Table of Contents

SODD Unity Framework

Welcome to the SODD Unity Framework, a robust tool for architecting your games using ScriptableObjects in Unity.

Introduction

Overview of the SODD Framework

The ScriptableObject Driven Development (SODD) Framework is an innovative and efficient approach to game development in Unity, designed to enhance modularity, maintainability, and scalability. Building on the foundational concepts introduced by Ryan Hipple at the Unite Austin conference in 2017, the SODD Framework leverages the power of ScriptableObjects to streamline the development process and overcome common challenges associated with traditional MonoBehaviour-centric architectures.

ScriptableObjects in Unity provide a robust way to manage data independently of the GameObjects in the scene, promoting a more organized and decoupled system. By encapsulating game events, variables, and collections within ScriptableObjects, the SODD Framework enables developers to create self-contained, reusable components that can be easily managed and modified without directly affecting other parts of the game.

Benefits and Applications

The SODD Framework offers several key benefits that make it a valuable tool for both independent developers and large development teams:

- **Modularity**: The framework allows for the creation of independent, self-contained game components. This reduces interdependencies and makes it easier to manage and update individual parts of the game without impacting the entire system. Modular components can be reused across different projects, saving development time and effort.
- **Editability**: With ScriptableObjects, game data can be easily modified through the Unity Inspector, enabling rapid prototyping and iterative development. Designers and developers can adjust game parameters on the fly, without needing to dive into the codebase, thus enhancing flexibility and responsiveness during the development process.
- **Debuggability**: The framework includes features for logging and tracking events and data changes, which simplifies debugging and testing processes. By maintaining clear and detailed logs of event invocations and variable updates, developers can quickly identify and resolve issues, ensuring a smoother development experience.

- **Scalability**: The decoupled nature of ScriptableObjects allows for scalable game architectures. As the game grows in complexity, the framework ensures that new features and systems can be added without causing significant disruptions to the existing structure.

Purpose and Goals

The primary goal of the SODD Framework is to provide a comprehensive toolset that enhances productivity and collaboration in Unity's development environment. By offering a structured approach to managing game data and events, the framework aims to:

1. **Simplify Dependency Management**: Reduce the complexity of interactions between different game systems, making it easier to maintain and expand the game over time.
2. **Enhance Collaboration**: Enable designers, developers, and other team members to work more effectively by providing clear interfaces and tools for modifying game behavior and data.
3. **Improve Code Quality**: Promote best practices in game engineering, such as modularity, data-driven design, and event-driven architecture, to create cleaner, more maintainable codebases.

Target Audience

The SODD Framework is designed for a wide range of Unity developers, from beginners to seasoned professionals. Whether you are an indie developer working on a small project or part of a large team developing a complex game, the framework provides the tools and methodologies to improve your development process. Familiarity with Unity and C# programming is recommended to fully leverage the capabilities of the SODD Framework.

Installation

> **NOTE**
> The SODD Framework requires Unity version 2022.3 or later.

To integrate the **SODD Framework** into your Unity project, follow these steps:

1. In your project, open the Package Manager by going to `Window` > `Package Manager`.
2. In the Package Manager, click on the `+` icon in the top left corner and select '**Add package from git URL**'. For more detailed instructions, see the Unity documentation on installing from a Git URL.
3. Enter the following URL into the text field and click '**Add**':

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

> **IMPORTANT**
> The SODD Framework requires Unity's Input System package and TextMeshPro package. If it is not already installed in your project, Unity will prompt you to install it as a dependency when adding the SODD Framework.

Once the above steps are completed, the latest version of the SODD Framework will appear in your project packages list, indicating successful installation.

Theory behind the SODD Framework

Context

Unity 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 a novel approach to game development employing ScriptableObjects. Hipple's proposal advocates for a more modular and manageable game architecture in Unity, harnessing the potential of ScriptableObjects to mitigate common development challenges.

Problem: Common Scripting Challenges in Unity

▶ Table of Contents

Singleton Problems

Rigid Connections

One of the significant issues with Singletons in Unity is the creation of rigid connections between different systems. When systems are tightly coupled, modifying or extending one part of the system necessitates changes in others, which reduces flexibility and can introduce new bugs. This tight coupling means that components cannot easily be reused or reconfigured without affecting other parts of the game, leading to a more fragile and less maintainable codebase.

Loss of Polymorphism

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 (a Singleton), 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 behaviors, thus reducing the overall flexibility of the architecture.

