Centres universitaris adscrits a la

Universitat Pompeu Fabra Barcelona

Degree in Videogame Design and Production

Creating a Unity Framework for Scriptable Object Driven Development (SODD)

Alex Ruiz Rabasseda

Tutor: Dr. Enric Sesa Nogueras

TecnoCampus Mataró-Maresme

Acknowledgements

I would like to express my deepest gratitude to my tutor, Enric Sesa, for his invaluable advice and guidance throughout this project.

I am profoundly thankful to my significant other, Andrea Kunze, for her unwavering support, patience, and love during the entire process of this project; even in the moments when I became consumed in my work.

Special thanks to my talented teammates at God Games, Alex Navarro and Alejandro Vega. Alex, the best artist I know, provided the gorgeous art and animations for the sample videogame of this project. Alejandro, a brilliant technical artist and expert in Unity’s shaders, created exceptional shaders and particle systems that significantly enhanced the visual quality of the final game.

Finally, special thanks to Casey Hofland for his innovative adaptation of DocFX for Unity packages. By replicating the behaviour of Unity's internal documentation generation tool for their packages, Casey provided an invaluable solution that Unity package developers had long needed but never received. His contribution significantly eased the process of generating and hosting documentation, saving me countless hours and greatly contributing to the success of this project.

Abstract

This project investigates and expands upon a novel approach to game architecture using ScriptableObjects in Unity. The goal is to create a framework that implements these ideas, establishing a robust foundation for a development workflow based on ScriptableObjects, coined in this project as ScriptableObject Driven Development. To demonstrate the viability of this framework, a videogame has been successfully developed, utilizing the tools provided by the framework.

Resum

Aquest projecte investiga i amplia un enfocament innovador sobre l’ús de ScriptableObjects com a base per l’arquitectura de videojocs a Unity. L’objectiu és crear un framework que implementi aquestes idees, establint una base sòlida cap a una metodologia de desenvolupant centrada en ScriptableObjects, empremtada en aquest projecte com ScriptableObject Driven Development (SODD). Per demostrar la viabilitat d’aquest framework, s’ha desenvolupat amb èxit un videojoc utilitzant totes les eines proporcionades per el framework.

Resumen

Este proyecto investiga y amplia un innovador enfoque sobre el uso de ScriptableObjects como base para la arquitectura de videojuegos en Unity. El objetivo es crear un framework que implemente estas ideas, estableciendo una base sólida hacia una metodología de trabajo centrada en ScriptableObjects, acuñada en este proyecto como ScriptableObject Driven Development (SODD). Para demostrar la viabilidad de este framework, se ha desarrollado con éxito un videojuego utilizando todas las herramientas proporcionadas por el framework.

Contents

List of Figures

List of Tables

Glossary of Terms

API	Application Programming Interface. A set of protocols, routines, and tools for building software applications. APIs specify how software components should interact and are used when programming graphical user interface components.
Build	The result of compiling and packaging a Unity project into an executable form that can be run on a target platform, such as Windows, Mac, Linux, or a specific console.
Component	A modular piece of functionality in Unity that can be attached to GameObjects to define behaviour and appearance.
Data-Driven Design	A design approach where game data is separated from game logic.
Dependency Injection	A design pattern used to implement IoC (Inversion of Control), allowing for the decoupling of dependencies from the objects that use them.
DocFX	A static site generator for producing documentation from source code.
Editor Time	The period when developers are working within the Unity Editor, modifying and configuring the game. Changes made during editor time affect the design and structure of the game but are not executed in real-time.
Framework	A reusable set of libraries or classes for a software system or subsystem. Frameworks provide particular functionality and dictate the architecture of applications developed with them.

Inspector	A window in the Unity Editor that allows developers to view and edit properties of selected objects.
Inspector-Friendly Tools	Tools in Unity that leverage the Inspector to create user-friendly interfaces for editing components and assets.
IoC	Inversion of Control. A design principle in which the control flow of a program is inverted, meaning that custom-written portions of a program receive the flow of control from a generic framework.
LTS	Long-Term Support. A version that receives updates and fixes for an extended period, typically two years. LTS versions are recommended for projects that require a stable and reliable development environment over a long duration.
MonoBehaviour	The base class from which every Unity script derives. It grants the use of Unity's built-in lifecycle methods.
Play Mode	A mode in the Unity Editor that executes the game in real-time as if it were running as a standalone build.
Persistence	The characteristic of state that outlives the process that created it.
Prefab	A pre-configured GameObject that acts as a template to be reused and instantiated across multiple scenes.
Runtime	The period during which the game is actively running, either within the Unity Editor's Play Mode or as a standalone build. Changes and interactions that occur during runtime are part of the game's live execution environment.
ScriptableObject	A modular piece of functionality in Unity that can exist in memory during runtime without having to be attached to a GameObject.

SODD	ScriptableObject Driven Development. A development methodology that utilizes ScriptableObjects to manage game data, events, and logic in Unity.
Transform	A component in Unity that represents the position, rotation, and scale of an object.
Unity Editor	The primary interface for building games and applications in Unity, where developers can create and manipulate GameObjects, Components, and other game elements.
UnityEvent	A type of event in Unity that can be hooked to several functions within the context of the current scene from the Inspector window.
UPM	Unity’ Package Manager.
User Story	A simple, clear description of a feature from the perspective of the person who desires the new capability, usually a user or customer of the system.

1 Introduction

Unity holds the position of being one of the most versatile, easy to learn and, consequently, popular engines in the game development landscape. As a result, Unity serves as a widely used tool for both independent and major game development projects. However, the development of videogames, as any other software, requires commitment to good practices, patterns, and architecture.

While Unity offers a diverse range of features and capabilities, developers often face challenges related to dependency management and interactions between systems within their projects. These issues, if not addressed adequately, can lead to complex and rigid game structures, hindering scalability and maintainability.

It is in this context that Ryan Hipple, principal engineer of Schell Games, introduces in the Unite Austin conference of 2017 (Hipple, 2017) a novel approach to game development employing Scriptable Objects. Hipple's proposal advocates for a more modular and manageable game architecture in Unity, harnessing the potential of Scriptable Objects to mitigate common development challenges.

Hipple's approach promotes a development paradigm centred on Scriptable Objects, referred to as Scriptable Object Driven Development (SODD) throughout this project. Despite the benefits of SODD, there is a noticeable absence of a comprehensive and practical implementation of these architectural principles in Unity.

This final year project aims to fill this gap in the landscape of Unity's development tools by developing a comprehensive framework based on Hipple's principles of ScriptableObject Driven Development.

2 Theoretical Framework

2.1 Software Patterns in Game Development

Software patterns, a concept popularized by the seminal work "Design Patterns: Elements of Reusable Object-Oriented Software" by Gamma et al. (1995), represent solutions to common design problems in software engineering. These patterns offer a standardized methodology to address recurring challenges, promoting code reuse and system robustness. They have become a cornerstone in software development, including game development, due to their ability to simplify complex design decisions and enhance code maintainability and scalability.

In game development, the unique challenges posed by dynamic, interactive environments make the application of software patterns particularly crucial. Games often require the management of complex state systems, real-time input handling, and the orchestration of numerous interactive elements. The use of well-established patterns can streamline these processes, ensuring that game developers can focus on the creative aspects of game design without being bogged down by underlying technical complexities.

2.1.1 Command Pattern

The Command pattern consists in encapsulating a request as an object, which in term allows for parameterization, queuing and logging of requests, and supporting undoable operations.

A key application of the Command pattern in game development is in configuring input controls. Games often translate user inputs, like button presses and mouse clicks, into in-game actions. The Command pattern offers a flexible method for mapping these inputs to various game actions, facilitating user-configurable input mappings and significantly simplifying input management (Nystrom, 2014).

Moreover, the Command pattern is instrumental in managing game characters, especially in scenarios driven by AI. By decoupling commands from specific actors, the pattern allows for more generic and reusable commands. This is crucial in the development of complex AI systems in games, where AI engines generate commands executed by game actors, thus providing flexibility and enhancing AI behaviour (Gatteschi, Lamberti, Montuschi, & Sanna, 2016).

Another significant implementation of the Command pattern is in developing undo and redo functionalities in games, commonly seen in strategy games and game design tools. This feature allows players to reverse or repeat actions, adding a strategic dimension to gameplay and providing a safety net during game design processes (Mechtley & Trowbridge, 2011).

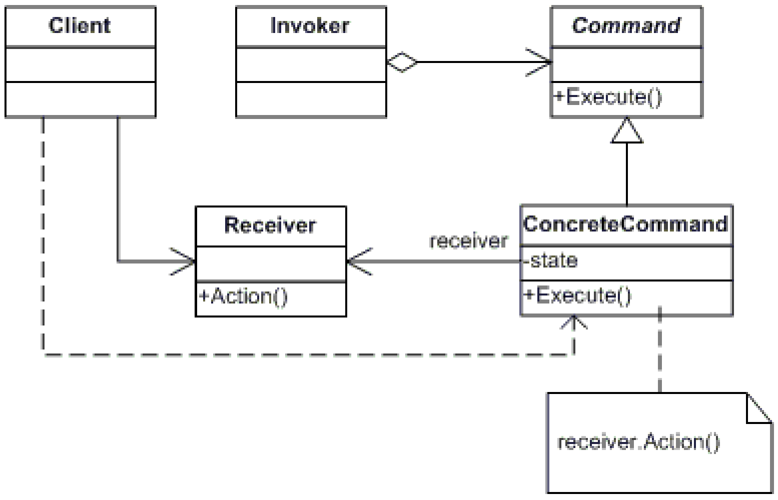


Figure 1: UML diagram for Command Pattern. Source: Khosravi & Guéhéneuc, 2004

2.1.2 Singleton Pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

One of the primary advantages of using the Singleton pattern in game development is its ability to provide controlled access to shared resources. For instance, a game might have a central manager for handling game states, audio management, or global settings. Implementing these managers as Singletons ensures that there's only one instance of each manager throughout the game, providing a single source of truth and preventing issues like conflicting states or redundant resource allocation (Nystrom, 2014).

Another significant aspect of the Singleton pattern is its memory efficiency. By ensuring only one instance of a class is created, it reduces the memory footprint, which is a critical consideration in game development, especially for mobile or resource-constrained platforms (Rautakopra, 2018).

However, it is essential to use the Singleton pattern judiciously. Overuse or misuse can lead to problems like increased coupling between classes and difficulty in testing due to the global state. Game developers must balance the need for Singleton instances with proper design practices to ensure maintainability and scalability of the game code (Freeman, The Singleton Pattern, 2015).

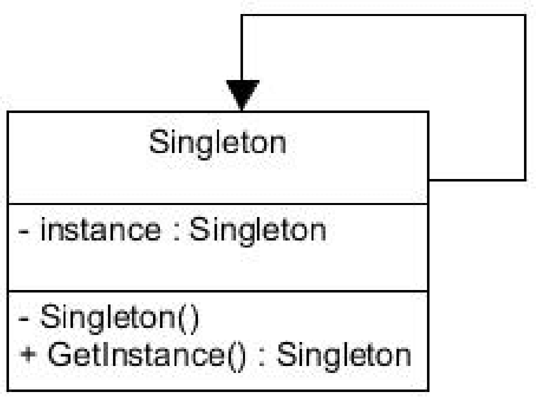


Figure 2: UML diagram for Singleton Pattern. Source: Contreras & Rene, 2017

2.1.3 Observer Pattern

The Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its observers are notified and updated automatically.

One of the key applications of the Observer pattern in game development is in the implementation of event handling systems. Games often involve numerous events, such as player actions, game state changes, or external inputs. The Observer pattern allows various parts of the game, like the UI, AI, and game logic, to observe these events and react accordingly without being tightly coupled to the event source. This decoupling promotes a more modular and maintainable codebase, which is essential for complex game development projects (Freeman, The Observer Pattern, 2015).

Another advantage of using the Observer pattern is its scalability. As games grow in complexity, the number of events and interactions can increase exponentially. The Observer pattern helps manage this complexity by allowing independent classes to communicate with each other through a well-defined interface, without needing to understand the internal workings of each other (Nystrom, 2014).

However, it is important to note that the Observer pattern can introduce overhead, especially if not implemented efficiently. Excessive notifications or poorly managed observers can lead to performance bottlenecks. Therefore, game developers need to carefully manage observer registrations and notifications, ensuring that only necessary updates are communicated.

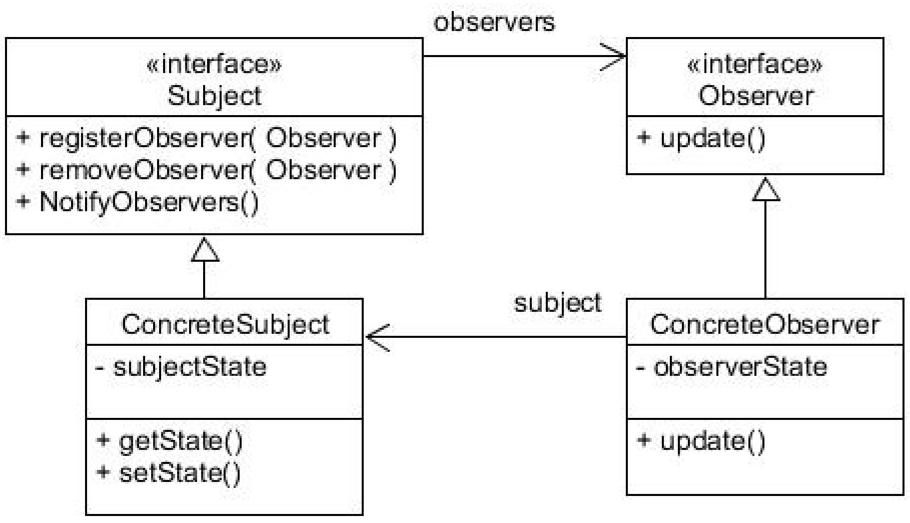


Figure 3: UML diagram for Observer Pattern. Source: Contreras & Rene, 2017

2.1.4 State Pattern

The State pattern allows an object to alter its behaviour when its internal state changes, appearing as if the object changed its class.

In game development, the State pattern is used to manage different states of game characters or the game environment. For instance, a character could have states like walking, jumping, or idle. Each state has distinct behaviours, and the character's actions and animations depend on its current state. Implementing these states allows for a clean, organized, and easily maintainable code structure (Nystrom, 2014).

One key advantage of using the State pattern is that it reduces conditional complexity. Instead of using multiple conditional statements to check the state, the pattern encapsulates state-specific behaviours into separate classes. This encapsulation not only makes the code more readable but also eases the addition of new states as the game evolves (Qu, Wei, & Song, 2014).

Another significant application of the State pattern is in AI programming in games. Different states of AI, like attack, patrol, or flee, can be implemented using the State pattern. This approach makes AI decisions and transitions between different behaviours more organized and easier to debug.

However, it's crucial to ensure that the State pattern is not overused or misapplied, as it can lead to an unnecessary proliferation of classes, making the codebase harder to manage. Proper design and planning are necessary to balance flexibility and complexity.

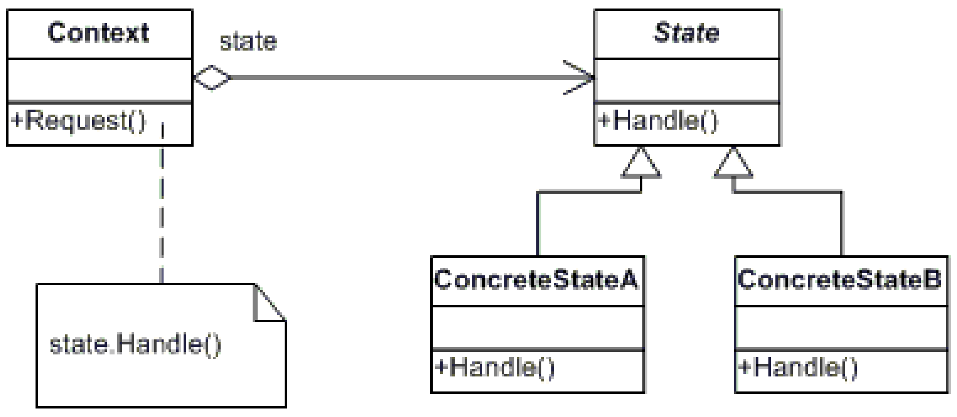


Figure 4: UML diagram for State Pattern. Source: Khosravi & Guéhéneuc, 2004

2.2 An Overview of Unity as a Game Engine

Unity is a widely recognized game engine that has made a significant impact on the field of game development. Functioning as an all-encompassing platform, it is utilized for the development of both 2D and 3D games, applications, and interactive experiences. Unity provides a comprehensive suite of tools and services that support the entire game development lifecycle, from the initial stages of creation to post-launch updates and maintenance. Its flexibility has made it a preferred choice among a diverse range of developers, from independent creators to large gaming studios (Singh & Kaur, 2022).

Central to Unity's functionality is the Unity Editor, which is noted for its robust and user-friendly interface. The Editor offers a broad spectrum of features that streamline the processes of game creation, management, and iteration. Its intuitive design is especially beneficial for beginners in game development, as it reduces entry barriers and simplifies the process of game creation (Jackson, 2015).

One of Unity's key advantages is its support for a wide array of platforms. It allows developers to publish games and applications on more than 20 different platforms, including mobile, desktop, console, and virtual reality systems. This multi-platform capability is a critical aspect of Unity, enabling developers to maximize the exposure and reach of their games (Hu, y otros, 2023).

In terms of customization and scripting, Unity employs C# as its scripting language. C# is known for being modern, flexible, and relatively easy to learn, which contributes to Unity's accessibility to a wide range of developers, from beginners to experienced programmers (Lukosek, 2016).

The Unity ecosystem is further complemented by a large and active community, alongside an extensive Asset Store. The Asset Store offers a wide range of assets, from textures and models to complete project templates and editor extensions, which aid in the rapid development and iteration of game projects. This collaborative environment promotes innovation and creative development within the Unity developer community (Baglie, Neto, Guimarães, & Brega, 2017).

2.2.1 Scripting in Unity

Unity’s User Manual (Unity Technologies, 2022) describes scripting as one of the most important aspects of Unity application development, playing a crucial role in enabling interactivity and responsiveness within games. The primary functions of scripts include processing player inputs, orchestrating in-game events, generating graphical effects, managing the physical behaviour of objects, and potentially implementing custom AI systems for characters.

Unity's runtime environment is structured around scenes, with each scene being populated by GameObjects. These GameObjects represent entities within the engine’s runtime and serve as containers for Components, which ultimately govern the behaviour of the GameObject they are attached to during scene execution.

While Unity's built-in Components offer versatility, custom gameplay features often necessitate the development of new Components via scripting. These custom scripts are pivotal for triggering specific game events, dynamically altering Component properties, and handling user inputs.

The development of new Components orbits around the MonoBehaviour class, the foundation of Unity’s scripting framework. This class is essential for all scripts, providing a structured approach to attaching scripts to GameObjects within the editor. MonoBehaviour also grants access to event call-backs of the scene’s lifecycle during execution. In addition, the GameObject class provides an array of methods for scripts to interact with their GameObject. This includes finding and connecting GameObjects, modifying their attributes, facilitating communication between them, and managing the components attached to them.

2.2.2 Unity’s Input System

One of the critical areas of improvement over the years in Unity has been its Input System. Unity's Input System, as defined in the official manual (Unity Technologies, 2024) is a flexible framework that allows developers to manage and process input from various devices efficiently. This system offers a modern, streamlined approach to handling user interactions, facilitating the creation of responsive and intuitive controls for games and applications.

The Traditional Method of Handling Input in Unity

Before the new Input System, Unity developers relied on the legacy Input Manager. While the Input Manager was straightforward and easy to use, it came with several significant limitations:

1. **Hardcoded Inputs**: Traditionally, inputs were hardcoded, requiring developers to explicitly define each input action within their scripts. This method lacked flexibility, making it challenging to modify or extend input configurations without altering the codebase.

2. **Limited Device Support**: The legacy system had restricted support for various input devices. It primarily catered to basic inputs like keyboards and mice, with minimal capabilities for handling game controllers or touch inputs.

3. **Single-Threaded Processing**: Input processing in the legacy system was single-threaded, potentially leading to performance bottlenecks in complex applications that required real-time input handling.

Comparison with the New Input System

The new Input System addresses many of the shortcomings of the legacy Input Manager. The following points highlight the key differences:

1. **Configuration vs. Hardcoding**: Unlike the old system, the new Input System allows for configuration-driven input handling. Developers can define input actions and bindings in a configuration file, streamlining the management and modification of inputs without necessitating code changes.

2. **Enhanced Device Support**: The new Input System provides extensive support for a wide range of input devices, including game controllers, VR headsets, and mobile touch inputs. It also facilitates the easy integration of custom devices.

3. **Multi-Threaded Performance**: Designed to be multi-threaded, the new system improves performance by distributing input processing across multiple threads. This enhancement is particularly beneficial for applications requiring high responsiveness and low latency.

How the New Input System Works

The new Input System introduces several core components that enhance its functionality:

1. **Input Actions**: These are high-level representations of input commands (e.g., Jump, Move, Fire). Input actions can be defined and configured within the Unity Editor, allowing developers to establish input mappings without writing extensive code.

2. **Input Binding**: Input bindings map physical device inputs (e.g., keyboard keys, gamepad buttons) to input actions, enabling flexible and customizable input configurations.

3. **Input Devices**: The system supports a variety of input devices out of the box, including keyboards, mice, gamepads, and touchscreens, and allows for the creation of custom devices.

4. **Action Maps**: Action maps are collections of input actions and bindings that can be activated or deactivated as needed, facilitating the management of different input schemes for various parts of a game (e.g., menu navigation, gameplay).

2.2.3 Existing Patterns in Unity

Lin et al. (2022) indicated that several patterns are already implemented in Unity's core as a game engine in order to simplify the development process. These include:

- **Game Loop**: A cycle that repeats throughout the runtime of a game regardless of hardware variations. Unity manages this loop, obviating the need for developers to implement it manually. This management involves maintaining consistent performance across devices with varying processing speeds.

Developers are instead required to focus on gameplay elements using MonoBehaviour methods such as ‘Update’, ‘LateUpdate’, and ‘FixedUpdate’.

- **Update Mechanism**: In game development, updating object behaviour in each frame is a common practice. Unity streamlines this process through the MonoBehaviour class. It automates the frame-by-frame update process, allowing developers to effortlessly modify GameObjects and components in sync with the game clock.
- **Prototype Pattern**: Replicating objects without altering the original is a common need in game development. Unity's Prefab system is a practical application of the Prototype pattern. It enables the duplication of template objects, complete with their components. This system facilitates the creation of Prefab Variants and hierarchical Prefab nesting, streamlining the process of object instantiation and variation.
- **Component Pattern**: This pattern advocates for the construction of smaller, focused components rather than large, multifunctional classes. Unity's approach allows developers to combine these components to achieve complex behaviour, enriching each GameObject with diverse functionalities.

2.3 Challenges and Limitations of Scripting in Unity

2.3.1 The MonoBehaviour Problem

Fine (2016) highlighted that despite its critical role, MonoBehaviour's utility in extensive projects is often overshadowed by several inherent challenges:

1. **Shared vs. Non-Shared State Management**: MonoBehaviour's primary challenge involves the management of shared and non-shared states. In game development terms, shared state pertains to attributes that are uniform across all script instances, whereas non-shared state refers to unique characteristics specific to each instance. MonoBehaviour tends to merge these states, creating a complex and often confusing scenario, especially in large-scale projects that require a vast amount of scripting.

2. **Loss of Changes in Play Mode**: The transient nature of changes made during the Unity play mode. Modifications made to GameObjects, or their states are not retained post-exit from the play mode, a design choice intended to prevent enduring unintended alterations. However, this feature restricts the real-time iterative development process, often essential in game development, as it hinders the preservation of adjustments made during playtesting.

3. **Collaboration Issues**: MonoBehaviour's file-level granularity, or lack thereof, complicates version control and teamwork. Given that MonoBehaviour scripts are attached to GameObjects within scene or prefab files, multiple developers working on the same file can encounter version control conflicts. This situation is particularly problematic in larger teams where simultaneous editing of scene or prefab files can result in overwrites and complex merge conflicts, impeding workflow efficiency.

4. **Callback Chaos**: MonoBehaviour also heavily relies on callbacks, leading to a situation termed as 'callback chaos'. In projects where numerous scripts are simultaneously active, managing and tracking the activity in each frame becomes exceedingly challenging. This complexity often obscures the understanding of the game's operational flow, posing significant challenges in debugging and game state management.

5. **Lack of Prefab Flexibility**: Workarounds are commonly employed to mitigate some of the MonoBehaviour limitations, such as the use of uninstantiated prefabs for shared state management. While this approach can theoretically separate shared from instance-specific data, it diverges from the original intent of prefabs and is prone to errors, including accidental scene inclusion or unintended modifications.

6. **Overreliance on C# Statics for Shared States**: Additionally, the employment of C# static variables for managing shared data, while a viable alternative, comes with its set of drawbacks. This method lacks integrated serialization support in Unity, complicates inspector integration, and requires additional handling during domain reloads, often necessitating a more do-it-yourself approach that might not be ideal in complex projects.

2.3.2 The Singleton Problem

Hipple (2017) emphasized on the increasing number of setbacks deriving from the common use of Singleton for managing dependencies between game systems and handling global state. The short-term solution of applying this pattern can lead to greater issues in the long term:

1. **Rigid Connections**: One of the significant issues with using Singletons in Unity is that they create rigid connections between different systems. This tight coupling implies that modifying or extending one part of the system necessitates changes in others, reducing overall flexibility and potentially introducing new bugs. Such tight coupling hinders the reusability and reconfiguration of components without impacting other parts of the game, resulting in a more fragile and less maintainable codebase.

2. **Global State Management**: Singletons typically maintain state across different scenes, which can lead to unintended side effects. For example, if a Singleton retains data that should be reset between scenes, it can cause bugs that are difficult to track down. This persistent state breaks the clean slate principle, where each scene should start with a known and controlled state.

3. **Polymorphism Limitations**: Singletons undermine the object-oriented principle of polymorphism, which is the ability to substitute objects of different types through a common interface. When a system relies on a single instance of a class, it becomes difficult to replace that instance with a different implementation. This limitation makes it harder to create variants of systems for testing or to provide alternative behaviours, thus reducing the overall flexibility of the architecture.

4. **Testing and Debugging Challenges**: The hidden dependencies introduced by Singletons present significant challenges for unit testing. Since Singletons are globally accessible and maintain state, tests can become interdependent, leading to unreliable or flaky outcomes. Isolating a single component for testing is difficult when it relies on the global state maintained by Singletons, which complicates identifying the source of a failure.

5. **Dependency Nightmares**: As the number of Singletons in a project increases, managing the dependencies between them becomes increasingly complex. This complexity often leads to race conditions, where the order of initialization and access to these Singletons causes unpredictable behaviour. Such dependency issues make the system more error-prone and harder to debug.

6. **Single Instance Limitation**: By design, the Singleton pattern restricts the system to a single instance of a class. This limitation can become problematic if the game's requirements evolve to support multiple instances of a system. For instance, a game might initially have a single player but later need to support multiple players or sessions. Revisiting and refactoring the code to remove the Singleton restriction can be time-consuming and error-prone.

2.4 Introduction to ScriptableObjects

During the 2016 Unite conference presentation “Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution”, Richard Fine advocated for the use of ScriptableObjects in Unity as a solution to the limitations posed by the MonoBehaviour class, especially in large-scale project development. The ScriptableObject approach offers a paradigm shift in how data and behaviours are managed within Unity, providing a range of benefits and new possibilities.

The fundamental concept of ScriptableObjects revolves around creating objects not attached to GameObjects. This detachment from the scene's GameObjects is a critical aspect that underpins the advantages highlighted by Fine. Unlike MonoBehaviour, which is closely tied to the GameObject lifecycle and scene structure, Scriptable Objects exists independently, allowing for more flexible and modular designs:

1. **Separation of Shared and Instance-Specific Data**: One of the primary advantages of Scriptable Objects is the efficient separation of shared and instance-specific data. This characteristic allows developers to segregate data such as common enemy attributes from individual-specific states like health. The result is a more efficient data management system, significantly reducing redundancy and enhancing overall project maintainability.

2. **Persistent Data Management**: ScriptableObjects also offer persistent data management. In stark contrast to MonoBehaviour, changes to a Scriptable Object are retained after exiting play mode, facilitating a smoother and more effective development workflow. This persistence is particularly advantageous during game balancing and iterative development phases, where real-time adjustments are crucial.

3. **Enhanced Project Organization and Collaboration**: From a collaboration standpoint, Scriptable Objects mitigate the version control issues often encountered in MonoBehaviour-based development. Its ability to exist streamlining the workflow and enhancing productivity.

4. **Reduced Complexity and Improved Performance**: By not being bound to GameObjects, ScriptableObject instances don't carry the overhead of unnecessary GameObject components, potentially leading to better runtime performance. Moreover, they steer clear of the "callback chaos" associated with MonoBehaviour, as they are not part of the GameObject lifecycle methods like Update and Start. This clarity leads to more maintainable and understandable code.

5. **Flexible and Extensible Design Patterns**: ScriptableObjects enable the implementation of various design patterns, such as Singleton or Factory, in a cleaner and more efficient manner. It allows for a data-driven design approach where game behaviour can be modified using different ScriptableObject instances without altering the core code.

6. **Ease of Scalability in Game Development**: ScriptableObject promotes a more scalable architecture, particularly useful in large-scale projects where managing a multitude of GameObjects and MonoBehaviours can become unwieldy. This scalability is beneficial not just in terms of project size but also in terms of team size, facilitating easier onboarding and division of work among different team members.

2.5 Game Architecture with ScriptableObjects

During the Unite Austin 2017 conference presentation “Game Architecture with ScriptableObjects,” Ryan Hipple introduced an innovative approach to game architecture using ScriptableObjects. This methodology focuses on creating modular, editable, and easy-to-debug game systems. Hipple emphasized the importance of reducing dependencies and global state management in game development, proposing ScriptableObjects as a versatile solution to these challenges.

2.5.1 The Three Principles of Game Engineering

Hipple’s approach delineates three key principles of game engineering: Modularity, Editability, and Debuggability. These principles are foundational in this unique application of ScriptableObjects to create more efficient and maintainable game architectures.

Modularity

The first principle, modularity, states that game systems should be designed as separate and interchangeable modules, components, or units. Each module performs a distinct function and operates independently of the others. Modularity offers several benefits:

- **Reduced Interdependencies**: By ensuring modules are self-contained, changes in one module have minimal impact on others, reducing the risk of a change causing a cascade of issues across the game.
- **Reusability**: Modular components can be reused across different parts of a game or even in different projects, saving development time and resources.
- **Flexible Design**: Modularity allows developers to assemble and reassemble components in various configurations, aiding in experimentation and innovation in game design.
-

Editability

The second principle, editability, states that game systems and data should be easily modified without requiring modifications to the source code. This principle is particularly important for enabling designers and other team members to tweak game elements without needing programming expertise.

- **Data-Driven Design**: This approach involves separating data from the logic of the game, allowing non-programmers to edit data directly.
- **Inspector-Friendly Tools**: In Unity, making systems editable often involves leveraging the Inspector window to create user-friendly interfaces for modifying game data.
- **Runtime Changes**: Allowing changes to game data at runtime aids in rapid prototyping and balancing, as adjustments can be made while the game is running, and their effects immediately observed.

Debuggability

The third principle, debuggability, advocates for the ease with which a game can be debugged or tested for errors. A well-designed game architecture should facilitate easy identification and fixing of bugs.

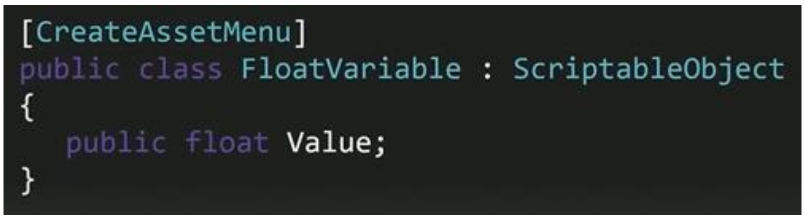
- **Isolation of Issues**: Modular design helps in isolating bugs to specific components, making it easier to identify the source of a problem.
- **Readable and Traceable**: The system should be transparent enough so that the flow of data and events can be easily followed and understood by the developers.
- **Tools and Visualizations**: Implementing tools that visualize the game's operations, such as showing event triggers or data changes in real-time, can significantly aid in debugging.

2.5.2 Modular Data

Hipple proposed the use of ScriptableObject Variables as modular containers of data.

Concept and Applications

A ScriptableObject Variable encapsulates a single value of a specific data type, such as integers, floats, or strings. This concept allows these modular containers of data to be shared and referenced across diverse systems within the game, preventing these systems from relying on each other to retrieve and modify the game state.



```
[CreateAssetMenu]
public class FloatVariable : ScriptableObject
{
   public float Value;
}
```

Figure 5: Sample code of a ScriptableObject Variable encapsulating a float value. Source: Hipple, 2017

In practical game development scenarios, ScriptableObject Variables can be used extensively to drive game logic and share game state. Hipple gives examples such as a ScriptableObject Variable holding a float value representing the player's health, which could be referenced in various systems like UI display and combat mechanics.

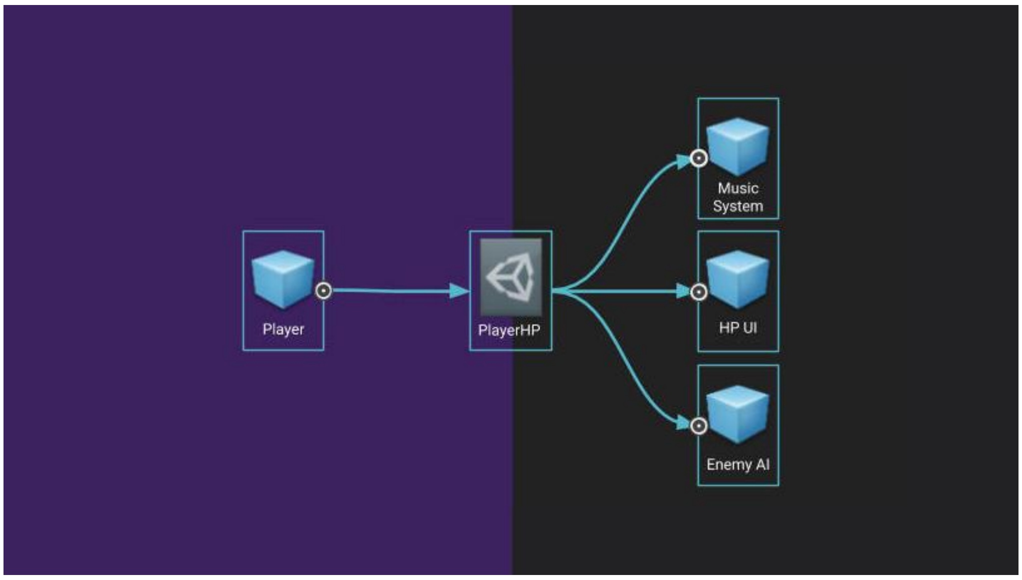


Figure 6: Diagram illustrating a ScriptableObject Variable holding the player's health. Multiple systems reference this variable for distinct purposes while being decoupled from each other. Source: Hipple, 2017

Advantages and Impact

One of the key aspects of ScriptableObject Variables is their accessibility through the Unity Editor. This allows for easy manipulation of their values directly in the Inspector, adhering to the editability principle. It empowers both developers and designers, particularly those without extensive programming backgrounds, to easily modify game parameters.

The adoption of ScriptableObject Variables also leads to centralized data management, where altering game data becomes a matter of modifying values in a single location, rather than navigating through numerous scripts.

Moreover, ScriptableObject Variables inherently promote modularity and reusability. As these data containers can be integrated across various game systems, they ensure consistency in data representation and usage, enhancing the overall coherence of the game architecture.

