", 2

instr 1
    // read from input control buses
    kin0 chnget "In Control 0"
    kin1 chnget "In Control 1"

    // some operations
    kvar0 = kin0 + 25
    kvar1 = kin1 * 3

    // write to the output control buses
    // the results of the previous operations
    chnset kvar0, "Out Control 0"
    chnset kvar1, "Out Control 1"
endin

```

3.14 Initializing Class Variables

Class variables are initialized at different stages of execution. The first set, which is related to MetaSounds node execution, will be initialized in the constructor.

```
TCsoundOperator(  
    const FOperatorSettings& InSettings,  
    const FTriggerReadRef& InPlayTrigger,  
    const FTriggerReadRef& InStopTrigger,  
    const FStringReadRef& InCsoundFile,  
    const TArray<FAudioBufferReadRef>& InAudioRefs,  
    const int32& InNumOutAudioChannels,  
    const TArray<FFloatReadRef>& InControlRefs,  
    const int32& InNumOutControlChannels)  
: PlayTrigger(InPlayTrigger)  
, StopTrigger(InStopTrigger)  
, CsoundFile(InCsoundFile)  
, FinishedTrigger(FTriggerWriteRef::CreateNew(InSettings))  
, AudioInRefs(InAudioRefs)  
, BuffersIn()  
, AudioOutRefs()  
, BuffersOut()  
, ControlInRefs(InControlRefs)  
, ControlInNames()  
, ControlOutRefs()  
, ControlOutNames()  
, OpSettings(InSettings)  
, CsoundInstance()  
, SpIndex(0)  
, Spin(nullptr)  
, Spout(nullptr)  
, FirstClearedFrame(InSettings.GetNumFramesPerBlock())  
, OpState(EOpState::Stopped)  
{  
  
    BuffersIn.Empty(InAudioRefs.Num());  
    for (int32 i = 0; i < InAudioRefs.Num(); i++)  
    {  
        BuffersIn.Add(InAudioRefs[i]->GetData());  
    }  
  
    AudioOutRefs.Empty(InNumOutAudioChannels);  
    BuffersOut.Empty(InNumOutAudioChannels);  
    for (int32 i = 0; i < InNumOutAudioChannels; i++)  
    {  
        AudioOutRefs.Add(FAudioBufferWriteRef::CreateNew(OpSettings));  
        BuffersOut.Add(AudioOutRefs[i]->GetData());  
    }  
  
    ControlInNames.Empty(ControlInRefs.Num());
```

```

    for (int32 i = 0; i < ControlInRefs.Num(); i++)
    {
        ControlInNames.Add("In Control " + FString::FromInt(i));
    }

    ControlOutRefs.Empty(InNumOutControlChannels);
    ControlOutNames.Empty(InNumOutControlChannels);
    for (int32 i = 0; i < InNumOutControlChannels; i++)
    {
        ControlOutRefs.Add(FFloatWriteRef::CreateNew());
        ControlOutNames.Add("Out Control " + FString::FromInt(i));
    }
}

```

All read and write data references related to the vertex interface are initialized here. Input references, created during the factory method, are copied using their copy constructors. For output references, the constructor handles their creation. The parameters `InNumOutAudioChannels` and `InNumOutControlChannels` indicate the number of output audio and control data references to be created for that instance.

For the audio channels, the `BuffersIn` and `BuffersOut` two-dimensional arrays have to be set. Each row will point to the raw data of a different audio buffer reference. For the control channels, the `ControlInNames` and `ControlOutNames` have to be set. The rows of these arrays will contain a string with the name of the control channel. This will be achieved by concatenating a string literal with the number of the bus.

The second set of variables pertains to Csound performance and will be initialized in the `Play` method, once the Csound object has compiled the `.csd` file. These variables rely on the Csound object post-compilation. Many of them will be initialized using the results from various methods of the Csound object. Most of these variables are straightforward, as many have been encountered in previous code sections.

```

CsoundNchnlsIn = CsoundInstance.GetNchnlsInput();
CsoundNchnlsOut = CsoundInstance.GetNchnls();
MinAudioIn = BuffersIn.Num() <= CsoundNchnlsIn ?
    BuffersIn.Num() : CsoundNchnlsIn;
MinAudioOut = BuffersOut.Num() <= CsoundNchnlsOut ?
    BuffersOut.Num() : CsoundNchnlsOut;

// ksmpls is the Csound equivalent of the audio buffer lenght,
// or the block size.
CsoundKsmpls = (uint32)CsoundInstance.GetKsmpls();

```

Two class variables remain to be discussed: `OpState` and `FirstClearedFrame`. These variables are explained in Section 3.16.1 and Section 3.16.6 respectively.

3.15 Controlling Csound's Performance

The `CsoundPerformKsmmps` method is responsible for initiating a new performance cycle on the `Csound` object. It resets the `SpIndex`, which tracks the frame index of the spin and spout, to zero. The method of the `Csound` object `PerformKsmmps` returns `false` (or zero) while the performance is ongoing, and `true` (or a non-zero value) when the performance is complete [7]. By repeatedly calling this method until it returns true, an entire score can be executed. This allows external applications to control the `Csound` object's execution and synchronize performance cycles with external audio inputs and outputs.

If `PerformKsmmps` returns `false`, indicating that the performance has finished, the `Stop` method will be invoked, passing the current node frame where the performance is stopped.

```
void CsoundPerformKsmmps(int32 CurrentFrame)
{
    SpIndex = 0;
    if (CsoundInstance.PerformKsmmps() != 0)
    {
        Stop(CurrentFrame);
    }
}
```

Unfortunately, this method introduces a sample delay. The frames written to the spin will be delayed by `ksmps` frames before the corresponding spout receives the updated information. This delay is inherent, as the information must first be placed in the spin and then, once the spin is full, a new performance call is made to `Csound`. Although this delay is unavoidable, it is generally imperceptible to the human ear.

3.16 Node States

The node can exist in various states, depending on the activated triggers, the status of the `Csound` performance, and the occurrence of any errors. The following section explains these states and the conditions that cause transitions between them.

3.16.1 Enumerating Operator States

When executing the node, it can be in one of three possible states: `Stopped`, `Playing`, or `Error`. An enumeration (enum) is used to track the current state of the node.

```
template<typename DerivedOperator>
class METAC SOUND_API TCsoundOperator :
    public TExecutableOperator<DerivedOperator>
{
    (...)

    enum class EOpState : uint8
    {
```

```

        Stopped,
        Playing,
        Error
    };

    EOpState OpState;

    (...)
}

```

Initially, the `OpState` is set to `Stopped` in the class constructor.

3.16.2 FTrigger Data References

An `FTrigger` data reference provides sample-accurate trigger detection. It contains an array of integers, with each integer representing the frame index where a trigger occurred [11]. For example, if a specific execution block contains two values (10 and 25) in the `FTrigger` array, this indicates that within this block of frames, triggers were signaled at frames 10 and 25.

3.16.3 Play State

If the node is not in the error state, the `Execute` method will listen for the play trigger. When the play trigger is activated, and if the node was previously stopped, the Csound file will start performing. If the node was already playing, the Csound performance will be rewound to the beginning and continue playing, effectively resetting the performance.

In the `Execute` method, a main loop iterates through all the frames. To integrate the `FTrigger` data structure into the loop while maintaining sample-accurate precision, the frame index is compared with the values in the `FTrigger` array. The following code demonstrates this design for the `PlayTrigger` reference. If a play trigger is detected, the `Play` method is called, passing the current frame index as a parameter.

```

for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
{
    for (int32 i = 0; i < PlayTrigger->Num(); i++)
    {
        if ((*PlayTrigger)[i] == f)
        {
            Play(f);
            (...)
            break;
        }
    }
    (...)
    if (OpState != EOpState::Playing)
    {
        continue;
    }
}

```

```

    }

    // Audio operations
    (...)
}

```

In the `Execute` method, after checking the `PlayTrigger`, the subsequent audio operations are carried out. However, these operations should only occur if the node's state is `Playing`. If the state is anything other than `Playing`, the `continue` statement is used to skip the remaining content inside the loop for the current iteration, effectively moving to the next frame's operations.

Parts of the `Play` method have been discussed earlier. This method compiles the `.csd` file and initializes variables related to the Csound performance. One key aspect not yet covered is the change of the `OpState` enum to `Playing`. Additionally, there is some error handling, which will be discussed in detail in Section 3.16.7. Finally, the method performs an initial Csound performance cycle and handles the control channels (see Sections 3.15 and 3.13).