Testing Difficulties

The hidden dependencies introduced by Singletons pose significant challenges for unit testing. Since Singletons are globally accessible and maintain state, tests can become interdependent, leading to flaky or unreliable tests. Isolating a single component for testing is challenging when it depends on the global state maintained by Singletons, making it harder to identify the source of a failure.

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 can cause unpredictable behavior. These dependency issues make the system more error-prone and harder to debug.

Global State

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.

Single Instance Limitation

By design, a Singleton pattern limits the system to a single instance of a class. This restriction 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 limitation can be time-consuming and error-prone.

Challenges with Traditional Event Systems

Unity Event Limitations

Unity’s built-in event system, UnityEvent, has several limitations. It relies on serialized function calls, which means the exact function to be called must be specified at design time. This approach is rigid and does not support dynamic binding or more complex event handling scenarios. Furthermore, UnityEvent can introduce garbage collection issues due to the way it resolves and invokes functions using reflection, which can impact performance, especially in performance-critical applications like games.

Hard References

UnityEvents create hard references between components, leading to maintenance difficulties and reduced modularity. Each event binding is a direct link from the event source to the event handler, making it challenging to refactor or extend the system. This tight coupling hinders the ability to decouple systems and create more modular and reusable components.

Issues with Enums

Code-Driven

Enums in Unity are defined in code, which makes them less flexible and harder to modify. Adding, removing, or reordering enum values requires changes in the codebase and recompilation, which can be cumbersome and error-prone, especially in large projects.

Difficult to Reorder or Remove

When enums are serialized, their values are stored based on their index positions. Changing the order or removing enum values can break the serialized data, causing runtime errors and inconsistencies. This limitation makes maintaining and evolving the game’s data structures more challenging over time.

No Additional Data

Enums cannot hold additional data, which limits their utility. Often, developers end up creating additional lookup tables or data structures to associate more data with each enum value. This extra complexity adds overhead and increases the risk of synchronization issues between the enum and the associated data.

Scene Management Problems

Transient Data Management

Managing transient data across scenes can be complex, particularly when using DontDestroyOnLoad objects. These objects persist across scene loads, which can clutter the global namespace and lead to bugs due to unintended interactions between persistent and transient data. Ensuring that only the necessary data persists while keeping the scene-specific data isolated is a delicate balance.

Clean Slate Scenes

Ensuring each scene loads as a clean slate is essential for maintaining modularity and flexibility. Persistent data or objects that carry over from previous scenes can introduce hard-to-track bugs and unintended behavior. By adhering to the clean slate principle, developers can ensure that each scene starts with a controlled and predictable state, reducing the risk of unexpected interactions.

Debugging Issues

Lack of Debugging Tools Debugging complex interactions between components and systems can be difficult without proper tools and strategies. A lack of built-in debugging support for custom architectures makes it challenging to diagnose issues. Ensuring that every feature has a clear debugging strategy and providing tools to expose runtime state and interactions are crucial for effective debugging.

General Architectural Challenges

Maintaining Flexibility and Extensibility

As games grow in complexity, maintaining an architecture that is both flexible and extensible becomes increasingly challenging. The architecture must allow for easy addition of new features and modifications without extensive refactoring. Achieving this balance requires careful planning and adherence to principles that promote modularity and decoupling.

Performance Overheads

Ensuring that architectural patterns do not introduce significant performance overheads is crucial, especially for resource-constrained platforms like mobile devices and VR. While achieving a clean and maintainable architecture is important, it should not come at the cost of runtime performance. Profiling and optimizing critical paths in the architecture are essential to maintain performance.

Design and Code Separation

Balancing the needs of designers and developers can be challenging. Designers require systems that are easy to use and flexible, while developers need robust and maintainable code. Creating an architecture that allows designers to work independently of developers, through data-driven and component-based systems, helps bridge this gap and improves collaboration.

Solution: Game Architecture with Scriptable Objects

Ryan Hipple's solution to common problems encountered in Unity game development revolves around the innovative use of ScriptableObjects. By leveraging ScriptableObjects, developers can create a more modular, editable, and debuggable game architecture. This approach addresses key issues such as rigid dependencies, inflexible systems, and debugging challenges. Central to this methodology are three core principles of game engineering: modularity, editability, and debuggability.

▶ Table of Contents

The Three Principles of Game Engineering

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.

Replacing Singletons

Modularizing Data with ScriptableObjects

One of the primary issues with Singletons is their tendency to create rigid and tightly coupled systems. Hipple's approach involves using ScriptableObjects to modularize data, eliminating the need for Singletons. ScriptableObjects allow data to be shared across multiple systems without creating direct dependencies. For instance, instead of a Singleton managing player health, a ScriptableObject can store the health data. This data can then be referenced by any system that needs it, such as the UI or enemy AI, without creating tight couplings.