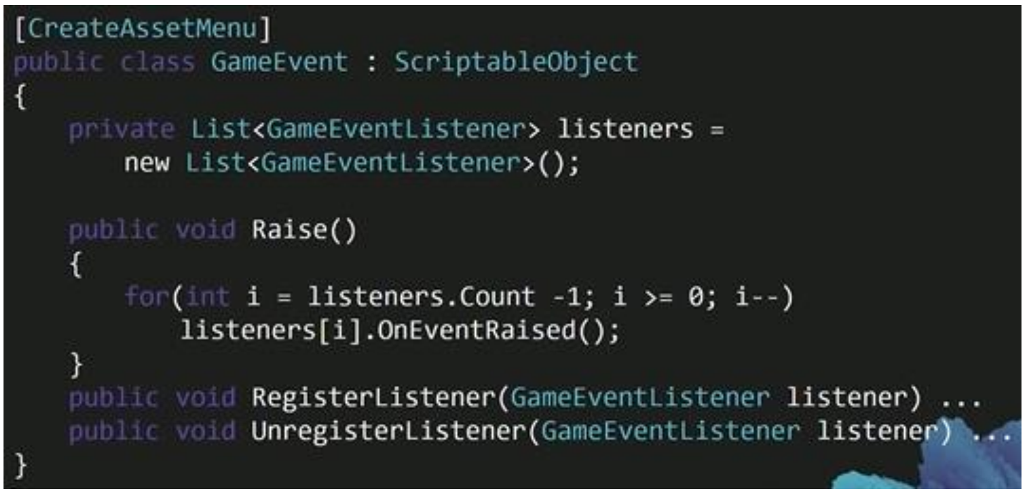
Most crucially, Hipple's concept of ScriptableObject Variables facilitate a data-driven design paradigm. This paradigm allows game behaviour to be modified through data adjustments rather than code alterations. Such an approach is particularly beneficial in a collaborative development environment, where different disciplines, from design to programming, interact seamlessly with the game's data layer.

2.5.3 Event-Driven Architecture

Hipple emphasized on the application of Scriptable Objects to represent game events, creating an event-driven architecture with ScriptableObject Events.

Concept and Applications

A ScriptableObject Event, referred also by Hipple as Game Event, consists in a Scriptable Object that holds a list of listeners, allowing for diverse systems within the game to subscribe and unsubscribe to this list and raise the event, notifying the rest of listeners. With this approach, each event in the game becomes a standalone entity represented by a Scriptable Object.



```
[CreateAssetMenu]
public class GameEvent : ScriptableObject
{
    private List<GameEventListener> listeners =
        new List<GameEventListener>();

    public void Raise()
    {
        for(int i = listeners.Count -1; i >= 0; i--)
            listeners[i].OnEventRaised();
    }
    public void RegisterListener(GameEventListener listener) ...
    public void UnregisterListener(GameEventListener listener) ...
}
```

Figure 7: Code sample of a ScriptableObject Event. Source: Hipple, 2017

In practical terms, an event could signify a change in the player’s health, the completion of a level, or an enemy encounter. These ScriptableObject Events act then as broadcasters, sending out signals when certain conditions in the game are met.

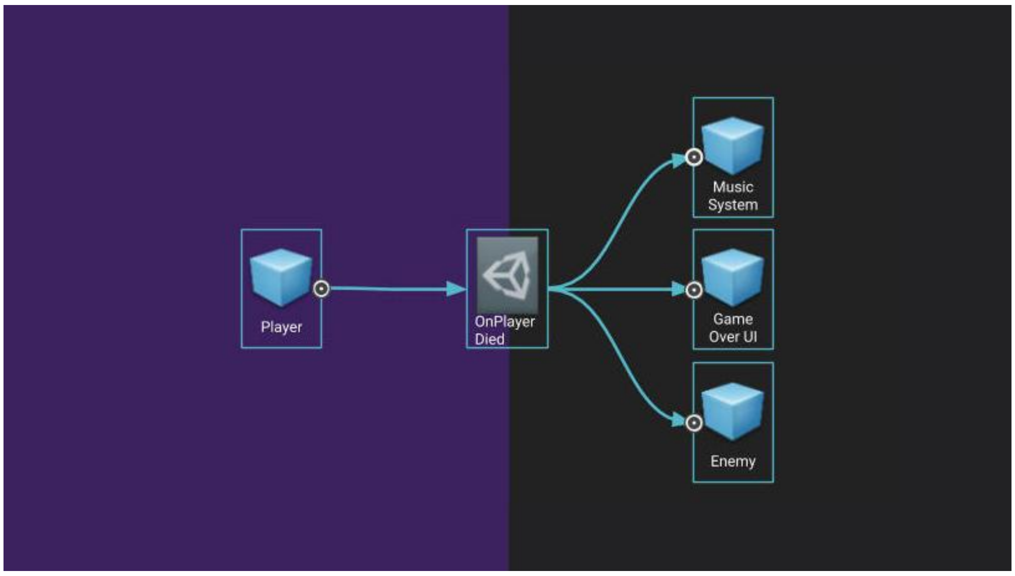
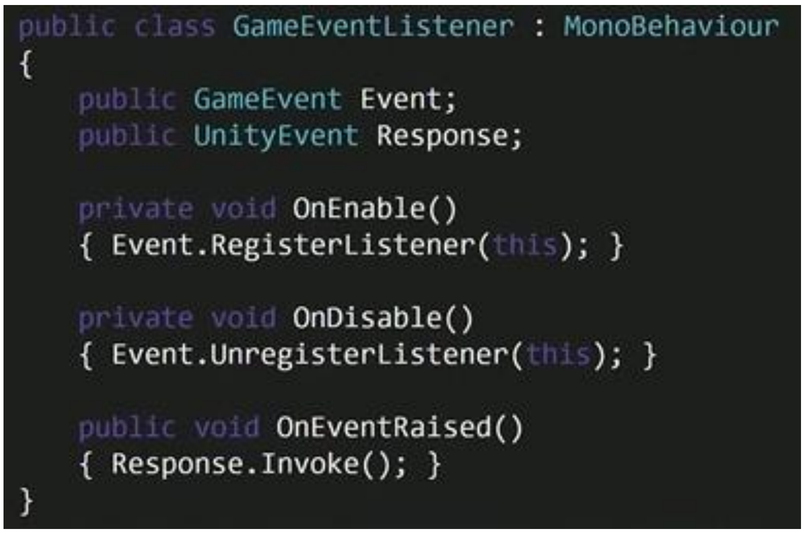


Figure 8: Diagram illustrating a ScriptableObject Event that signifies the death of the player. The Player script raises the event and other game systems listening to the event react accordingly. Source: Hipple, 2017

To facilitate this architecture, Hipple introduces the concept of Game Event Listeners. Listeners consist in game components that subscribe to specific events and react accordingly. When an event occurs, each listener executes its corresponding Unity Event, which can be hooked from the Unity Editor to functions within the context of the current scene, whether it be updating the UI, triggering a gameplay mechanic, or modifying the game state.



```
public class GameEventListener : MonoBehaviour
{
    public GameEvent Event;
    public UnityEvent Response;

    private void OnEnable()
    { Event.RegisterListener(this); }

    private void OnDisable()
    { Event.UnregisterListener(this); }

    public void OnEventRaised()
    { Response.Invoke(); }
}
```

Figure 9: Code sample of an Event Listener. Source: Hipple, 2017

Advantages and Impact

One of the most notable advantages of ScriptableObject Events is the significant reduction in coupling it achieves. In traditional game development paradigms, events are often tightly integrated into the game's components, leading to a high degree of interdependency. By abstracting events into Scriptable Objects, game components can react to events without being directly bound to the event source.

Moreover, this approach offers enhanced flexibility and scalability. Developers can easily introduce new listeners to existing events or create entirely new events without disrupting the existing game structure. This flexibility is particularly advantageous when scaling up the game or introducing new features.

Another critical aspect is the ease of debugging and testing it affords. With each component acting independently, isolating and testing specific event reactions becomes much more manageable. This modularity ensures that each part of the system can be individually verified for correct behaviour.

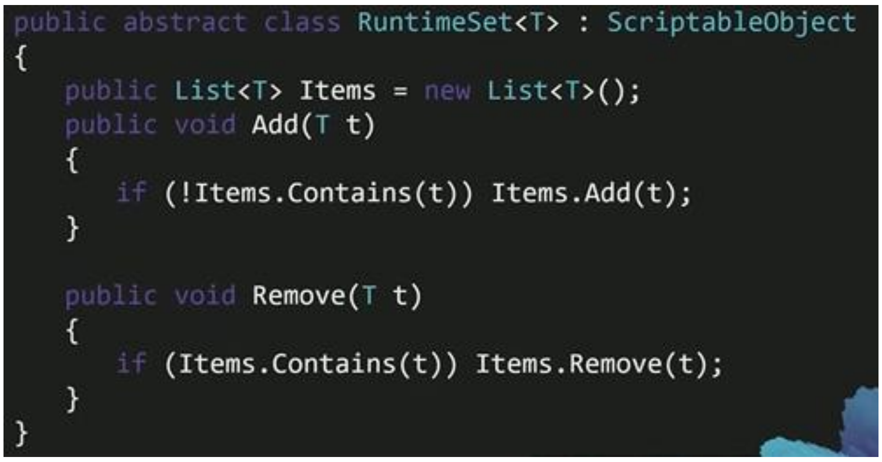
Additionally, Hipple's ScriptableObject Event-driven architecture provides easy collaboration in development. Designers, for instance, can modify and create events using Unity's Editor, while programmers can focus on more complex logic and underlying systems that react to these events.

2.5.4 Runtime Object Management

Another concept introduced by Hipple is Runtime Sets, which addresses a common challenge in game development: managing and tracking a dynamic set of items or objects throughout the game's runtime.

Concept and Applications

At its core, a Runtime Set is a Scriptable Object employed to maintain a dynamic list of items or objects during the game's runtime. This list can be constantly updated, with items being added or removed as the game progresses. The key feature of Runtime Sets is their ability to function as central repositories for specific types of objects, such as enemies, collectible items, or interactive game elements.



```
public abstract class RuntimeSet<T> : ScriptableObject
{
    public List<T> Items = new List<T>();
    public void Add(T t)
    {
        if (!Items.Contains(t)) Items.Add(t);
    }

    public void Remove(T t)
    {
        if (Items.Contains(t)) Items.Remove(t);
    }
}
```

Figure 10: Code sample of a generic Runtime Set. Source: Hipple, 2017

In practical terms, Runtime Sets can be employed in various scenarios where tracking and managing a group of similar objects is essential. For instance, in a game that involves combat against multiple enemies, a Runtime Set can function to keep track of all active enemies in the current level. This Set is then updated whenever an enemy is spawned or defeated, providing a real-time overview of the enemies present in the game at any given moment.

With this application of Scriptable Objects, each object that needs to be tracked would register itself to a Runtime Set upon being created or activated and deregister when destroyed or deactivated. This registration and deregistration process ensures that the Runtime Set always contains an up-to-date record of the objects in question.

Advantages and Impact

The traditional approach to tracking such dynamic elements often involves each object managing its own state or the game performing constant searches through all game objects to identify relevant items. Runtime Sets offer a more efficient alternative by centralizing the tracking process. This centralization not only makes the management of these objects more straightforward but also can lead to performance optimizations, particularly in complex scenes with numerous objects.

Another key strength of Runtime Sets is their adaptability and scalability. They provide a flexible framework that can easily accommodate changes in the number of objects being tracked, making them suitable for games with varying levels of complexity and object interactions.

2.6 Frameworks in Software Development

Frameworks in software development provide the skeleton of an application, allowing developers to customize and extend functionalities with greater ease. They often encompass a set of reusable design patterns and code components, streamlining the development process and promoting code reusability.

2.6.1 Inversion of Control

A key feature of frameworks is their "inversion of control" (IoC). This concept refers to the reversal of the flow of control compared to traditional library-based development. In traditional development, the flow of control is dictated by the application code, which calls specific library functions as needed. However, with IoC in a framework, this relationship is reversed – the framework calls into the application code. The framework dictates the overall flow of the application, and the application-specific code is plugged into it. This means the framework oversees the main loop, and it is the framework that determines when the application code is called. IoC is a powerful mechanism for decoupling components, leading to more modular and easier-to-maintain applications. It is central to the design of modern software frameworks and a critical concept in understanding how frameworks alter the development process compared to traditional library use (Pop, 2008).

2.6.2 Application Programming Interfaces (APIs)

As stated by Wu et al. (2015), Application Programming Interfaces (APIs) are central to the functioning of software frameworks, acting as the primary point of interaction between the framework and the client's application code. They define a set of contracts, or interfaces, through which different software components communicate and interact. This relationship is crucial for leveraging the capabilities of frameworks in application development.

API Evolution and Compatibility

One of the significant challenges with APIs in frameworks is managing their evolution. As frameworks evolve, their APIs may change, adding new features or altering existing ones. These changes can break backward compatibility, necessitating updates or rewrites in client applications that rely on older versions of the API. This evolution process requires a careful balance between introducing new features and maintaining compatibility with existing client code. The study by Wu et al. found that missing classes and methods are common in framework evolution, and these can significantly affect client programs, emphasizing the importance of understanding the potential impacts of API changes.

Framework-Client Dependency

The use of APIs in frameworks creates a dependency relationship between the framework and its client applications. This dependency means that client applications may need to be updated to accommodate changes in the framework's API. It is essential for developers to assess the cost and impact of these updates. Effective management of these dependencies requires robust version control and a keen understanding of the framework's roadmap and update cycles.

API Design and Usability

The design of an API significantly influences the usability and effectiveness of a framework. A well-designed API simplifies complex tasks, promotes efficient coding practices, and enhances the developer experience. Conversely, a poorly designed API can lead to increased development time, potential errors, and frustration. Thus, API design is a critical aspect of framework development, requiring a deep understanding of the users' needs and the application domain.

Encapsulation and Modularization

APIs in frameworks often encourage encapsulation and modularization. By providing a set of well-defined interfaces, frameworks allow developers to compartmentalize functionality. This modular approach facilitates easier maintenance, testing, and scaling of applications. It also enables developers to use only the parts of the framework necessary for their application, leading to leaner and more efficient code.

2.6.3 Framework Design

Frameworks need to address specific challenges related to scalability, maintainability, and flexibility. Drawing from the experiences in large-scale industrial banking projects, Bäumer et al. (1997) proposed several key concepts to framework design.

Domain Partitioning

Large systems often encompass diverse functionalities and requirements. Domain partitioning involves dividing the system into distinct segments based on functionality, business logic, or other criteria. This approach allows for targeted development and management of each segment, enhancing the modularity and clarity of the system. Frameworks must support this partitioning to enable developers to work on different segments independently, reducing complexity and potential conflicts.

Framework Layering

Layering is a technique where various levels of the framework provide distinct functionalities, such as data access, business logic, and presentation layers. In large systems, framework layering is crucial for separating concerns and managing dependencies. Each layer should have a well-defined role and interface, enabling developers to make changes in one layer without significantly affecting others. This separation of concerns is pivotal in managing the complexity of large-scale systems.

Framework Construction and Integration

The construction of the framework involves defining its architecture, components, and interfaces. For large systems, it is essential that the framework is designed with flexibility and adaptability in mind, allowing it to evolve as requirements change. Integration strategies are also crucial, as large systems often involve integrating the framework with existing systems or other frameworks. This integration should be achieved without tight coupling, maintaining the system's modularity and ease of maintenance.

Accommodating Domain Needs

Frameworks in large systems need to be tailored to the specific needs of the domain they are serving. This tailoring involves understanding the unique requirements and challenges of the domain and designing the framework to address these effectively. For instance, in banking applications, security and transaction management might be areas of particular focus.

Evolution and Reusability

Given the size and longevity of large systems, the frameworks used must be designed for evolution. They should support easy updates and extensions without significant overhauls. Reusability is another critical factor, as components and patterns developed for one part of the system should be reusable in others, fostering efficiency and consistency across the system.

3 Objectives

3.1 Principal Objectives

- Develop a specialized framework for Unity that provides a solid foundation for ScriptableObject Driven Development (SODD). The significance of this framework lies in implementing and expanding the fundamental principles of modularity, editability and debuggability introduced by Hipple's game architecture with ScriptableObjects. The envisioned outcome is a robust and comprehensive tool that enhances productivity and collaboration in Unity's development environment.

- Produce exhaustive and well-structured documentation for the framework. As key to any software tool, this documentation will serve as a comprehensive guide for users to understand, implement and effectively use the framework. The desired outcome is a user-friendly documentation that makes the framework's features and functionalities accessible and understandable to developers, encouraging its use.

- Create a sample videogame using the developed framework. The purpose is to demonstrate the practical application and effectiveness of the framework in a real-world use case. The game acts as a proof of concept, illustrating the streamlined development process facilitated by the framework while simultaneously serving as a reference for future users.

3.2 Secondary Objectives

- Publish the developed framework on platforms like GitHub and Unity's Asset Store. Making the framework publicly available broadens its reach to potential users while additionally promoting community engagement, feedback and attracting potential collaborators, contributing to the open-source community.

4 Methodological design and timeline

4.1 Methodology

The development of the SODD Framework employs a hybrid approach, incorporating elements from Agile and Lean methodologies to create a flexible, adaptive development process. This methodological fusion ensures a balance between rapid delivery and process efficiency.

4.1.1 The MoSCoW method

The initial step involves gathering all necessary requirements for the framework. This includes functional requirements as well as non-functional requirements such as performance and usability.

Once defined, the requirements are categorized following the MoSCoW method to define their relevance and priority within the tool landscape the framework offers. Ahmad et al. (2017) describe the MoSCoW method as a prioritization technique used in management, business analysis, project management, and software development to define the importance of each requirement. This method categorizes requirements into four categories:

- **Must Have**: These are non-negotiable requirements that the software cannot function without. They form the backbone of the project and must be implemented for the framework to be considered viable. This category often includes core functionalities that define the framework's purpose and objectives.

- **Should Have**: Requirements classified under "Should have" are important but not critical for the launch. They enhance the framework's utility or user experience but can be delayed without compromising the framework's core functionality. These features should be included in the initial releases if time and resources permit.

- **Could Have**: These are desirable features that, while beneficial, have the least impact on the framework's overall goals. They are typically implemented if extra time becomes available after the development of higher-priority features. "Could have" features offer a way to add value to the framework without detracting from the essential work needed to meet "Must have" and "Should have" requirements.

- **Won’t Have**: This category is sometimes added to the list to explicitly state which features will not be included in the current release/delivery but might be considered in the future.

4.1.2 User Stories, DoR and DoD

Once the priority of the requirements and features is determined, each one is translated into a User Story. User Stories are a fundamental aspect of Agile methodologies, offering a simple yet effective way to capture user-centric requirements. A User Story typically follows a simple template:

As a [type of user], I want [an action] so that [a benefit/a value].

This format helps in breaking down complex requirements into manageable, implementable tasks that directly contribute to the user's experience with the framework (Kumar, Tiwari, & Dobhal, 2022).

User Stories contain a "Definition of Ready" (DoR) and a "Definition of Done" (DoD), these components act as a quality gate to ensure they meet the project’s standards and requirements at every stage of the development process.

- **Definition of Ready (DoR)**: The DoR specifies the conditions a User Story must meet before the development can begin. This includes having a clear description, acceptance criteria, and ensuring that the Story is feasible within the current scope and resources (Power, 2014).

- **Definition of Done (DoD)**: The DoD establishes the criteria for when a User Story is considered complete. It typically includes passing all specified tests, code reviews, integration into the main branch without issues, and documentation updates. The DoD is crucial for maintaining quality, as it expected value before being marked as complete (Diebold, Theobald, Wahl, & Rausch, 2018).

4.1.3 Version Control and CI/CD

The development of the defined User Stories relies on the use of GIT for version control and CD/CI to follow modern standardized development and deployment practices.

GIT is a distributed version control system that plays a central role in managing source code, enabling developers to work on the same project without interfering in each other's progress. GIT allows for the tracking of changes, reverting to previous versions of the code, and managing different development branches efficiently. The use of GIT ensures that the framework's codebase remains organized, versions are well-managed, and the integration of changes is seamless (Casquina & Montecchi, 2021).

Continuous Integration (CI) and Continuous Deployment (CD) are practices that automate the process of integrating code changes and deploying them to production environments, automating the process of getting the software from version control to the end user.

- **Continuous Integration (CI)** involves automatically building and testing the framework every time a new code change is pushed to the repository. This ensures that new changes do not break or negatively impact the existing functionality. CI aims to detect and fix integration errors as quickly as possible, maintaining the codebase's health and facilitating a smoother development process. Tools like Jenkins, Travis CI, or GitHub Actions can be used to automate the CI process, running tests and builds on every commit (Naveen, et al., 2023).

- **Continuous Deployment (CD)** extends the CI process by automatically deploying the code to production environments after it passes all tests. This means that new features, fixes, and updates can be delivered to users quickly user needs and feedback. CD minimizes the time taken from the development of a new feature to its deployment, enabling a rapid iteration cycle that is vital for maintaining the quality of the framework (Ali, 2023).

4.2 Planning

4.2.1 Definition and categorization of requirements and features

Building upon the principles of a ScriptableObject-based game architecture, as proposed by Hipple (2017) and previously examined in this document, the SODD Framework has been delineated around three core features: Scriptable Events, Scriptable Variables and Runtime Sets. Other complimentary functionalities have been additionally defined for the framework, with lesser development priorities.

Scriptable Events

Scriptable Events are conceptualized as ScriptableObjects designed to encapsulate an event. This encapsulation facilitates other components to reference, subscribe, unsubscribe, and invoke these events.

The implementation of Scriptable Events should be versatile, supporting payloads that include Unity's fundamental data types as well as additional types to cover a broad spectrum of use cases. The prioritization of each Scriptable Event's implementation is determined by the significance of the payload's data type it supports.

Inherently, the concept of Scriptable Events implements Hipple's first principle of game engineering: modularity. To further adhere to the principles of debuggability, it is required for Scriptable Event implementations to incorporate an inspector option that enables the logging of event invocations. Additionally, to implement the principle of editability, they should provide fields for the specification of payload values and mechanisms for event triggering from the Unity inspector.

Scriptable Variables

Scriptable Variables are defined as a value encapsulated within a ScriptableObject, facilitating access and modification by other components. Aligning with the approach taken with Scriptable Events, the development of Scriptable Variables should encompass a wide array of data types. The prioritization for implementing these variables is guided by the relevance of their data types.

The principle of modularity is intrinsic to the concept of Scriptable Variables. It is essential for Scriptable Variables to permit modifications via the Unity inspector and to feature mechanisms for logging changes. This approach ensures adherence to the principles of editability and debuggability.

As an extension to Hipple's initial concept of Scriptable Variables, a new requirement has been added. This addition stipulates that Scriptable Variables must be capable of notifying other components about changes in their values. Such a feature enhances the reactivity of the system, allowing components that reference a Scriptable Variable to update automatically, thereby obviating the need for inefficient polling methods.

Runtime Sets

Runtime Sets, now termed Scriptable Collections within the framework's context, are conceptualized as ScriptableObjects that encapsulate a collection of values or references. These collections provide components with the ability to reference them for querying the presence of items, as well as adding or removing elements.

Consistent with the development priorities of Scriptable Events and Scriptable Variables, the emphasis for Scriptable Collections is placed on the types of elements they contain.

Following the previously mentioned ScriptableObject implementations, Scriptable Collections integrate in their concept the principle of modularity but require additional mechanisms for logging changes—such as items being added or removed—and enable modifications from the inspector in order to comply with the debuggability and editability principles.

An enhancement to the Scriptable Collections concept is the incorporation of a requirement for mechanisms that notify observers about changes within the collection, particularly the addition or removal of items. This advancement mirrors the improvements made to Scriptable Variables by introducing a reactive update system, thereby eliminating the need for inefficient polling techniques.

Event Listeners

Event Listeners, as conceptualized by Hipple, serve as components designated to subscribe to a Scriptable Event. Upon the invocation of such event, these listeners trigger a corresponding Unity Event, allowing developers to specify in the editor actions to be executed in the scene.

Event Listeners, as a concept, inherently carry the principles of modularity and editability. The principle of debuggability, while not directly integrated into the Event Listeners themselves, is addressed through the events to which they subscribe, as previously outlined in the requirements of Scriptable Events.

The prioritization of Event Listener implementations directly correlates with that of the Scriptable Events they are designed to support. For instance, an Event Listener designed for integer-based events (Int Event Listener) is prioritized in alignment with the implementation priority of its corresponding Scriptable Event (Int Event).

Value Reference

Value References, another complimentary concept idealized by Hipple, are designed to streamline the workflow for designers and enhance component editability. These references allow a script to utilize a value that may either be derived from the script's internal variable or from an external Scriptable Variable. This flexibility is crucial as the requirements dictating the reliance on internal versus external values can frequently shift throughout the iterative phases of game design. By employing Value References, developers are afforded the convenience of toggling between internal and external sources for a given value directly from the Unity inspector, negating the need for code modifications to accommodate evolving design requirements.

Given their role in facilitating flexibility and adaptability in the design process, Value References are recognized as an important but supplementary feature within the development framework. Consequently, they are assigned a lower priority compared to core elements such as Scriptable Events, Variables, and Collections.

Variable Repository

In the Unity Editor, modifications to values of ScriptableObject instances are retained across Play Mode sessions, facilitating the development workflow. However, this persistence is not maintained in the final game builds, presenting a challenge for maintaining state between game sessions. Addressing this challenge necessitates the development of a mechanism that ensures the continuity of scriptable variable values across executions.

The proposed solution, termed as Variable Repository, is designed to bridge this gap. It is conceptualized as a specialized entity responsible for managing a collection of references to Scriptable Variables that require their values to be preserved across game sessions. The repository functions by persistently storing and retrieving these values in response to specific lifecycle events, such as the loading and unloading of scenes or the initiation and termination of the game.

Despite the significance of Variable Repositories for managing the persistence of Scriptable Variable values, they are not indispensable for the core functionality of the framework. Consequently, the implementation of this feature is considered a secondary priority.

Variable Observers

Variable Observers, building upon the concept of Event Listeners, represent an addition to complement the framework's ecosystem. These components, once attached to a GameObject, are linked to a Scriptable Variable and trigger a Unity Event in response to updates in the variable's value. Additionally, a toggle option is included, which in term allows for the Unity Event to be triggered upon initialization. This functionality aims to benefit components such as UI elements, enabling them to be updated not only with the initial value of the variable but also whenever the variable's value changes thereafter.

This component implements the principles of modularity and editability, reducing the need for scripts dedicated to reading a value of a variable and assigning it to another component that feeds from this value. The implementation of the principle of debuggability is ensured by the Scriptable Variable to which the observer is assigned, maintaining consistency with the design considerations outlined for Scriptable Events and Variables.

Given that Variable Observers are a complementary component rather than a core feature, they are assigned a lower priority in the development process. However, they represent an important aspect to consider for subsequent integration once foundational functionalities have been established.

Input Action Handlers

Input Action Handlers are a concept introduced with the objective to expand upon Hipple's use of ScriptableObjects for managing events and data. These ScriptableObjects are designed to monitor events initiated by an Input Action and, in response, trigger Scriptable Events. This functionality bridges de gap between events occurring in Unity's Input System and the event system featured by the framework. Additionally, the capability to debug event invocations, a feature previously defined in the Scriptable Event requirements, is extended to Input Action Handlers, enhancing the ability to log input actions in runtime from the framework's event system.

While this feature is considered secondary, it is prioritized immediately following the implementation of the core functionalities.

Control Scheme Handlers

Control Scheme Handlers further extend the integration between Unity's Input System and the SODD Framework's event system.

Unity's Input System supports the creation of control schemes. Each control scheme can be bound to one or multiple devices, enabling developers to define how input from these devices translates to input actions.

Control Scheme Handlers are conceptualized as ScriptableObjects that monitor the currently active control scheme of an Input Action Asset at runtime, and respond to changes by triggering corresponding Scriptable Events. This approach allows for game systems to react to these changes by listening to the Scriptable Events invoked by the Control Scheme Handlers. For instance, if the control scheme switches from a gamepad to a mouse, react by showing the cursor and updating UI elements to reflect mouse-based controls. Conversely, switching to a gamepad can prompt to hide the cursor and update the UI to display gamepad controls.

While the integration of Control Scheme Handlers is beneficial and useful, it is not essential to the core functionality of the framework. Therefore, its implementation is prioritized after the foundational features.

4.2.2 List of categorized requirements

Following the defined requirements and their assigned priority, the following table has been made containing all the planned framework functionalities.

Feature	**Priority**	**Description**
Void event	Must have	Event with no payload
Bool event	Must have	Event with boolean payload
Int event	Must have	Event with integer payload
Float event	Must have	Event with float payload
String event	Must have	Event with string payload
Vector2 event	Must have	Event with 2D vector payload
Vector3 event	Must have	Event with 3D vector payload
GameObject event	Must have	Event with a GameObject reference payload
Bool variable	Must have	Variable holding a boolean value
Int variable	Must have	Variable holding an integer value
Float variable	Must have	Variable holding a float value

String variable	Must have	Variable holding a string value
Vector 2 variable	Must have	Variable holding a 2D vector value
Vector3 variable	Must have	Variable holding a 3D vector value
GameObject collection	Must have	Collection of GameObject references
Void event listener	Must have	Reacts to void events
Bool event listener	Must have	Reacts to bool events
Int event listener	Must have	Reacts to int events
Float event listener	Must have	Reacts to float events
String event listener	Must have	Reacts to string events
Vector 2 event listener	Must have	Reacts to Vector2 events
Vector3 event listener	Must have	Reacts to Vector3 events
GameObject event listener	Must have	Reacts to GameObject events
Transform event	Should have	Event with a Transform reference payload
ScriptableObject event	Should have	Event with a ScriptableObject reference payload
Component event	Should have	Event with a generic component reference payload
Object event	Should have	Event with a generic object reference payload
AudioClip event	Should have	Event with an AudioClip payload

GameObject variable	Should have	Variable holding a GameObject reference
Transform variable	Should have	Variable holding a Transform reference
LayerMask variable	Should have	Variable holding a LayerMask value
Color variable	Should have	Variable holding a color value
Transform collection	Should have	Collection of Transform references
Component collection	Should have	Collection of Component references
Object collection	Should have	Collection of Object references
AudioClip collection	Should have	Collection of Object references
Void Input Action Handler	Should Have	Listens for input actions with no data
Bool Input Action Handler	Should Have	Listens for input actions with boolean data
Int Input Action Handler	Should Have	Listens for input actions with integer data
Float Input Action Handler	Should Have	Listens for input actions with float data
Vector2 Input Action Handler	Should Have	Listens for input actions with Vector2 data

Vector3 Input Action Handler	Should Have	Listens for input actions with Vector3 data
Value Reference	Should Have	Switches between instance and variable values
Variable Repository	Should Have	Save and load scriptable variable values
Transform event listener	Should have	Reacts to Transform events
ScriptableObject event listener	Should have	Reacts to ScriptableObject events
Component event listener	Should have	Reacts to Component events
Object event listener	Should have	Reacts to Object events
AudioClip event listener	Should have	Reacts to AudioClip events
Color event	Could have	Event with a color payload
LayerMask event	Could have	Event with a LayerMask payload
ScriptableObject variable	Could have	Variable holding a ScriptableObject reference
Component variable	Could have	Variable holding a generic Component reference
Object variable	Could have	Variable holding a generic Object reference
AudioClip variable	Could have	Variable holding an AudioClip reference

Int collection	Could have	Collection of integer values
String collection	Could have	Collection of string values
Color collection	Could have	Collection of Object references
Scriptable Dictionary	Could Have	A dictionary editable from the inspector
Bool Variable Observer	Could Have	Reacts to changes in a bool variable
Int Variable Observer	Could Have	Reacts to changes in an int variable
Float Variable Observer	Could Have	Reacts to changes in a float variable
String Variable Observer	Could Have	Reacts to changes in a String variable
Vector2 Variable Observer	Could Have	Reacts to changes in a Vector2 variable
Vector3 Variable Observer	Could Have	Reacts to changes in a Vector3 variable
GameObject Variable Observer	Could Have	Reacts to changes in a GameObject variable
Transform Variable Observer	Could Have	Reacts to changes in a Transform variable
ScriptableObject variable Observer	Could Have	Reacts to changes in a ScriptableObject variable
Component variable Observer	Could Have	Reacts to changes in a Component variable
Object variable Observer	Could Have	Reacts to changes in a Object variable
LayerMask variable Observer	Could Have	Reacts to changes in a LayerMask variable
Color variable Observer	Could Have	Reacts to changes in a Color variable

AudioClip variable Observer	Could Have	Reacts to changes in a AudioClip variable
Color event listener	Could have	Reacts to Color events
LayerMask event listener	Could have	Reacts to LayerMask events
Bool collection	Won't have	Collection of boolean values
Float collection	Won't have	Collection of float values
Vector2 collection	Won't have	Collection of Vector2 values
Vector3 collection	Won't have	Collection of Vector3 values
LayerMask collection	Won't have	Collection of LayerMask references

Table 1: Planned functionalities for the SODD Framework, categorized using MoSCoW. (Own elaboration)

4.2.3 Definition of project's environment and workflow

Unity Version

The development of the SODD Framework will be anchored to Unity 2022.3, which is the most recent LTS version.

Development Platform

GitHub has been selected as the primary platform for hosting the framework's repository. This choice is grounded in the following key advantages:

- **Popularity:** GitHub is widely used, especially for open-source projects, making it a familiar environment for contributors.
- **Project Management Tools:** The inclusion of GitHub Projects facilitates organized and efficient project management.

- **CI/CD Tools:** GitHub Actions provide a seamless integration for Continuous Integration/Continuous Deployment pipelines, enhancing development workflows.
- **Documentation Hosting:** GitHub Pages offers a convenient and free solution for hosting project documentation, ensuring accessibility and ease of maintenance.

GIT branch management

The project's branch management strategy is structured as follows:

- **Main Branch:** This branch hosts the latest stable version of the code, serving as the foundation for all releases.
- **Develop Branch:** Originating from the main branch, the develop branch is designated for integrating new features. This branch acts as a staging area for the next version's features.
- **Feature Branches:** For each new feature, a dedicated branch is created from the develop branch. This approach isolates development work, allowing for focused implementation and testing. Once a feature is completed and stable, it is merged back into the develop branch.

Development workflow

The development workflow for the SODD Framework is structured as follows:

1. **User Story Creation:** Initially, every requirement and feature is translated into a User Story, which is then created from the project management tool as a GitHub Issue that requests a new functionality. These Issues are tagged based on the feature type, such as a Scriptable Event or a Scriptable Variable, and assigned a priority level, such as “Must Have” or “Should Have.”
2. **Branch Creation:** Following the selection of a User Story for development, a dedicated branch is created from the develop branch. This new branch is directly linked with the GitHub Issue containing the User Story, ensuring a clear and

traceable link between the development work and its corresponding requirement.

3. **Development and Pull Request:** After the development of the User Story is completed, the process continues with the creation of a pull request aimed at merging the new features into the develop branch. This step leverages GitHub Actions to automatically run Unity tests and generate a temporary build, verifying the functionality and compatibility of the new features. Successful completion of these checks allows for the merge to proceed, at which point the pull request is closed, and the Issue is marked as completed.

4. **Integration and Verification:** The merging of features into the develop branch signifies that the User Story has met all the criteria outlined in the Definition of Done, marking the development of that feature as complete and closing the User Story.

5. **Preparation for Release:** As the development progresses and the required User Stories are completed, a pull request is prepared to transition the develop branch into the main branch. This step involves another round of automated Unity tests and a build process, conducted through GitHub Actions, to ensure that the new version maintains stability and is ready for release.

6. **Documentation and Release:** The final step in the workflow activates further GitHub Actions upon the successful merge into the main branch. These actions are responsible for generating up-to-date documentation from the code and comments, deploying this documentation to GitHub Pages. Additionally, a new release of the framework is created using semantic versioning, accompanied by comprehensive release notes detailing the changes. The changelog is also updated to reflect the changes in the generated version.

5 Development

5.1 Project Setup

This section outlines the preliminary steps taken for setting up the SODD Framework project before commencing development.

5.1.1 Unity Package Structure

The file structure of the SODD Framework has been organized according to the Unity standard package structure. This structure encompasses the following components:

- **package.json**: This manifest file includes metadata about the package, such as its name, version, dependencies, and additional pertinent information that enables Unity’s Package Manager to accurately identify and manage the package.

- **README**, **CHANGELOG**, **and LICENSE**: These documentation files are essential for users of the package. The README file provides a comprehensive description of the package, the CHANGELOG details the modifications and updates between versions, and the LICENSE outlines the terms under which the package is distributed.