```

void Play(int32 CurrentFrame)
{
    (...)
    const int32 ErrorCode =
        CsoundInstance.Compile(FullPathANSI, SrOptionANSI, "-n");
    if (ErrorCode != 0)
    {
        (...)
        return;
    }
    else
    {
        OpState = EOpState::Playing;
    }
    (...)
    SetInputControlChannels();
    CsoundPerformKsmps(CurrentFrame);
    if (OpState == EOpState::Playing)
    {
        GetOutputControlChannels();
    }
}

```

3.16.4 Stop State

There are two ways in which the node's state can transition to `Stopped`. The first occurs when the `Stop` event is triggered while the previous state was `Playing`. Similar to the handling of the `PlayTrigger` data reference described earlier, each frame in the `Execute`

method is compared against the values in the **FTrigger** array. This ensures that when a **Stop** trigger is detected, the node's state is updated to **Stopped**.

```
for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
{
    (...)
    for (int32 i = 0;
        i < StopTrigger->Num() && OpState != EOpState::Stopped;
        i++)
    {
        if ((*StopTrigger)[i] == f)
        {
            Stop(f);
            break;
        }
    }
    (...)
}
```

The node can also transition to **Stopped** if the Csound performance has ended, as detailed in Section 3.15. This scenario will similarly invoke the **Stop** method.

The **Stop** method performs a few key functions. It updates the **OpState** enum to **Stopped**, clears the audio buffers to remove any residual data from previous calls, and invokes the **OnFinished** event. Details on the buffer clearing process and the **OnFinished** event will be covered in the following sections.

```
void Stop(int32 StopFrame = 0)
{
    OpState = EOpState::Stopped;

    ClearChannels(StopFrame);
    FinishedTrigger->TriggerFrame(StopFrame);
}
```

3.16.5 On Finished Event

As previously explained, the vertex interface of the node includes a trigger output parameter named *On Finished*. This trigger signals when the node transitions from the **Playing** state to the **Stopped** state. This transition is managed by invoking the **TriggerFrame** method on the **FinishedTrigger** data reference. The frame index at which the node stopped must be provided to maintain sample-accurate precision. This process is demonstrated in the previous section.

At the beginning of the **Execute** method, the **AdvanceBlock** method is called for the **FinishedTrigger** data reference. This method shifts all triggered frames within the reference by the block size [12]. Shifting causes the frame decrease in value. When they eventually become negative, they disappear from the **FTrigger** array as they no longer represent valid frame indexes. By advancing by a block size, all triggered frames in the

array are effectively set to negative values, resetting the data. This step ensures that the output data is updated correctly and avoids repeatedly sending the same information in each `Execute` call.

```
FinishedTrigger->AdvanceBlock();
```

3.16.6 Clearing Channels

The `ClearChannel` method is responsible for setting all or part of the audio buffers, as well as the float inputs and outputs, to zero. It takes a single parameter, which indicates the frame index from which the clearing process should begin. This is particularly useful when only a portion of the buffer needs to be cleared, such as when a `Stop` event is triggered at a specific frame. In such cases, the samples before that frame should remain unchanged, while the samples after it should be cleared.

When clearing the entire buffer, the `Zero` method can be called on the audio buffer data references. If only part of the buffer needs to be cleared, each buffer is manually cleared using a `for` loop.

```
void ClearChannels(int32 StartClearingFrame = 0)
{
    if (FirstClearedFrame <= StartClearingFrame)
    {
        // Already cleared
        return;
    }

    for (int32 i = 0; i < AudioOutRefs.Num(); i++)
    {
        if (StartClearingFrame == 0)
        {
            AudioOutRefs[i]->Zero(); // Clear the whole buffer
        }
        else
        {
            // Clear from StartClearingFrame integer to the end of the buffer
            for (int32 f = StartClearingFrame;
                f < OpSettings.GetNumFramesPerBlock(); f++)
            {
                BuffersOut[i][f] = 0.;
            }
        }
    }

    FirstClearedFrame = StartClearingFrame;

    for (int32 i = 0; i < ControlOutRefs.Num(); i++)
    {
        *ControlOutRefs[i] = 0.;
    }
}
```

```
    }  
}
```

This method will be called during each **Execute** call when the node is in a **Stopped** state. To avoid unnecessarily clearing frames that have already been cleared, a new class variable is introduced to keep track of the first frame that was cleared in previous calls to this method. This optimization prevents redundant operations and improves efficiency. Consequently, when the node transitions from **Stopped** to **Playing**, this variable must be reset to a value equal to or greater than the block size. This ensures that when the node returns to the **Stopped** state, it can clear the buffers correctly.

```
void Play(int32 CurrentFrame)  
{  
    (...)  
    FirstClearedFrame = OpSettings.GetNumFramesPerBlock();  
}
```

It is possible that during the **Execute** call where the node transitions from **Playing** to **Stopped**, some frames may still contain data. However, in the subsequent call, the entire buffer should be cleared. For this reason, **ClearChannels** is called at the beginning of the **Execute** method. A zero value is passed as the parameter, ensuring that the entire length of the audio channels is cleared.

```
void Execute()  
{  
    (...)  
    else if (OpState == EOpState::Stopped)  
    {  
        ClearChannels(0);  
    }  
    (...)  
}
```

3.16.7 Error State

When compiling the **.csd** file, errors may occur due to various reasons, such as syntax errors within the file or the file not being found. If an error arises, the node should cease all operations and take no further action. Therefore, the third possible state of the node, in addition to **Stopped** and **Playing**, is **Error**. In the **Play** method, if an error is detected, the **OpState** enum is updated accordingly, and the method halts its execution by invoking **return**.

```
void Play(int32 CurrentFrame)  
{  
    (...)  
    const int32 ErrorCode =  
        CsoundInstance.Compile(FullPathANSI, SrOptionANSI, "-n");  
    if (ErrorCode != 0)
```

```

{
    UE_LOG(LogMetaSound, Error,
        TEXT("Not able to compile Csound file: \"%s\"), *FullPath);
    OpState = EOpState::Error;
    return;
}
(...)
}

```

The `UE_LOG` macro is used to send messages to the console in the Unreal Editor [12]. The `Error` log category indicates that the message is indeed an error.

In the `Execute` method, after invoking the `Play` method or at the start of a new call, the `OpState` should be checked. If the state is `Error`, the method should return immediately, preventing further execution.

```

void Execute()
{
    if (OpState == EOpState::Error)
    {
        // If error, do nothing
        return;
    }
    (...)
    for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
    {
        for (int32 i = 0; i < PlayTrigger->Num(); i++)
        {
            if ((*PlayTrigger)[i] == f)
            {
                Play(f);
                // Check for error after Csound compilation
                if (OpState == EOpState::Error)
                {
                    return;
                }
                break;
            }
        }
    }
    (...)
}
}

```

3.16.8 Node State Flux Diagram

After explaining the three possible states of the node and the methods associated with them, a flux diagram illustrating all possible transitions can be helpful for visualizing the process. This is depicted in Figure 3.8.

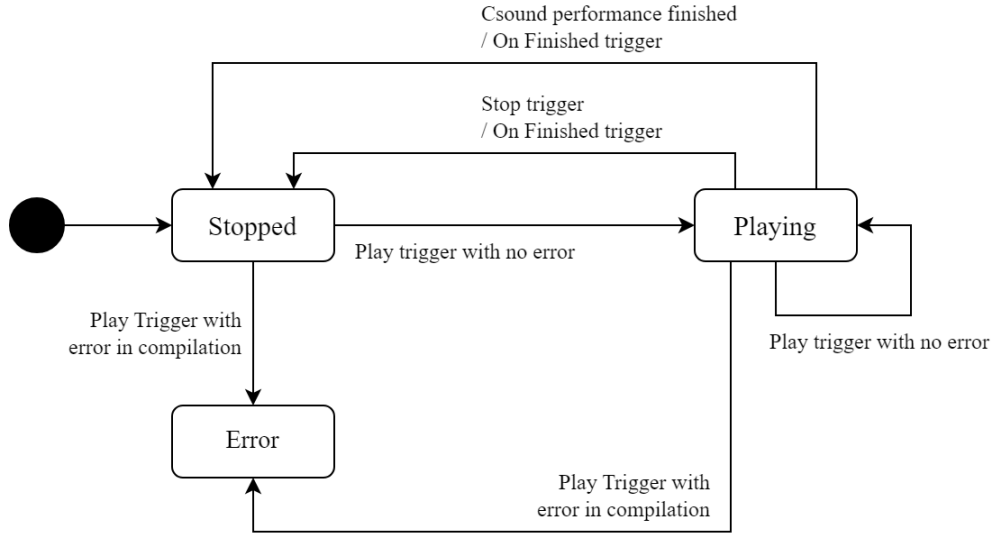


Figure 3.8: State transition diagram of the Csound node.

3.17 Full Implementation of the Execute Method

Now that we have covered all the operations related to audio processing and examined the individual components of the Execute method, it is time to present the entire method in full. This will provide a comprehensive perspective on how all the parts work together to handle the node's execution efficiently.