Runtime Sets

To manage lists of objects dynamically, Hipple introduces the concept of Runtime Sets. A Runtime Set is a ScriptableObject that maintains a list of objects at runtime. Objects can register themselves with these sets when they are instantiated and deregister when they are destroyed. This approach eliminates the need for Singleton managers to keep track of objects, thereby reducing dependency issues and race conditions. For example, an EnemyRuntimeSet can track all active enemies in the game, making it easy for various systems to access the list without relying on a Singleton.

Dependency Injection Principles

Hipple emphasizes the use of dependency injection principles to further decouple systems. In Unity, this can be achieved through the inspector, which acts as a dependency injector. Components can be assigned their dependencies via the inspector, reducing the need for hard-coded references and enabling easy substitution of different implementations. This method enhances modularity and makes testing and extending systems simpler.

Enhancing Event Systems

Unity’s built-in event system (UnityEvent) has several limitations, such as rigid bindings and performance overheads due to garbage collection. Hipple suggests creating a custom event system using ScriptableObjects. In this system, ScriptableObjects represent events, and listeners register with these events to receive notifications. This decouples event producers from consumers, allowing for more flexible and maintainable event handling. New listeners can be added or removed without modifying the event source, and events can be raised without knowing which objects will respond.

Replacing Enums

Hipple suggests using Scriptable Objects to represent enumerated types instead of traditional enums. Each possible value is a Scriptable Object, which can hold additional data and behavior, allowing for dynamic modification of the set of values. For example, instead of using an enum for different weapon types (e.g., Sword, Bow, Magic), create a Scriptable Object for each weapon type. These Scriptable Objects can contain additional data like damage values, range, and special effects. This approach allows for adding new weapons without changing the codebase and avoids serialization issues.

Improving Scene Management

To ensure that each scene loads as a clean slate, Hipple advocates minimizing the use of `DontDestroyOnLoad` objects. Instead, persistent data should be managed through ScriptableObjects. This approach prevents unwanted data from carrying over between scenes and ensures that each scene starts with a controlled and predictable state. By using ScriptableObjects to manage persistent data, developers can maintain a modular and flexible scene structure.

Debugging Enhancements

Hipple highlights the importance of integrating debugging tools into the Unity Inspector. By exposing runtime states and allowing events to be raised directly from the Inspector, developers can diagnose and fix issues more efficiently. This method makes debugging more intuitive and less time-consuming, providing immediate feedback on the game's state and interactions.

Data-Driven Design

Hipple’s approach strongly favors data-driven design, where game systems are configured through data rather than hard-coded logic. ScriptableObjects play a central role in this design philosophy by storing configurations, variables, and references. This method allows

for more flexible and dynamic game behavior, as changes can be made through data adjustments without altering the underlying code.

Single Responsibility Principle

Adhering to the single responsibility principle is crucial in Hipple’s architecture. Each component or system in the game should have a well-defined responsibility. This principle makes the system more modular, easier to manage, and simpler to extend. Components can be developed, tested, and debugged in isolation, reducing overall system complexity.

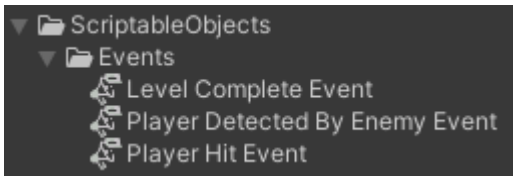
Scriptable Events

Concept

A Scriptable 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.

In traditional game development paradigms, events are often tightly woven into the fabric of the game's components, leading to a high degree of coupling and interdependency. The approach of Scriptable Events, however, conceptualizes events as independent modules, encapsulated within Scriptable Objects. This modularization of events allows for a more dynamic and loosely coupled system design, where various game components can react to events without being directly bound to the event source.

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



How It Works

- **Event Creation**: A Scriptable Event is created as a ScriptableObject, defining an event that can carry an optional payload (data associated with the event). There are two ways to create these events:
  - From the **Create menu**: Right-click in the Project window, navigate to `Create > SODD > Events`, and select the desired event type (e.g., `IntEvent`).
  - From the **SODD menu**: In the main menu, go to `Tools > SODD > Events`, and choose the desired event type from the provided options.
- **Listeners**: Components interested in the event can register as listeners. These listeners are notified when the event is triggered.
- **Event Invocation**: The event can be invoked from any script, notifying all registered listeners and passing along the optional payload.
- **Event Debugging**: Scriptable Events have a built-in debug option that logs invocations to the console. This helps in tracking event triggers and payloads, making it easier to debug and understand event flow.
- **Event Testing**: Scriptable Events can be invoked directly from the Unity Inspector. This allows developers to test events without writing additional code. The Inspector provides

an `Invoke` button and a field to enter the payload value, enabling quick and easy testing.