- **Runtime/**: This directory contains all runtime scripts and assets that are incorporated into the final build. These elements constitute the core of the package and are utilized within a game or application to employ its functionality.

- **Editor/**: Situated here are scripts and assets that find their utility solely within the Unity Editor; they are not included in the final game build. Commonly, this includes custom editors, property drawers, and utilities specifically designed for enhancing editor functionality.

- **Tests/**: This folder is reserved for unit and integration tests that can be executed within the Unity Editor. The tests are critical for verifying the functionality and reliability of the package, ensuring that each component performs as expected.

5.1.2 License

The selection of an appropriate license is a crucial decision in software project development, delineating the legal framework within which the software can be used, modified, and distributed. It ensures clarity regarding the legal boundaries and freedoms of both the creators and the users of the software.

For open-source software projects, the MIT License emerges as a highly preferred option due to its permissive nature and simplicity. The characteristics that make the MIT License particularly appealing include:

- **Simplicity and Permissiveness**: The MIT License offers broad freedoms to users and developers, allowing virtually any use of the project, including using, copying, modifying, merging, publishing, distributing, sublicensing, and even selling copies of the software. The only requirement is that the original copyright and license notice be included with any substantial portion of the software distributed. This degree of freedom makes it an attractive choice for both commercial ventures and open-source initiatives.

- **Encourages Open-Source Development**: Its permissive nature not only facilitates the use and distribution of software but also encourages active participation in open-source development. Developers can contribute to or use the project without the burden of navigating through restrictive licensing terms, promoting a culture of collaboration and shared growth.

- **Broad Compatibility**: The MIT License is known for its compatibility with numerous other licenses, simplifying the process of integrating multiple opensource projects. This is particularly beneficial for developers looking to merge different libraries or tools without encountering license incompatibilities.

- **Industry Acceptance**: The straightforward, lenient terms of the MIT License have garnered wide acceptance across the software industry. Its popularity among companies and individual developers alike furthers greater adoption and contribution, enhancing the project's development and reach.

5.1.3 GitHub Actions

To fulfil the Continuous Integration and Continuous Deployment (CI/CD) requirements outlined in the planning phase of the SODD Framework, a suite of GitHub Actions has been established within the repository. These actions are designed to automate various aspects of the development workflow, enhancing efficiency and ensuring adherence to quality standards:

- **Branch Check**: This action is activated upon pull request initiation and serves to safeguard the integrity of the *main* branch by permitting merges solely from the *develop* branch. By verifying the source and target branches of pull requests, this action enforces the designated branching strategy, thereby preserving the stability of the codebase.

- **DocFX Unity Package**: Triggered by pushes to the main branch or manually through *workflow_dispatch*, this action utilizes DocFX to generate and update the project's documentation automatically. The documentation is then deployed to the *gh-pages* branch, ensuring that the project's documentation remains synchronized with the latest code changes in the *main* branch.

- **Release**: Executed in response to pushes to the main branch, this action facilitates the release process by automating the generation of semantic versioning tags, compiles changelogs, and creates GitHub releases. This streamlines the publication of new framework versions, maintaining an organized and accessible record of the project's release history.

- **Unity CI**: This action is invoked by pull requests and can also be triggered manually via *workflow_dispatch*. It is responsible for running Unity tests to validate the framework's stability and functionality. Covering unit tests, coverage reports, and, where applicable, integration tests, this action ensures that all modifications adhere to the project's stringent quality criteria.

5.2 Implementation of features

Every feature in the SODD Framework has been provided with a menu entry, hierarchically organized based on feature type (e.g., Scriptable Event, Scriptable Variable) and data type (e.g., **integer**, **string**). This unified access point—accessible from the Tools section in Unity's menu—speeds up the workflow, allowing developers to access and instantiate any ScriptableObject and Component implementation from the framework's dedicated menu.

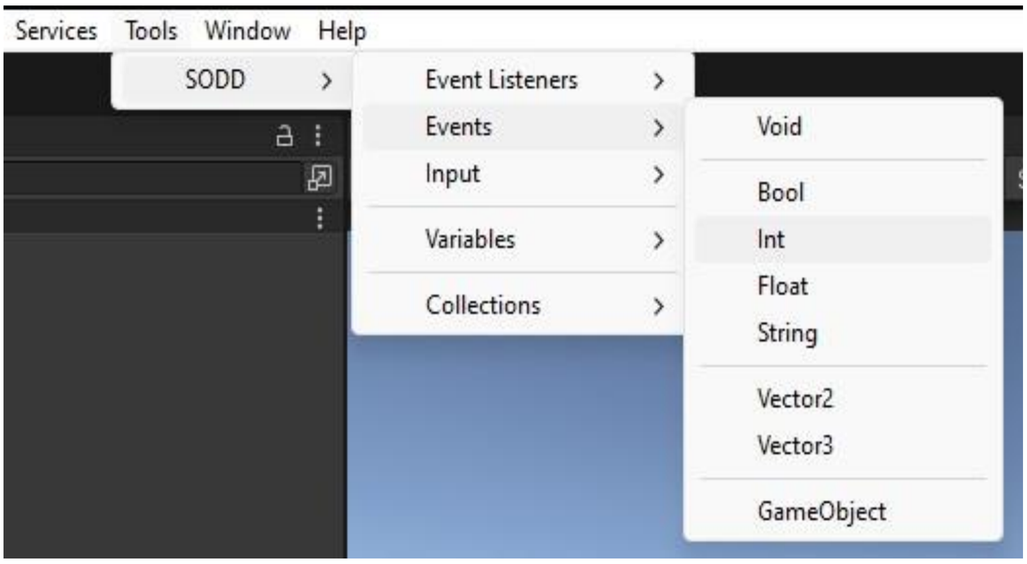
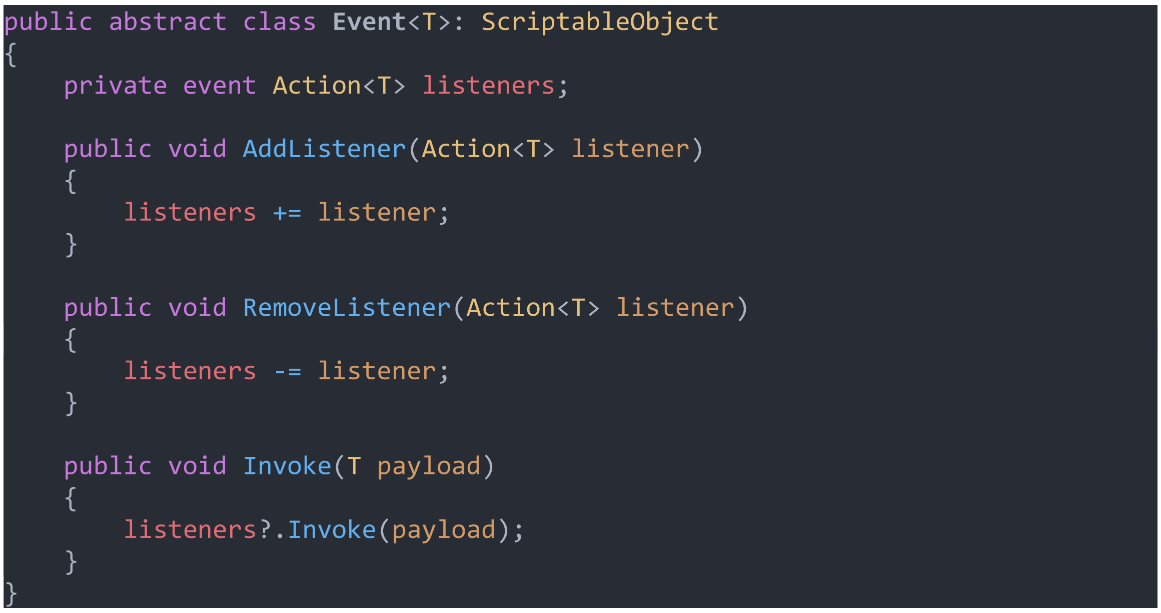


Figure 11: Menu entries for the SODD Framework in Unity, organized by feature and data type. Source: Own elaboration

5.2.1 Scriptable Events

The original concept of Scriptable Events, as introduced by Hipple, is centred around the creation of a list of listeners. Listeners are added to or removed from this list upon subscribing or unsubscribing. When an event is triggered, the ScriptableObject iterates through this list, notifying each subscribed listener in turn.

Building on this concept, the implementation of Scriptable Events has been enhanced through the integration of C# native delegates. By adopting delegates, the system not only maintains its original functionality but also achieves a significant improvement in performance.



```
public abstract class Event<T>: ScriptableObject
{
    private event Action<T> listeners;

    public void AddListener(Action<T> listener)
    {
        listeners += listener;
    }

    public void RemoveListener(Action<T> listener)
    {
        listeners -= listener;
    }

    public void Invoke(T payload)
    {
        listeners?.Invoke(payload);
    }
}
```

Figure 12: Simplified code sample of the base class implementation for all Scriptable Events. Source: Own elaboration

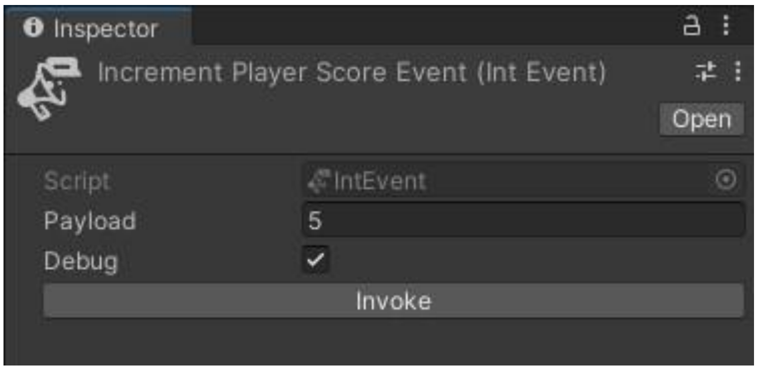
The naming of methods such as **AddListener()**, **RemoveListener()**, and **Invoke**, instead of other names like **Subscribe()**, **Unsubscribe()**, **Raise()** or **Trigger()**, has been a design decision to be consistent with Unity, where UnityEvents provide the same naming. Furthermore, C# delegates also employ the **Invoke()** method name for triggering events. This consistency ensures a more intuitive and unified approach for developers familiar with the Unity ecosystem.

Custom Editor Scripts have been developed for each Scriptable Event type, ensuring that developers can easily edit and debug event properties directly within the Unity Inspector, adhering therefore to the project's stipulations for editability and debuggability.

Result Evaluation

In order to evaluate the functionality of Scriptable Event implementations, a practical use case has been proposed and tested—the management of a player's score in a game.

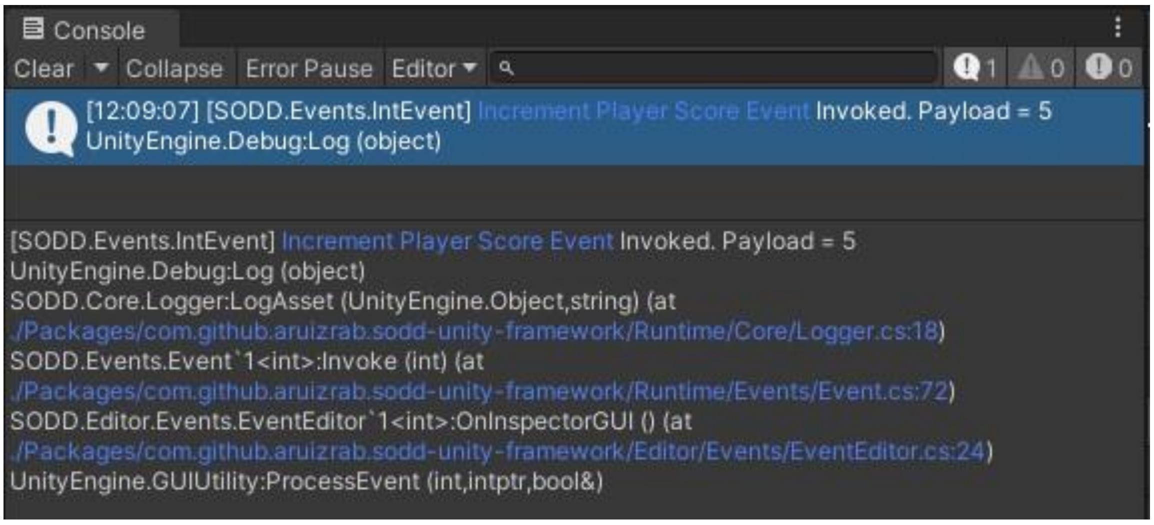
The first step involves creating an **IntEvent** with the goal of signaling the increment of the player's score by a certain amount. By naming the Scriptable Event instance as *Increment Player Score Event*, its' purpose becomes evident.



```
Script      IntEvent
Payload     5
Debug       ✓
```

Figure 13: Inspector view of the Increment Player Score Event. Source: Own elaboration

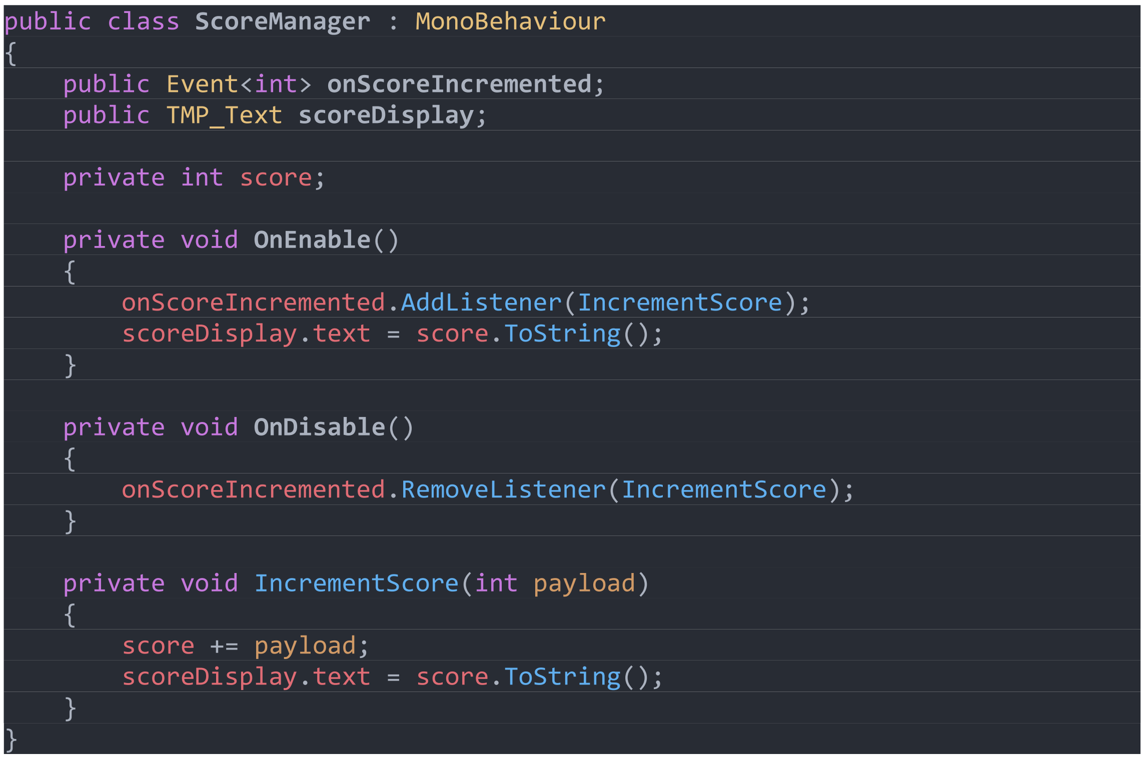
Testing is facilitated through the Inspector window. A developer can specify the payload, enable the debug mode by checking the corresponding toggle, and invoke the event using the provided button. A successful test is indicated by a new entry in the Console, which should detail the event type, the instance name, the payload value, and a clickable link directing to the asset that triggered the event. This log entry not only confirms the event's invocation but also serves as a debugging tool, providing immediate feedback and transparency.



```
[12:09:07] [SODD.Events.IntEvent] Increment Player Score Event Invoked. Payload = 5
UnityEngine.Debug:Log (object)

[SODD.Events.IntEvent] Increment Player Score Event Invoked. Payload = 5
UnityEngine.Debug:Log (object)
SODD.Core.Logger:LogAsset (UnityEngine.Object,string) (at
./Packages/com.github.aruizrab.sodd-unity-framework/Runtime/Core/Logger.cs:18)
SODD.Events.Event`1<int>:Invoke (int) (at
./Packages/com.github.aruizrab.sodd-unity-framework/Runtime/Events/Event.cs:72)
SODD.Editor.Events.EventEditor`1<int>:OnInspectorGUI () (at
./Packages/com.github.aruizrab.sodd-unity-framework/Editor/Events/EventEditor.cs:24)
UnityEngine.GUIUtility:ProcessEvent (int,intptr,bool&)
```

Figure 14: Console output after invoking the Increment Player Score Event with a payload of 5. Source: Own elaboration

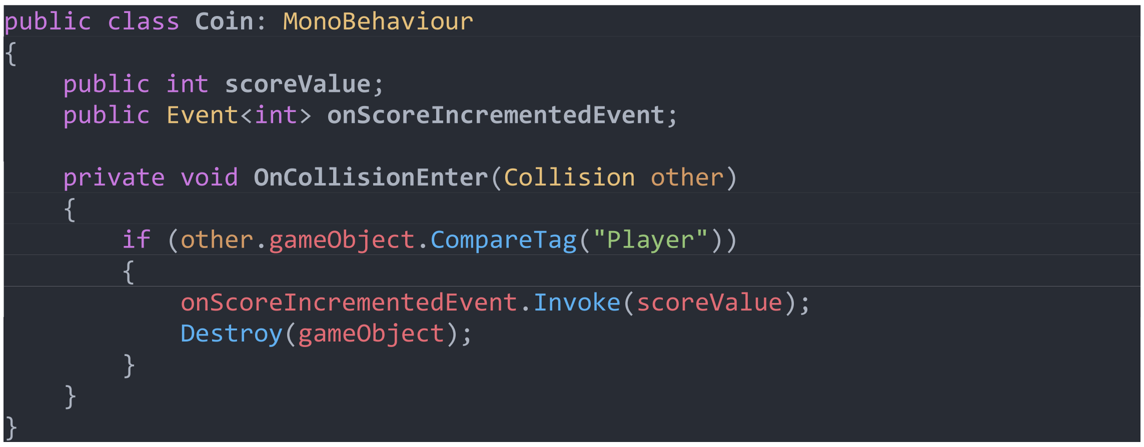
Subsequently, a script dedicated to score management is developed. This script subscribes to the *Increment Player Score Event* and, upon the event's invocation, updates the player's score accordingly. The updated score is then reflected within a UI component, visualizing the change in real-time.



```
public class ScoreManager : MonoBehaviour
{
    public Event<int> onScoreIncremented;
    public TMP_Text scoreDisplay;

    private int score;

    private void OnEnable()
    {
        onScoreIncremented.AddListener(IncrementScore);
        scoreDisplay.text = score.ToString();
    }

    private void OnDisable()
    {
        onScoreIncremented.RemoveListener(IncrementScore);
    }

    private void IncrementScore(int payload)
    {
        score += payload;
        scoreDisplay.text = score.ToString();
    }
}
```

Figure 15: ScoreManager script subscribing to the Increment Player Score Event and updating the score display. Source: Own elaboration

Finally, any script within the game's environment can trigger the *Increment Player Score Event*, for example a coin upon colliding with the player. The **ScoreManager** script, already listening for this event, receives the notification and updates the score without requiring a direct reference to the scripts that trigger the score change. This approach exemplifies the modular and decoupled nature of Scriptable Events, as it allows for disparate game systems to communicate without tightly coupled dependencies.

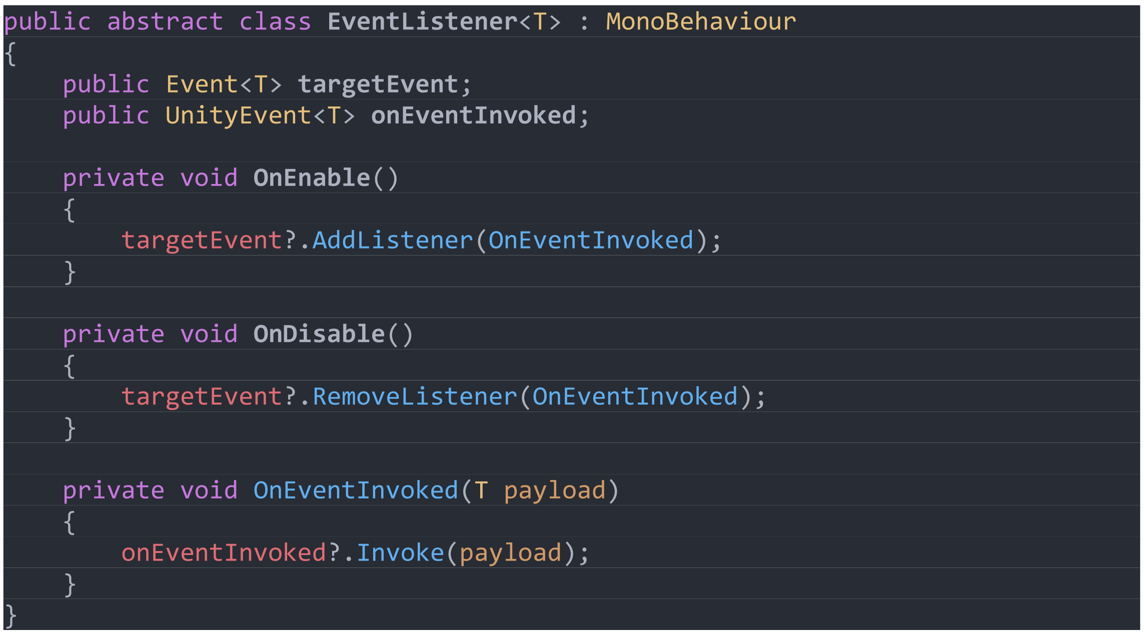


```
public class Coin: MonoBehaviour
{
    public int scoreValue;
    public Event<int> onScoreIncrementedEvent;

    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            onScoreIncrementedEvent.Invoke(scoreValue);
            Destroy(gameObject);
        }
    }
}
```

Figure 16: Coin script triggering the Increment Player Score Event upon collision with the player. Source: Own elaboration

5.2.2 Event Listeners

An Event Listener has been incorporated into the framework for each Scriptable Event type, adhering to the requirements established in the planning stage. These Event Listeners partake in the role of components that trigger a UnityEvent whenever the Scriptable Event they listen to is invoked.



```
public abstract class EventListener<T> : MonoBehaviour
{
    public Event<T> targetEvent;
    public UnityEvent<T> onEventInvoked;

    private void OnEnable()
    {
        targetEvent?.AddListener(OnEventInvoked);
    }

    private void OnDisable()
    {
        targetEvent?.RemoveListener(OnEventInvoked);
    }

    private void OnEventInvoked(T payload)
    {
        onEventInvoked?.Invoke(payload);
    }
}
```

Figure 17: Simplified code sampple of the base class implementation for all Event Listeners. Source: Own elaboration

This design might initially seem redundant, but it serves an important purpose in separating the responsibilities of Scriptable Events from UnityEvents. Scriptable Events function as a global broadcast mechanism, allowing any component across various scenes to subscribe and respond to the events. UnityEvents, on the other hand, are local to the components they are attached to, allowing for the definition of actions within a more constrained context, such as within the components of a GameObject or Prefab.

Moreover, given that the actions reacting to a Scriptable Event are created and modified from the Unity Event inspector, the need for changes in the code is removed when an alteration in behavior requirements occur.

Result Evaluation

The evaluation of Event Listener implementations expands upon the example stated earlier in the evaluation of Scriptable Events, consisting in the management of a player’s score.

To start, an Int Event Listener is added to the GameObject that holds the Score Manager component. Through the Unity Inspector, the "Increment Player Score Event" is assigned to this listener. Within the Unity Event section of the listener, the Score Manager's method responsible for incrementing the score is called, with the event's payload passed dynamically.

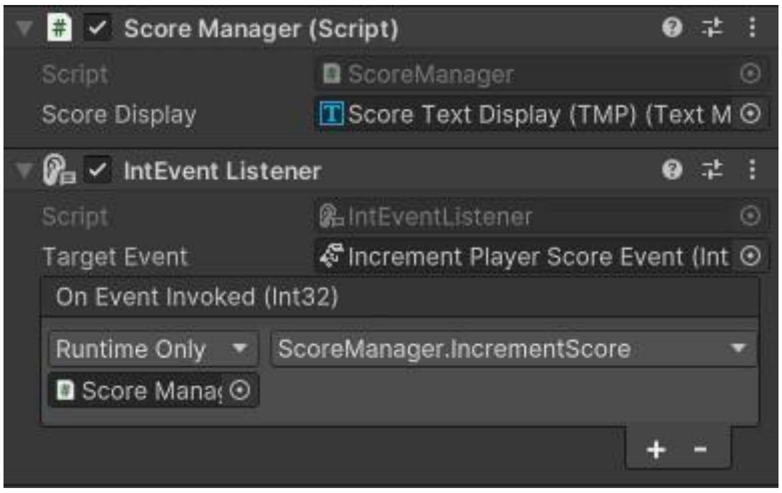
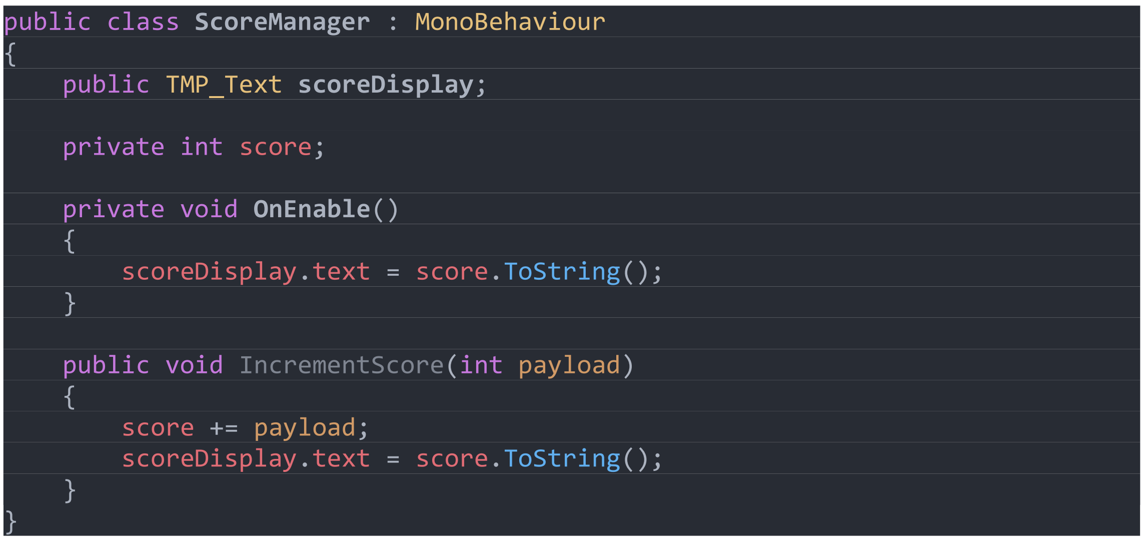

Score Manager (Script)
Script: ScoreManager
Score Display: Score Text Display (TMP) (Text M
IntEvent Listener
Script: IntEventListener
Target Event: Increment Player Score Event (Int
On Event Invoked (Int32)
Runtime Only
ScoreManager.IncrementScore
Score Mana(

Figure 18: IntEvent Listener setup in the Unity Inspector, linking the Increment Player Score Event to the ScoreManager's increment method. Source: Own elaboration

This configuration decouples the Score Manager from the Int Event. Previously, the Score Manager script would have to directly reference the event and include logic for subscription and notification handling. With the introduction of the Event Listener, the Score Manager is now streamlined to focus solely on score-related logic. In practice, this means the Score Manager no longer needs to be aware of the existence of the Scriptable event, its only concern is to update the score when instructed.



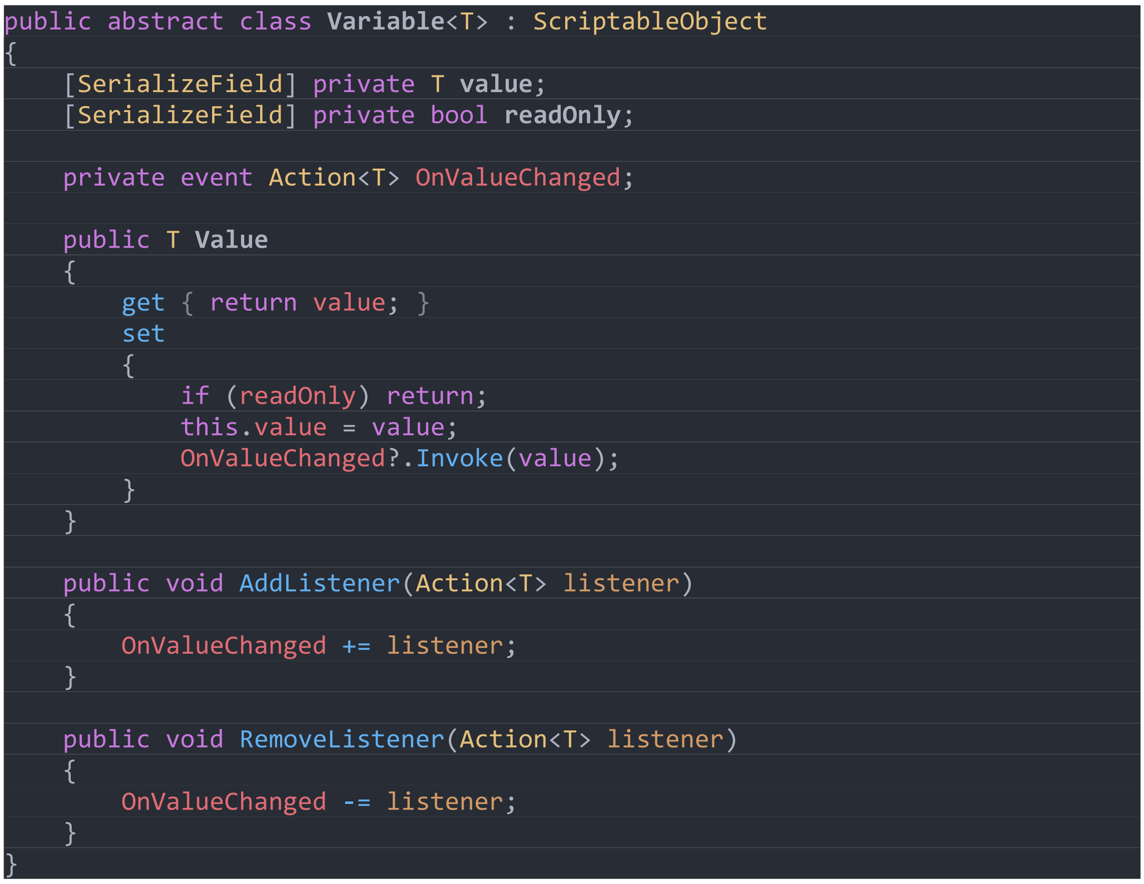
```
public class ScoreManager : MonoBehaviour
{
    public TMP_Text scoreDisplay;

    private int score;

    private void OnEnable()
    {
        scoreDisplay.text = score.ToString();
    }

    public void IncrementScore(int payload)
    {
        score += payload;
        scoreDisplay.text = score.ToString();
    }
}
```

Figure 19: Simplified ScoreManager script after introducing Event Listeners, focusing solely on score-related logic. Source: Own elaboration

For game designers, this modular approach is highly advantageous. They are given the freedom to dictate game behaviours from the Unity Editor—like what happens when a player's score needs to be updated—without modifying the underlying code. This not only simplifies the Score Manager script but also empowers designers to prototype, iterate, and define game behaviours with greater independence and efficiency.

5.2.3 Scriptable Variables

In line with the set requirements, Scriptable Variables have been implemented to encapsulate a value while furthermore providing the functionality to invoke an event upon any change to this value. The invocation event for a Scriptable Variable is designed using a C# delegate, mirroring the strategy employed in the implementation of Scriptable Events.



```
public abstract class Variable<T> : ScriptableObject
{
    [SerializeField] private T value;
    [SerializeField] private bool readOnly;

    private event Action<T> OnValueChanged;

    public T Value
    {
        get { return value; }
        set
        {
            if (readOnly) return;
            this.value = value;
            OnValueChanged?.Invoke(value);
        }
    }

    public void AddListener(Action<T> listener)
    {
        OnValueChanged += listener;
    }

    public void RemoveListener(Action<T> listener)
    {
        OnValueChanged -= listener;
    }
}
```

Figure 20: Simplified code sample of the base class implementation for all Scritpable Variables. Source: Own elaboration

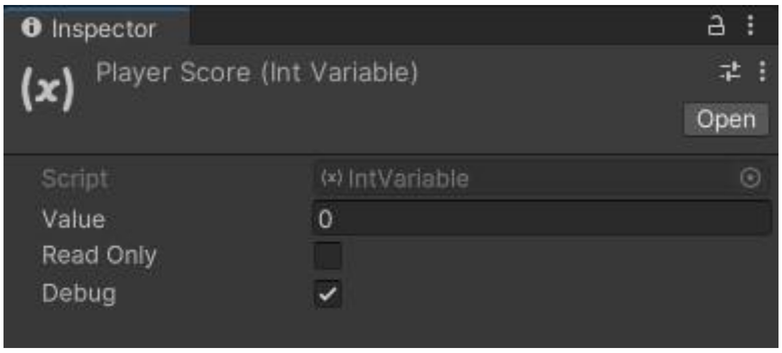
The development of Scriptable Variables also includes the creation of custom editor scripts in order to ensure editability and debuggability.

Result Evaluation

The evaluation of Scriptable Variable implementations focuses on continuing the use case of managing player score.

In the current arrangement, the score value is maintained within the Score Manager. This requires any script needing to access the current score to reference the score manager. Furthermore, the score manager bears the responsibility for both managing the score and its display within the user interface. Ideally, the task of user interface management should be delegated to a distinct script to ensure a separation of concerns and enhance modularity.

As a solution, an Int Variable has been instantiated, with the purpose of storing the current player’s score. The chosen name for the variable instance, "Player Score," effectively communicates its intended purpose to developers and game designers.