```

void Execute()
{
    if (OpState == EOpState::Error)
    {
        return;
    }
    else if (OpState == EOpState::Stopped)
    {
        ClearChannels(0);
    }

    FinishedTrigger->AdvanceBlock();

    for (int32 f = 0; f < OpSettings.GetNumFramesPerBlock(); f++)
    {
        for (int32 i = 0; i < PlayTrigger->Num(); i++)
        {
            if ((*PlayTrigger)[i] == f)
            {

```

```

        Play(f);
        if (OpState == EOpState::Error)
        {
            return;
        }
        break;
    }
}

for (int32 i = 0;
     i < StopTrigger->Num() && OpState != EOpState::Stopped;
     i++)
{
    if ((*StopTrigger)[i] == f)
    {
        Stop(f);
        break;
    }
}

if (OpState != EOpState::Playing)
{
    continue;
}

for (int32 i = 0; i < MinAudioIn; i++)
{
    Spin[SpIndex * CsoundNchnlsIn + i] = (double)(BuffersIn[i][f]);
}

for (int32 i = 0; i < MinAudioOut; i++)
{
    BuffersOut[i][f] = (float)Spout[SpIndex * CsoundNchnlsOut + i];
}

if (++SpIndex >= CsoundKsmps)
{
    SetInputControlChannels();
    CsoundPerformKsmps(f);
    if (OpState == EOpState::Playing)
    {
        GetOutputControlChannels();
    }
}
}
}

```

3.18 Ensuring Node Visibility and Functionality in MetaSounds

Now that we have thoroughly covered the core methods, we will delve into additional methods that are crucial for making the node visible and functional within the MetaSounds environment. These methods often utilize the `GET_PARAM` macros discussed in Section 3.9.2. These macros are essential for retrieving information related to the vertex interface, other metadata from the node, and static information in the `DerivedOperator` type, thanks to the use of CRTP.

3.18.1 GetNodeInfo Method

The `GetNodeInfo` function is a key requirement for the node, as it gathers and returns essential metadata [17]. This includes the class name, complete with the MetaSound namespace, which determines where the node will appear within the MetaSound editor's subcategories. The function also supplies other critical details, such as the version number, a localizable display name and description, the author's name, and a user prompt for instances when the node is missing.

Crucially, `GetNodeInfo` also returns the default vertex interface definition. This definition outlines the node's input and output objects and their respective types, effectively shaping how the node interacts with other components.

```
static const FNodeClassMetadata& GetNodeInfo()
{
    auto CreateNodeClassMetadata = []() -> FNodeClassMetadata
    {
        FNodeClassMetadata Info;
        Info.ClassName = DerivedOperator::GetClassName();
        Info.MajorVersion = 1;
        Info.MinorVersion = 0;
        Info.DisplayName = DerivedOperator::GetDisplayName();
        Info.Description = DerivedOperator::GetDescription();
        Info.Author = TEXT("Albert Madrenys Planas");
        Info.PromptIfMissing = Metasound::PluginNodeMissingPrompt;
        Info.DefaultInterface = DeclareVertexInterface();
        Info.CategoryHierarchy = {
            LOCTEXT("MetaCsound_NodeCategory", "Csound") };

        return Info;
    };

    static const FNodeClassMetadata Metadata = CreateNodeClassMetadata();
    return Metadata;
}
```

Although the `GetNodeInfo` method is part of the template base class, it benefits from the static polymorphism enabled by CRTP. This allows access to static methods in the `DerivedOperator` as if they were virtual functions. This is advantageous because each

`DerivedOperator` can have its own class name, display name, and description retrieval functions without necessitating duplication of the entire `GetNodeInfo` function code for each derived operator.

The `GetNodeInfo` method utilizes the `DeclareVertexInterface` static function, which will be detailed in the following section.

Note the use of a *lambda expression* and a static constant variable initialized by this lambda. A lambda expression is a concise way to define an anonymous function within the code [23]. It is often employed when a small, quick function is required without the overhead of a full function declaration.

By leveraging a lambda expression, the computation or initialization of a static constant variable can be deferred until it is first needed. This is particularly useful for values that are costly to compute, dependent on runtime conditions, or should be calculated only when the function is invoked. In this case, the `Metadata` variable will be computed only once, at its first use. Subsequent calls to `GetNodeInfo` will return the precomputed `Metadata` value without recalculating it.

3.18.2 DeclareVertexInterface Method

To properly define a MetaSound node, it is crucial for the graph builder to comprehend the vertices involved in constructing the node. To streamline this process, it's beneficial to define a static local helper function that creates a `FVertexInterface` object [17]. This function is referenced in multiple parts of the node definition, thus centralizing the vertex construction logic in a single location.

```
static const FVertexInterface& DeclareVertexInterface()
{
    using namespace NodeParams;

    FInputVertexInterface InputVertex;
    InputVertex.Add(TInputDataVertex<FTrigger>
        (METASOUND_GET_PARAM_NAME_AND_METADATA(PlayTrig)));
    InputVertex.Add(TInputDataVertex<FTrigger>
        (METASOUND_GET_PARAM_NAME_AND_METADATA(StopTrig)));
    InputVertex.Add(TInputDataVertex<FString>
        (METASOUND_GET_PARAM_NAME_AND_METADATA(CsoundFile)));
    for (int32 i = 0; i < DerivedOperator::NumAudioChannelsIn; i++)
    {
        InputVertex.Add(TInputDataVertex<FAudioBuffer>
            (METASOUND_GET_PARAM_NAME_WITH_INDEX_AND_METADATA(InA, i)));
    }

    for (int32 i = 0; i < DerivedOperator::NumControlChannelsIn; i++)
    {
        InputVertex.Add(TInputDataVertex<float>
            (METASOUND_GET_PARAM_NAME_WITH_INDEX_AND_METADATA(InK, i)));
    }
}
```

```

FOutputVertexInterface OutputVertex;
OutputVertex.Add(TOutputDataVertex<FTrigger>
    (METASOUND_GET_PARAM_NAME_AND_METADATA(FinTrig)));
for (int32 i = 0; i < DerivedOperator::NumAudioChannelsOut; i++)
{
    OutputVertex.Add(TOutputDataVertex<FAudioBuffer>
        (METASOUND_GET_PARAM_NAME_WITH_INDEX_AND_METADATA(OutA, i)));
}

for (int32 i = 0; i < DerivedOperator::NumControlChannelsOut; i++)
{
    OutputVertex.Add(TOutputDataVertex<float>
        (METASOUND_GET_PARAM_NAME_WITH_INDEX_AND_METADATA(OutK, i)));
}

static const FVertexInterface VertexInterface(InputVertex, OutputVertex);
return VertexInterface;
}

```

Note how the `VertexInterface` return variable is constructed from both the `InputVertex` and `OutputVertex` components. Additionally, observe the use of indexed versions of the `METASOUND_GET_PARAM` macros to retrieve multiple vertices, where the only distinction between them is the index used for string formatting. The application of CRTP is also noteworthy, as it allows access to static values from the `DerivedOperator` to ensure that each instance of the template maintains the correct configuration for input and output channels, including audio and control channels.

3.18.3 BindInputs and BindOutputs Methods

These virtual functions are what allow the MetaSound graph to interact with the inputs and outputs of the nodes, respectively, via returning data reference collections [17].

These functions have pretty simple implementations, utilizing the function `BindReadVertex` with the name of the vertex retrieved by a macro and the private read/write data reference as parameters.