Implementations in the Framework

The SODD Framework provides several implementations of Scriptable Events to cover various data types and use cases. Below is a table detailing the core event types included in the framework:

Event Type	Description
Void Event	An event with no payload.
Bool Event	An event carrying a boolean payload.
Int Event	An event carrying an integer payload.
Float Event	An event carrying a float payload.
String Event	An event carrying a string payload.
Vector2 Event	An event carrying a 2D vector payload.
Vector3 Event	An event carrying a 3D vector payload.
GameObject Event	An event carrying a GameObject reference payload.

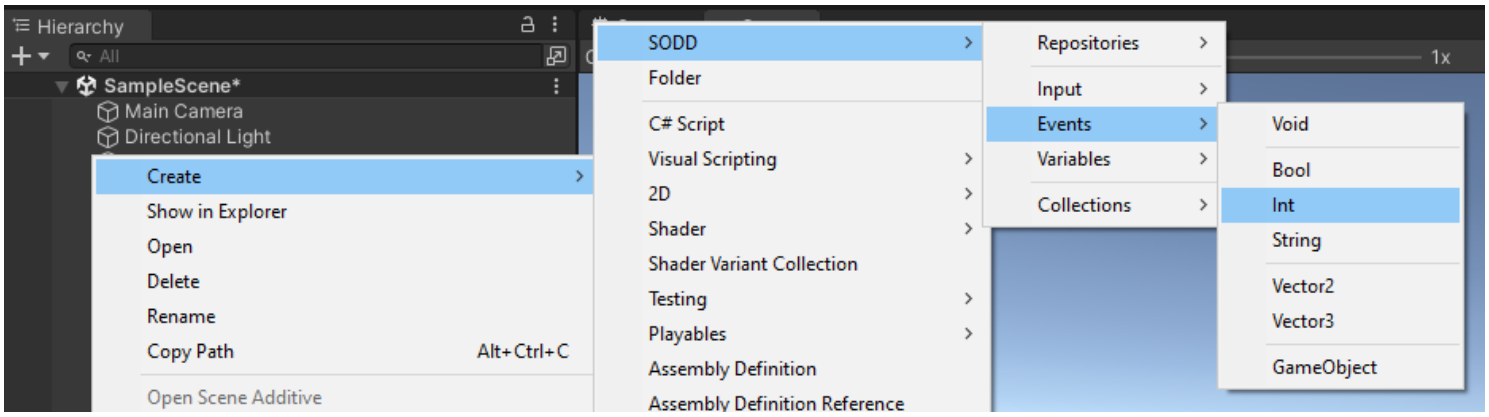
Practical Example

Let's walk through a practical example of managing a player's score using an `IntEvent`.

Step 1: Creating the Scriptable Event

First, we need to create a Scriptable Event that will signal when the player's score should be incremented.

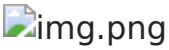
1. Right-click in the Project window of Unity.
2. Navigate to `Create > SODD > Events > Int`.



3. Name the newly created event `IncrementPlayerScoreEvent`.

This Scriptable Event will now be used to notify the system whenever the player's score needs to be increased.

In the inspector window of the newly created `IncrementPlayerScoreEvent`, you can specify the payload, enable the debug mode by checking the corresponding toggle, and invoke the event using the provided button. By doing so, you will see a new entry in the Unity 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.



Step 2: Creating the ScoreManager Script

Next, we'll create the `ScoreManager` script that will handle the score updates directly. This script will reference, subscribe to, and unsubscribe from the `IncrementPlayerScoreEvent`.