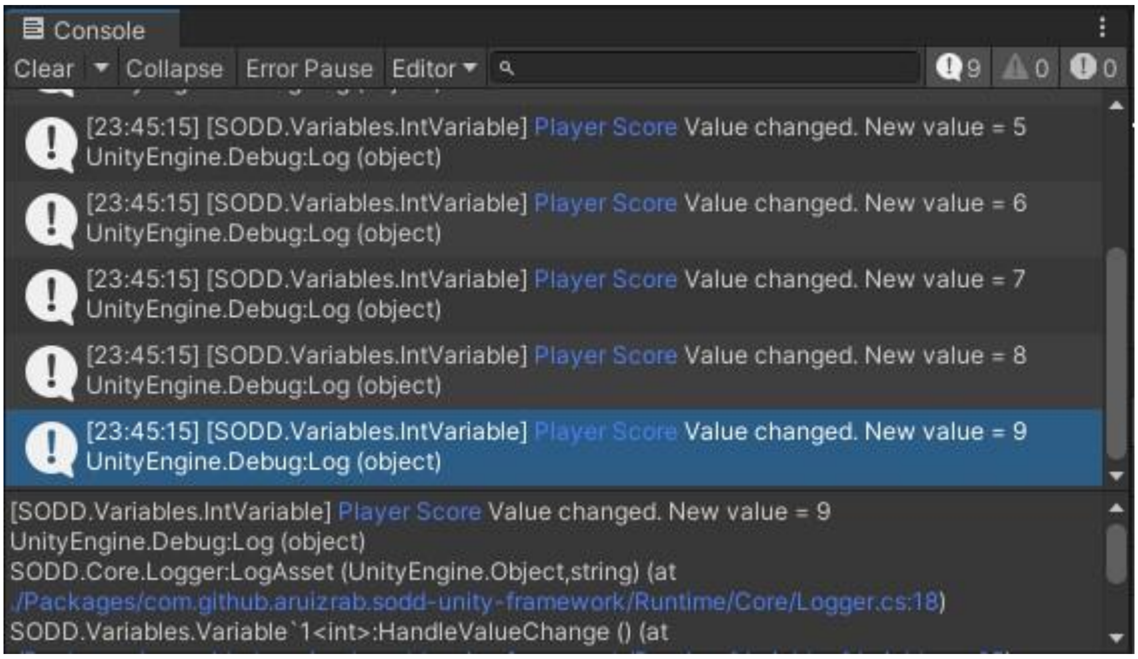


Player Score (Int Variable)

Script	(x) IntVariable
Value	0
Read Only	☐
Debug	✓

Figure 21: Inspector view of the Player Score Scriptable Variable with options for setting the value, read-only status, and debug mode. Source: Own elaboration

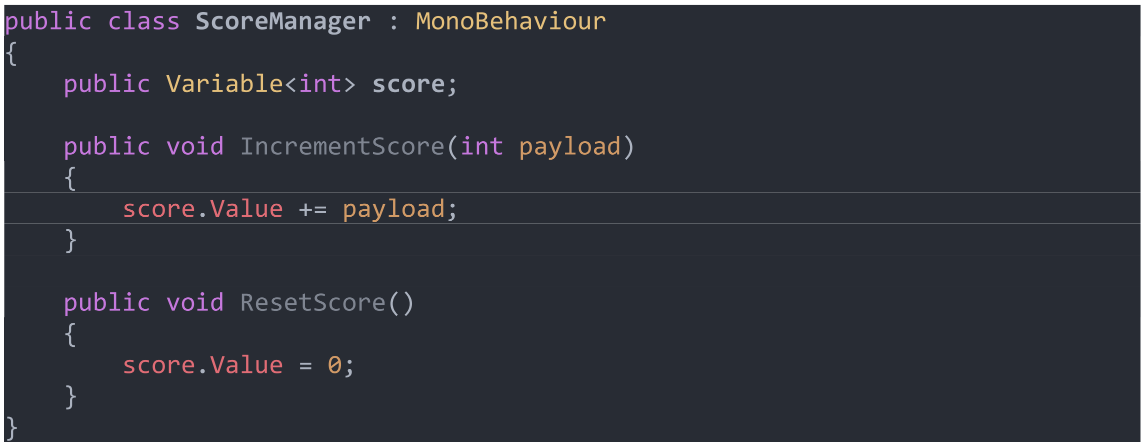
The variable's current value is accessible within Unity's inspector, enabling developers to observe and adjust its value both during edit mode and runtime. Additionally, toggle options are available to restrict value modifications from scripts, in case the variable is intended to be constant, and to activate the debug mode. The debugging of value alterations in the variable can be easily tested by modifying the value in the inspector with the debug mode activated, leading to the logging of these changes in the console. The log format adheres to the format established in the implementation of Scriptable Events, detailing the variable type, instance name, and updated value.



```
[23:45:15] [SODD.Variables.IntVariable] Player Score Value changed. New value = 5
UnityEngine.Debug:Log (object)
[23:45:15] [SODD.Variables.IntVariable] Player Score Value changed. New value = 6
UnityEngine.Debug:Log (object)
[23:45:15] [SODD.Variables.IntVariable] Player Score Value changed. New value = 7
UnityEngine.Debug:Log (object)
[23:45:15] [SODD.Variables.IntVariable] Player Score Value changed. New value = 8
UnityEngine.Debug:Log (object)
[23:45:15] [SODD.Variables.IntVariable] Player Score Value changed. New value = 9
UnityEngine.Debug:Log (object)

[SODD.Variables.IntVariable] Player Score Value changed. New value = 9
UnityEngine.Debug:Log (object)
SODD.Core.Logger:LogAsset (UnityEngine.Object,string) (at
./Packages/com.github.aruizrab.sodd-unity-framework/Runtime/Core/Logger.cs:18)
SODD.Variables.Variable`1<int>:HandleValueChange () (at
```

Figure 22: Console output showing the Player Score value changes with debug mode activated, detailing the new values. Source: Own elaboration

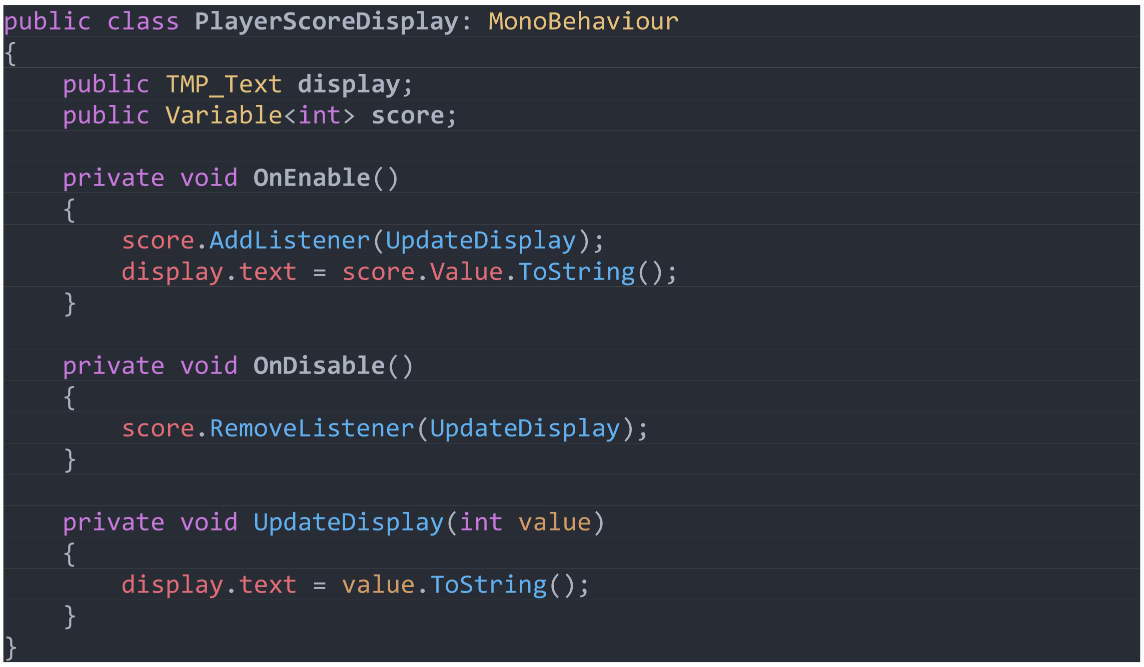
In this stage, the Score Manager can be revised to reference this external variable, eliminating the need to internally store the score value. Consequently, the Score Manager now utilizes a Scriptable Variable reference, enabling the straightforward referencing of the "Player Score" variable via drag-and-drop from the Unity inspector. This adjustment refines the Score Manager's role to exclusively managing score alterations in response to game events.



```
public class ScoreManager : MonoBehaviour
{
    public Variable<int> score;

    public void IncrementScore(int payload)
    {
        score.Value += payload;
    }

    public void ResetScore()
    {
        score.Value = 0;
    }
}
```

Figure 23: Revised ScoreManager script referencing an external Scriptable Variable for managing player score. Source: Own elaboration

Finally, the development of a new script dedicated to displaying the value of the player’s score in a UI component has been carried out. By subscribing to changes in the variable's value, this script ensures the UI is updated in a reactive fashion. Although this new script is aimed at presenting the current score.



```
public class PlayerScoreDisplay: MonoBehaviour
{
    public TMP_Text display;
    public Variable<int> score;

    private void OnEnable()
    {
        score.AddListener(UpdateDisplay);
        display.text = score.Value.ToString();
    }

    private void OnDisable()
    {
        score.RemoveListener(UpdateDisplay);
    }

    private void UpdateDisplay(int value)
    {
        display.text = value.ToString();
    }
}
```

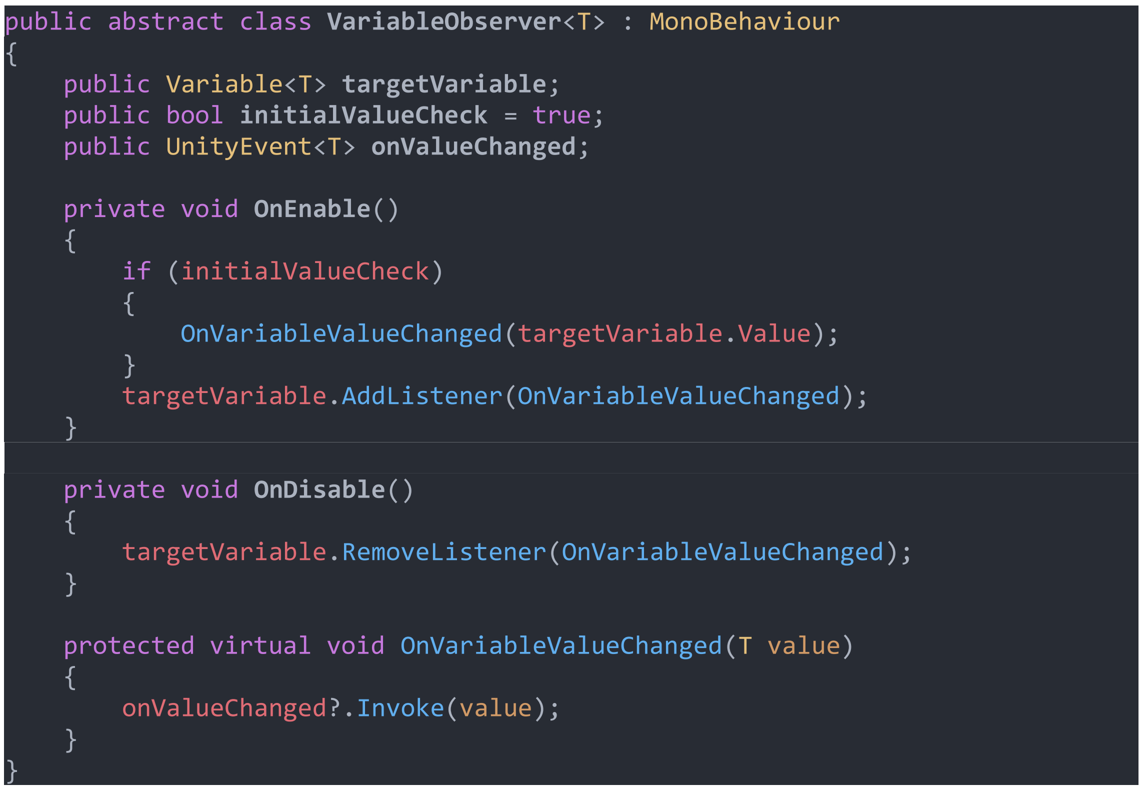
Figure 24: PlayerScoreDisplay script updating the UI component with the current player score by subscribing to the Scriptable Variable. Source: Own elaboration

This example showcases Scriptable Variables as a potent tool within game development. When synergized with Scriptable Events, these variables facilitated the creation of a score management system distinguished by its decoupling, modularity, and the simplicity with which it can be edited and debugged directly from the Unity Editor. Moreover, this system's design simplifies the process of scaling or enhancing the game's features. For instance, implementing new functionalities such as augmenting the score by a different value upon defeating an enemy, or resetting the score following the player's death, can be achieved with minimal effort. These enhancements involve the creation of new events and configuring the desired behaviours in the Event Listener inspector, negating the necessity for direct code modifications. This approach provides game designers with an intuitive and efficient workflow, thereby enabling rapid iteration and testing of new game behaviours.

5.2.4 Variable Observers

Variable Observers, as delineated in the requirements, have been implemented analogously to Event Listeners. Each Variable Observer references a Scriptable

Variable of a specific type, subscribes for changes in its' value, and triggers a UnityEvent in response, passing the new value as a parameter. From the Inspector, behavior can be defined within the UnityEvent, such as updating a value in a UI component. Additionally, an initial state check option has been included, allowing the observer to trigger the UnityEvent with the initial value of the variable when enabled.



```
public abstract class VariableObserver<T> : MonoBehaviour
{
    public Variable<T> targetVariable;
    public bool initialValueCheck = true;
    public UnityEvent<T> onValueChanged;

    private void OnEnable()
    {
        if (initialValueCheck)
        {
            OnVariableValueChanged(targetVariable.Value);
        }
        targetVariable.AddListener(OnVariableValueChanged);
    }

    private void OnDisable()
    {
        targetVariable.RemoveListener(OnVariableValueChanged);
    }

    protected virtual void OnVariableValueChanged(T value)
    {
        onValueChanged?.Invoke(value);
    }
}
```

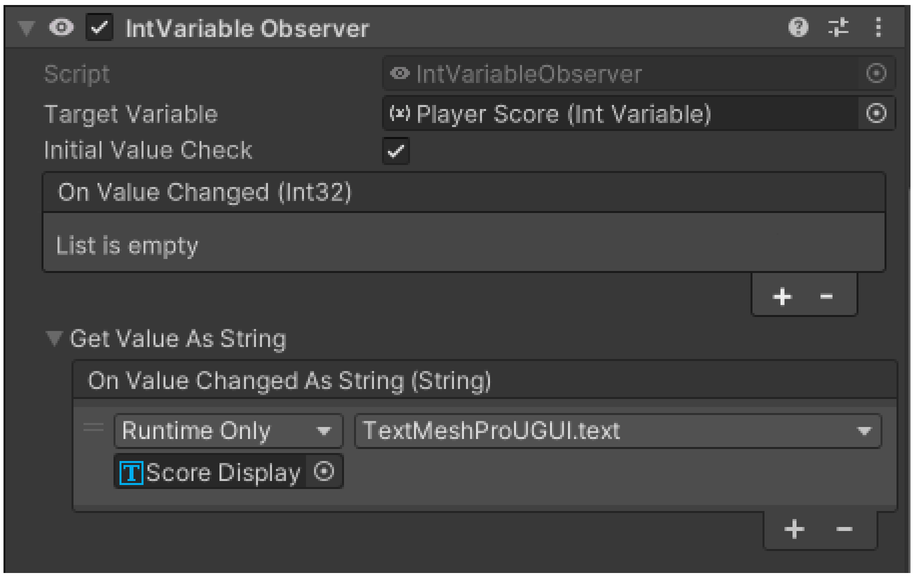
Figure 25: Simplified code sample of the base class implementation for all Variable Observer implementations. Source: Own elaboration

Result Evaluation

The evaluation of Variable Observer implementations builds upon the previous use case of managing player score.

Previously, the script PlayerScoreDisplay was utilized to display the value of the Player Score variable in the UI. However, by employing an Int Variable Observer, which includes an additional UnityEvent that converts and passes the Player Score value as a string when it changes, the IntVariableDisplay script becomes redundant.

The function to update the UI component is specified directly within the UnityEvent of the observer.


IntVariable Observer

Script: IntVariableObserver

Target Variable: Player Score (Int Variable)

Initial Value Check: ✓

On Value Changed (Int32)

List is empty

Get Value As String

On Value Changed As String (String)

Runtime Only | TextMeshProUGUI.text

Score Display

Figure 26: IntVariable Observer setup in the Unity Inspector, linking the Player Score variable to the UI component for real-time updates. Source: Own elaboration

After setting up the Variable Observer and entering Play Mode, the UI element displaying the Player Score is initially set to the variable's value and updates automatically whenever the variable's value changes.

This use case exemplifies how Variable Observers enhance the modularity and responsiveness of the game architecture, ensuring that UI elements and other components remain synchronized with the underlying data without requiring additional scripting effort.

5.2.5 Variable Repository

The Variable Repository has been developed to manage the persistent storage and retrieval of Scriptable Variables, ensuring efficient saving and loading of data such as player settings and game states across game sessions.

To achieve this functionality, the Variable Repository is implemented as a ScriptableObject containing a serialized list of Scriptable Variables. Developers can add variables to this list that they wish to persist across game sessions. The variables can then be loaded and saved when the game starts and closes, either through configuration toggles in the repository settings or manually by invoking the provided public methods. The data is stored using binary serialization, ensuring non-human-readable, compact, and secure storage.

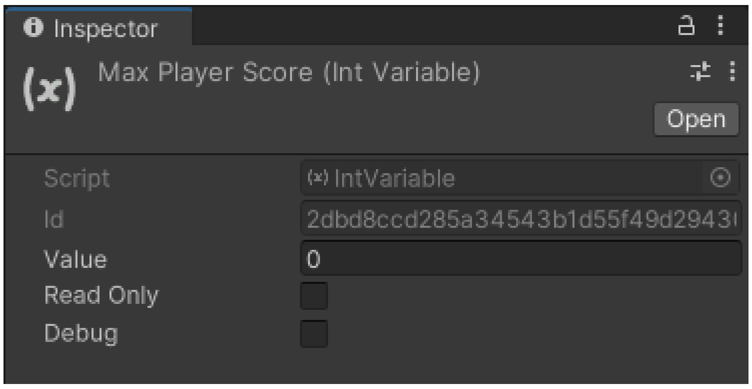
Additionally, a custom editor script has been developed for the Variable Repository for two primary purposes. First, it adds a debug option in the repository inspector to log whenever variable data is loaded or saved. Second, it provides buttons in the inspector for developers to manually save and load variables for testing purposes.

As a game expands in size and complexity, the number of Scriptable Variables increases significantly. Manually adding these variables to the Variable Repository can become a tedious task, particularly if future requirements change and some variables no longer need to be persisted. To streamline this process, Scriptable Variables now extend PersistentScriptableObject class, which includes a new property displayed in the inspector: a toggle to indicate whether the specific variable should be persisted. Subsequently, the custom Editor Script for the Variable Repository type has been enhanced with an additional button. When clicked, this button scans the project files for variables marked for persistence and automatically adds them to the repository list. This feature allows for easy updates to Variable Repositories whenever new variables with the persistence toggle enabled are created or when existing variables need to be removed or marked as non-persistent.

Result Evaluation

The evaluation of the Variable Repository’s functionality builds upon the previous example of handling the player’s score. In this case, two variables are used: the existing Player Score variable and a new variable for tracking the player’s maximum score. Both variables will be stored and retrieved whenever the game is closed and opened.

The first step involves creating an additional Int Variable named “Max Player Score” to store the player’s maximum score.

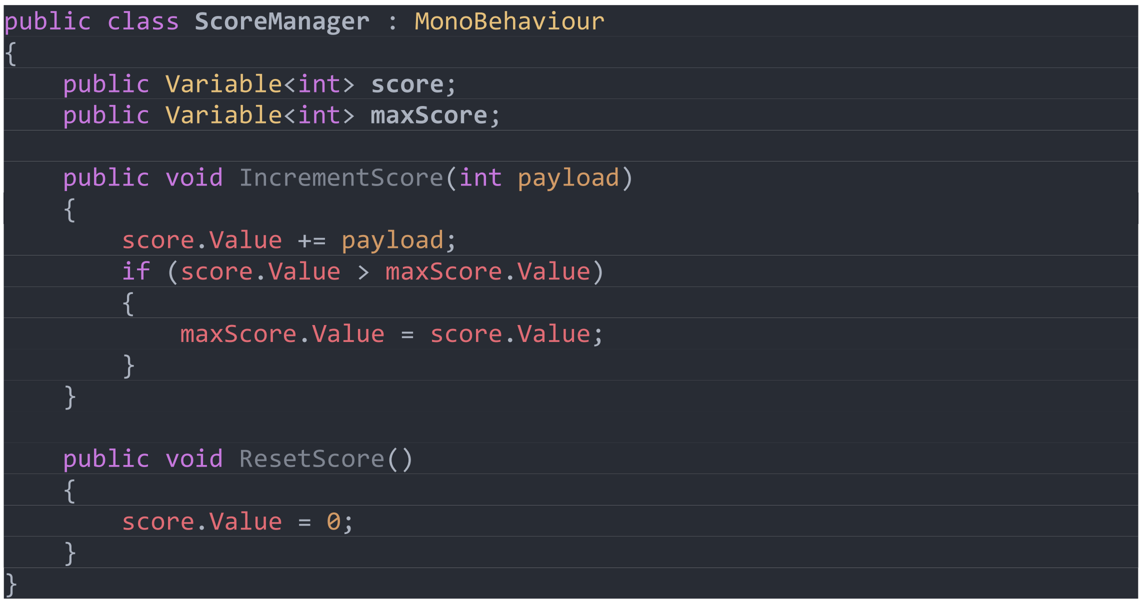


(x) Max Player Score (Int Variable)

Script	(x) IntVariable
Id	2dbd8ccd285a34543b1d55f49d2943
Value	0
Read Only	☐
Debug	☐

Figure 27: Inspector view of the Max Player Score Scriptable Variable. Source: Own elaboration

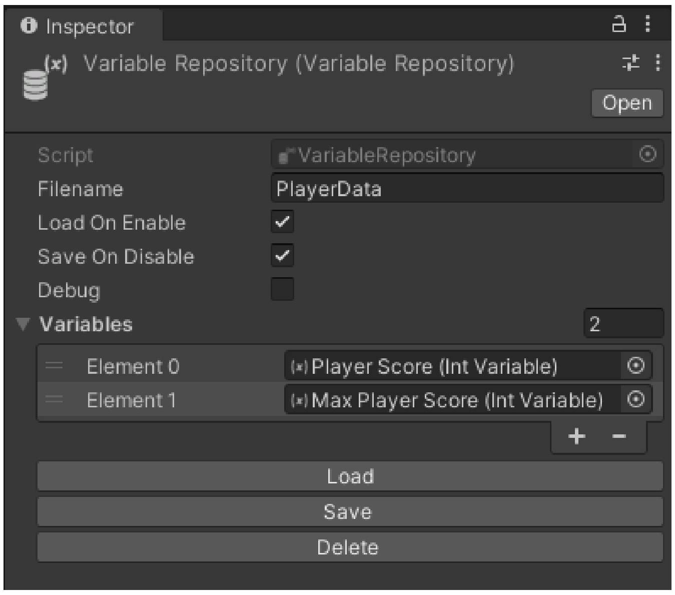
The ScoreManager script, defined in the previous section, has been updated to modify the Player Max Score's value whenever the current score exceeds the maximum score.



```
public class ScoreManager : MonoBehaviour
{
    public Variable<int> score;
    public Variable<int> maxScore;

    public void IncrementScore(int payload)
    {
        score.Value += payload;
        if (score.Value > maxScore.Value)
        {
            maxScore.Value = score.Value;
        }
    }

    public void ResetScore()
    {
        score.Value = 0;
    }
}
```

Figure 28: Updated ScoreManager script to modify the Player Max Score whenever the current score exceeds the maximum score. Source: Own elaboration

The next step involves creating a Variable Repository instance and specifying the file name where the data will be stored. By enabling the options to load data when the game starts and save it when the game closes, it is ensured that the values of the variables remain updated during runtime and across playing sessions. The Player Score and Max Player Score variables can then be added to the list of variables to be persisted in the created Variable Repository.



Inspector

Variable Repository (Variable Repository)

Open

Script VariableRepository

Filename PlayerData

Load On Enable ✓

Save On Disable ✓

Debug

Variables 2

Element 0 Player Score (Int Variable)

Element 1 Max Player Score (Int Variable)

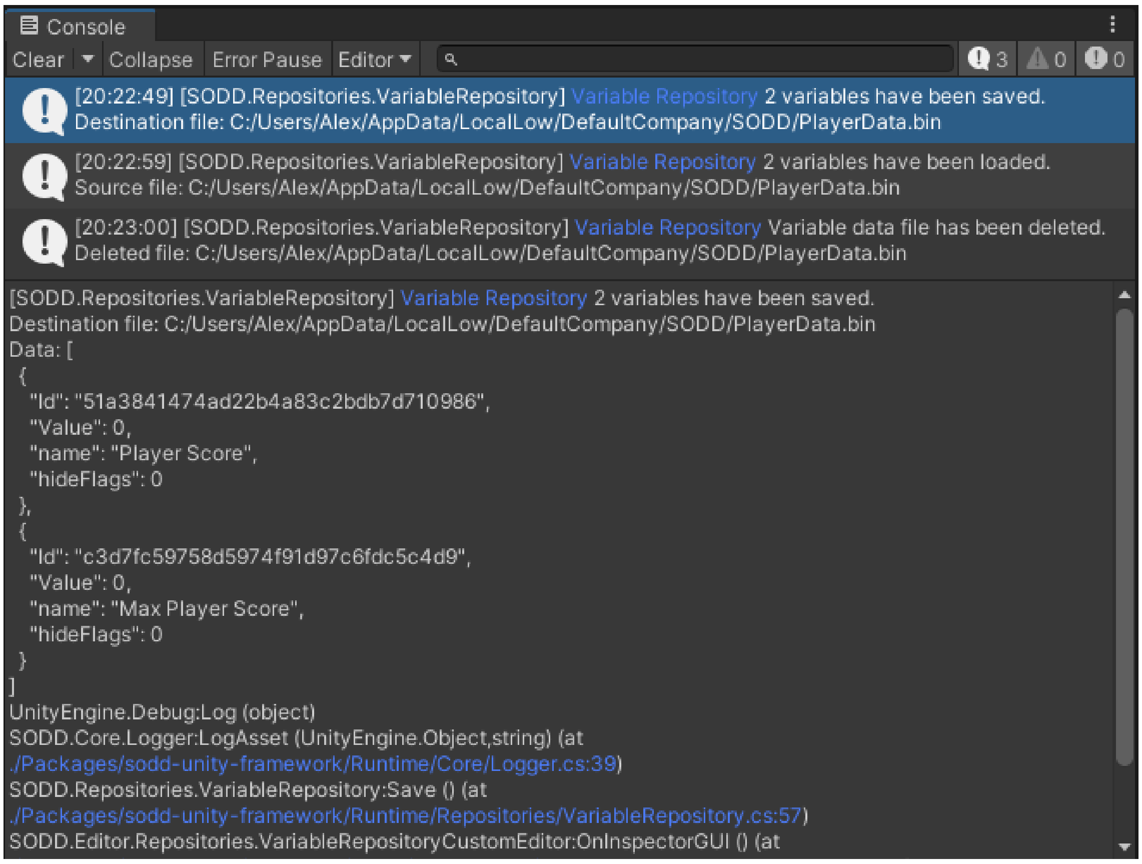
Load

Save

Delete

Figure 29: Variable Repository with Player Score and Max Player Score variables added to the list of persistent variables. Source: Own elaboration

At this stage, testing the repository’s functionality is facilitated through the Inspector window, where buttons allow for saving, loading, and deleting the data stored in the file system of the variables. Additionally, enabling the debug option generates new entries in the Unity Console for every action, providing information about the action performed, the file path where the data is stored, and details about the affected variables.



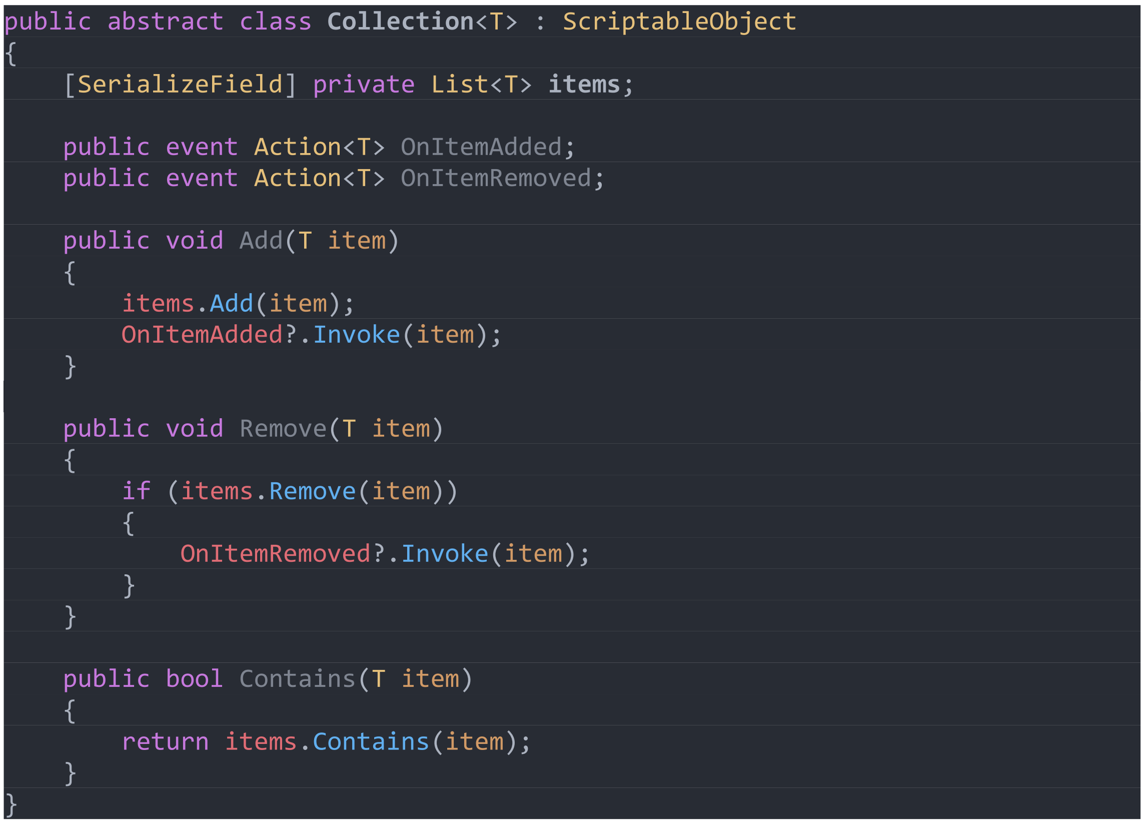
```
[20:22:49] [SODD.Repositories.VariableRepository] Variable Repository 2 variables have been saved.
Destination file: C:/Users/Alex/AppData/LocalLow/DefaultCompany/SODD/PlayerData.bin
[20:22:59] [SODD.Repositories.VariableRepository] Variable Repository 2 variables have been loaded.
Source file: C:/Users/Alex/AppData/LocalLow/DefaultCompany/SODD/PlayerData.bin
[20:23:00] [SODD.Repositories.VariableRepository] Variable Repository Variable data file has been deleted.
Deleted file: C:/Users/Alex/AppData/LocalLow/DefaultCompany/SODD/PlayerData.bin

[SODD.Repositories.VariableRepository] Variable Repository 2 variables have been saved.
Destination file: C:/Users/Alex/AppData/LocalLow/DefaultCompany/SODD/PlayerData.bin
Data: [
  {
    "Id": "51a3841474ad22b4a83c2bdb7d710986",
    "Value": 0,
    "name": "Player Score",
    "hideFlags": 0
  },
  {
    "Id": "c3d7fc59758d5974f91d97c6fdc5c4d9",
    "Value": 0,
    "name": "Max Player Score",
    "hideFlags": 0
  }
]
UnityEngine.Debug:Log (object)
SODD.Core.Logger:LogAsset (UnityEngine.Object,string) (at
./Packages/sodd-unity-framework/Runtime/Core/Logger.cs:39)
SODD.Repositories.VariableRepository:Save () (at
./Packages/sodd-unity-framework/Runtime/Repositories/VariableRepository.cs:57)
SODD.Editor.Repositories.VariableRepositoryCustomEditor:OnInspectorGUI () (at
```

Figure 30: Unity Console output showing actions performed by the Variable Repository, including saving, loading, and deleting variable data with debug information. Source: Own elaboration

Finally, by creating a build of the game and executing it, it can be observed that Player Score and Max Player Score are successfully persisted between game sessions, thereby overcoming the issue of Unity not persisting ScriptableObject data in the builds.

5.2.6 Scriptable Collections

The implementation of Scriptable Collections has been executed in alignment with the specifications outlined during the planning phase. These implementations include a serialized list that can be edited from the Unity inspector, the invocation of events upon the addition and removal of items, and a debug toggle to log these changes.



```
public abstract class Collection<T> : ScriptableObject
{
    [SerializeField] private List<T> items;

    public event Action<T> OnItemAdded;
    public event Action<T> OnItemRemoved;

    public void Add(T item)
    {
        items.Add(item);
        OnItemAdded?.Invoke(item);
    }

    public void Remove(T item)
    {
        if (items.Remove(item))
        {
            OnItemRemoved?.Invoke(item);
        }
    }

    public bool Contains(T item)
    {
        return items.Contains(item);
    }
}
```

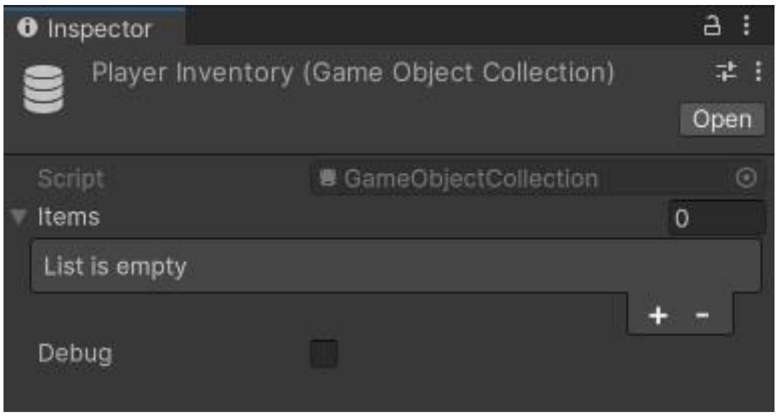
Figure 31: Simplified code sample of the base class implementation for all Scriptable Collections. (Own elaboration)

Due to the nuances of Unity's serialization of lists in the inspector and the constraints on how editor scripts interact with serialized lists, it has not been feasible to enable log changes made to a collection directly from the inspector. Consequently, the debugging functionality is limited to modifications performed at runtime through scripts.

Result Evaluation

To evaluate the functionality of the Scriptable Collection implementations, a practical use case, termed "lock and key," has been introduced. This scenario involves a door that can only be unlocked by the player using a specific key. The level may contain multiple doors and keys, but each door requires a distinct key to unlock.

Firstly, a GameObject Collection named "Player Inventory" has been created to represent the player's inventory throughout the level.