```

virtual void BindInputs
    (Metasound::FInputVertexInterfaceData& InOutVertexData) override final
{
    using namespace NodeParams;

    InOutVertexData.BindReadVertex(
        METASOUND_GET_PARAM_NAME(PlayTrig), PlayTrigger);
    InOutVertexData.BindReadVertex(
        METASOUND_GET_PARAM_NAME(StopTrig), StopTrigger);
    InOutVertexData.BindReadVertex(
        METASOUND_GET_PARAM_NAME(CsoundFile), CsoundFile);
}

```

```

    for (int32 i = 0; i < AudioInRefs.Num(); i++)
    {
        InOutVertexData.BindReadVertex(
            METASOUND_GET_PARAM_NAME_WITH_INDEX(InA, i), AudioInRefs[i]);
    }

    for (int32 i = 0; i < ControlInRefs.Num(); i++)
    {
        InOutVertexData.BindReadVertex(
            METASOUND_GET_PARAM_NAME_WITH_INDEX(InK, i), ControlInRefs[i]);
    }
}

virtual void BindOutputs
    (Metasound::FOutputVertexInterfaceData& InOutVertexData) override final
{
    using namespace NodeParams;

    InOutVertexData.BindReadVertex(
        METASOUND_GET_PARAM_NAME(FinTrig), FinishedTrigger);

    for (int32 i = 0; i < AudioOutRefs.Num(); i++)
    {
        InOutVertexData.BindReadVertex(
            METASOUND_GET_PARAM_NAME_WITH_INDEX(OutA, i), AudioOutRefs[i]);
    }

    for (int32 i = 0; i < ControlOutRefs.Num(); i++)
    {
        InOutVertexData.BindReadVertex(
            METASOUND_GET_PARAM_NAME_WITH_INDEX(OutK, i), ControlOutRefs[i]);
    }
}

```

Observe that, unlike other static functions within this template class, these two functions are non-static. As a result, they can leverage the lengths of the arrays to iterate through the vertex interface, rather than relying on static expressions from the `DerivedOperator`.

3.18.4 Facade Class

The final step is to create and register the node to make it available in the MetaSound Editor. To streamline this process, we can create a class that inherits from the helper class `FNodeFacade`, which automates much of the boilerplate code needed for new MetaSound nodes [17].

For each derived operator, there should be a corresponding `FNodeFacade` derived class. This approach is logical because we need to register the specific nodes rather than the template itself.

```
class METASOUND_API FCsoundNode2 : public FNodeFacade
```

```

{
public:
    FCSoundNode2(const FNodeInitData& InitData) : FNodeFacade(
        InitData.InstanceName, InitData.InstanceID,
        TFacadeOperatorClass<FCsoundOperator2>()
    )
    { }
};

```

The example provided illustrates the registration process for `FCsoundOperator2`. However, similar declarations can be found in the header files for each derived operator.

3.18.5 Node Registration

To register the `FNodeFacade` derived classes into the environment, use the `METASOUND_REGISTER_NODE` macro. This macro requires the node class as its sole parameter. Each derived operator must be explicitly registered in its respective header file.

```
METASOUND_REGISTER_NODE(FCSoundNode2);
```

Lastly, in addition to using the `METASOUND_REGISTER_NODE` macro, it's essential to invoke the following function when initializing the `MetaCsound` module (see Section 3.6.8). This ensures that the `MetaSounds` frontend will recognize and register any new nodes defined within the module.

```

void FMetaCsoundModule::StartupModule()
{
    FMetasoundFrontendRegistryContainer::Get()->RegisterPendingNodes();
    (...)
}

```

3.19 Extending MetaCsound

The three available `Csound` nodes (stereo, quadraphonic, and octophonic) cover a wide range of needs. However, developers may require a custom `Csound` node with a more specific vertex interface. Because the base template operator will be public and accessible to other Unreal modules that include `MetaCsound` as a dependency, developers can easily create new derived operators. Implementing a derived operator is straightforward: it involves setting the number of input and output audio and control channels and defining some static functions to provide descriptions and names.

Additionally, each new derived node operator must be accompanied by its corresponding facade and the necessary register node macro. This makes `MetaCsound` a highly accessible and easily extendable plugin.

3.20 Chapter Conclusions

This chapter has detailed the implementation of the Unreal Engine plugin for Csound, focusing on key aspects such as spin and spout indexing, and the handling of audio and control channels. Effective indexing ensures accurate synchronization between Csound and MetaSounds, while the factory method facilitates the creation and initialization of nodes, adapting to various input and output configurations. The handling of `.csd` files (essential for compiling Csound performance scripts) was integrated to ensure seamless interaction between Csound's runtime environment and Unreal Engine. The use of CRTP further streamlines the process by allowing static polymorphism, enabling derived operators to have distinct properties and behaviors without redundant code.

In addition to the core implementation, the plugin's architecture plays a critical role. It leverages API macros for symbol visibility, utilizes UBT for managing build settings and dependencies, and configures the `.uplugin` file for module metadata. These architectural elements, along with the factory method and CRTP, ensure the plugin's effective integration and stability within the Unreal Engine environment. Clear design goals were established to ensure that the plugin meets relevant needs, providing a robust and functional tool. The handling of node states and triggers, including transitions and error conditions, is crucial for maintaining responsive audio processing and ensuring the reliability of the Csound integration.

4 Testing and Discussion

This chapter presents the testing and evaluation of the developed plugin. The chapter begins by outlining the testing procedures and results, followed by a discussion on whether the outcomes align with the expectations set during the design phase. The discussion will cover what has been successfully implemented, what remains incomplete, and the challenges encountered throughout the development process. Potential improvements for future iterations will also be proposed.

The development and testing were conducted using Unreal Engine version 5.3.2 and Csound version 6.18.0, ensuring compatibility and functionality within these specific versions of the software.

4.1 Examples

In order to test if the new nodes are working, I have been working with three MetaSound graphs, at ultimately have become examples on how to use the plugin.

To test the functionality of the new nodes, three MetaSound graphs have been developed as practical examples of the plugin in use. These examples leverage the *FirstPersonTemplate* provided by Unreal Engine when creating a new project. This template includes pre-built player movement and actions such as jumping, picking up a weapon, and shooting (see Fig. 4.1). By utilizing this template, we can focus exclusively on the audio aspects of the project without needing to manage player movement, physics, and shooting mechanics.

4.1.1 Wind Test

The first example is the *Wind* MetaSound, which serves as the simplest illustration of the plugin’s capabilities. This example is inspired by a tutorial that demonstrated creating a wind effect using pink noise generators and low-pass filters [11]. The tutorial’s approach involved modulating the cutoff frequency of the filters with a low-frequency oscillator (LFO). For this implementation, this DSP logic has been transferred from the MetaSound graph into the `.csd` file, simplifying the graph by offloading the signal processing tasks to the `.csd` file.

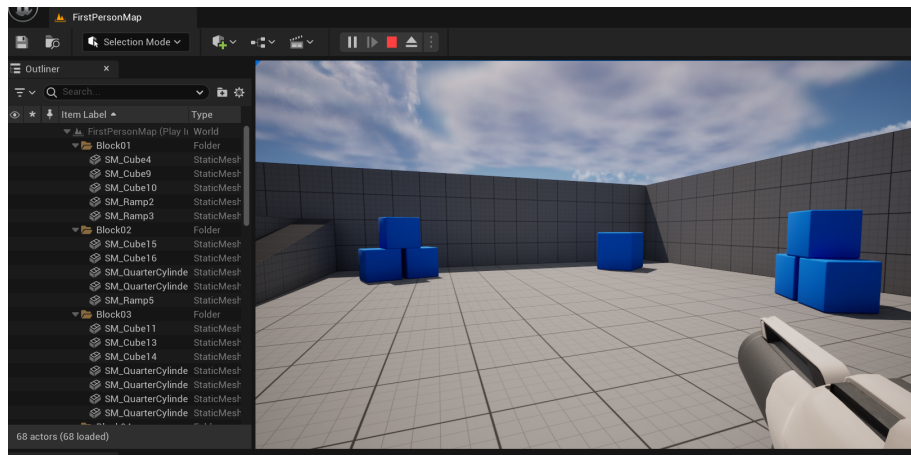


Figure 4.1: Playing a simple level of the FirstPerson Unreal template.

```

<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
nchnls = 2
Odbfs = 1

chn_k "In Control 0", 1

instr 1
  kamp chnget "In Control 0"
  kamp = (kamp*4)+0.5

  apink1 pinker
  apink2 pinker
  kcutfreq = oscili(30, 0.1) + 50 // between 20 and 80

  kport port kamp, 0.05
  atone1 tone apink1, kcutfreq
  atone2 tone apink2, kcutfreq

  outs atone1*kport, atone2*kport
endin

schedule(1, 0, -1)

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>

```

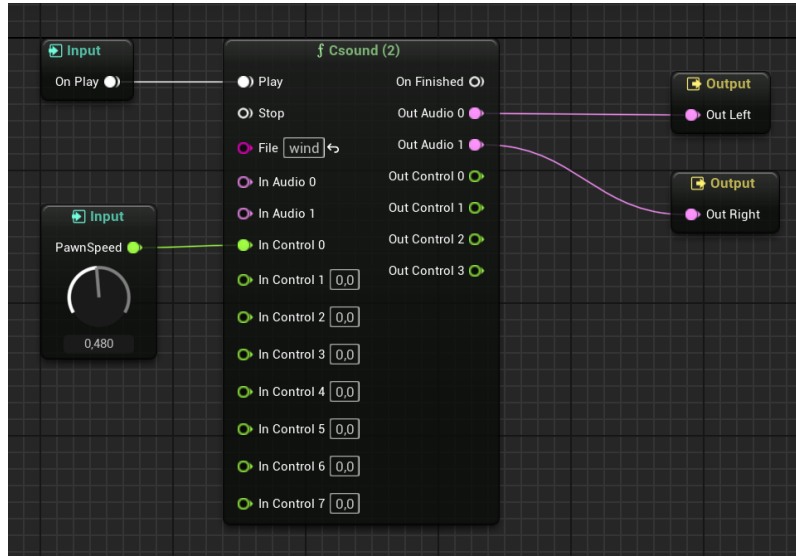


Figure 4.2: MetaSounds graph of the wind example.

Even though this example is relatively straightforward, it includes important aspects worth noting. Firstly, the audio output is stereo, allowing us to test the multi-channel capabilities of the plugin. Secondly, there is a control-rate input directly tied to the player’s movement speed; as the player moves, the output gain increases. While simple, this example provides a solid foundation and yields satisfying results. Its successful performance validates the functionality of the plugin, paving the way for more complex tests.

In Figure 4.2, the MetaSound graph is illustrated, and Figure 4.3 shows the Level Blueprint responsible for spawning the MetaSound graph and updating the player’s speed.

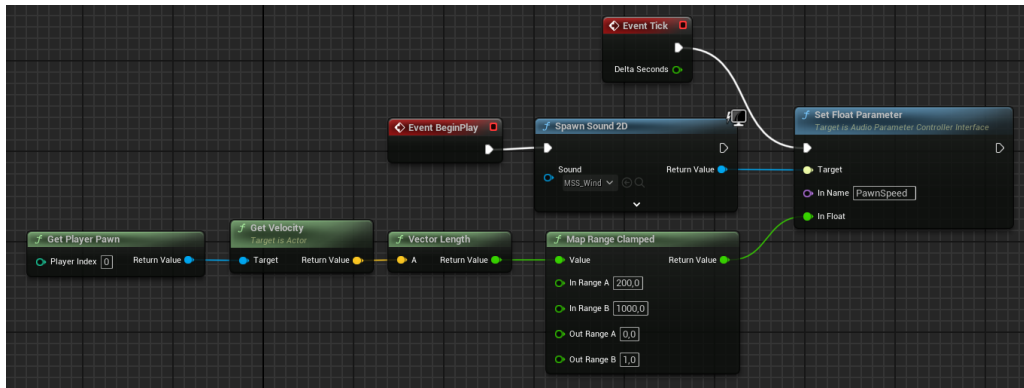


Figure 4.3: Level Blueprint that sends the player speed to MetaSounds.

4.1.2 Bomb Test

This example represents a significant step up in complexity compared to the previous one. It is also based on the same tutorial [11]. While the earlier example focused on continuous sound linked to the Level Blueprint, this new example introduces a sound designed to emerge and fade away after a set period.

```
<CsoundSynthesizer>
<CsOptions>
</CsOptions>
<CsInstruments>
nchnls = 1
0dbfs = 1

instr 1
    ain in

    kcutfreq expseg 100, 2, 2000, 1, 2000
    alp butterlp ain, kcutfreq

    out alp
endin

schedule(1, 0, -1);

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

In this scenario, the output is mono rather than stereo, eliminating the need for multichannel handling. The `.csd` file remains straightforward, reading from the input audio channel, applying a filter, and outputting the result. The primary focus of this example is to demonstrate the functionality of the start, stop, and finished triggers.

The sound will be a *bomb* explosion. When the player shoots with the weapon, a projectile will be instantiated. The Blueprint of that projectile is the one that spawns the MetaSounds graph, and gets attached to the position of the projectile for the first phase. The first phase will be a fire sparkling sound. When the projectile surpasses a certain amount of time existing, or when it colides with one of the blue boxes in the level, it will explode. The projectile will desappear, but on doing that, it will make the metasound instance detach from the projectile, and trigger the “explode” trigger. This will signal the change of phase. The second phase is the bomb exploding sound. Once the sound has finished, the graph will be deleted thanks to the On Finished output of the Oneshot interface. The Blueprint logic of this test can be seen in Figure 3.2.

The sound in this example simulates a *bomb* explosion. When the player shoots a weapon, a projectile is instantiated. The Blueprint of the projectile is responsible for spawning the MetaSounds graph and attaching it to the projectile’s position during the initial phase.

This phase generates a fire-sparkling sound. If the projectile exists for a certain amount of time or collides with one of the blue boxes in the level, it triggers an explosion. The projectile then disappears, causing the MetaSound instance to detach from the projectile and trigger the `explode` event. This event signals the transition to the second phase, where the bomb explosion sound is played. Once the sound finishes, the graph is deleted via the `On Finished` output of the `OneShot` interface. The Blueprint logic for this test is illustrated in Figure 3.2.

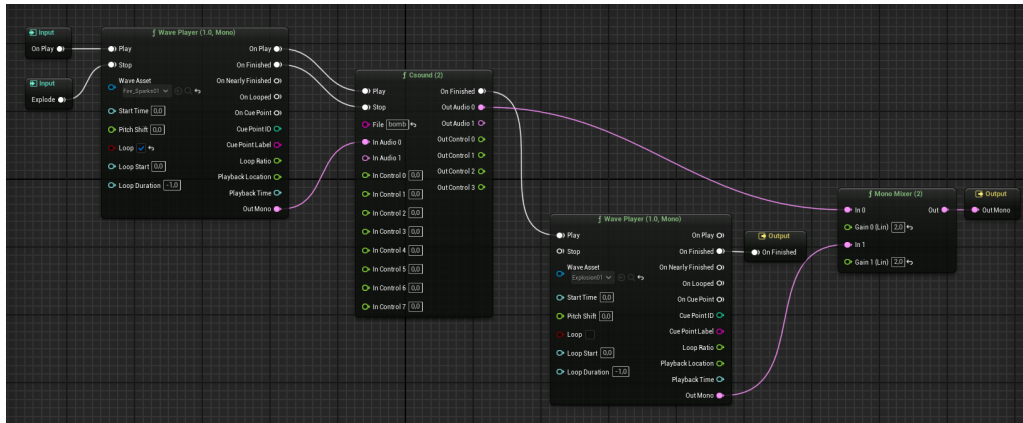


Figure 4.4: MetaSounds graph of the bomb example.