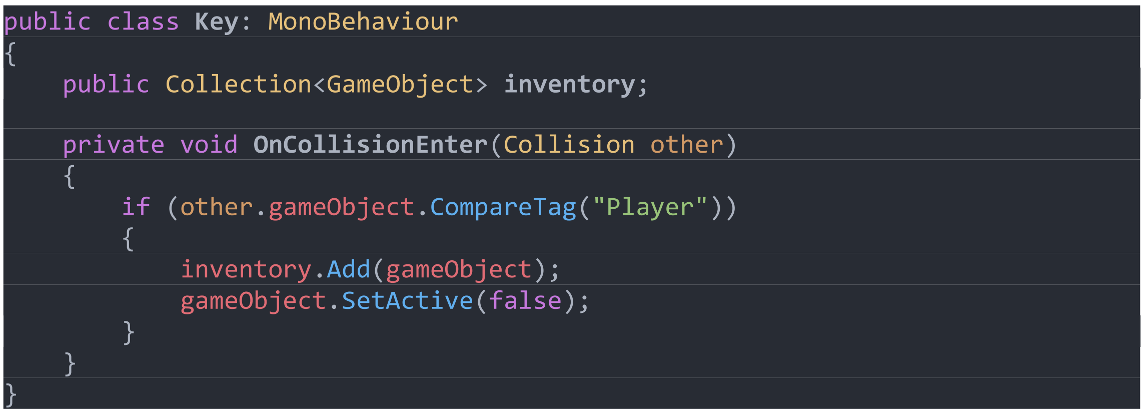


Player Inventory (Game Object Collection)

Script GameObjectCollection
Items 0
List is empty
Debug

Figure 32: Inspector view of the Player Inventory Scriptable Collection for managing the player's inventory items. Source: Own elaboration

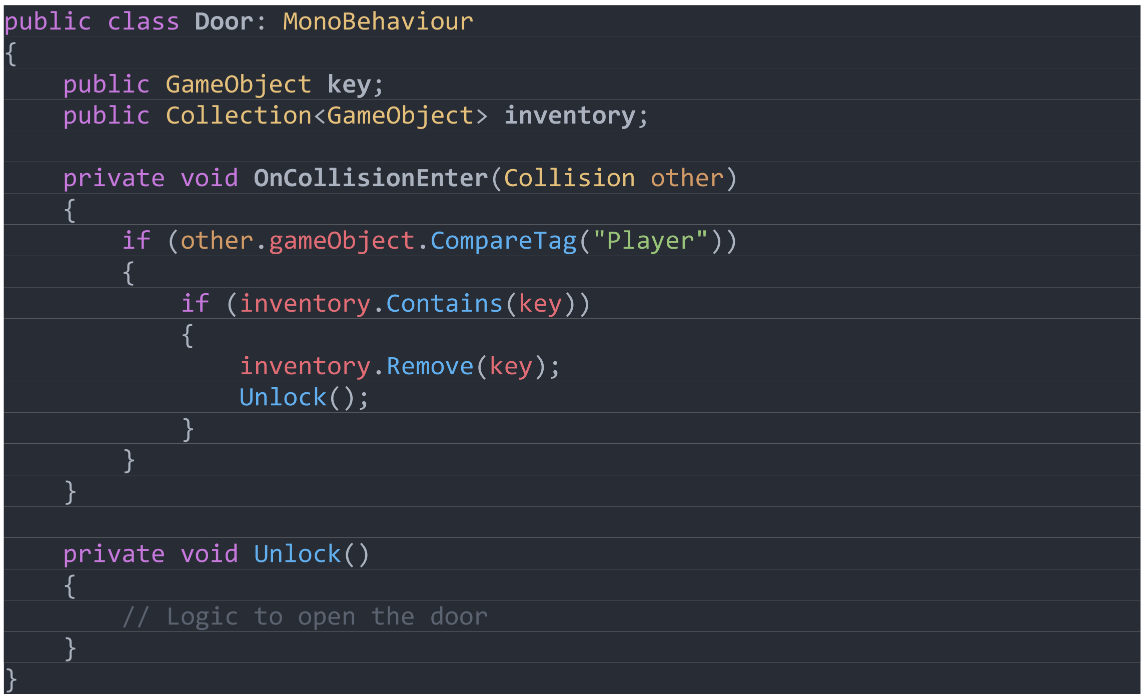
Keys are represented by GameObjects equipped with scripts that, upon collision with the player, add themselves to the "Player Inventory" collection.



```
public class Key: MonoBehaviour
{
    public Collection<GameObject> inventory;

    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            inventory.Add(gameObject);
            gameObject.SetActive(false);
        }
    }
}
```

Figure 33: Key script adding the key GameObject to the Player Inventory collection upon collision with the player. Source: Own elaboration

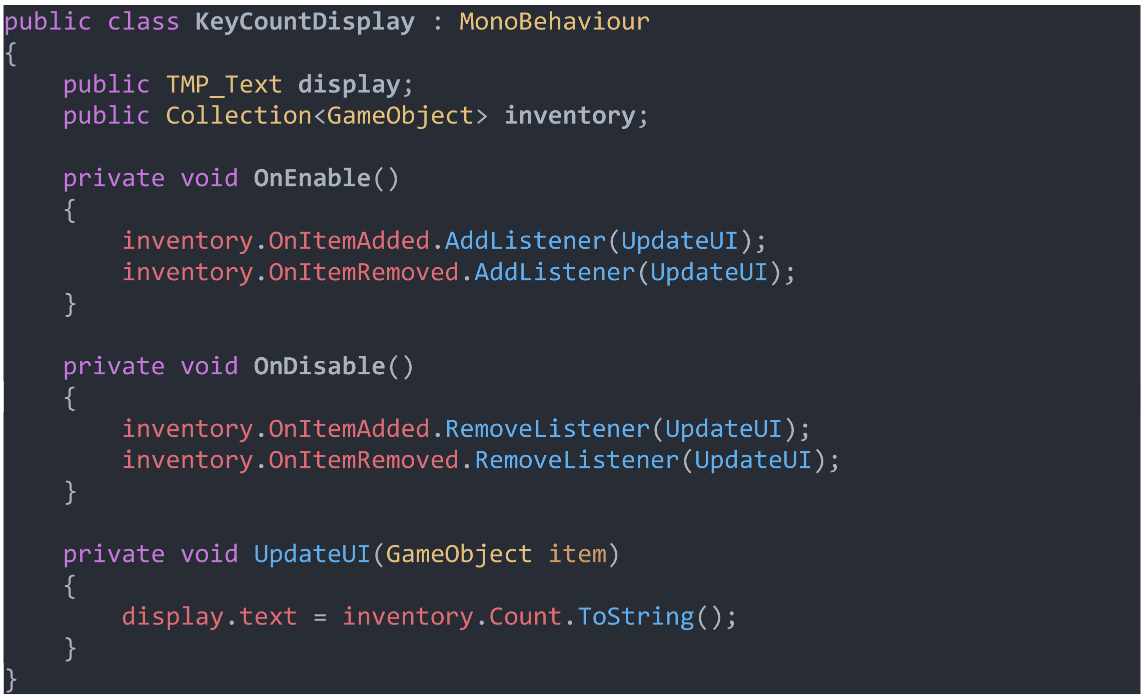
Similarly, each door is a GameObject with a script that references the inventory and the specific key required for unlocking. When the player collides with a door, the script checks if the required key is present in the player's inventory.



```
public class Door: MonoBehaviour
{
    public GameObject key;
    public Collection<GameObject> inventory;

    private void OnCollisionEnter(Collision other)
    {
        if (other.gameObject.CompareTag("Player"))
        {
            if (inventory.Contains(key))
            {
                inventory.Remove(key);
                Unlock();
            }
        }
    }

    private void Unlock()
    {
        // Logic to open the door
    }
}
```

Figure 34: Door script checking the player's inventory for the required key and unlocking the door upon collision with the player. Source: Own elaboration

Moreover, an additional script has been implemented to display the number of keys in the player's inventory on a UI element. This script responds to the addition and removal events in the collection by updating a counter accordingly.



```
public class KeyCountDisplay : MonoBehaviour
{
    public TMP_Text display;
    public Collection<GameObject> inventory;

    private void OnEnable()
    {
        inventory.OnItemAdded.AddListener(UpdateUI);
        inventory.OnItemRemoved.AddListener(UpdateUI);
    }

    private void OnDisable()
    {
        inventory.OnItemAdded.RemoveListener(UpdateUI);
        inventory.OnItemRemoved.RemoveListener(UpdateUI);
    }

    private void UpdateUI(GameObject item)
    {
        display.text = inventory.Count.ToString();
    }
}
```

Figure 35: KeyCountDisplay script updating the UI element to reflect the number of keys in the player's inventory. Source: Own elaboration

This example highlights the benefits of utilizing Scriptable Collections, as it demonstrates how different systems can interact with and modify a shared collection in a manner that is both decoupled and efficient.

5.2.7 Input Action Handlers

Input Action Handlers have been implemented to translate Unity’s Input System actions into the framework's Scriptable Events. Each handler is responsible for a single Input Action, monitoring its events when the action is started, performed, and cancelled, and triggering a corresponding Scriptable Event in response.

```
public abstract class InputActionHandler<T> : ScriptableObject
{
    public InputActionReference reference;
    public Event<T> onActionStarted;
    public Event<T> onActionPerformed;
    public Event<T> onActionCanceled;

    private void OnEnable()
    {
        reference.action.started += OnStarted;
        reference.action.performed += OnPerformed;
        reference.action.canceled += OnCanceled;
        reference.action.Enable();
    }

    private void OnDisable()
    {
        reference.action.started -= OnStarted;
        reference.action.performed -= OnPerformed;
        reference.action.canceled -= OnCanceled;
        reference.action.Disable();
    }

    private void OnStarted(InputAction.CallbackContext context)
    {
        onActionStarted.Invoke(context.ReadValue<T>());
    }

    private void OnPerformed(InputAction.CallbackContext context)
    {
        onActionPerformed.Invoke(context.ReadValue<T>());
    }

    private void OnCanceled(InputAction.CallbackContext context)
    {
        onActionCanceled.Invoke(context.ReadValue<T>());
    }
}
```

Figure 36: Simplified code sample of the base class implementation for all Input Action Handler implementations. Source: Own elaboration

The implementation of Input Action Handlers accommodates various data types, contingent on the nature of the Input Action. For instance, a Bool Action Handler manages actions involving buttons, using a Boolean value to indicate whether the button is pressed or released. Another example is a Vector2 Action Handler, which manages actions involving two-dimensional axis, such as a joystick.

Result Evaluation

To evaluate the functionality of the Input Action Handler implementations, a practical use case involving player movement has been tested.

Initially, an InputAction asset was created and configured with an action named "Move," which reads the 2D vector input from the left joystick of a gamepad.

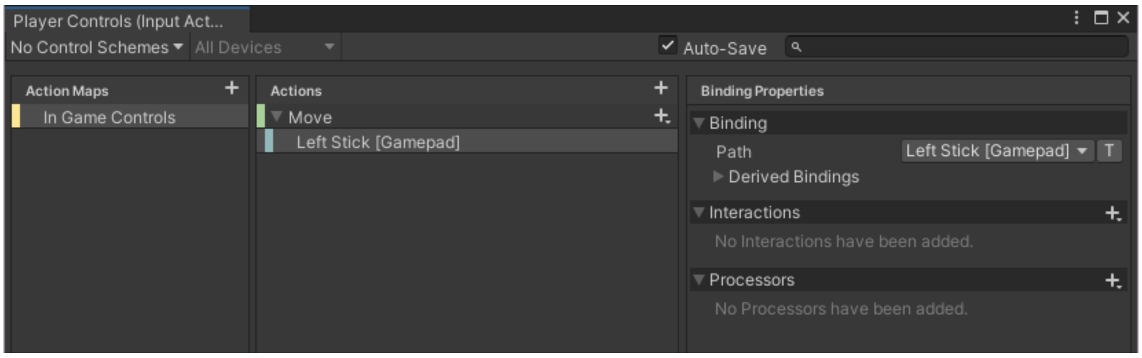
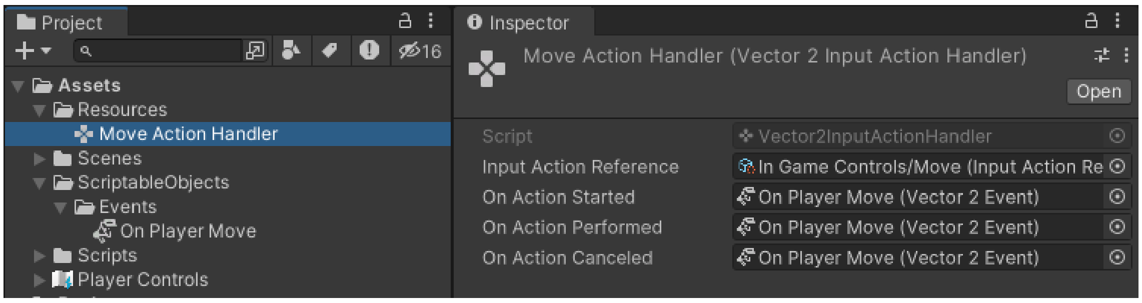


Figure 37: InputAction asset configuration for player movement, reading 2D vector input from the left joystick of a gamepad. Source: Own elaboration

Subsequently, a Vector2 Action Handler, named "Move Action Handler," was created to manage the "Move" action in the previously configured InputAction asset.

The next step involved creating the respective Scriptable Events that would be triggered by the handler in response to the input action. For this case, a single Vector2 Event, named "On Player Move," was created. This event carries a 2D vector payload representing the joystick's value during each phase of the action.



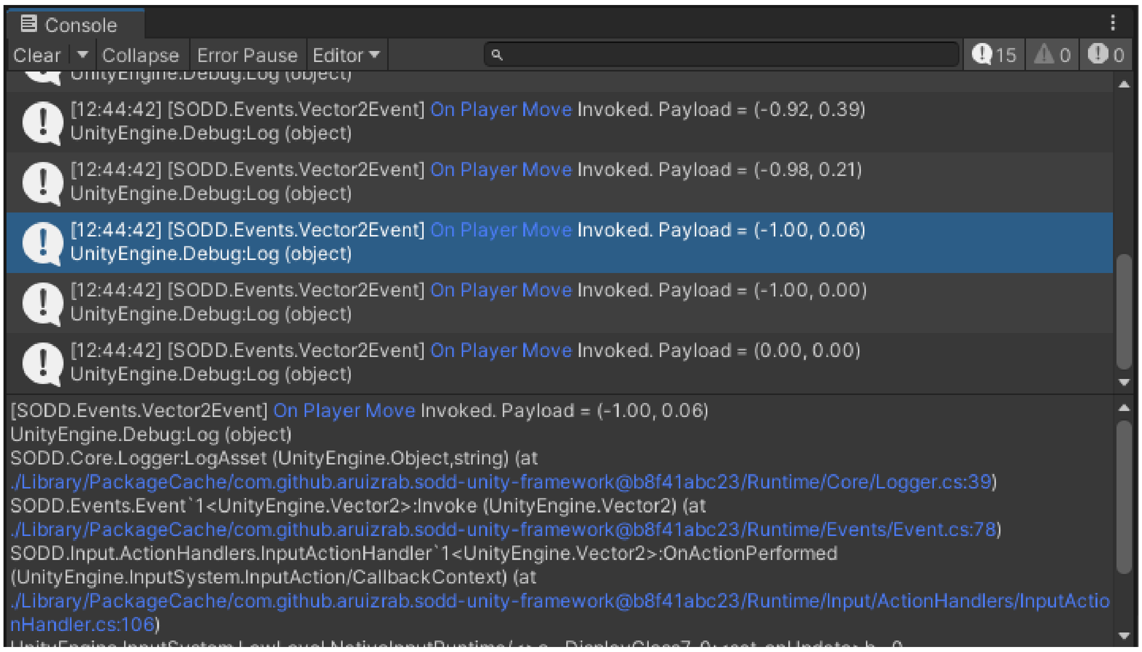
Assets
Resources
Move Action Handler
Scenes
ScriptableObjects
Events
On Player Move
Scripts
Player Controls

Move Action Handler (Vector 2 Input Action Handler)

Script	Vector2InputActionHandler
Input Action Reference	In Game Controls/Move (Input Action Re
On Action Started	On Player Move (Vector 2 Event)
On Action Performed	On Player Move (Vector 2 Event)
On Action Canceled	On Player Move (Vector 2 Event)

Figure 38: Move Action Handler setup in the Unity Inspector, linking the "Move" InputAction to the "On Player Move" Vector2 Event for handling player movement. Source: Own elaboration

Upon completing this setup, the debug option for the "On Player Move" event was activated. By running Play Mode and moving the left joystick on a connected gamepad, new entries appeared in the Unity Console, indicating invocations of the event. This confirms that the Input Action Handler is functioning correctly.



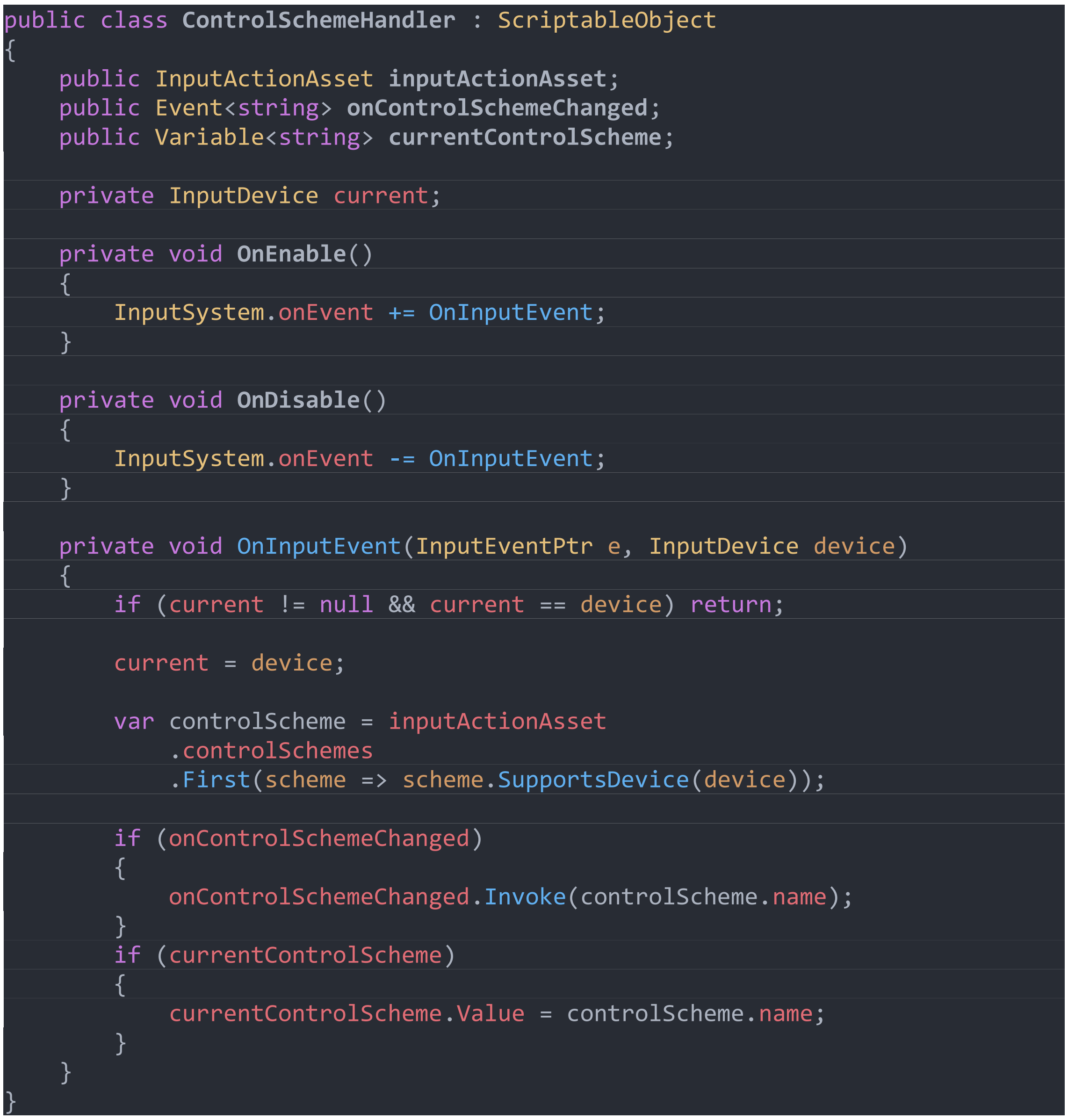
```
UnityEngine.Debug:Log (object)
[12:44:42] [SODD.Events.Vector2Event] On Player Move Invoked. Payload = (-0.92, 0.39)
UnityEngine.Debug:Log (object)
[12:44:42] [SODD.Events.Vector2Event] On Player Move Invoked. Payload = (-0.98, 0.21)
UnityEngine.Debug:Log (object)
[12:44:42] [SODD.Events.Vector2Event] On Player Move Invoked. Payload = (-1.00, 0.06)
UnityEngine.Debug:Log (object)
[12:44:42] [SODD.Events.Vector2Event] On Player Move Invoked. Payload = (-1.00, 0.00)
UnityEngine.Debug:Log (object)
[12:44:42] [SODD.Events.Vector2Event] On Player Move Invoked. Payload = (0.00, 0.00)
UnityEngine.Debug:Log (object)

[SODD.Events.Vector2Event] On Player Move Invoked. Payload = (-1.00, 0.06)
UnityEngine.Debug:Log (object)
SODD.Core.Logger:LogAsset (UnityEngine.Object,string) (at
./Library/PackageCache/com.github.aruizrab.sodd-unity-framework@b8f41abc23/Runtime/Core/Logger.cs:39)
SODD.Events.Event`1<UnityEngine.Vector2>:Invoke (UnityEngine.Vector2) (at
./Library/PackageCache/com.github.aruizrab.sodd-unity-framework@b8f41abc23/Runtime/Events/Event.cs:78)
SODD.Input.ActionHandlers.InputActionHandler`1<UnityEngine.Vector2>:OnActionPerformed
(UnityEngine.InputSystem.InputAction/CallbackContext) (at
./Library/PackageCache/com.github.aruizrab.sodd-unity-framework@b8f41abc23/Runtime/Input/ActionHandlers/InputActio
nHandler.cs:106)
```

Figure 39: Unity Console output showing the "On Player Move" event invocations with vector payloads, confirming the Input Action Handler functionality. Source: Own elaboration

With this configuration, a player script can listen for the "On Player Move" event, either directly or through the previously demonstrated Event Listeners, to execute the game logic required for moving the player. This approach ensures that the player movement logic is cleanly separated from the input reading logic, and can be easily adjusted or expanded upon, adhering to principles of modular and decoupled design.

5.2.8 Control Scheme Handler

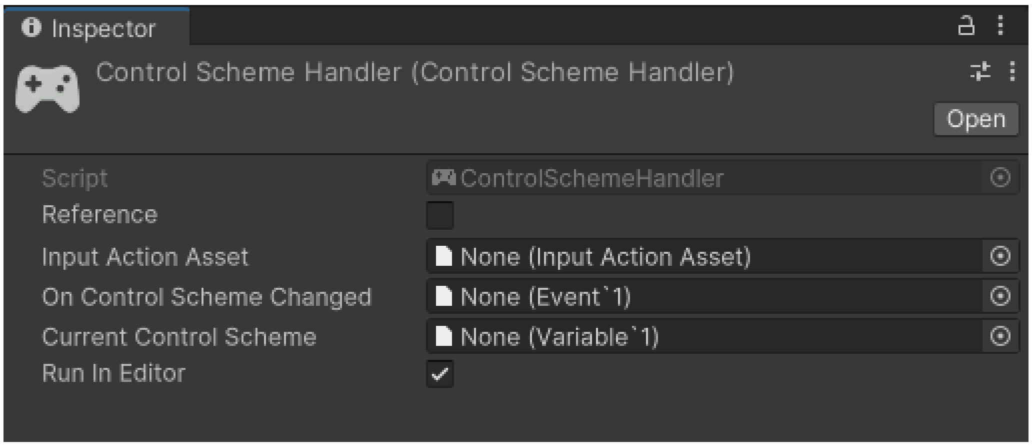
The Control Scheme Handler has been implemented as specified in the requirements to manage changes in Input Control Schemes. This handler listens for generic input events within Unity's Input System and determines the active Control Scheme in its referenced Input Action Asset based on the current device triggering the input event. Upon detecting a change in the Control Scheme, the handler can invoke a Scriptable Event, if a reference is provided, or set the current control scheme to a Scriptable Variable, if a reference is available.



```
public class ControlSchemeHandler : ScriptableObject
{
    public InputActionAsset inputActionAsset;
    public Event<string> onControlSchemeChanged;
    public Variable<string> currentControlScheme;

    private InputDevice current;

    private void OnEnable()
    {
        InputSystem.onEvent += OnInputEvent;
    }

    private void OnDisable()
    {
        InputSystem.onEvent -= OnInputEvent;
    }

    private void OnInputEvent(InputEventPtr e, InputDevice device)
    {
        if (current != null && current == device) return;

        current = device;

        var controlScheme = inputActionAsset
            .controlSchemes
            .First(scheme => scheme.SupportsDevice(device));

        if (onControlSchemeChanged)
        {
            onControlSchemeChanged.Invoke(controlScheme.name);
        }
        if (currentControlScheme)
        {
            currentControlScheme.Value = controlScheme.name;
        }
    }
}
```

Figure 40: Simplified code sample of the Control Scheme Handler class implementation. (Own elaboration)

Since Unity's Input System triggers the generic input events tracked by the Control Scheme Handler even during editor time, tests can be conducted without entering play mode. However, this may lead to undesired behaviors during editor time and create performance overheads. To mitigate this issue, a toggle option has been created, allowing developers to decide whether Control Scheme Handlers should run in editor mode or exclusively at runtime.



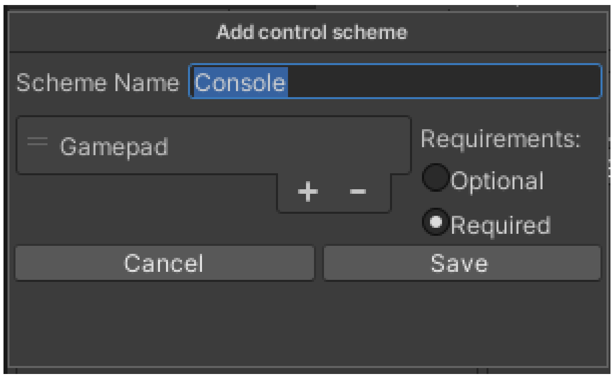
Control Scheme Handler (Control Scheme Handler)

Script	ControlSchemeHandler
Reference	☐
Input Action Asset	None (Input Action Asset)
On Control Scheme Changed	None (Event`1)
Current Control Scheme	None (Variable`1)
Run In Editor	✓

Figure 41: Inspector view of the Control Scheme Handler setup for managing changes in Input Control Schemes. Source: Own elaboration

Result Evaluation

To evaluate the functionality of the Control Scheme Handler, a practical use case involving dynamic control scheme switching has been tested. This scenario involves the creation of an input action asset with two control schemes: one for keyboard and mouse and another for gamepad. A StringEvent was created to carry the scheme change event and a StringVariable to be modified whenever the control scheme changes. An instance of the Control Scheme Handler was then created and the values assigned.


Add control scheme

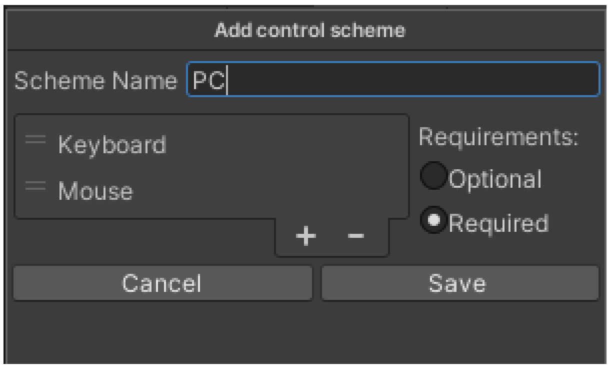
Scheme Name: Console

Gamepad

Requirements: Optional [✓ Required]

Cancel Save

Figure 42: Adding the "Console" control scheme with the required Gamepad device. Source: Own elaboration.


Add control scheme

Scheme Name: PC

Keyboard

Mouse

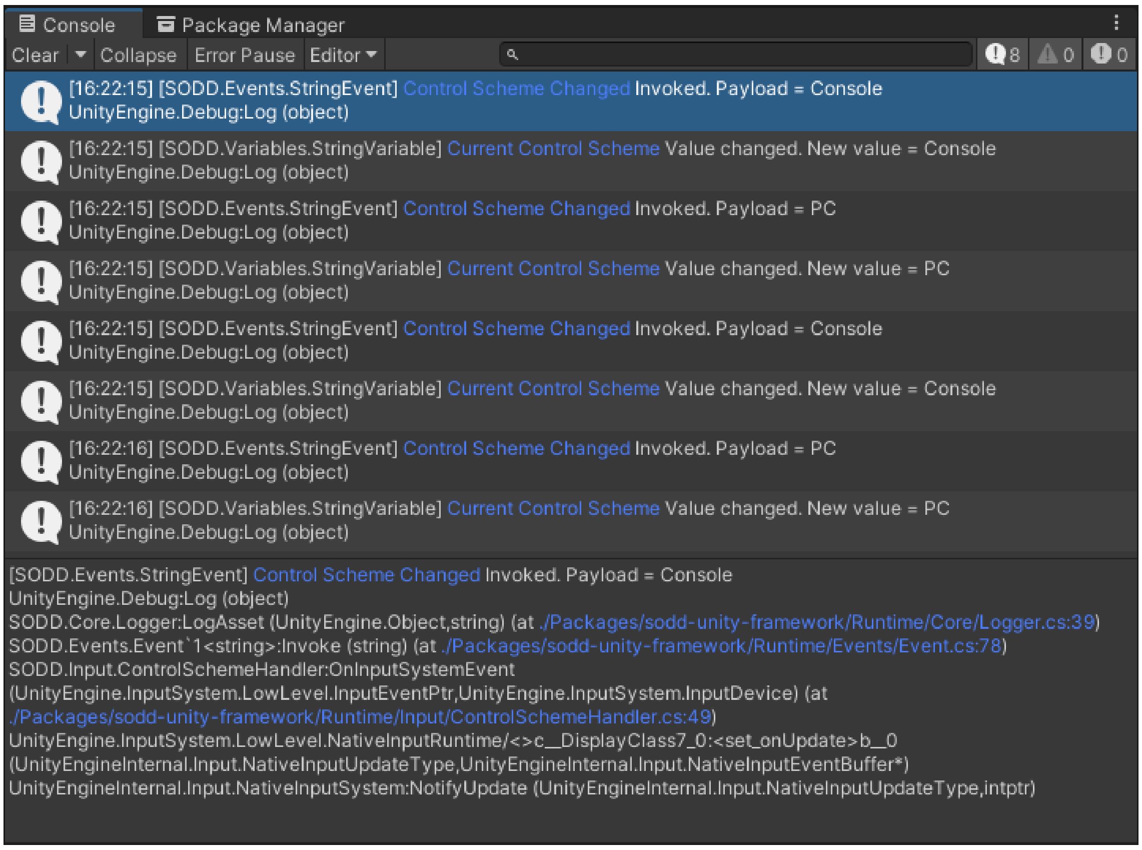
Requirements: Optional [✓ Required]

Cancel Save

Figure 43: Adding the "PC" control scheme with the required Keyboard and Mouse devices. Source: Own elaboration.

Given that previous evaluations of Scriptable Events and Variables have demonstrated their proper functioning, the testing of Control Scheme Handlers can be reliably conducted in Editor Mode. By enabling the debug option of both the created StringEvent and StringVariable, it can be ensured that new log entries appear in the Console when the control scheme changes.

By performing actions such as moving the mouse or pressing keyboard keys, logs of the event being triggered and the variable being set to the name of the control scheme supporting keyboard and mouse successfully appear in the Console. Similarly, when performing any input action with a connected gamepad, such as moving a joystick or pressing a button, the appropriate logs are observed, confirming the correct functionality of the Control Scheme Handler.



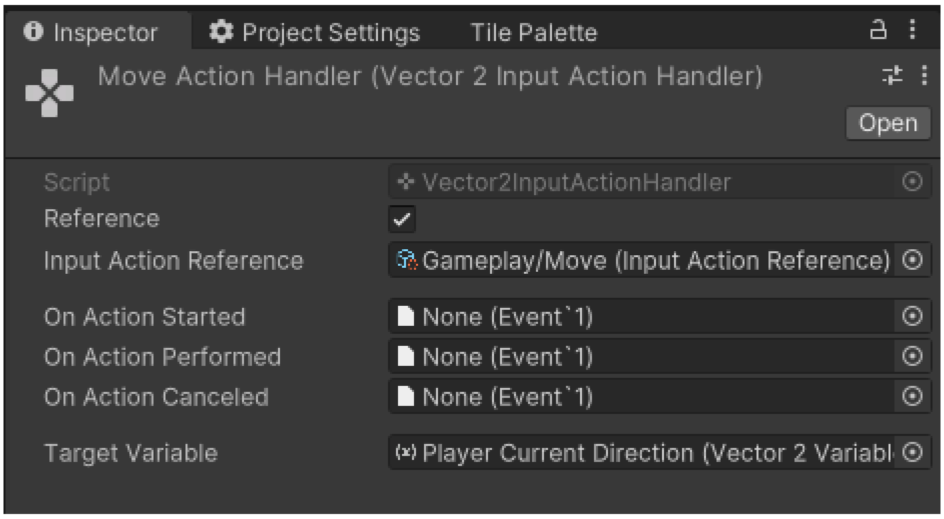
```
[16:22:15] [SODD.Events.StringEvent] Control Scheme Changed Invoked. Payload = Console
UnityEngine.Debug:Log (object)
[16:22:15] [SODD.Variables.StringVariable] Current Control Scheme Value changed. New value = Console
UnityEngine.Debug:Log (object)
[16:22:15] [SODD.Events.StringEvent] Control Scheme Changed Invoked. Payload = PC
UnityEngine.Debug:Log (object)
[16:22:15] [SODD.Variables.StringVariable] Current Control Scheme Value changed. New value = PC
UnityEngine.Debug:Log (object)
[16:22:15] [SODD.Events.StringEvent] Control Scheme Changed Invoked. Payload = Console
UnityEngine.Debug:Log (object)
[16:22:15] [SODD.Variables.StringVariable] Current Control Scheme Value changed. New value = Console
UnityEngine.Debug:Log (object)
[16:22:16] [SODD.Events.StringEvent] Control Scheme Changed Invoked. Payload = PC
UnityEngine.Debug:Log (object)
[16:22:16] [SODD.Variables.StringVariable] Current Control Scheme Value changed. New value = PC
UnityEngine.Debug:Log (object)

[SODD.Events.StringEvent] Control Scheme Changed Invoked. Payload = Console
UnityEngine.Debug:Log (object)
SODD.Core.Logger:LogAsset (UnityEngine.Object,string) (at ./Packages/sodd-unity-framework/Runtime/Core/Logger.cs:39)
SODD.Events.Event`1<string>:Invoke (string) (at ./Packages/sodd-unity-framework/Runtime/Events/Event.cs:78)
SODD.Input.ControlSchemeHandler:OnInputSystemEvent
(UnityEngine.InputSystem.LowLevel.InputEventPtr,UnityEngine.InputSystem.InputDevice) (at
./Packages/sodd-unity-framework/Runtime/Input/ControlSchemeHandler.cs:49)
UnityEngine.InputSystem.LowLevel.NativeInputRuntime/<>c__DisplayClass7_0:<set_onUpdate>b__0
(UnityEngineInternal.Input.NativeInputUpdateType,UnityEngineInternal.Input.NativeInputEventBuffer*)
UnityEngineInternal.Input.NativeInputSystem:NotifyUpdate (UnityEngineInternal.Input.NativeInputUpdateType,intptr)
```

Figure 44: Debug information logged in the Console regarding the invocation of the created Scriptable Event and the value changes of the created Scriptable Variable. Source: Own elaboration

5.2.9 Passive ScriptableObjects

Unity does not load ScriptableObjects into memory during runtime unless they are referenced in a scene. Features of the SODD Framework, such as Input Action Handlers or Control Scheme Handlers, are not directly referenced in scenes—they exist passively in the file system and interact with the framework's ScriptableObject-based systems. Therefore, these implementations require a mechanism to ensure they are loaded into memory during runtime.

To address this, a new concept called Passive ScriptableObjects has been introduced. This is represented by an abstract class that inherits from the ScriptableObject type. Passive ScriptableObjects are ScriptableObjects that don't populate a scene but provide functionality passively as assets loaded into memory. Each Passive ScriptableObject includes a toggle option in the Inspector, which is enabled by default, indicating whether the specific instance should be loaded into memory.



Move Action Handler (Vector 2 Input Action Handler)

Script	Vector2InputActionHandler
Reference	✓
Input Action Reference	Gameplay/Move (Input Action Reference)
On Action Started	None (Event`1)
On Action Performed	None (Event`1)
On Action Canceled	None (Event`1)
Target Variable	Player Current Direction (Vector 2 Variabl

Figure 45: A Move Action Handler—now inheriting from the new Passive ScriptableObject type—with the reference toggle activated, indicating that it needs to be passively loaded into memory. Source: Own elaboration

The final element to ensure these Passive ScriptableObjects are loaded consists in a new menu option has been added to the SODD Framework's Menu. This menu option triggers a functionality that scans the project files for Passive ScriptableObjects with the enabled toggle option. It then creates a GameObject with a Component that contains a list of all the collected Passive ScriptableObjects.