The two phases in this example are represented by two wave players in the MetaSound graph, each responsible for playing a sound file corresponding to a phase. When the `explode` trigger activates, it signals the first wave player to stop. As a result, the `On Finished` trigger signals the `Csound` node to stop as well. This stopping action then triggers the second wave player to start playing the explosion sound file. This chain reaction-style design is advantageous in MetaSounds as it helps reduce visual clutter and maintain a cleaner graph structure (see Fig. 4.4). The `Csound` node responds as expected, confirming the success of this test.

4.1.3 Third Test

The third test, referred to simply as *test* within the project, is the most frequently used example for testing the new nodes. Unlike the previous examples, this test is not tied to any gameplay elements, as there is no Blueprint that spawns the MetaSounds object or interacts with it. Instead, this test runs entirely within the MetaSound graph editor, focusing on the core functionality of the `Csound` node. As long as the node operates correctly within the graph, the test is considered successful.

In this example, the `.csd` file performs minimal processing. It reads from two audio channels, modulates their amplitude using two offset LFOs, and outputs the result. Addi-

tionally, the amplitude is modified by reading the first two input control buses, and the same values are written to the first two output control buses. This test is designed to ensure that the Csound node correctly handles both audio-rate and control-rate data, a critical aspect of the project's functionality.

```
<CsoundSynthesizer>
<CsOptions>
--sample-rate=41100
</CsOptions>
<CsInstruments>
nchnls = 2
nchnls_i = 2
0dbfs = 1

chn_k "In Control 0", 1
chn_k "In Control 1", 1
chn_k "Out Control 0", 2
chn_k "Out Control 1", 2

instr 1
  ain0, ain1 inch 1, 2

  aosc0 = oscili(0.5, 1)+0.5
  aosc1 = oscili(0.5, 1, -1, 0.5)+0.5

  kamp0 chnget "In Control 0"
  kamp1 chnget "In Control 1"

  chnset kamp0, "Out Control 0"
  chnset kamp1, "Out Control 1"

  out ain0*aosc0*kamp0, ain1*aosc1*0.2*kamp1
endin

schedule(1, 0, -1);

</CsInstruments>
<CsScore>
</CsScore>
</CsoundSynthesizer>
```

To clearly differentiate both audio channels, one channel is feeded with a noise generator, while the other one with a sawtooth wave oscillator. The first output of the node goes to the final stereo mixer, but the second channel of the mixer can be fed with the second output of the Csound node, or by the original sawtooth wave, but with its amplitude controlled by the output control-rate of the Csound node. That way, we can test if the out control-rate is working, which is something that we haven't test yet.

To clearly differentiate the two audio channels, one channel is fed with a noise generator, while the other utilizes a sawtooth wave oscillator. The first output of the node is sent to

the final stereo mixer, while the second channel of the mixer can either receive the second output of the Csound node or the original sawtooth wave, with its amplitude controlled by the output control-rate of the Csound node. This setup allows for testing the functionality of the output control-rate, which hadn't been tested until this point.

Finally, the start and stop events manage the performance, with the finished trigger used to initiate a wave player, similar to the bomb example discussed earlier. Figure 4.5 shows the full MetaSounds graph.

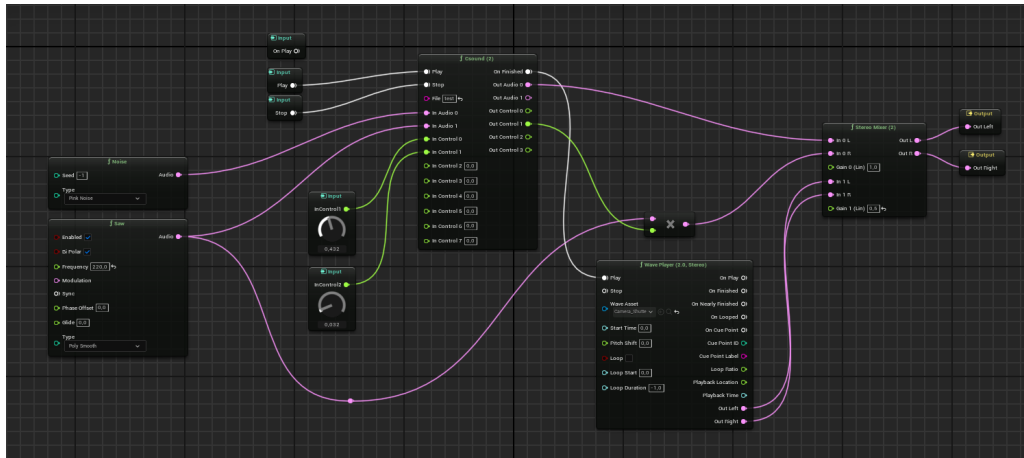


Figure 4.5: MetaSounds graph of the test example.

This third example, although not applicable in a real game scenario, proves to be highly valuable for testing purposes. Everything functions as expected, even when the graph is stressed by rapidly changing control values or repeatedly triggering the play and stop events. This demonstrates that the system is both stable and robust.

4.1.4 Testing with Packaged Game

Two examples have been tested using packaged game versions, and they worked correctly, behaving exactly as they did in the editor. However, when new users execute the packaged game, they must have the Csound DLL installed beforehand, as it is not currently packaged with the game. This is intentional, as it provides the flexibility to choose and control the preferred version of Csound.

4.1.5 Results of the Tests

The quality of the testing is highly satisfactory. Each of the three tests targeted different aspects of the tool's functionality. The first test involved a simple, continuous sound that interacted with the player's speed, with the .csd file handling a significant amount of

DSP operations. The second test featured a short sound that transitioned between phases, controlled by the play, stop, and finished events of the Csound node. Finally, the third test focused on verifying the correct operation of both input and output channels, including audio buffers and k-rate (float) channels.

The results were very positive. All three tests performed as expected, with no bugs or unexpected behavior, even under conditions designed to stress the system.

4.2 Discussion

4.2.1 Evaluation of Final Results

The testing phase demonstrated that the new family of nodes performs reliably under various conditions, including when the game is packaged. The results are consistently positive, reflecting a robust and well-structured implementation.

The core functionality of the node, which processes `.csd` files, works effectively. Users can introduce complex DSP operations via these files, promoting a clean and organized project structure. The node supports both control signals and audio signals, facilitating smooth integration between MetaSounds and Csound environments. It accommodates a range of audio configurations, including stereo, quadraphonic, and octophonic setups. While it may appear restrictive, the available nodes are versatile. For example, mono audio can be achieved using the stereo node, and unused channels are handled appropriately based on the `.csd` file configuration. Additionally, developers requiring more extensive audio channels or control buses can leverage the MetaCsound plugin API to extend the plugin capabilities and create custom operators derived from the base Csound operator, allowing for customized vertex interfaces.

In reviewing the design goals outlined in Section 3.1, we find the following:

- **Novel Integration:** The integration represents a pioneering effort, bridging previously incompatible systems.
- **Relevance:** The integration enhances MetaSounds with Csound’s advanced tools, providing MetaSounds users with new capabilities for crafting interactive audio experiences. The future impact of this tool will hinge on the maturation of MetaSounds beyond its current beta phase. Should MetaSounds advance, MetaCsound and, by extension, Csound, has the potential to become a prominent plugin for expanding the functionalities of MetaSounds.
- **Ease of Use:** MetaCsound is intuitive to use. It does not have lots of complex features, instead it focuses on an intuitive, clear and concise communication between the two systems. Being a new set of MetaSounds nodes, it is very easy and integrated with the rest of the MetaSounds environment, and the Csound code specifics don’t

deviate to the basics of the language, making the tool approachable for developers that come from either of the two environments..

Overall, the final result aligns well with the initial design goals, confirming that the project meets the intended objectives and provides a functional and practical solution.

4.2.2 Difficulties Encountered

The project faced several challenges throughout its development.

One recurring issue involved *linker errors*, stemming from the author's limited experience with Unreal Engine's development environment. These problems were eventually resolved through proper use of API macros and effective dependency management via the plugin descriptor and UBT files.