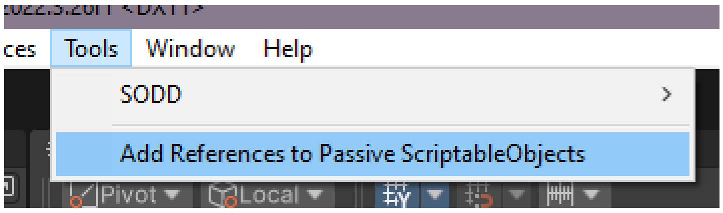
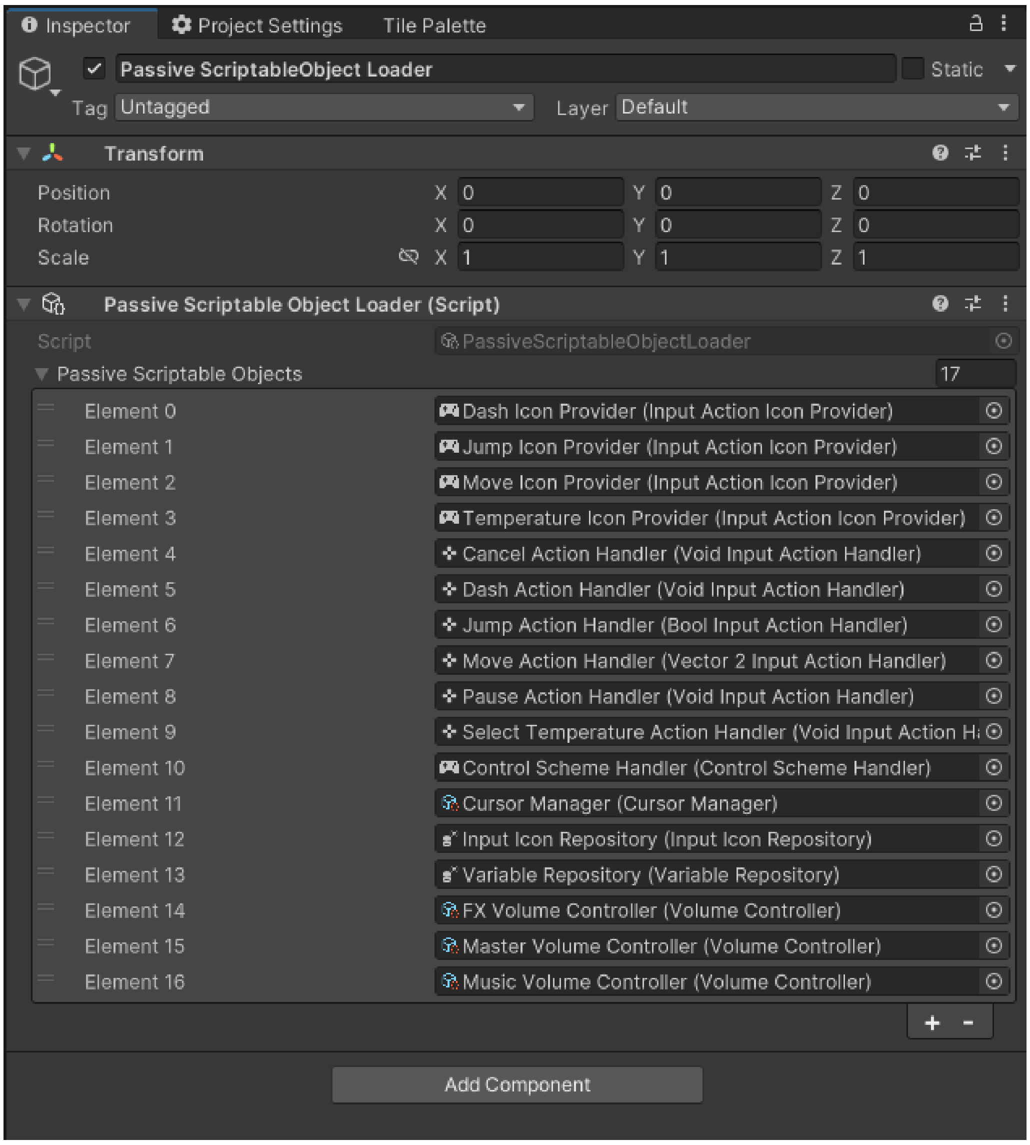


Figure 46: The created menu command, responsible for referencing Passive ScriptableObjects in the scenes. Source: Own elaboration

The generated GameObject is added to all the scenes that will be included in the final build. The sole purpose of this created GameObject is to reference the Passive ScriptableObjects, ensuring they are loaded into memory during runtime.



Passive ScriptableObject Loader

Tag Untagged Layer Default

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

Passive Scriptable Object Loader (Script)

Script PassiveScriptableObjectLoader

Passive Scriptable Objects 17

Element 0	Dash Icon Provider (Input Action Icon Provider)
Element 1	Jump Icon Provider (Input Action Icon Provider)
Element 2	Move Icon Provider (Input Action Icon Provider)
Element 3	Temperature Icon Provider (Input Action Icon Provider)
Element 4	Cancel Action Handler (Void Input Action Handler)
Element 5	Dash Action Handler (Void Input Action Handler)
Element 6	Jump Action Handler (Bool Input Action Handler)
Element 7	Move Action Handler (Vector 2 Input Action Handler)
Element 8	Pause Action Handler (Void Input Action Handler)
Element 9	Select Temperature Action Handler (Void Input Action H
Element 10	Control Scheme Handler (Control Scheme Handler)
Element 11	Cursor Manager (Cursor Manager)
Element 12	Input Icon Repository (Input Icon Repository)
Element 13	Variable Repository (Variable Repository)
Element 14	FX Volume Controller (Volume Controller)
Element 15	Master Volume Controller (Volume Controller)
Element 16	Music Volume Controller (Volume Controller)

Figure 47: The generated GameObject, containing the references of all Passive ScriptableObjects in a project. Source: Own elaboration

This menu command is an effective solution to the problem given the project's time constraints, but more efficient solutions may be explored for the future of the framework.

5.3 Developing a Sample Videogame

This section details the development of a small video game using the ScriptableObject Driven Development (SODD) tools provided by the framework. The aim is to demonstrate the usability, reliability, and viability of these tools in a practical game development scenario.

It is important to note that various aspects of the development process that do not directly pertain to the utilization of SODD Framework's features have been omitted from this section. These aspects include, but are not limited to:

- Level design.
- Setting up scenes, layer masks, collision detection and other environment configurations.
- Setting up animations and animation state machines.
- User interface design and implementation.
- Asset management and importing

5.3.1 Game Proposal

Overview

- **Title:** Ice Heat
- **Description**: Ice Heat is a 2D level-based platformer inspired by the videogame "Celeste".
- **Concept**: In Ice Heat, players take control of a character with the unique power of producing temperature expansion waves to boil or freeze water around them, overcoming platforming challenges to progress through the levels of the game.

Game Mechanics

- **Jump**: The player can perform a quick, short jump with a brief press of the jump button. Holding the jump button allows for a higher and longer jump. This mechanic provides basic platforming functionality, common in 2D platformers.
- **Directional Dash**: The player can dash in a specified direction by pointing to the direction and pressing the dash button. Horizontal and vertical movement are completely arrested before and after the dash, ensuring a predictable behaviour even when moving. This mechanic allows the player to quickly move across the screen, aiding in overcoming larger gaps and avoiding obstacles.
- **Temperature Wave**: By holding a button, time slows down, and a small UI in the form of a radial menu appears, letting the player choose to perform either a heating or cooling temperature wave. This mechanic enables the player to manipulate the state of water, which is essential for navigating through the levels.

Game Elements

- **Water Platforming**: Water platforming is a central gameplay mechanic in Ice Heat. Water exists in three states, each affecting the player's ability to navigate levels:
    - **Ice**: As a solid state, ice serves as walkable platforms or obstacles. It can be transformed into normal water with a heating wave, allowing players to create or remove solid ground as needed.
    - **Normal Water**: Players can swim through normal water, which can slow their momentum, particularly when falling. This state can be turned into ice with a cooling wave or boiling water with a heating wave.
    - **Boiling Water**: Boiling water produces high-pressure vapour, propelling the player upwards. It can be cooled into normal water, adding a vertical dimension to platforming and enabling quick escapes or access to higher platforms.
- **Collectible Coins**: Each level contains a collectible coin placed in difficult-to-reach locations, which require mastering game mechanics in order to collect

them. Once collected, the coins are displayed above their respective levels in the selection menu, offering visual progress tracking.

Game Structure

Ice Heat" is structured into three levels, each becoming available once the previous one has been completed. This progression system ensures a steady increase in difficulty and complexity.

The first level serves as a tutorial for the dash mechanic, guiding players on how to use the directional dash to overcome obstacles and navigate the environment effectively.

The second level acts as a tutorial for the temperature wave mechanic. Players learn to slow down time and choose between heating and cooling waves to manipulate the state of water.

The third and final level puts the combination of the dash and temperature wave mechanics to the test. Players must use their acquired skills in tandem to navigate through tougher platforming challenges. This level integrates all the learned mechanics, providing a comprehensive test of the player's abilities and mastery of the game.

5.3.2 Setup

The setup of Ice Heat involves creating a new Unity 2D project, using the same Unity version selected for the framework: 2022.3 LTS. Subsequently, the SODD Framework is imported via UPM using the URL of the repository that hosts the framework on GitHub. During the import process, Unity automatically includes the package dependencies, which consist of Unity's Input System and the Text Mesh Pro package.

5.3.3 Input Management

The management of input in Ice Heat involves the creation of an Input Action asset, encompassing Action Maps, Control Schemes, Input Actions, and Input Bindings.

Action Maps

Two primary action maps have been defined to delineate the context of player interactions:

1. **Gameplay:** This map covers actions that are pertinent during gameplay, such as Move, Jump, Dash, and Select Temperature. These actions represent fundamental mechanics that the player will utilize while engaging with the game.
2. **UI:** This map contains actions relevant to menu navigation, providing a set of interactions when the player is outside of the core gameplay environment.

Control Schemes

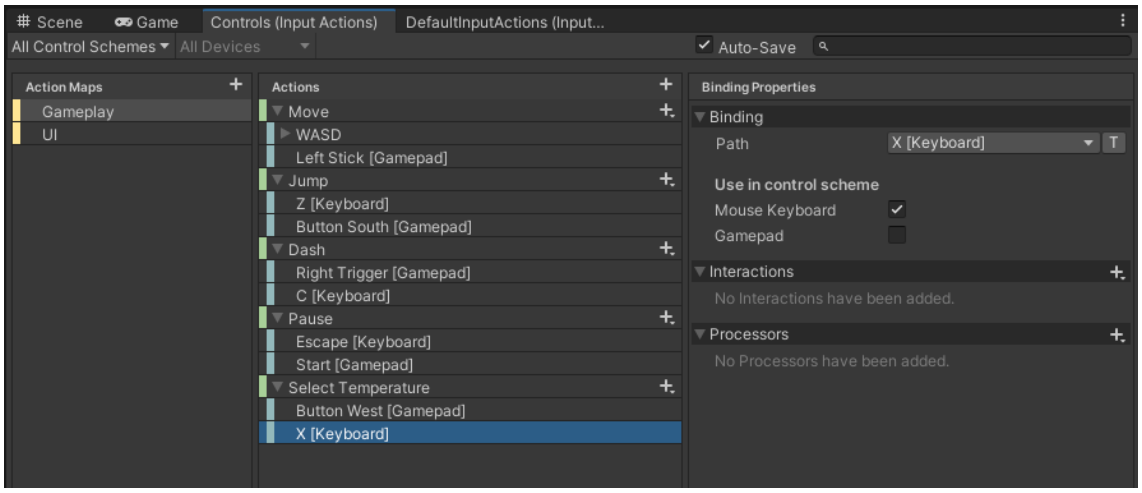
To accommodate different input devices, two control schemes are established. One configured to support players using a gamepad and another designed for players utilizing a keyboard and mouse.

Input Actions and Bindings

Within the Gameplay action map, each input action is tied to a specific game mechanic.

- **Move:** Handles player movement.
- **Jump:** Manages the jumping mechanic.
- **Dash:** Controls the dashing action.
- **Select Temperature:** Triggers the process of selecting whether to produce a heating expansion wave, or a cooling one.

Each action is configured with two input bindings corresponding to the two control schemes. This configuration step showcases the versatility of Unity's Input System, which translates diverse device inputs into unified game actions that define how the player interacts with the game.



Scene | Game | Controls (Input Actions) | DefaultInputActions (Input...

All Control Schemes | All Devices | ✓ Auto-Save

Action Maps
- Gameplay
- UI

Actions
- Move
  - WASD
  - Left Stick [Gamepad]
- Jump
  - Z [Keyboard]
  - Button South [Gamepad]
- Dash
  - Right Trigger [Gamepad]
  - C [Keyboard]
- Pause
  - Escape [Keyboard]
  - Start [Gamepad]
- Select Temperature
  - Button West [Gamepad]
  - X [Keyboard]

Binding Properties

Binding
Path: X [Keyboard]

Use in control scheme
Mouse Keyboard ✓
Gamepad

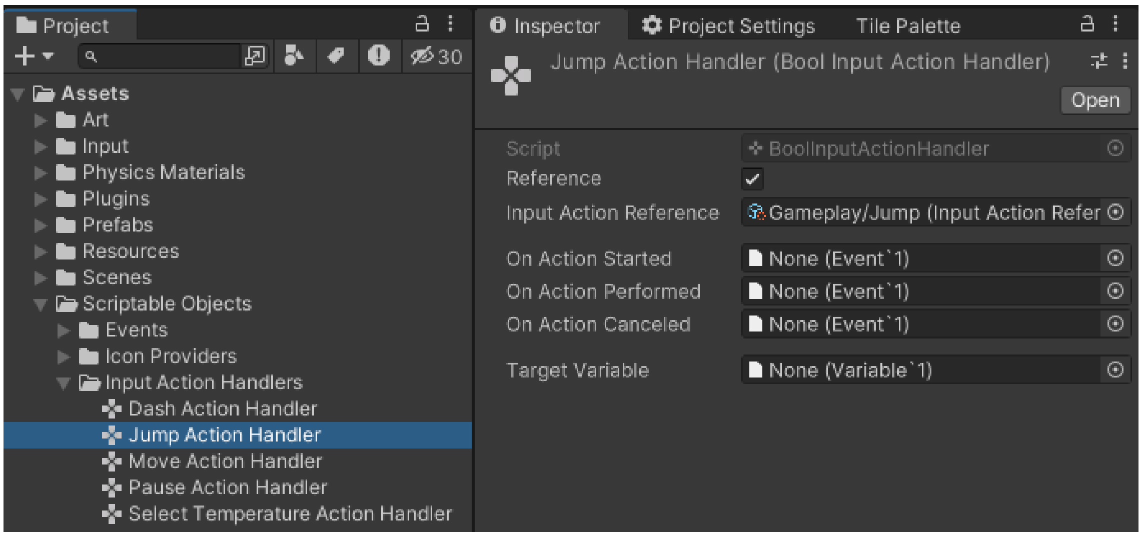
Interactions
No Interactions have been added.

Processors
No Processors have been added.

Figure 48: Input Action asset setup with Action Maps for Gameplay and UI, including bindings for both keyboard and gamepad control schemes. Source: Own elaboration

Input Action Handlers

For each defined action, an Input Action Handler is created. These handlers are responsible for processing the input and triggering corresponding Scriptable Events and Scriptable Variables. The events and variables referenced by these handlers will be added in future stages, once the game scripts define how they need to access the input data.



Project

- Assets
  - Art
  - Input
  - Physics Materials
  - Plugins
  - Prefabs
  - Resources
  - Scenes
  - Scriptable Objects
    - Events
    - Icon Providers
    - Input Action Handlers
      - Dash Action Handler
      - Jump Action Handler
      - Move Action Handler
      - Pause Action Handler
      - Select Temperature Action Handler

Inspector | Project Settings | Tile Palette

Jump Action Handler (Bool Input Action Handler) — Open

Script	BoolInputActionHandler
Reference	✓
Input Action Reference	Gameplay/Jump (Input Action Refer
On Action Started	None (Event`1)
On Action Performed	None (Event`1)
On Action Canceled	None (Event`1)
Target Variable	None (Variable`1)

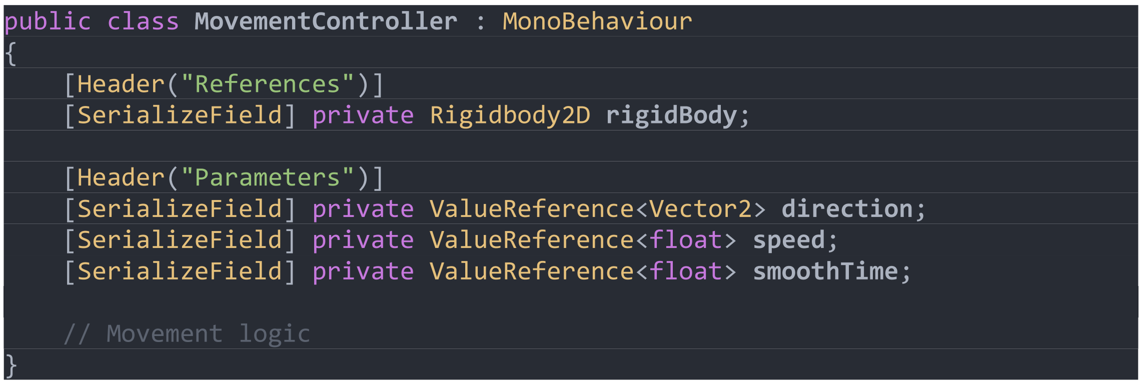
Figure 49: Project view of the created Input Action Handlers. Source: Own elaboration

5.3.4 Player Control

To create a comprehensive player control system, a set of scripts have been developed, each focusing on a specific aspect of player control. This modular approach ensures that each script can be independently managed and maintained, promoting a clean and organized codebase.

Movement

The MovementController script is designed to read from a Scriptable Variable that holds the desired direction for player movement. It moves the player's Rigidbody horizontally based on this direction, governed by parameters for movement speed and the time required to reach full speed. This latter parameter provides a smoother transition between the player being stationary and moving, enhancing the fluidity of the movement.

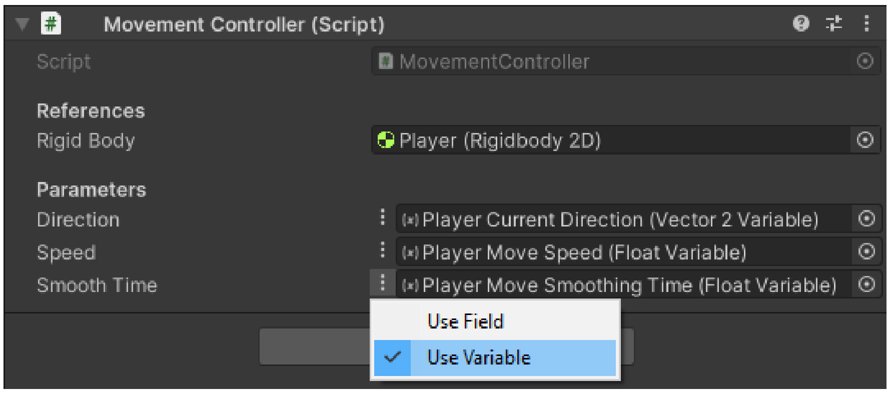


```
public class MovementController : MonoBehaviour
{
    [Header("References")]
    [SerializeField] private Rigidbody2D rigidBody;

    [Header("Parameters")]
    [SerializeField] private ValueReference<Vector2> direction;
    [SerializeField] private ValueReference<float> speed;
    [SerializeField] private ValueReference<float> smoothTime;

    // Movement logic
}
```

Figure 50: Code fragment showing the parameters of the MovementController script.

The script’s behaviour is entirely dependent on these parameters, which are exposed in the Unity Inspector. This exposure facilitates a data-driven design, allowing developers to adjust movement characteristics without altering the code.

Additionally, all parameter values are referenced through a Value Reference. This allows for flexibility in switching between directly inputted values in the script's editor or references to Scriptable Variables. This design pattern ensures adherence to good practices discussed in this document and is consistently applied throughout the development of all game scripts.



Movement Controller (Script)

Script: MovementController

References

Rigid Body: Player (Rigidbody 2D)

Parameters

Direction: Player Current Direction (Vector 2 Variable)

Speed: Player Move Speed (Float Variable)

Smooth Time: Player Move Smoothing Time (Float Variable)

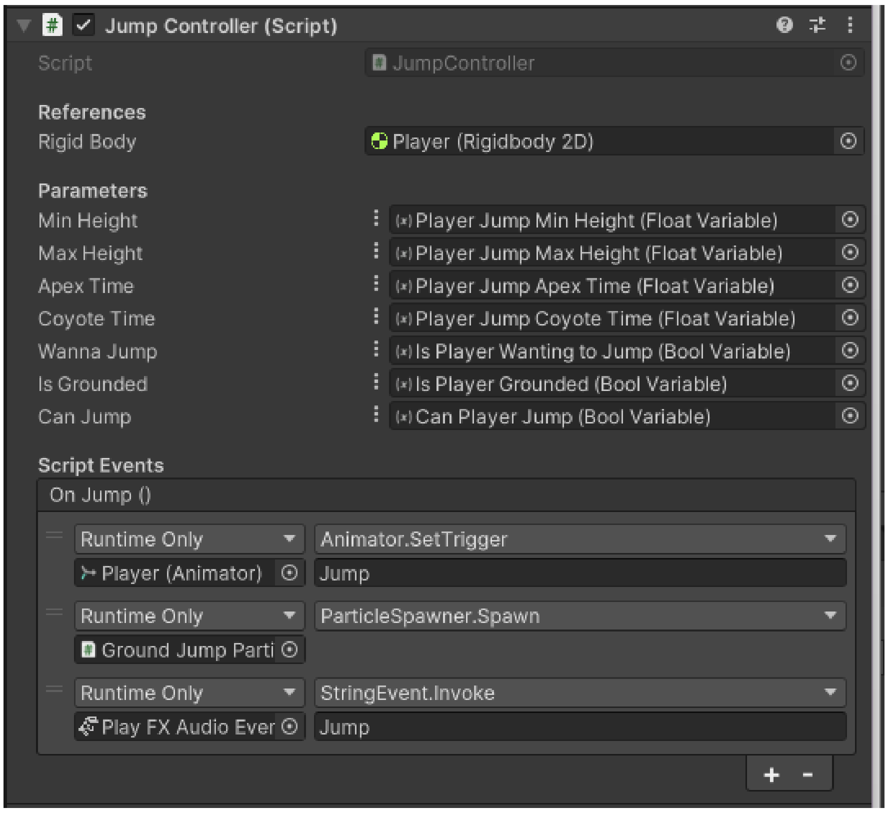
Use Field
[✓ Use Variable]

Figure 51: MovementController script setup in the Unity Inspector, using Scriptable Variables for direction, speed, and smoothing time parameters. Source: Own elaboration

The **MovementController** script introduces the first input requirement: a Scriptable Variable that holds the real-time direction in which the player is pointing. To meet this requirement, a Vector2 Variable can be created. This variable is referenced by both the MovementController script and the Input Action Handler responsible for the Move action. The Input Action Handler dynamically updates the variable based on the player's input, ensuring that the player's direction is accurately reflected in real-time.

Jump

The **JumpController** script is designed to manage the vertical movement of the player. It utilizes a set of parameters encapsulated as Value References, including the minimum and maximum height of the jump and the time required to reach the apex of the jump trajectory. These parameters can be adjusted in real-time using Scriptable Variables, allowing developers to fine-tune the jump mechanics until the desired movement and feel are achieved. As Scriptable Variables are ScriptableObjects, they retain their values even after exiting Play Mode, ensuring persistence and consistency.


Jump Controller (Script)

Script: JumpController

References

Rigid Body: Player (Rigidbody 2D)

Parameters

Min Height	Player Jump Min Height (Float Variable)
Max Height	Player Jump Max Height (Float Variable)
Apex Time	Player Jump Apex Time (Float Variable)
Coyote Time	Player Jump Coyote Time (Float Variable)
Wanna Jump	Is Player Wanting to Jump (Bool Variable)
Is Grounded	Is Player Grounded (Bool Variable)
Can Jump	Can Player Jump (Bool Variable)

Script Events

On Jump ()

Runtime Only	Animator.SetTrigger
Player (Animator)	Jump
Runtime Only	ParticleSpawner.Spawn
Ground Jump Parti	
Runtime Only	StringEvent.Invoke
Play FX Audio Ever	Jump

Figure 52: JumpController script setup in the Unity Inspector, managing jump mechanics with Scriptable Variables for parameters and UnityEvents for additional behaviors. Source: Own elaboration

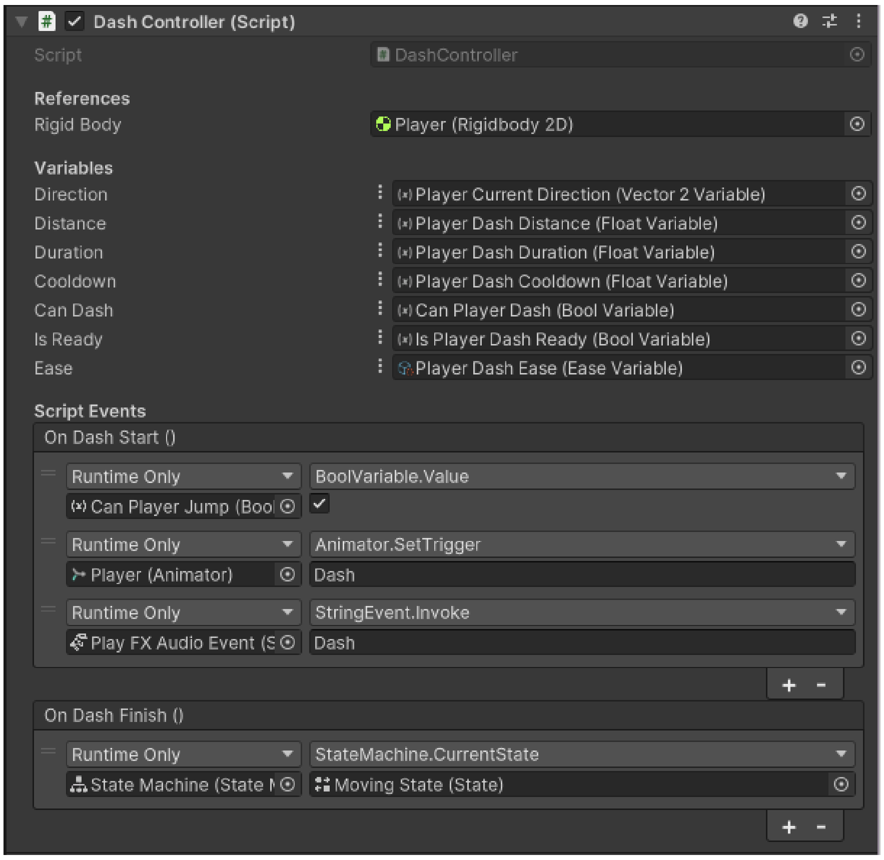
The **JumpController** script references two particular variables, **isGrounded** and **canJump**, which are updated by external scripts. The **isGrounded** Boolean value is set to true when an external ground detection systems confirms that the player is colliding with the ground. By delegating collision detection to another script, the **JumpController** remains focused solely on managing jump logic, enhancing modularity. The **canJump** Boolean value is used by game mechanics, such as the Dash, to reset the player's ability to jump while airborne.

The script necessitates a Boolean value to indicate whether the player intends to jump, updated in real-time based on input. This Boolean value influences the height of the jump, determined by the duration the jump button is pressed. A Bool Variable is created for this purpose and referenced by both the JumpController and the Input Action Handler responsible for the Jump action.

Lastly, it is important to remark that a UnityEvent is invoked whenever the player performs a jump. This event can be linked to additional behaviours, such as spawning particles, playing sounds, or triggering animations. This extensibility allows new features to be added without modifying the **JumpController** script, maintaining its single responsibility for managing jump logic. By leveraging UnityEvents, the system remains adaptable and scalable as game requirements evolve.

Dash

The DashController script adheres to the same principles of exposing all parameters in the Unity Inspector and encapsulating them with ValueReferences. Since the dash mechanic is directional and depends on the player's orientation, this script references the same Vector2 Variable used by the previously discussed MovementController, which holds the current input direction set by the Input Action Handler.



Dash Controller (Script)

Script	DashController
References	
Rigid Body	Player (Rigidbody 2D)
Variables	
Direction	Player Current Direction (Vector 2 Variable)
Distance	Player Dash Distance (Float Variable)
Duration	Player Dash Duration (Float Variable)
Cooldown	Player Dash Cooldown (Float Variable)
Can Dash	Can Player Dash (Bool Variable)
Is Ready	Is Player Dash Ready (Bool Variable)
Ease	Player Dash Ease (Ease Variable)

Script Events

On Dash Start ()

Runtime Only	BoolVariable.Value
Can Player Jump (Boo	✓
Runtime Only	Animator.SetTrigger
Player (Animator)	Dash
Runtime Only	StringEvent.Invoke
Play FX Audio Event (S	Dash

On Dash Finish ()

Runtime Only	StateMachine.CurrentState
State Machine (State M	Moving State (State)

Figure 53: DashController script setup in the Unity Inspector, managing dash mechanics with Scriptable Variables for parameters and UnityEvents for additional behaviors. Source: Own elaboration

An additional input requirement for the DashController is to start the dash whenever the Dash button is pressed. This is accomplished by creating a Void Event and referencing it in the Input Action Handler responsible for the Dash action. This setup ensures that the event is invoked whenever the Dash button is pressed. A Void Event Listener is then added alongside the DashController, referencing this Void Event. The provided UnityEvent in the listener is used to hook the DashController's dash execution method.

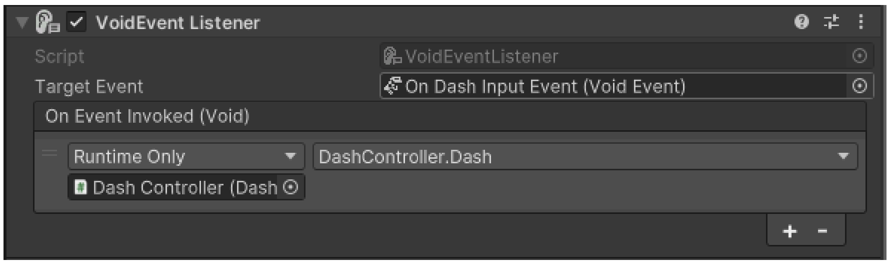

VoidEvent Listener
Script: VoidEventListener
Target Event: On Dash Input Event (Void Event)
On Event Invoked (Void)
Runtime Only | DashController.Dash
Dash Controller (Dash

Figure 54: VoidEvent Listener setup in the Unity Inspector, linking the "On Dash Input Event" to the DashController's dash execution method. Source: Own elaboration

Similar to the JumpController script, the DashController offers UnityEvents for intrinsic events such as when the dash begins and ends. These UnityEvents facilitate the specification of additional behaviors without altering the code, thus promoting scalability. For example, figure 61 illustrates how the UnityEvent sets the canJump Bool Variable to true, enabling the player to jump again while airborne after performing a dash.

Temperature Waves

This mechanic is more intricate than the previously described mechanics and requires multiple single-responsibility scripts, orchestrated through Scriptable Events, Scriptable Variables, and UnityEvents.

Firstly, the script **PlayerSelectionController** is activated whenever the temperature selection button is held down. This behavior is facilitated with two Void Events, referenced in the Input Action Handler. One event is triggered when the button begins to be pressed, and the other is triggered upon release. While the button is pressed and the selection script is active, it reads the current player direction by referencing the Vector2 Variable used for player direction. This direction selection is visually represented in the game UI, which displays two options for the player: right for heating and left for cooling.

When the selection button is released, the **PlayerSelectionController** script reads the current direction. Based on whether the direction is right or left, it triggers a corresponding UnityEvent. These UnityEvents are then connected to various scripts that perform visual and functional effects for the expansion wave. These effects include:

- **Changing Player Color:** Adjusting the player's material variables to reflect the heating or cooling selection.
- **Animating the Expansion Wave:** Creating a visual animation that shows the expansion wave emanating from the player.
- **Water State Detection:** Detecting water within the area of the expansion wave and changing its state accordingly (e.g., heating or cooling the water).

The integration of these scripts through Scriptable Events and UnityEvents ensures a modular and decoupled design. Each script handles a specific aspect of the mechanic, from player input to visual feedback and environmental interactions. This separation of concerns not only simplifies the development process but also enhances the maintainability and scalability of the codebase. Developers can easily tweak or extend individual components without affecting the overall system.

Results

During the development of player controls, a distinction has been made regarding the Scriptable Variables used in the player's functionality. These variables fall into two categories:

- **Parameters**: Read by the scripts to define the player's behaviour, such as movement speed, jump height, and dash duration. These are generally static and are used to configure the player's capabilities.
- **Runtime Variables:** Dynamically updated by the scripts to reflect the current state of the player, such as the player's current direction, whether they are

grounded, and if they can perform another jump or dash. These variables change during gameplay and are crucial for the real-time functionality of the player.

To organize these variables efficiently, they have been grouped into two distinct folders within the file system. Both of these folders are nested within a parent folder that clearly indicates these Scriptable Variables belong to the player.

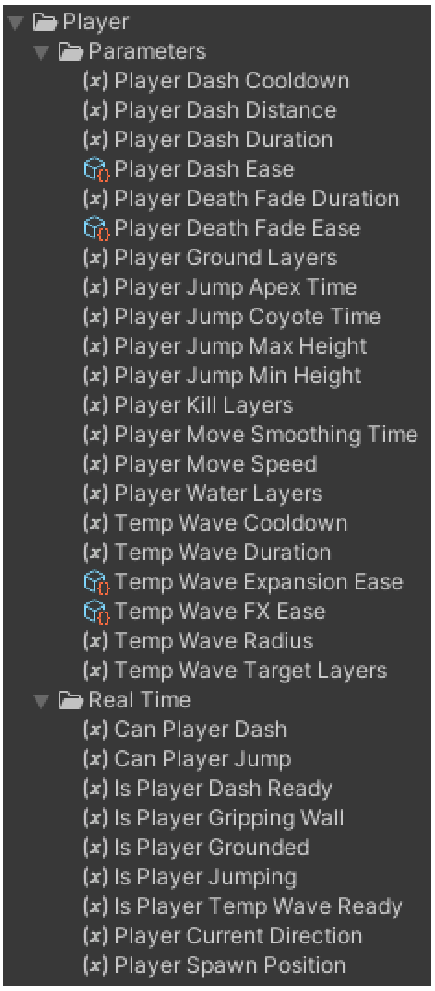

```
Player
  Parameters
    (x) Player Dash Cooldown
    (x) Player Dash Distance
    (x) Player Dash Duration
    Player Dash Ease
    (x) Player Death Fade Duration
    Player Death Fade Ease
    (x) Player Ground Layers
    (x) Player Jump Apex Time
    (x) Player Jump Coyote Time
    (x) Player Jump Max Height
    (x) Player Jump Min Height
    (x) Player Kill Layers
    (x) Player Move Smoothing Time
    (x) Player Move Speed
    (x) Player Water Layers
    (x) Temp Wave Cooldown
    (x) Temp Wave Duration
    Temp Wave Expansion Ease
    Temp Wave FX Ease
    (x) Temp Wave Radius
    (x) Temp Wave Target Layers
  Real Time
    (x) Can Player Dash
    (x) Can Player Jump
    (x) Is Player Dash Ready
    (x) Is Player Gripping Wall
    (x) Is Player Grounded
    (x) Is Player Jumping
    (x) Is Player Temp Wave Ready
    (x) Player Current Direction
    (x) Player Spawn Position
```

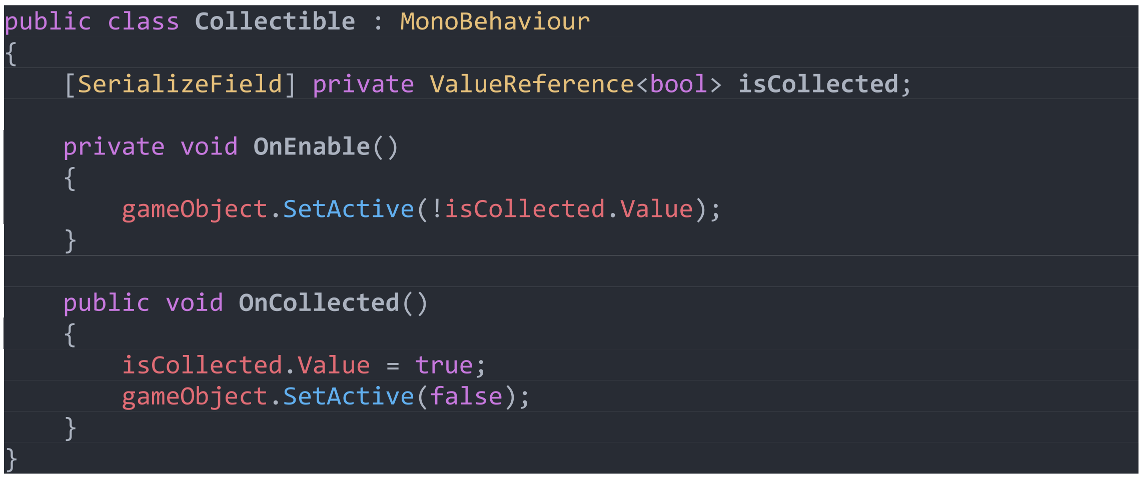
Figure 55: Organized folder structure for Player Scriptable Variables, separating parameters and runtime variables. Source: Own elaboration

This organization provides a single point of access to all the parameters and the runtime state of the player. Consequently, developers and designers can easily access, consult, and modify these variables either in Editor Mode or Play Mode. This setup eliminates the need to navigate through the scripts and GameObjects that make up the player's Prefab, streamlining the development, testing and debugging process.

5.3.5 Collectible Coins

As outlined in the game proposal, each level contains a collectible coin that players can obtain by navigating platforming challenges. The mechanism governing these coins is straightforward: it detects collisions with the player and notifies a **Collectible** script.

Each coin has an assigned Bool Variable, referenced through a Value Reference, indicating whether the coin has already been collected. This Bool Variable ensures that collected coins are not visible in the level, while those that still need to be collected remain visible. Upon collection, the Bool Variable's state of the specific coin is set to true, marking it as collected.



```
public class Collectible : MonoBehaviour
{
    [SerializeField] private ValueReference<bool> isCollected;

    private void OnEnable()
    {
        gameObject.SetActive(!isCollected.Value);
    }

    public void OnCollected()
    {
        isCollected.Value = true;
        gameObject.SetActive(false);
    }
}
```

Figure 56: Collectible script using a Bool Variable to manage the visibility and state of collectible coins. Source: Own elaboration

The use of these Bool Variables play a role in the UI system for the level selection menu, displaying which coins have been collected. This provides players with a clear visual representation of their progress. Moreover, the state of these variables is intended to be persisted across game sessions, ensuring that players' progress is saved and maintained. The implementation and management of this persistence will be further detailed in the section dedicated to Variable Repositories.

5.3.6 UI Management

Level Selection

To ensure that levels cannot be accessed until the previous level has been completed—with the exception of the first level—the level selection menu needs to dynamically enable or disable access to levels based on the player's progress

Each level selection button incorporates a **LevelSelectionController** script. This script references a Bool Variable that indicates whether the level is available. External systems are responsible for updating these Bool Variables to reflect the player's progress, ensuring a clear separation of concerns. The **LevelSelectionController** script itself does not handle the logic for determining level availability; it merely references the variable to enable or disable the level button based on its state.

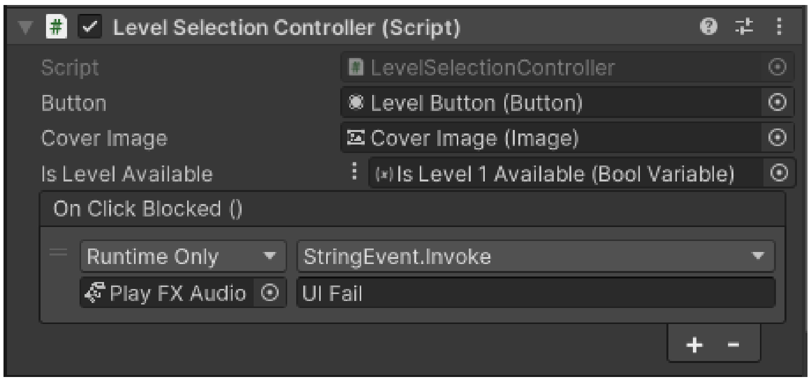
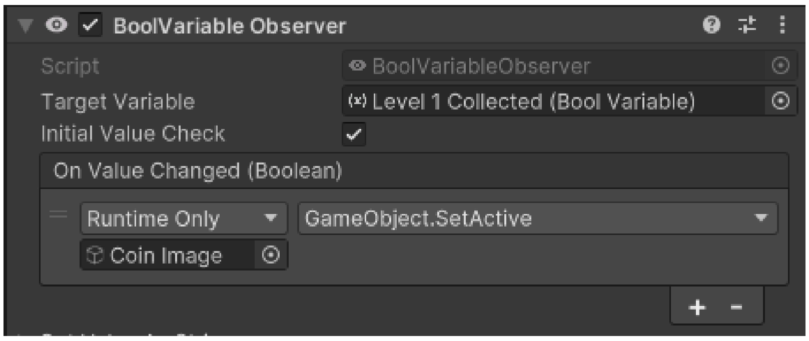

Level Selection Controller (Script)
Script: LevelSelectionController
Button: Level Button (Button)
Cover Image: Cover Image (Image)
Is Level Available: Is Level 1 Available (Bool Variable)
On Click Blocked ()
Runtime Only | StringEvent.Invoke
Play FX Audio | UI Fail

Figure 57: LevelSelectionController script setup, using a Bool Variable to manage the availability of levels and invoking a StringEvent if the level is blocked. Source: Own elaboration

Displaying Collected Coins

The level buttons exhibit different appearances and interaction behaviours depending on whether they are enabled or disabled. The **LevelSelectionController** script manages such behaviours, ensuring that each button reflects its current state accurately.

On the other hand, displaying a collected coin on top of the level button when a coin has been collected in that level is a simpler behaviour, which can be managed by leveraging Variable Observers. Each level button includes a Variable Observer that references the Bool Variable indicating whether a coin has been collected in that specific level. The state of this variable is transmitted to an attached UnityEvent, which is linked to the UI component responsible for displaying the coin. When the BoolVariable indicates that the coin has been collected, the UI component is dynamically enabled to display the coin; otherwise, it remains disabled.



BoolVariable Observer
Script: BoolVariableObserver
Target Variable: Level 1 Collected (Bool Variable)
Initial Value Check: ✓
On Value Changed (Boolean)
Runtime Only | GameObject.SetActive
Coin Image

Figure 58: BoolVariable Observer setup to display a collected coin on the level button by dynamically enabling the UI component based on the Bool Variable state. Source: Own elaboration

In-Level Tutorial Prompts

Another crucial functionality is the implementation of HUD prompts that teach the player how to use the controls during the first two levels, which serve as a tutorial for the game's mechanics. This feature ensures that players are guided effectively through the initial stages of the game, enhancing their learning experience.

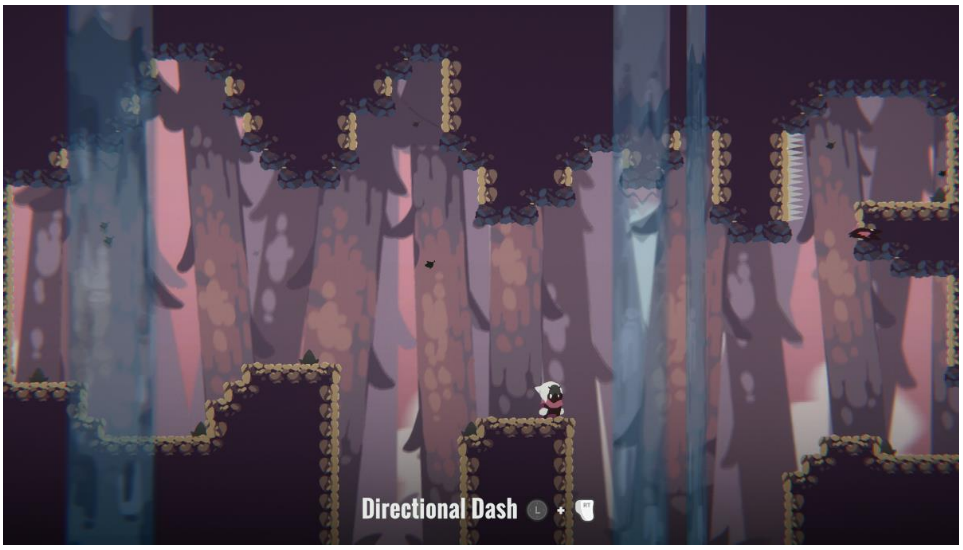
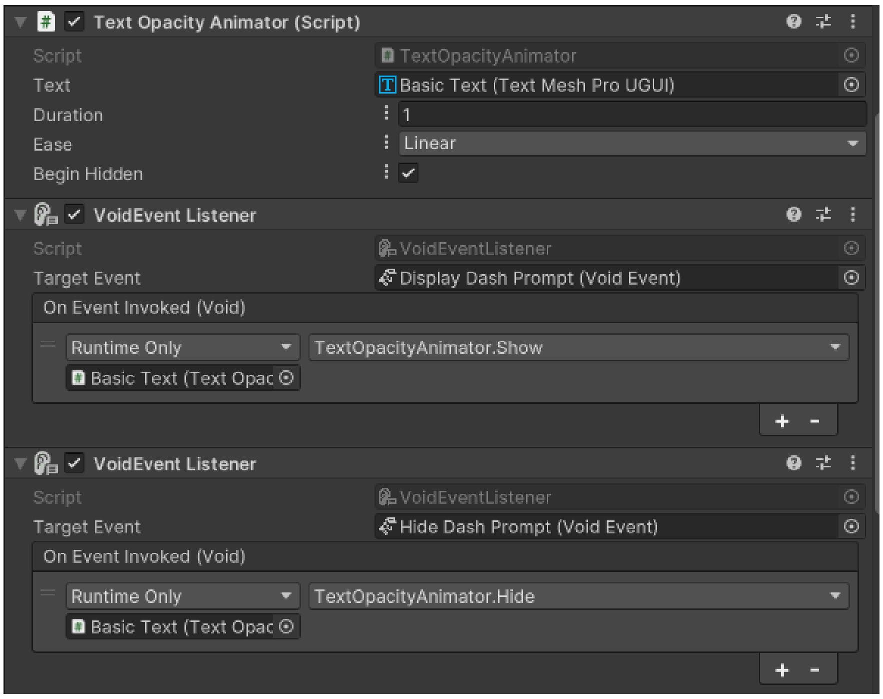


Figure 59: In-level tutorial prompt guiding the player on how to use the directional dash mechanic. Source: Own elaboration

Initially, a Heads-Up Display (HUD) element is created to display tutorial prompts, including a text component with an accompanying script that animates the alpha channel. This animation creates a fade-in and fade-out effect, drawing the player's attention to the prompts in a non-intrusive manner.

To manage the display of these tutorial prompts, two Void Events are defined: one to show the prompts and another to hide them. These events enable the prompts to be displayed and hidden at the appropriate times without embedding the logic directly into the gameplay scripts. A Void Event Listener is attached to the HUD, referencing these Void Events. The listener calls the necessary functions within the HUD script to control the visibility of the tutorial prompts, ensuring that the logic for displaying the prompts remains modular and easily adjustable.


Text Opacity Animator (Script)
Script: TextOpacityAnimator
Text: Basic Text (Text Mesh Pro UGUI)
Duration: 1
Ease: Linear
Begin Hidden: ✓

VoidEvent Listener
Script: VoidEventListener
Target Event: Display Dash Prompt (Void Event)
On Event Invoked (Void)
Runtime Only | TextOpacityAnimator.Show
Basic Text (Text Opac

VoidEvent Listener
Script: VoidEventListener
Target Event: Hide Dash Prompt (Void Event)
On Event Invoked (Void)
Runtime Only | TextOpacityAnimator.Hide
Basic Text (Text Opac

Figure 60: TextOpacityAnimator script setup with VoidEvent Listeners to manage the display and hide events for in-level tutorial prompts. Source: Own elaboration

To trigger the display of the tutorial prompts, a collision detection script is added to specific areas within the levels. When the player enters a designated area, the collision detector triggers the Void Event to show the tutorial prompt. Conversely, when the player exits the area, the event to hide the prompt is triggered. This use of Void Events and Event Listeners decouples the tutorial prompt logic from the main gameplay scripts, providing greater flexibility and maintainability.

Dynamic Input Icons

In modern video games, displaying dynamic input icons in the UI is crucial for informing players how to interact with the game. These icons must change dynamically based on the input device currently in use, such as displaying keyboard icons when using a keyboard and mouse, or gamepad icons when using a gamepad. This feature is particularly important for tutorial levels, which instruct players on how to perform game mechanics like dashing or temperature waves, and for the settings menu, which provides a comprehensive overview of all game controls.

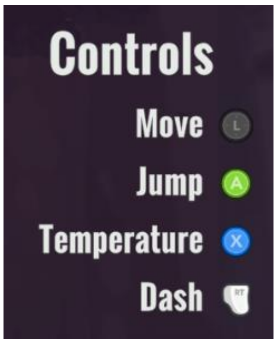
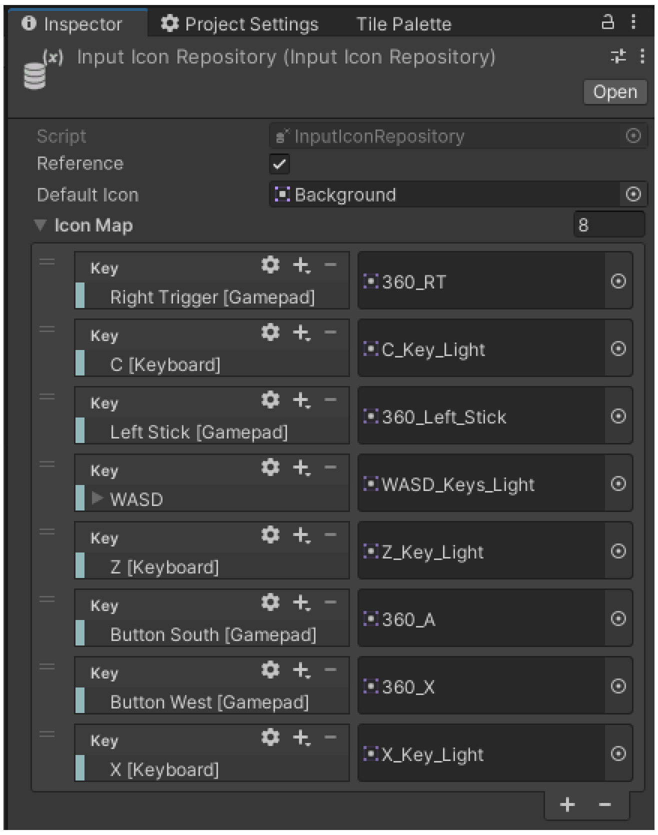


Figure 61: Settings menu displaying input icons based on the current input device, providing players with updated control information. Source: Own elaboration

In Ice Heat, this feature can be achieved by leveraging SODD Framework's Control Scheme Handlers. The implementation begins with the creation of a Control Scheme Handler instance. This handler is assigned the game's input action asset and is responsible for managing the current control scheme. A String Variable is added to the handler to hold the name of the current control scheme. This variable dynamically updates to reflect changes in the control scheme, ensuring that the correct icons are displayed based on the input device in use.

Next, it is necessary to map the input bindings of actions based on the current control scheme. For example, the binding for jumping is the 'Z' key on the keyboard and mouse control scheme and the 'A' button on the gamepad control scheme. To facilitate this mapping, an Input Icon Repository is created. This ScriptableObject holds a mapping of input bindings to their respective input icons, providing a centralized repository for managing these icons.



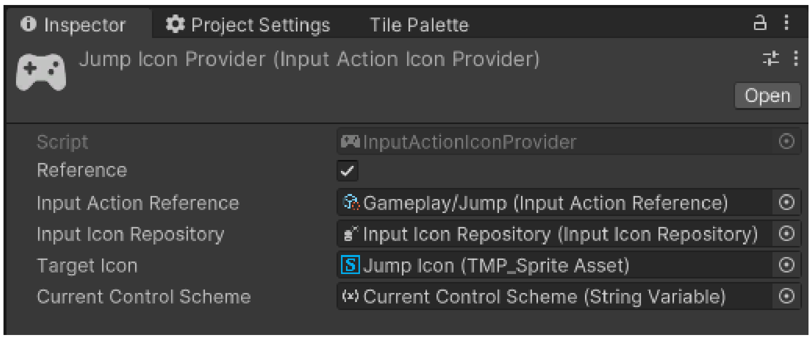
Input Icon Repository (Input Icon Repository)

Script	InputIconRepository
Reference	✓
Default Icon	Background
Icon Map	8

Key	
Right Trigger [Gamepad]	360_RT
C [Keyboard]	C_Key_Light
Left Stick [Gamepad]	360_Left_Stick
WASD	WASD_Keys_Light
Z [Keyboard]	Z_Key_Light
Button South [Gamepad]	360_A
Button West [Gamepad]	360_X
X [Keyboard]	X_Key_Light

Figure 62: Input Icon Repository setup in the Unity Inspector, mapping input bindings to their respective icons for different control schemes. Source: Own elaboration

An **InputIconProvider** script is then created to handle the dynamic updating of input icons. This script references an Input Action, the created Input Icon Repository, a Sprite Asset, and the String Variable holding the current control scheme. A Sprite Asset, provided by the Unity Text Mesh Pro package, contains a collection of sprites that can be used as inline icons within text components, allowing for dynamic icon changes.



Jump Icon Provider (Input Action Icon Provider)

Script	InputActionIconProvider
Reference	✓
Input Action Reference	Gameplay/Jump (Input Action Reference)
Input Icon Repository	Input Icon Repository (Input Icon Repository)
Target Icon	Jump Icon (TMP_Sprite Asset)
Current Control Scheme	Current Control Scheme (String Variable)

Figure 63: Jump Icon Provider setup in the Unity Inspector, linking the Jump action to the Input Icon Repository and dynamically updating the sprite based on the current control scheme. Source: Own elaboration

The **InputIconProvider** listens for changes in the control scheme. When a change is detected, it searches the referenced input action for the input binding associated with the current control scheme. Once the appropriate binding is found, the script retrieves the corresponding input icon from the icon repository and updates the Sprite Asset with the new icon. Text components that reference this Sprite Asset will automatically update to display the new icon, ensuring that the UI reflects the current input device.

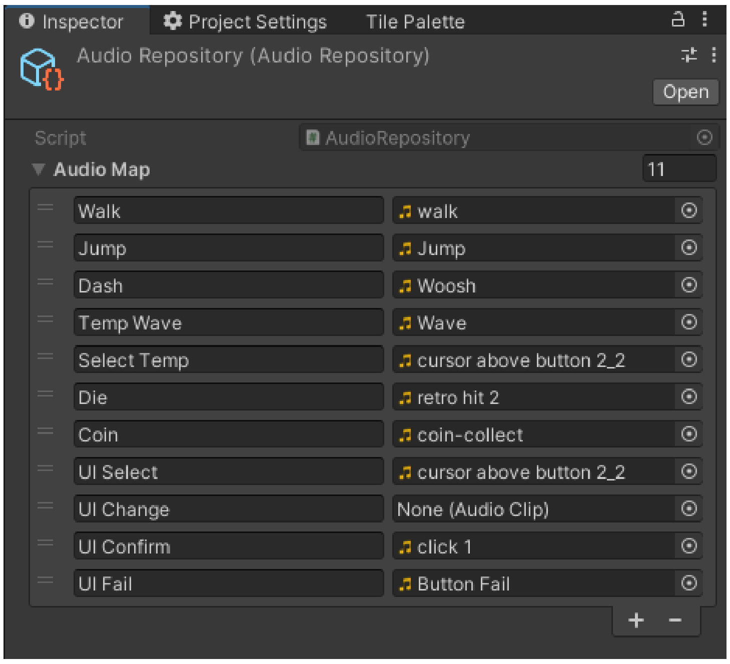
It is important to note that **InputIconProvider**s have been made to inherit from **PassiveScriptableObject**, which will allow them to be passively referenced in scenes alongside other Passive ScriptableObjects, such as Input Action Handlers, since they are not actively referenced in scenes and provide functionality exclusively from the project files.

Finally, an InputIconProvider is created for each action, with a respective Sprite Asset to display the icon for each action. This setup ensures that all text components dynamically reflect the correct input icons based on the player's current control scheme, enhancing the usability and accessibility of the game.

5.3.7 Audio System

A notable application of the SODD Framework’s Scriptable Events is in the creation of a decoupled and flexible audio system, allowing for centralized management and playback of audio events across the game.

The foundation of the audio system is the Audio Repository, a custom ScriptableObject that contains a map linking the names of audio events occurring in the game to the corresponding audio clips that should be played in response. This audio map can be easily accessed and modified through the Unity Inspector, facilitating the addition of new audio events and clips as the game evolves. This approach ensures that audio management remains centralized and easily maintainable.



Script	AudioRepository
Audio Map	11
Walk	walk
Jump	Jump
Dash	Woosh
Temp Wave	Wave
Select Temp	cursor above button 2_2
Die	retro hit 2
Coin	coin-collect
UI Select	cursor above button 2_2
UI Change	None (Audio Clip)
UI Confirm	click 1
UI Fail	Button Fail

Figure 64: Audio Repository setup in the Unity Inspector, mapping audio events to corresponding audio clips. Source: Own elaboration

Next, an AudioManager script is created. This script references an Audio Source component designated for playing audio clips and the Audio Repository. The AudioManager script is responsible for handling the playback of audio clips based on events received.

To handle audio events, a String Event is created. This event represents an audio event and carries the name of the audio event as its payload. By using a String Event Listener, this event is linked to the AudioManager. When an audio event occurs, the String Event Listener triggers the AudioManager to play the corresponding audio clip from the Audio Repository.

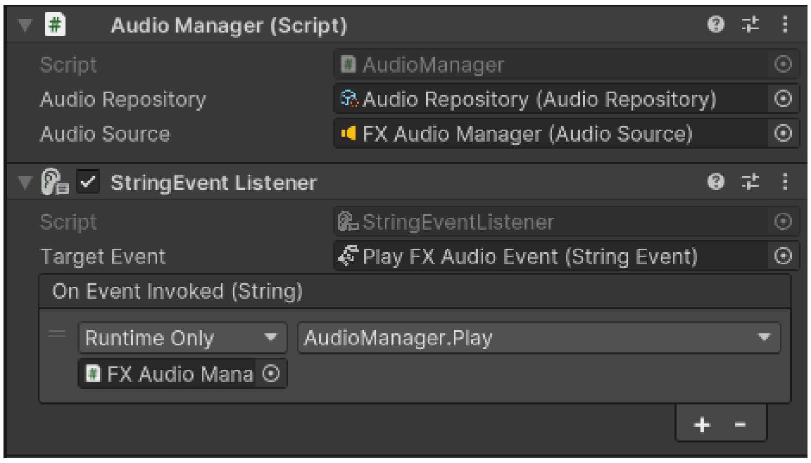

Audio Manager (Script)
Script AudioManager
Audio Repository Audio Repository (Audio Repository)
Audio Source FX Audio Manager (Audio Source)
StringEvent Listener
Script StringEventListener
Target Event Play FX Audio Event (String Event)
On Event Invoked (String)
Runtime Only AudioManager.Play
FX Audio Mana

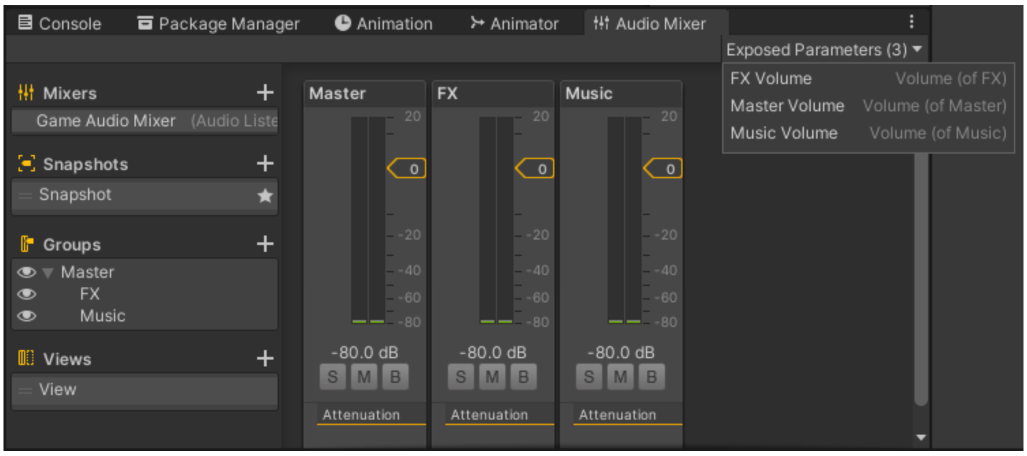
Figure 65: AudioManager script setup in the Unity Inspector, using a StringEvent Listener to trigger audio playback from the Audio Repository based on received events. Source: Own elaboration

This entire setup is encapsulated within an Audio Manager prefab. This prefab includes the **AudioManager** script, the Audio Source component, and references to the Audio Repository and String Event Listener. The prefab can be added to any scene, eliminating the need to manually add references or configure settings in each scene. All communication to trigger audio playback is handled through the String Event, which is stored in the file system.

5.3.8 Game Settings

Scriptable Variables can also be utilized for managing game settings, specifically audio settings within the context of Ice Heat. Due to time constraints, the implementation is limited to audio settings, excluding graphics settings and input rebinding.

The setup begins with the creation of a Float Variable for each audio setting: master volume, music volume, and FX volume. An audio mixer is configured with a separate mixing group for each audio setting, with each group having an exposed parameter for volume control. These mixing groups are then assigned to the audio sources managed by the previously created Audio Manager.

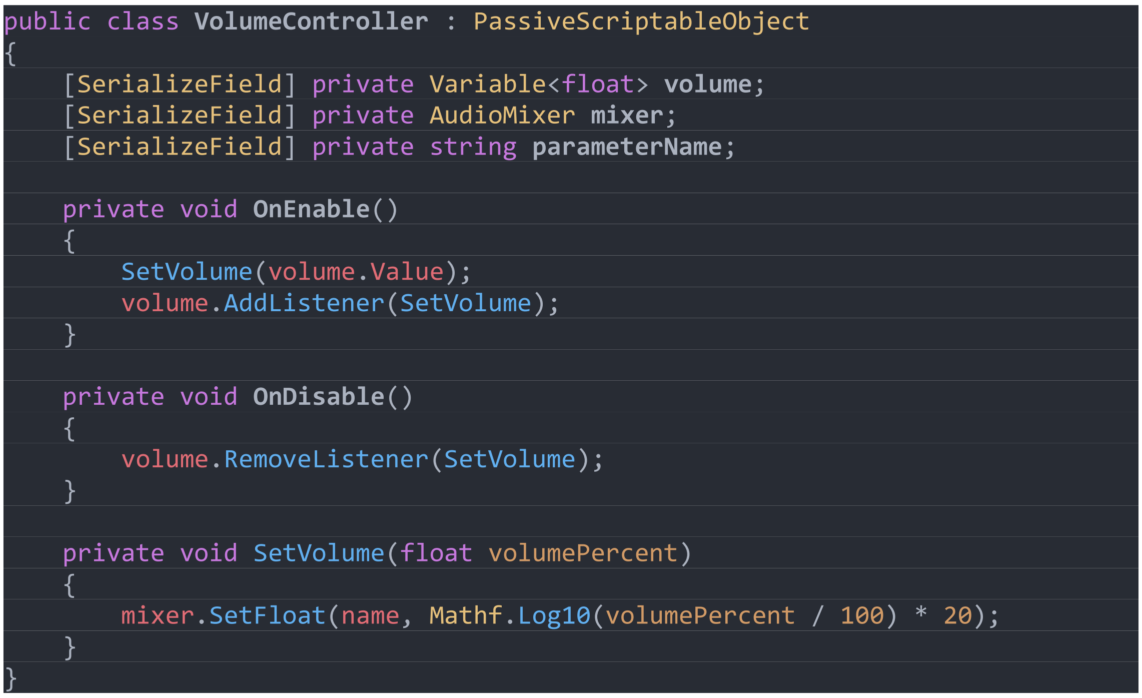


Exposed Parameters (3)

FX Volume	Volume (of FX)
Master Volume	Volume (of Master)
Music Volume	Volume (of Music)

Figure 66: Audio mixer setup in Unity, with separate mixing groups for master, FX, and music volumes, each with an exposed volume parameter. Source: Own elaboration

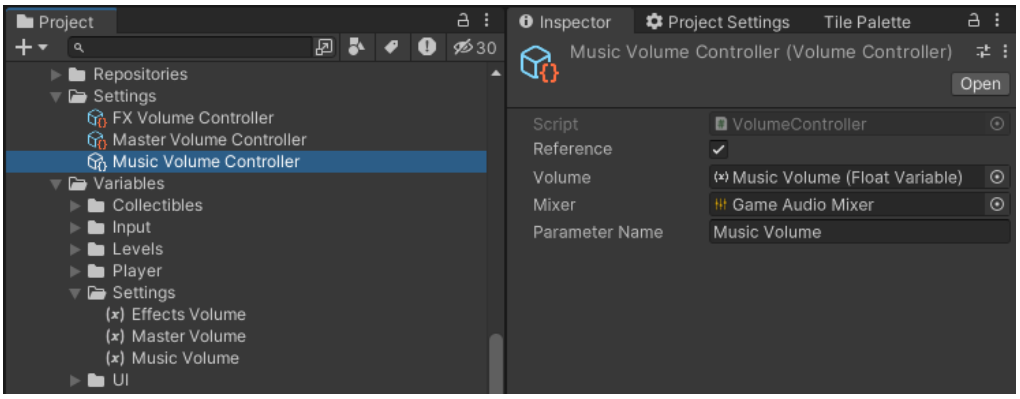
To manage these settings, a VolumeController script is developed. This script functions as another PassiveScriptableObject, providing functionality from the project files. It takes as parameters a Float Variable representing the volume, the audio mixer, and the name of the parameter to modify within the audio mixer. Initially, it sets the audio mixer's volume to match the Float Variable's value and listens for changes in the variable to reactively adjust the volume. To ensure correct behavior, the value of the Float Variable is clamped to a range of 0 to 100, representing the percentage of volume. This requirement suggests a potential future enhancement for the framework: the addition of value constraints to Scriptable Variables.



```
public class VolumeController : PassiveScriptableObject
{
    [SerializeField] private Variable<float> volume;
    [SerializeField] private AudioMixer mixer;
    [SerializeField] private string parameterName;

    private void OnEnable()
    {
        SetVolume(volume.Value);
        volume.AddListener(SetVolume);
    }

    private void OnDisable()
    {
        volume.RemoveListener(SetVolume);
    }

    private void SetVolume(float volumePercent)
    {
        mixer.SetFloat(name, Mathf.Log10(volumePercent / 100) * 20);
    }
}
```

Figure 67: VolumeController script setting audio mixer volumes based on Float Variables and updating them dynamically on value changes. Source: Own elaboration

A **VolumeController** instance is created for each mixing group, resulting in controllers for master volume, music, and audio effects. Each **VolumeController** is configured with its respective Float Variable, audio mixer, and parameter name corresponding to the mixer volume of its group.



```
▶ Repositories
▼ Settings
    FX Volume Controller
    Master Volume Controller
    Music Volume Controller
▼ Variables
  ▶ Collectibles
  ▶ Input
  ▶ Levels
  ▶ Player
  ▼ Settings
      (x) Effects Volume
      (x) Master Volume
      (x) Music Volume
  ▶ UI
```

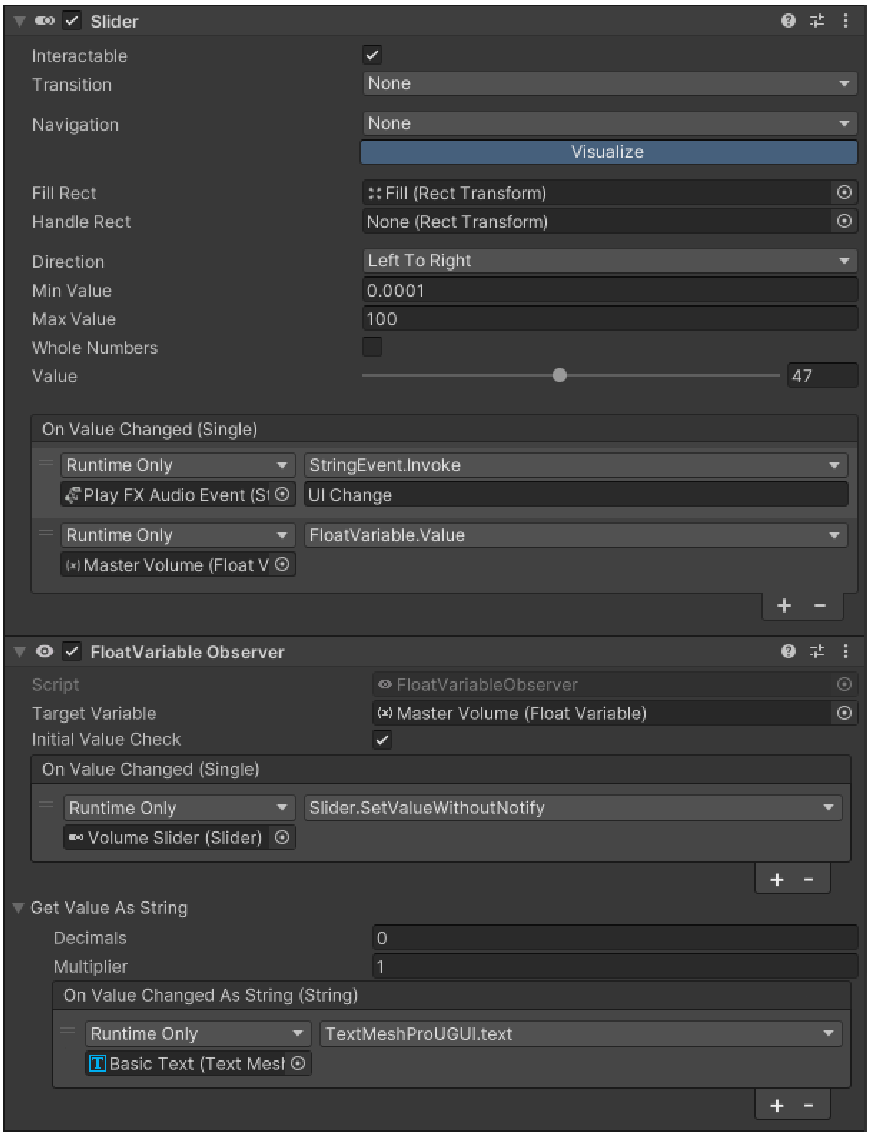
Music Volume Controller (Volume Controller)

Script	VolumeController
Reference	✓
Volume	Music Volume (Float Variable)
Mixer	Game Audio Mixer
Parameter Name	Music Volume

Figure 68: VolumeController instance setup in the Unity Inspector for managing music volume, referencing the Float Variable, Audio Mixer, and parameter name. Source: Own elaboration

The final step involves displaying these audio settings in the game's settings menu, allowing for visualization and modification. For each volume setting, a Slider and a Text component are created. The Slider allows users to adjust the volume, while the

Text component displays the exact volume percentage. Using the Slider's integrated UnityEvent, which triggers whenever the slider's value changes, and Variable Observers, no additional scripting is required. The Slider's UnityEvent, on one hand, sets the value of the corresponding Float Variable. On the other hand, the Variable Observer monitors changes to the Float Variable and uses its UnityEvent to update the slider and text components accordingly.



Slider ✓
Interactable ✓
Transition: None
Navigation: None
Visualize
Fill Rect: Fill (Rect Transform)
Handle Rect: None (Rect Transform)
Direction: Left To Right
Min Value: 0.0001
Max Value: 100
Whole Numbers: ☐
Value: 47
On Value Changed (Single)
Runtime Only | StringEvent.Invoke
Play FX Audio Event (St | UI Change
Runtime Only | FloatVariable.Value
Master Volume (Float V

FloatVariable Observer ✓
Script: FloatVariableObserver
Target Variable: Master Volume (Float Variable)
Initial Value Check ✓
On Value Changed (Single)
Runtime Only | Slider.SetValueWithoutNotify
Volume Slider (Slider)
Get Value As String
Decimals: 0
Multiplier: 1
On Value Changed As String (String)
Runtime Only | TextMeshProUGUI.text
Basic Text (Text Mesh

Figure 69: Settings menu setup with Slider and Text components for audio volume control, utilizing UnityEvents and Variable Observers to manage value changes and display updates. Source: Own elaboration

This setup provides a robust, modular, and scalable system for managing audio settings within the game. By leveraging Scriptable Variables, VolumeControllers, and UnityEvents, the system ensures that changes to audio settings are dynamically reflected in real-time, offering a flexible and easily editable solution for game settings management. This approach aligns with the best practices discussed in this document, supporting efficient and maintainable game development.