Another notable challenge encountered was the limited *documentation* available for MetaSounds, which is still in its beta phase. This limitation presented difficulties when developing more complex features, as existing resources were often sparse and insufficient for advanced implementations. While basic node creation can be accomplished using available tutorials, more intricate features required extensive code examination and trial-and-error. Enhanced documentation will be essential for supporting users and developers as MetaSounds continues to evolve and gain broader adoption.

A major obstacle was the implementation of *score* events in Csound. Score events, which schedule new instrument instances, appeared straightforward to integrate initially. However, a major flaw was discovered: while triggers are sample-accurate, string data references are only updated once per execute call. This discrepancy led to synchronization issues, where multiple triggers might be signaled, but only one string would be processed, potentially overriding previous events. Although workarounds were explored, they added significant complexity, leading to the cancellation of this feature. A potential solution would be the development of a new pin type capable of handling *sample-accurate strings*, allowing multiple strings to be processed per execute call.

Another challenge was MetaSounds' inability to create *dynamic vertex interfaces*. The initial design aimed to adapt the node's parameters based on the `.csd` file, displaying only the necessary audio channels and bus parameters. However, MetaSounds requires static interface definitions, which are set using preprocessor directives. Consequently, this dynamic approach was abandoned in favor of using multiple derived operators, each with a predefined vertex interface. While this solution is less flexible than the original concept, it offers a practical compromise. Developers can still create custom derived operators to meet specific needs, adjusting the number of input and output channels as required.

Implementing these derived operators presented its own set of challenges. The project employed *CRTP* to facilitate static polymorphism, enabling the base class to access static methods and expressions from derived operators. This approach effectively managed the differences between derived operators and their common base, achieving a clean and functional design.

4.2.3 Improvements

1. **MIDI Integration:** The initial design considered including MIDI support, as both MetaSounds and Csound handle MIDI messages. Further investigation could explore an elegant method for integrating MIDI communication between these systems.
2. **Cross-Platform Compatibility:** Currently, MetaCsound operates only on Windows, whereas Csound supports Linux and macOS, and Unreal Engine is compatible with macOS and Ubuntu Linux. By accommodating the requirements of different operating systems for linking dynamic libraries in the `.build.cs` file, UBT could facilitate Csound integration on other platforms.
3. **File Management:** The current iteration requires `.csd` files to be placed in a specific folder with a predetermined name. Improving file management could involve adding an option in the plugin preferences to customize the folder name. This folder could then be automatically included in the additional packaging folders, eliminating the need for manual inclusion on each project.
4. **File Browser Support:** The Unreal Editor folder browser currently does not support `.csd` files, leading to their omission. Consequently, users cannot view or manipulate `.csd` files within Unreal Editor. Exploring solutions that do not require creating a new asset type for Unreal, and that are compatible with other Csound frontends, could address this issue. Alternatively, implementing a new node pin to directly accept `.csd` files, similar to how the `WavePlayer` node selects sound files, could be considered.
5. **Score Event Communication:** As noted in Section 4.2.2, implementing support for Csound's score events is a desirable feature. This would require a new pin type capable of handling sample-accurate strings, allowing multiple strings to be processed per frame to represent Csound score events.
6. **DLL Deployment:** Currently, Csound must be installed separately on new systems. While this design choice offers flexibility in version selection, providing an optional and easier installation process would be beneficial. Simplifying the deployment, especially during game packaging, would help avoid inconveniences for end-users or clients who might otherwise need to install Csound manually.

4.3 Chapter Conclusions

In this chapter, the functionality and performance of the MetaCsound plugin have been evaluated through various tests, and the outcomes and challenges have been discussed. The testing phase has involved three distinct MetaSound graphs, each designed to assess different aspects of the plugin's capabilities.

Testing with the packaged game has confirmed that the plugin performs as expected in a final build, maintaining consistency and functionality outside the Unreal Editor environment. The results of the tests have been overall positive, with the plugin demonstrating reliable performance across various scenarios. The robust design has allowed it to manage different types of audio and control data effectively, even under stress testing conditions.

The discussion section has reviewed the quality of the final result, confirming that the design goals have been achieved and that the plugin provides a meaningful integration between MetaSounds and Csound. Challenges such as linker errors, limited documentation, and the creation of dynamic vertex interfaces have been identified and addressed, with appropriate solutions and workarounds provided.

Suggestions for improvements have been made to enhance the plugin's flexibility and usability, including better file management, support for additional operating systems, and simplified DLL deployment. Addressing these areas could further increase the plugin's utility and user experience.

Overall, this chapter demonstrates that MetaCsound is a functional and effective tool for integrating Csound with Unreal Engine's MetaSounds, with a clear path outlined for future enhancements and refinements.

5 Conclusions

In this thesis, the integration of *Csound* within *Unreal Engine*'s MetaSounds was explored to enhance real-time audio processing and provide a robust solution for *adaptive audio* in gaming. The primary objective was to develop a plugin that leverages Csound's advanced audio capabilities within the *MetaSounds* framework, addressing the limitations of existing solutions and improving the user experience for developers.

- **Novel Integration:** Offering a new way to integrate Csound with Unreal Engine, allowing for complex audio processing and real-time interaction.
- **Relevance:** Expanding the functionality of MetaSounds with Csound's powerful audio features, thus providing developers with a more versatile tool for creating immersive audio experiences.
- **Ease of Use:** Ensuring ease of use and seamless integration within the MetaSounds environment, making advanced audio techniques accessible to developers.

The testing phase confirmed that the plugin performs reliably in both the Unreal Editor and packaged game builds, demonstrating its robustness and effectiveness in various scenarios. Despite these successes, there are areas for future improvement, such as enhanced file management, cross-platform support, and integration of additional audio features.

As MetaSounds evolves and gains broader adoption, MetaCsound has the potential to become a key tool in the development of interactive audio within Unreal Engine. The project's source code is available on GitHub at: <https://github.com/AlbertMadrenys/MetaCsoundProject>, providing a resource for ongoing development and community contributions.

In summary, this work fulfills the initial objectives outlined in the introduction and sets a strong foundation for future enhancements, contributing valuable advancements to the field of real-time audio processing in game development.

Bibliography

- [1] ABRAHAM, D., AND GURTOVOY, A. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley Professional, 2004.
- [2] ALEXANDRESCU, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ in-depth series. Addison-Wesley, 2001.
- [3] AUDIOKINETIC. Wwise official website. <https://www.audiokinetic.com/en/wwise/>, 2024. [Online; accessed 31-August-2024].
- [4] BRINKMANN, P., KIRN, P., LAWLER, R., MCCORMICK, C., ROTH, M., AND STEINER, H.-C. Embedding pure data with libpd: Design and workflow. Bauhaus-Universität Weimar, https://www.uni-weimar.de/kunst-und-gestaltung/wiki/PDCON:Conference/Embedding_Pure_Data_with_libpd:_Design_and_Workflow, 2011. [Online; accessed 31-August-2024].
- [5] CABRERA, A., ET AL. CsoundQT Official Site. <https://csoundqt.github.io>, 2022. [Online; accessed 31-August-2024].
- [6] CENTER FOR COMPUTER RESEARCH IN MUSIC AND ACOUSTICS (CCRMA), STANFORD UNIVERSITY. OpenSoundControl Website. <https://ccrma.stanford.edu/groups/osc/>, 2021. [Online; accessed 31-August-2024].
- [7] CSOUND CONTRIBUTORS. Csound API Documentation. <https://csound.com/docs/api/index.html>, 2024. [Online; accessed 31-August-2024].
- [8] CSOUND CONTRIBUTORS. Csound Web-IDE Documentation. <https://ide.csound.com/documentation>, 2024. [Online; accessed 31-August-2024].
- [9] CSOUND CONTRIBUTORS. The Canonical Csound Reference Manual. <https://csound.com/docs/manual/>, 2024. [Online; accessed 31-August-2024].
- [10] CSOUND CONTRIBUTORS. The Csound FLOSS Manual. <https://flossmanual.csound.com>, 2024. [Online; accessed 31-August-2024].
- [11] EPIC GAMES. Metasounds quick start. <https://dev.epicgames.com/documentation/en-us/unreal-engine/metasounds-quick-start>, 2023. [Online; accessed 31-August-2024].

-
- [12] EPIC GAMES. Metasounds quick start guide. <https://dev.epicgames.com/documentation/en-us/unreal-engine/metasounds-quick-start>, 2024. [Online; accessed 31-August-2024].
- [13] EPIC GAMES. Unreal engine documentation. <https://dev.epicgames.com/documentation/en-us/unreal-engine/>, 2024. [Online; accessed 31-August-2024].
- [14] EPIC GAMES. Unreal engine official website. <https://www.unrealengine.com>, 2024. [Online; accessed 31-August-2024].
- [15] FIRELIGHT TECHNOLOGIES PTY. LTD. Fmod official website. <https://www.fmod.com>, 2024. [Online; accessed 31-August-2024].
- [16] GODOT ENGINE CONTRIBUTORS. Godot engine official website. <https://godotengine.org>, 2024. [Online; accessed 31-August-2024].
- [17] LANTZ, A. Creating metasound nodes in c++ quick start. <https://dev.epicgames.com/community/learning/tutorials/ry7p/unreal-engine-creating-metasound-nodes-in-c-quickstart>, 2023. [Online; accessed 31-August-2024].
- [18] LAZZARINI, V., YI, S., FFITCH, J., HEINTZ, J., ØYVIND BRANDTSEGG, AND MC-CURDY, I. *Csound: A Sound and Music Computing System*. Springer International Publishing, Cham, 2016.
- [19] MENDIZABAL, W. Godot-csound Repository. <https://github.com/nonameentername/godot-csound>, 2024. [Online; accessed 31-August-2024].
- [20] MICROSOFT CORPORATION, INC. _declspec Keyword Documentation. <https://learn.microsoft.com/en-us/cpp/cpp/declspec>, 2022. [Online; accessed 31-August-2024].
- [21] PLAYDOTS, INC. UnityPd Repository. <https://github.com/playdots/UnityPd>, 2024. [Online; accessed 31-August-2024].
- [22] PURE DATA CONTRIBUTORS. The Pure Data Site. <https://puredata.info/>, 2023. [Online; accessed 31-August-2024].
- [23] STROUSTRUP, B. *The C++ Programming Language*, 3rd ed. Addison-Wesley, 1997.
- [24] SUPERCOLLIDER CONTRIBUTORS. SuperCollider Documentation Home. <https://docs.supercollider.online/>, 2024. [Online; accessed 31-August-2024].
- [25] UNITY TECHNOLOGIES. Unity Engine Official Website. <https://unity.com/>, 2024. [Online; accessed 31-August-2024].

-
- [26] WALSH, R. CsoundUnity Documentation. <https://rorywalsh.github.io/CsoundUnity/>, 2023. [Online; accessed 31-August-2024].
- [27] WALSH, R., MCCURDY, I., AND BOYLE, G. Cabbage Official Site. <https://cabbageaudio.com/>, 2023. [Online; accessed 31-August-2024].

A Git Repository

The source code for this project is available at the following GitHub repository:

`https://github.com/AlbertMadrenys/MetaCsoundProject`

This repository contains the complete codebase, documentation, and licensing information for the project.

B Documentation

MetaCsound is a plugin for *Unreal Engine* that integrates the audio signal processing engine *Csound* within the *MetaSounds* environment. This integration results in a new set of available nodes for MetaSounds. When executed, they will create a new Csound instance, compile the input Csound file, and start processing the resulting audio.

B.1 Installing

MetaCsound has been developed using Unreal Engine version 5.3.2 and Csound version 6.18.0. Currently, it is only compatible with Windows machines.

1. **Install MetaSounds Plugin:** First, you need to install the MetaSounds plugin in your project, since MetaCsound depends on it. You can do this by searching for it in Edit -> Plugin Manager.
2. **Ensure Csound is Installed:** Make sure that the correct version of Csound is installed in the default location.
3. **Copy MetaCsound Folder:** Download the MetaCsound folder from this repository and copy it into the plugin folder of your project.
4. **Compile the Code:** Close the Unreal Editor and compile the code (most Windows users will use Visual Studio for this). Do not use Unreal Editor's live coding feature for the first compilation with the plugin.
5. **Setup Csound Files:** Open the Editor. In your project content folder, create a new folder called CsoundFiles. Place your .csd files here or in subfolders.
6. **Verify Node Availability:** To check if the new nodes are available, create a new MetaSounds graph and try to add some of the new nodes from the drop-down menu, such as Csound (2).

B.2 Using

Using MetaCsound is straightforward. In your MetaSounds graph, simply add one of the available Csound nodes. All nodes have a similar layout, differing only in the number of input and output audio and control channels.

-
- **Play:** Compiles the Csound file and starts performing. If an error occurs, a message will appear in the console. Triggering **Play** again will restart the performance.
 - **Stop:** Stops the Csound performance. This will trigger **On Finished**.
 - **On Finished:** Triggered when the performance stops, either due to a **Stop** trigger or the Csound performance ending. To keep the performance running continuously, add `f0 z` to the score section of your `.csd` file.
 - **File:** The name of the `.csd` file to be performed. This file must be placed in the **CsoundFiles** folder within the **Content** folder of your project. Subdirectories are allowed and should be specified using the `/` character to indicate the relative path. Do not include the `.csd` extension in the filename. For example, specifying **Subdirectory/Test** corresponds to the file:
`/Content/CsoundFiles/Subdirectory/Test.csd`.
 - **In Audio X:** Input audio channel. The number of input channels in the `.csd` file is determined by defining `nchnls_i`, or defaults to `nchnls` if not explicitly defined. Csound can handle fewer channels than the node allows but cannot handle more. Use a MetaSound node with more audio inputs if needed. In the `.csd` files, input audio channels can be accessed using opcodes like `in` or `ins`.
 - **Out Audio X:** Output audio channel. The number of output channels in the `.csd` file is determined by defining `nchnls`. Csound can handle fewer channels than the node allows but cannot handle more. Use a MetaSound node with more audio outputs if needed. In the `.csd` files, output audio channels can be written using opcodes like `out` or `outs`.
 - **In Control X:** Input control channel. The control channel can be accessed from Csound using a control bus, defined with the same name as the corresponding MetaSounds pin. Define it in Csound using `[chn_k "In Control X", 1]`, where X is the input number. The value can be read using the opcode `chnget`.
 - **Out Control X:** Output control channel. The control channel can be accessed from Csound using a control bus, defined with the same name as the corresponding MetaSounds pin. Define it in Csound using `[chn_k "Out Control X", 2]`, where X is the output number. The value can be written using the opcode `chnset`.

B.3 Examples

Examples can be found in this repository. Three MetaSound graphs are located in **Content/MetaSounds**. Their corresponding `.csd` files are in **Content/CsoundFiles**.

B.4 Packaging your Game

The Csound DLL file is not included in the packaged game. This means new users will need to install Csound separately.

```
Edit -> Project -> Settings -> Packaging ->
      -> Additional Non-Asset Directories To Copy.
```

This option is an array that is initially empty. Add a new element with the value `CsoundFiles`. This step is necessary to include your `.csd` files in the final packaged game.

C License Information

The code for this project is released under the *GNU Lesser General Public License (LGPL) version 2.1*. This license allows for the code to be used and integrated into both open source and proprietary software with some conditions.

C.1 Key Points of LGPL 2.1

1. **Freedom to Use and Modify:** Users can freely use, modify, and distribute the code.
2. **Derivative Works:** Modifications to the LGPL code must also be licensed under LGPL 2.1. However, proprietary software that links to the LGPL code does not need to be open source.
3. **Source Code Availability:** If distributing modified versions of the LGPL code, the source code must be made available under the same license.

C.2 Implications for this Project

- **Integration:** The code can be used in proprietary applications without requiring the proprietary code to be open source.
- **Redistribution:** Users must include the LGPL license with any redistributed versions of the code and provide the source for any modifications.

C.3 Access to Code

The project's source code, including the LGPL license, is available at the following GitHub repository:

`https://github.com/AlbertMadrenys/MetaCsoundProject`