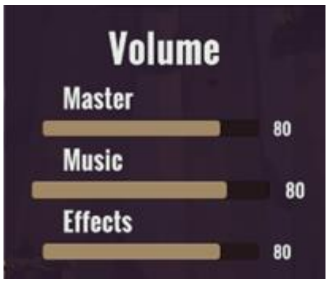


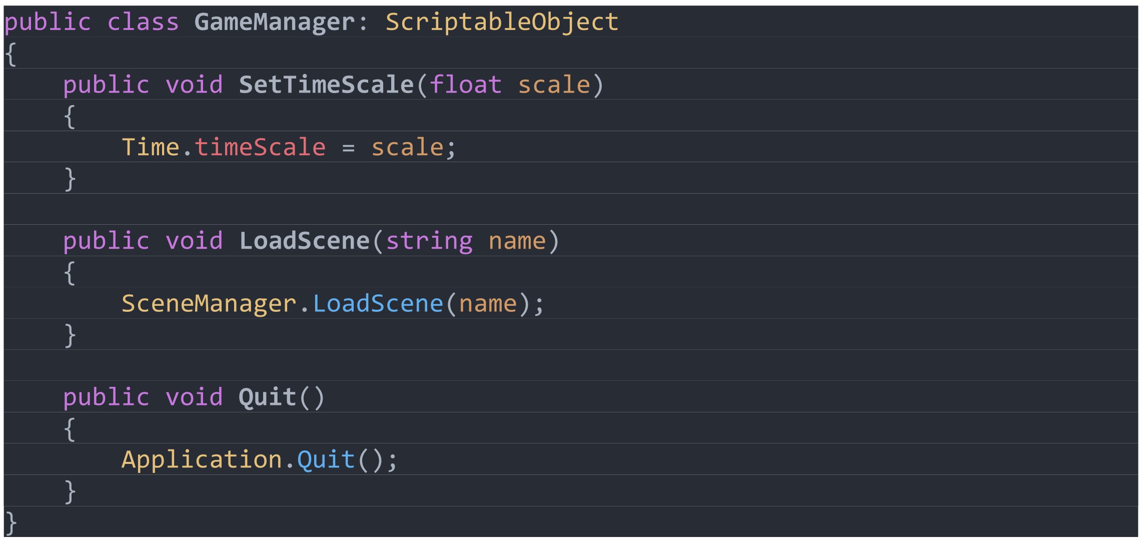
Figure 70: Resulting settings menu displaying audio volume sliders for Master, Music, and Effects. Source: Own elaboration

5.3.9 ScriptableObject Managers

One of the key teachings of Ryan Hipple in game architecture with ScriptableObjects is their use in creating game managers. This approach provides all the benefits of Singletons, commonly used for creating managers, but without the associated drawbacks. ScriptableObject managers facilitate modular and decoupled design, improving scalability and maintainability.

Game Manager

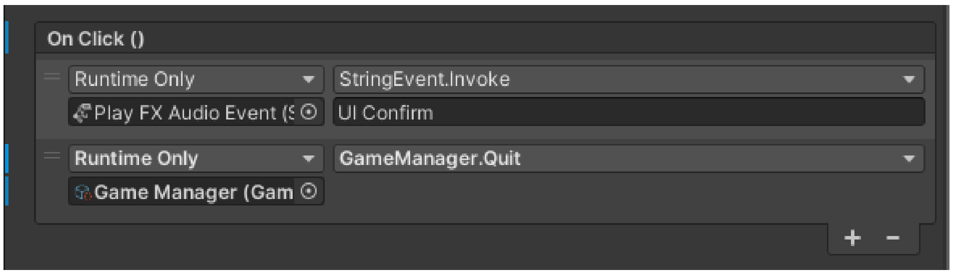
An example of this principle in Ice Heat is the Game Manager ScriptableObject, which provides public methods for performing simple game flow actions, such as changing the time scale or loading scenes.



```
public class GameManager: ScriptableObject
{
    public void SetTimeScale(float scale)
    {
        Time.timeScale = scale;
    }

    public void LoadScene(string name)
    {
        SceneManager.LoadScene(name);
    }

    public void Quit()
    {
        Application.Quit();
    }
}
```

Figure 71: GameManager ScriptableObject with methods to control time scale, load scenes, and quit the application. Source: Own elaboration

Its nature as a ScriptableObject allows it to be referenced anywhere in different scenes while staying loaded in memory between scenes. Practical uses of the Game Manager include adjusting the time scale to a slower rate when beginning the temperature selection for the player's expansion wave, setting it to 0 when pausing the game, changing scenes when a level is selected, or exiting the application.



On Click ()

Runtime Only | StringEvent.Invoke
Play FX Audio Event (S | UI Confirm

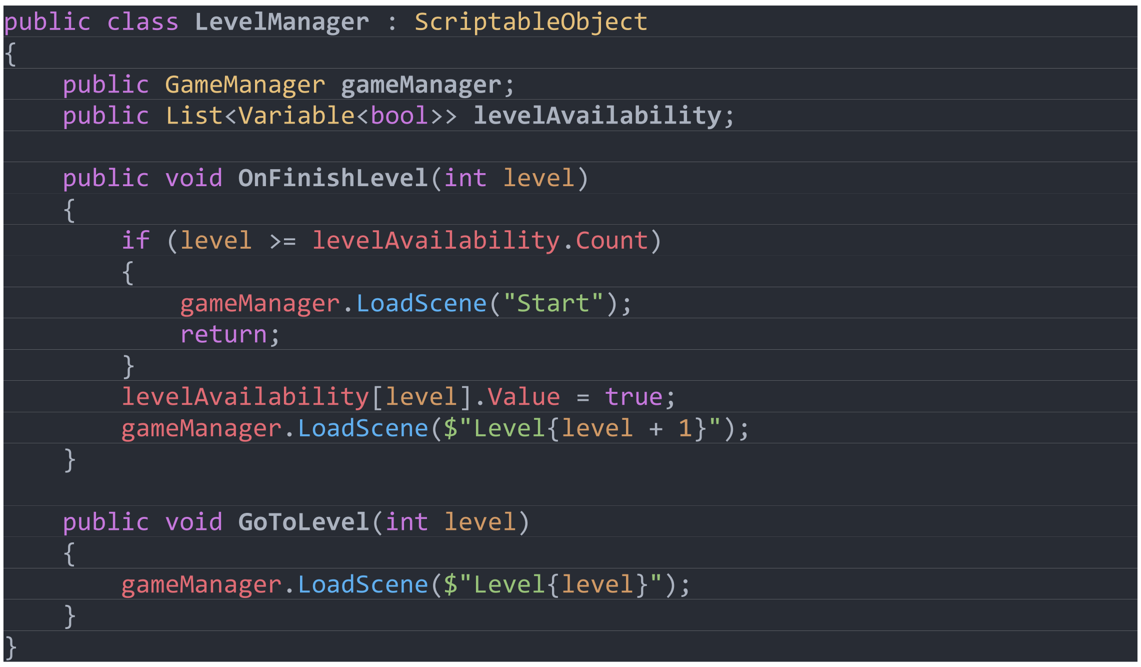
Runtime Only | GameManager.Quit
Game Manager (Gam

Figure 72: Unity Inspector showing the On Click event of the game's exit button setup to invoke the GameManager's Quit method and play a UI confirmation sound. Source: Own elaboration

Level Manager

Another example is the Level Manager, which holds references to the Bool Variables indicating the availability of the levels and controls the flow between levels. Since both the Level Manager and Game Manager are ScriptableObjects, the Level Manager can reference the Game Manager and call its methods when changing levels. This separation of concerns ensures that if scene loading logic changes in the future, such as displaying a loading screen while loading asynchronously, the

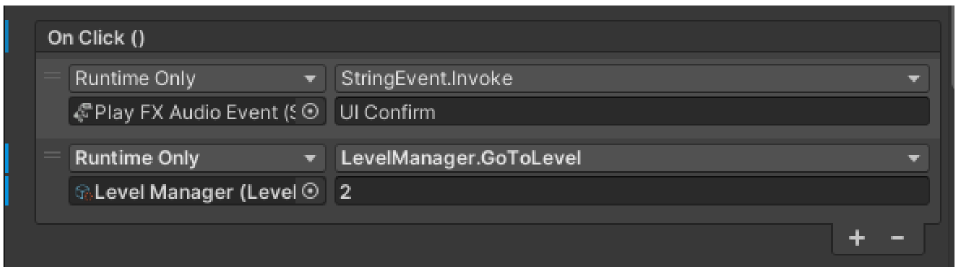
Level Manager does not need to know about these changes; it simply continues to call the same method in the Game Manager.



```
public class LevelManager : ScriptableObject
{
    public GameManager gameManager;
    public List<Variable<bool>> levelAvailability;

    public void OnFinishLevel(int level)
    {
        if (level >= levelAvailability.Count)
        {
            gameManager.LoadScene("Start");
            return;
        }
        levelAvailability[level].Value = true;
        gameManager.LoadScene($"Level{level + 1}");
    }

    public void GoToLevel(int level)
    {
        gameManager.LoadScene($"Level{level}");
    }
}
```

Figure 73: Level Manager ScriptableObject, with methods to control level flow. Source: Own elaboration.

The Level Manager is directly referenced from UnityEvents in the level selection buttons and the triggers that detect when a player reaches the end of a level. Since ScriptableObjects are scene-independent, they can be referenced directly from the prefabs of these elements without the need for significant logic changes or scene-specific references.



On Click ()

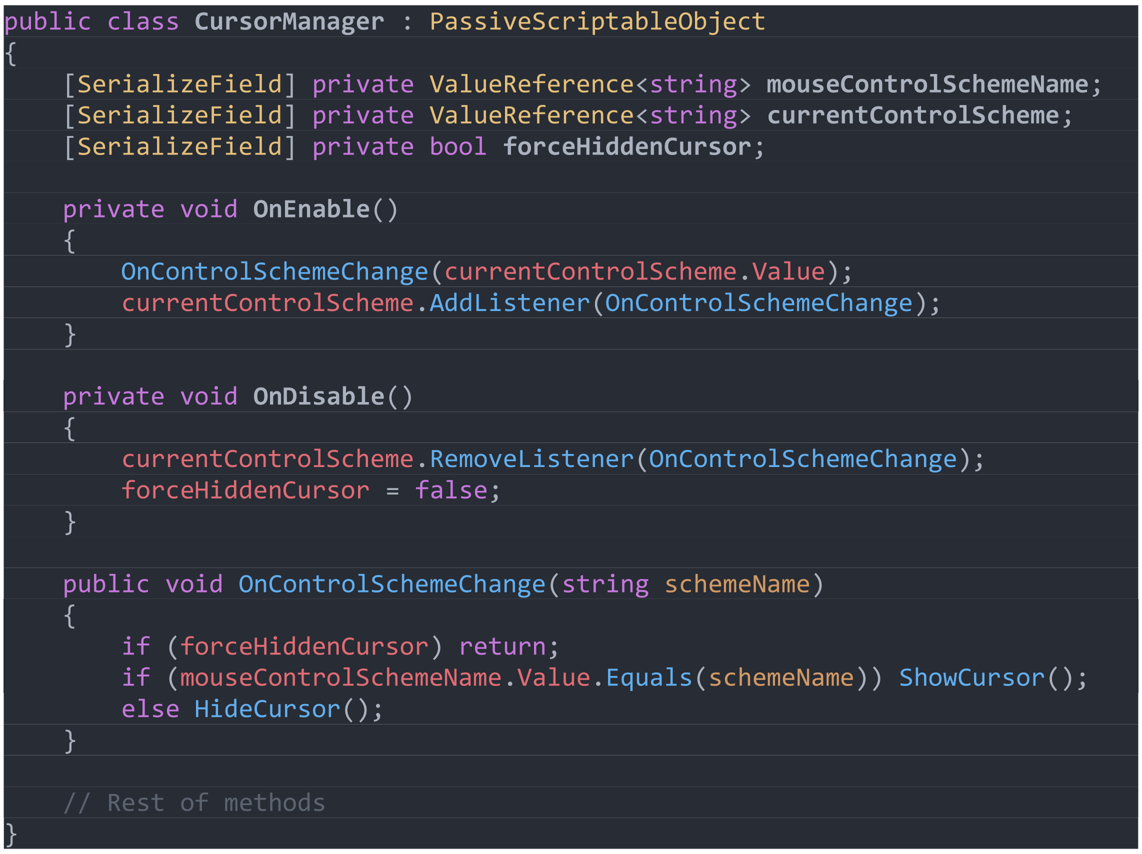
Runtime Only | StringEvent.Invoke
Play FX Audio Event (S | UI Confirm

Runtime Only | LevelManager.GoToLevel
Level Manager (Level | 2

Figure 74: Inspector view of the On Click UnityEvent of a level selection button, calling the Level Manager to navigate to level 2. Source: Own elaboration.

Cursor Manager

The Cursor Manager is another example of a ScriptableObject Manager. It centralizes the management of the cursor state, showing or hiding it as necessary. The Cursor Manager references the previously created Control Scheme Handler, allowing it to passively change the cursor's visibility based on the current control scheme. For instance, the cursor is made visible when the control scheme switches to one associated with the mouse and hidden when switched to a gamepad control scheme. Additionally, the cursor state can be forced to hidden independently of the control scheme when the player is in a level, ensuring that the cursor's visibility is consistent with the game's requirements.



```
public class CursorManager : PassiveScriptableObject
{
    [SerializeField] private ValueReference<string> mouseControlSchemeName;
    [SerializeField] private ValueReference<string> currentControlScheme;
    [SerializeField] private bool forceHiddenCursor;

    private void OnEnable()
    {
        OnControlSchemeChange(currentControlScheme.Value);
        currentControlScheme.AddListener(OnControlSchemeChange);
    }

    private void OnDisable()
    {
        currentControlScheme.RemoveListener(OnControlSchemeChange);
        forceHiddenCursor = false;
    }

    public void OnControlSchemeChange(string schemeName)
    {
        if (forceHiddenCursor) return;
        if (mouseControlSchemeName.Value.Equals(schemeName)) ShowCursor();
        else HideCursor();
    }

    // Rest of methods
}
```

Figure 75: CursorManager ScritpableObject, managing cursor state depending on changes in the value of the current control scheme. Source: Own elaboration.

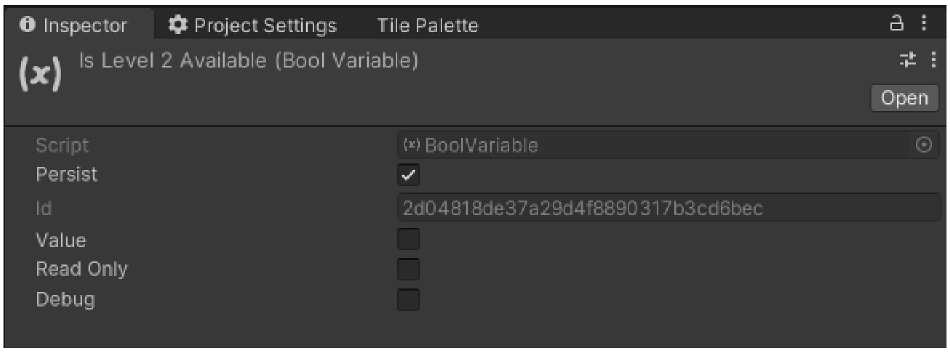
It is important to note that the Game Manager and Level Manager are not marked as Passive ScriptableObjects because they do not contain any functionality that requires them to be constantly loaded into memory. This contrasts with the Cursor Manager, which must remain passively loaded to handle real-time cursor state changes. This distinction ensures that only necessary objects remain persistently in memory, optimizing resource usage.

5.3.10 Game State Persistency

Some aspects of the game need to be persisted across play sessions to ensure continuity and a seamless player experience. These aspects include:

- **Levels:** The unlocked levels must remain unlocked.
- **Collectibles:** The collected coins must remain collected.
- **Settings:** The volume parameters configured by the player in the settings window must remain as last set.

This persistency can be ensured by leveraging Variable Repositories. First, it is important to identify which Scriptable Variables are part of the game state that needs to be persisted. In this case, these include the Bool Variables created to indicate the coins being collected, the Bool Variables indicating the availability of the levels, and the Float Variables indicating the value of the volume. These variables need to be marked as persistent by enabling the "persist" toggle option they integrate.

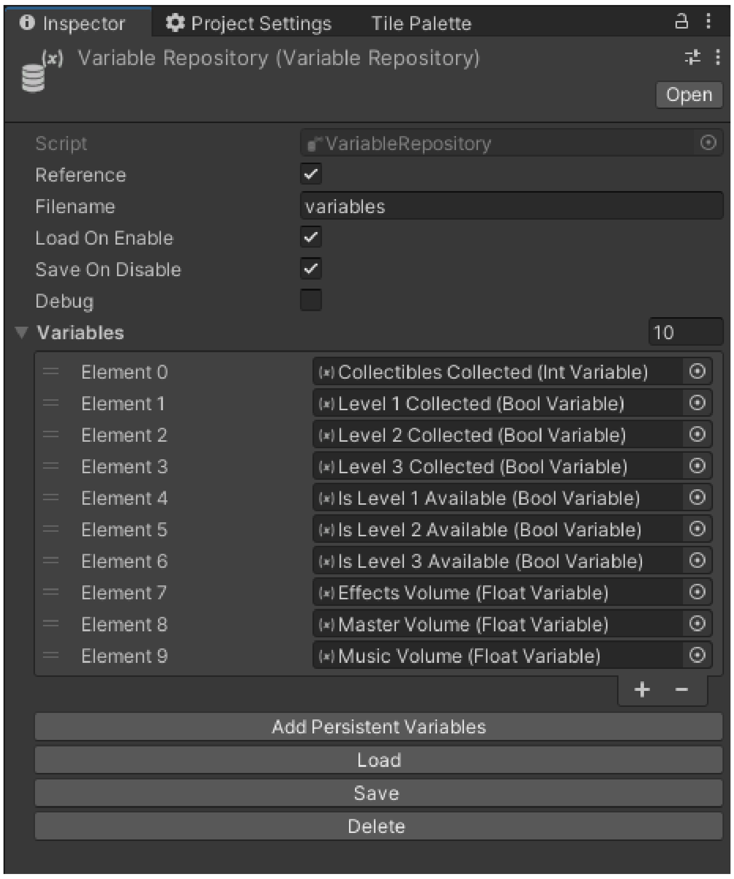


(x) Is Level 2 Available (Bool Variable)

Script (x) BoolVariable
Persist ✓
Id 2d04818de37a29d4f8890317b3cd6bec
Value
Read Only
Debug

Figure 76: Inspector view of Is Level 2 Available, one of the varaibles marked to be persisted. Source: Own elaboration.

Once the variables are marked as persistent, the next step is to create a Variable Repository. By pressing the "Add Persistent Variables" button displayed in the Inspector window, all variables marked as persistent are automatically added to the repository's list. The final step is to enable the options in the Variable Repository to automatically load the variables when the repository is loaded into memory and save them when it is unloaded, eliminating the need for manual loading and saving.



Script: VariableRepository
Reference: ✓
Filename: variables
Load On Enable: ✓
Save On Disable: ✓
Debug:
Variables: 10

Element 0	(x) Collectibles Collected (Int Variable)
Element 1	(x) Level 1 Collected (Bool Variable)
Element 2	(x) Level 2 Collected (Bool Variable)
Element 3	(x) Level 3 Collected (Bool Variable)
Element 4	(x) Is Level 1 Available (Bool Variable)
Element 5	(x) Is Level 2 Available (Bool Variable)
Element 6	(x) Is Level 3 Available (Bool Variable)
Element 7	(x) Effects Volume (Float Variable)
Element 8	(x) Master Volume (Float Variable)
Element 9	(x) Music Volume (Float Variable)

Figure 77: Inspector view of the Variable Repository, containing all the Scriptable Variables to be persisted. Source: Own elaboration.

Since the Variable Repository is a Passive ScriptableObject, it will be loaded at the beginning of the game, ensuring the variables are properly synchronized with their persistent state from the outset. Because all components and systems created in the development of this game are reactive to the values of these variables, the game will "configure itself" to the state of the last play session. This design removes the need for game managers that set up the game state at the beginning of a play session.

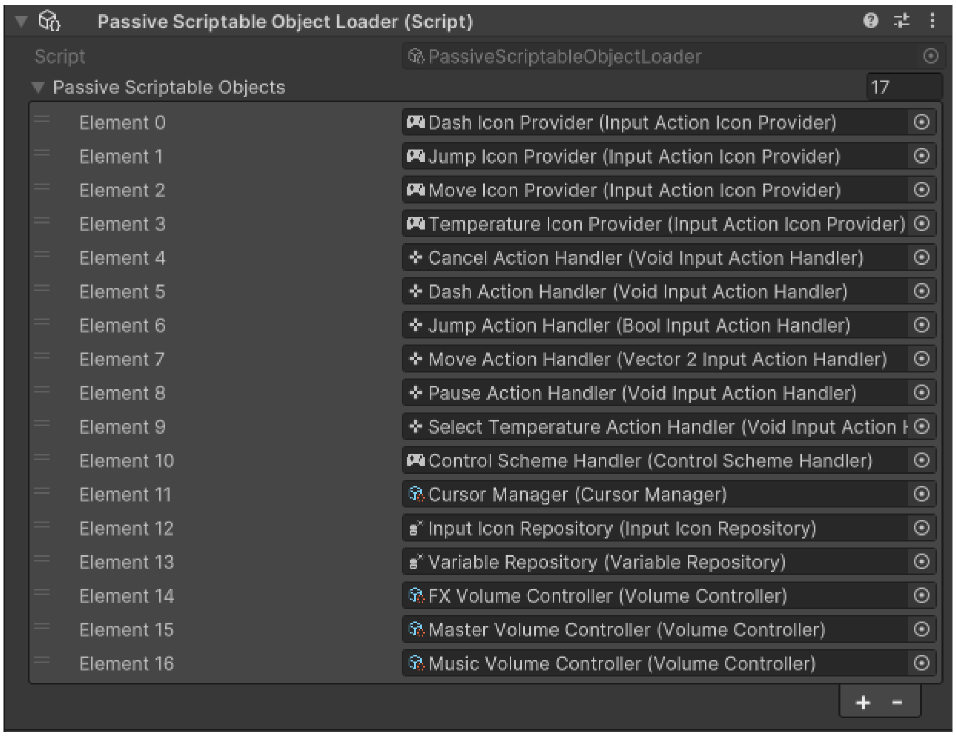
5.3.11 Managing Passive ScriptableObjects

To ensure all the created Passive ScriptableObjects are properly loaded into memory at runtime, it is crucial to verify that their toggle option, indicating they should be passively referenced, is set to true. This applies to both the Passive ScriptableObjects provided by the framework and those customized by extending the framework. The relevant ScriptableObjects include:

- Input Action Handlers**.**
- Input Icon Providers.
- Control Scheme Handler.
- Cursor Manager.
- Variable Repository.
- Volume Controllers.
- Variable Repository

By default, Passive ScriptableObjects have this toggle option set to true when created. However, there might be instances where they have been manually set to false to prevent them from being loaded, typically as the game requirements evolve. Ensuring the toggle remains set to true is essential for maintaining their passive loading behaviour.

The final step in managing Passive ScriptableObjects is to click on the “Add References to Passive ScriptableObjects” option in the SODD Framework’s menu. This action results in the creation of a GameObject in all scenes included in the final build of the game, which references all the Passive ScriptableObjects. This ensures that these objects are loaded into memory as intended. If new Passive ScriptableObjects need to be created in the future to accommodate new functionalities, clicking this menu option again will update the references accordingly.


Passive Scriptable Object Loader (Script)

Script	PassiveScriptableObjectLoader
Passive Scriptable Objects	17
Element 0	Dash Icon Provider (Input Action Icon Provider)
Element 1	Jump Icon Provider (Input Action Icon Provider)
Element 2	Move Icon Provider (Input Action Icon Provider)
Element 3	Temperature Icon Provider (Input Action Icon Provider)
Element 4	Cancel Action Handler (Void Input Action Handler)
Element 5	Dash Action Handler (Void Input Action Handler)
Element 6	Jump Action Handler (Bool Input Action Handler)
Element 7	Move Action Handler (Vector 2 Input Action Handler)
Element 8	Pause Action Handler (Void Input Action Handler)
Element 9	Select Temperature Action Handler (Void Input Action H
Element 10	Control Scheme Handler (Control Scheme Handler)
Element 11	Cursor Manager (Cursor Manager)
Element 12	Input Icon Repository (Input Icon Repository)
Element 13	Variable Repository (Variable Repository)
Element 14	FX Volume Controller (Volume Controller)
Element 15	Master Volume Controller (Volume Controller)
Element 16	Music Volume Controller (Volume Controller)

Figure 78: The generated object referencing all the Passive ScriptableObjects. Source: Own elaboration

6 Conclusions

The journey to develop a SODD Framework for Unity has been both challenging and rewarding. This project set out to explore and extend the possibilities of using ScriptableObjects in game architecture, aiming to create a tool that could streamline and enhance the game development process. This section reflects on the achievements, key learnings, setbacks, and future directions for the framework, as well as assess the extent to which the project objectives were met.

6.1 Objectives

- **Developing the framework**: The primary objective of developing a specialized framework for Scriptable Object Driven Development has been successfully achieved. The SODD framework effectively implements and expands upon the principles of modularity, editability, and debuggability. This achievement has resulted in a robust tool that enhances productivity and collaboration in Unity's development environment.

- **Documentation**: Comprehensive documentation has been created and hosted for the framework. Detailed documentation is provided directly in the source code as XML comments, making it accessible for anyone importing and using the framework in their Unity project. Additionally, the documentation has been generated and hosted on GitHub Pages, where users can refer to the manual, API reference, changelog, and license. This extensive documentation ensures that the framework's features and functionalities are accessible and understandable to developers, thereby encouraging its adoption.

- **Sample Videogame**: To evaluate and demonstrate the practical viability of the framework, a sample video game was developed using SODD as the primary methodology. This objective was achieved successfully, as the sample game, Ice Heat, effectively showcased the framework's capabilities and provided a practical validation of its design and functionality.

- **Publication and Accessibility**: The objective of publishing the framework has been achieved through its public hosting on GitHub as an open-source project.

The repository can be imported into any Unity project via UPM. Moreover, it can be expanded through open-source collaboration. This availability broadens the framework's reach to potential users, promotes community engagement, gathers feedback, and attracts potential collaborators, contributing significantly to the open-source community.

6.2 Learnings and Achievements

- **Improving Game Architecture**: One of the significant achievements of the SODD framework is its successful implementation of a modular approach to game architecture. By encapsulating game data and behaviours into Scriptable Objects, the framework reduces dependencies and enhances flexibility. This modular design allows for individual components to be developed, tested, and maintained independently, promoting reusability across different projects. The modularity ensures that changes in one part of the system do not adversely affect others, leading to a more stable and maintainable codebase.

- **Empowering Designers:** Another primary goal of the SODD Framework was to facilitate a more designer-friendly environment. The use of Scriptable Variables and Events allows game designers to modify game parameters and behaviours directly through the Unity Inspector without needing to dive into the codebase. This feature significantly reduces the iteration time during the game design process, empowering designers to experiment and fine-tune game elements efficiently. The editability provided by the framework enables rapid prototyping and adjustments, enhancing the overall development workflow.

- **Open-Source Project Management:** A key learning of the project was the creation, hosting, and maintenance of an open-source Unity Package that follows Unity's standards. This involved structuring the project to meet Unity's package requirements and ensuring that it could be easily integrated into any Unity project. Additionally, leveraging the potential of Continuous Integration/Continuous Deployment (CI/CD) in GitHub was crucial. The CI/CD pipelines were set up to automate the update, publication, and hosting of the package documentation, ensuring that the latest changes and improvements were always available to users.

6.3 Challenges and Setbacks

The primary setback of this project has been time constraints. Given its scope, it was sometimes necessary to omit certain elements to prioritize more critical requirements.

- **Left Out Implementations**: Some specific implementations for Scriptable Variables, Events, and Collections outlined in the planning phase were not completed. These implementations were classified as "Could Have" and "Won't Have" in the MoSCoW prioritization scheme, meaning they were not essential for the proper functioning of the SODD Framework.

- **Development Workflow**: The development methodology established in the planning phase included creating User Stories for each feature, defining clear requirements such as comprehensive documentation and proper test coverage. While this methodology was followed for most of the process, time constraints and the need to finalize certain features before developing the sample video game led to deviations from this methodology in some instances.

- **Manual for the Developed Videogame**: Ice Heat served as a successful example of the practical use of the Framework. However, beyond being a proof of concept, Ice Heat was intended to serve as a tutorial for users of the framework, demonstrating how functionalities are implemented. Unfortunately, a manual explaining the process of creating the video game and navigating the project was not created due to time constraints.

- **Lack of a Deeper Manual for the Framework**: While the technical documentation of the framework, also referred to as the Scripting API Reference in Unity Packages, was successfully created—fulfilling a key documentation requirement—the user manual, which serves as a guide to using each feature of the framework, is somewhat shallow. It requires more comprehensive coverage to fully support users.

- **Architecture decisions in the Videogame**: Although following the principles of SODD ensures good practices and results in a sound game architecture, time constraints led to design decisions that, in hindsight, could be improved for a more accurate application of these principles. This suggests that the SODD Framework alone does not guarantee proper practices; users need knowledge on how to apply SODD principles effectively for successful use of the framework.

6.4 Future Work

- **Framework Enhancements:** The SODD Framework provides a robust foundation, but there are numerous opportunities for further enhancements. Future work should focus on expanding the framework's capabilities by incorporating the implementations that have been omitted in this project, as well as providing additional Scriptable Object types and enhancing existing ones to support more complex use cases and data types. Moreover, introducing an editor window to consolidate all framework features into a central menu, rather than relying on Unity’s toolbar, would significantly improve user experience and efficiency.

- **Platform Expansion and Community Engagement:** Increasing the framework's visibility and adoption through additional platforms, such as the Unity Asset Store, represents another significant area of improvement. This would not only enhance accessibility but also foster community engagement. Collecting feedback from users and encouraging collaborative development will be invaluable for identifying areas of improvement, ensuring the framework evolves to meet the needs of a diverse and growing user base.

- **Documentation:** Expanding on the existing documentation and providing more tutorials will help new users quickly understand and adopt the framework. By providing clear and detailed guides, the framework's accessibility and usability can be significantly increased, leading to wider adoption within the Unity community.

6.5 Final Conclusion

The positive outcomes observed in this project highlight the potential for widespread adoption of Scriptable Object Driven Development within the Unity community. By achieving the established objectives and addressing common challenges in Unity development, the SODD Framework offers a potential new standard for efficient and effective game development practices.

7 References

Ahmad, K. S., Ahmad, N., Tahir, H., & Khan, S. (2017). Fuzzy_MoSCoW: A fuzzy based MoSCoW method for the prioritization of software requirements. *International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, (pp. 433-437).

Ali, J. M. (2023). DevOps and continuous integration/continuous deployment (CI/CD) automation. *Advances in Engineering Innovation*.

Baglie, L. S., Neto, M. P., Guimarães, M., & Brega, J. (2017). Distributed, Immersive and Multi-platform Molecular Visualization for Chemistry Learning. *Communication Systems and Applications*.

Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., & Züllighoven, H. (1997). Framework Development for Large Systems. *Communications of the ACM, 40*(10), 52-59.

Casquina, J. C., & Montecchi, L. (2021). A proposal for organizing source code variability in the git version control system. *Proceedings of the 25th ACM International Systems and Software Product Line Conference*, *A.*

Contreras, N., & Rene, A. (2017). Master Thesis in Mechatronics from the FH Aachen. "Use and adoption of software design patterns for PLC based systems".

Diebold, P., Theobald, S., Wahl, J., & Rausch, Y. (2018). An Agile transition starting with user stories, DoD & DoR. *Proceedings of the 2018 International Conference on Software and System Process.*

Fine, R. (2016). Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution. Retrieved from https://www.youtube.com/watch?v=6vmRwLYWNRo&t=1725s

Freeman, A. (2015). *The Observer Pattern.*

Freeman, A. (2015). *The Singleton Pattern.*

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.*

Gatteschi, V., Lamberti, F., Montuschi, P., & Sanna, A. (2016). Semantics-Based Intelligent Human-Computer Interaction. *IEEE Intelligent Systems, 31*(4), 11-21.

Hipple, R. (2017). Game Architecture with Scriptable Objects. Retrieved from https://www.youtube.com/watch?v=raQ3iHhE_Kk

Hu, X., Shen, Z., Chen, Z., Ran, X., Xiong, X., & Wu, Y. (2023). Research on the Technology of the Multi-operating System Co-development Based on the Unity Platform. *IEEE 3rd International Conference on Electronic Technology, Communication and Information (ICETCI)*, 468-472.

Jackson, S. K. (2015). *Unity 3D UI Essentials.*

Khosravi, K., & Guéhéneuc, Y.-G. (2004). A Quality Model for Design Patterns.

Kumar, B., Tiwari, U. K., & Dobhal, D. C. (2022). User Story Splitting in Agile Software Development using Machine Learning Approach. *Seventh International Conference on Parallel, Distributed and Grid Computing (PDGC)*, (pp. 167-171).

Lin, W., Krogh-Jacobsen, T., Andreasen, P., & Bilas, S. (2022). *Level Up Your Code With Game Programming Patterns.*

Lukosek, G. (2016). *Learning C# by developing games with Unity 5.x.*

Mechtley, A., & Trowbridge, R. M. (2011). *Maya Python for Games and Film: A Complete Reference for Maya Python and the Maya Python API.*

Naveen, B., Grandhi, J. K., Lasya, K., Reddy, E. M., Srinivasu, N., & Bulla, S. (2023). Efficient Automation of Web Application Development and Deployment Using Jenkins: A Comprehensive CI/CD Pipeline for Enhanced Productivity and Quality. *International Conference on Self Sustainable Artificial Intelligence Systems (ICSSAS)*, (pp. 751-756).

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

Pop, D. (2008). Introduction to Zend Framework. *Journal of Information Systems and Operations Management*(2), 507-512.

Power, K. (2014). Definition of Ready: An Experience Report from Teams at Cisco. *International Conference on Agile Software Development.*

Qu, J., Wei, Y., & Song, Y. (2014). Design patterns applied for networked first person shooting game programming. *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, (pp. 1-6).

Rautakopra, A. (2018). *Game Design Patterns: Utilizing Design Patterns in Game Programming.*

Singh, S., & Kaur, A. (2022). Game Development using Unity Game Engine. *2022 3rd International Conference on Computing, Analytics and Networks (ICAN)*, (pp. 1-6).

Unity Technologies. (2022). Unity User Manual 2022.3 (LTS). Retrieved from https://docs.unity3d.com/Manual/index.html

Unity Technologies. (2024). Input System 1.8.2. Retrieved from https://docs.unity3d.com/Packages/com.unity.inputsystem@1.8/manual/index.html

Unity. (n.d.). Unity Real-Time Development Platform | 3D, 2D, VR & AR Engine. Retrieved from https://unity.com

Wu, W., Khomh, F., Adams, B., Guéhéneuc, Y., & Antoniol, G. (2015). An exploratory study of api changes and usages based on apache and eclipse ecosystems. *Empirical Software Engineering*(21), 2366-2412.

8 Annex

8.1 Source code

SODD Framework Source Code:

(Root Folder)/Source/sodd-unity-framework-3.6.0.zip

Ice Heat Videogame Project:

(Root Folder)/Source/sample-videogame-project.zip

Ice Heat Videogame Build:

(Root Folder)/Source/sample-videogame-build.zip

Project's GitHub Repository:

https://github.com/aruizrab/sodd-unity-framework

8.2 Documentation

Note: To install the framework, it is necessary to follow the *Manual Installation* in the *Installation* section of the framework's manual, while using the annexed source code of the framework instead of the latest release download.

SODD Framework Manual:

(Root Folder)/Documentation/SODDFramework_Manual.pdf

SODD Framework API Reference:

(Root Folder)/Documentation/SODDFramework_API_Reference.pdf

8.3 Media

Ice Heat Gameplay:

(Root Folder)/Media/IceHeat_Gameplay.mp4