Here’s the code for the `ScoreManager` script:

```
public class ScoreManager : MonoBehaviour
{
    public Event<int> incrementScoreEvent; // Reference to the Scriptable Event
    public TMP_Text scoreDisplay; // Reference to the UI text component that
displays the score

    private int score; // Variable to keep track of the player's score

    private void OnEnable()
    {
        // Subscribe to the event
        incrementScoreEvent.AddListener(OnScoreIncremented);
        // Update the score display with the initial score
```

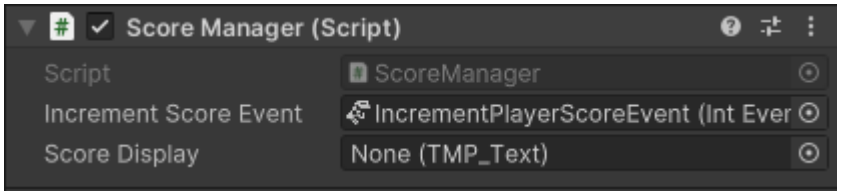
```
        scoreDisplay.text = score.ToString();
    }

    private void OnDisable()
    {
        // Unsubscribe from the event
        incrementScoreEvent.RemoveListener(OnScoreIncremented);
    }

    // Method to update the score and refresh the UI display
    private void OnScoreIncremented(int increment)
    {
        score += increment; // Increase the score by the given increment
        scoreDisplay.text = score.ToString(); // Update the score display text
    }
}
```

Once the script is created:

1. Create a new GameObject in the scene (e.g., ScoreManager).
2. Attach the new `ScoreManager` script to this GameObject.
3. Add the reference to the `IncrementPlayerScoreEvent` scriptable event in the `ScoreManager` script.



At this point, if you start the game and invoke `IncrementPlayerScoreEvent` by clicking the `Invoke` button on the inspector, you should see the score being updated in your UI component referenced by the `ScoreManager`.

Step 3: Triggering the Event

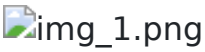
We need a way to trigger the `IncrementPlayerScoreEvent` inside the game. For this example, let's assume the player collects coins to increase their score. Each coin will trigger the event when collected.

Here’s the code for the `Coin` script:

```
public class Coin : MonoBehaviour
{
    public int scoreValue = 10; // Value to increase the score by
    public Event<int> incrementScoreEvent; // Reference to the Scriptable Event
```

```
    // Method called when another collider enters the trigger collider attached to
this GameObject
    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player")) // Check if the collider belongs
to the player
        {
            incrementScoreEvent.Invoke(scoreValue); // Trigger the event with the
score value
            Destroy(gameObject); // Destroy the coin GameObject
        }
    }
}
```

1. Create a Coin GameObject
2. Attach the new `Coin` script.
3. Add the reference to the `IncrementPlayerScoreEvent` scriptable event in the `Coin` script.



Step 4: Testing the Setup

Now, let's test the setup to ensure everything works as expected.

1. Run the game in the Unity Editor.
2. When the player collides with a coin, the `IncrementPlayerScoreEvent` is invoked.
3. The `ScoreManager` script, which has subscribed to the event, will receive the notification.
4. The `OnScoreIncremented` method in the ScoreManager script will be called, updating the player's score and refreshing the UI display.

Conclusion

By using Scriptable Events, we have successfully decoupled the `ScoreManager` and the `Coin` scripts. The `ScoreManager` script does not need a direct reference to the `Coin` objects, nor does it need to know how or when the score increment event is triggered. Similarly, the `Coin` script does not need to know about the existence of the `ScoreManager` or how it handles score updates. The role of the `IncrementPlayerScoreEvent` Scriptable Event is to act as a mediator that facilitates communication between these independent components.

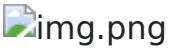
Scriptable Variables

Concept

A Scriptable Variable consists in a Scriptable Object that encapsulates a single value of a specific data type, such as integers, floats or strings. These modular data containers can then be shared and referenced across diverse systems within the game without said systems depending and relying on each other to retrieve and modify the game state, reducing direct dependencies and enhancing flexibility.

Additionally, the values of Scriptable Variables are exposed in the Inspector, allowing to see and manipulate them easily. This empowers both developers and designers, particularly those without extensive programming backgrounds, to easily modify game parameters.

In practical terms, Scriptable Variables can be used to store any value, such as game configuration, player settings, game state, etc.



How It Works

- **Variable Creation**: Scriptable Variables are created as ScriptableObject assets. These assets are designed to hold specific data types, such as integers, floats, strings, booleans, etc. There are two ways to create these variables:
  - From the **Create menu**: Right-click in the Project window, navigate to `Create > SODD > Variables`, and select the desired event type (e.g., `IntVariable`).
  - From the **SODD menu**: In the main menu, go to `Tools > SODD > Varaibles`, and choose the desired variable type from the provided options.
- **Data Access and Modification**: Components can reference these variables to get or set their values. If a component modifies a value, other components referencing the same variable will see the changes when reading the value.
- **Value Observers**: Components can register as listeners for value changes in Scriptable Variables. This allows for automatic updates when the variable's value changes, promoting a reactive architecture.
- **Value Change Debugging**: Scriptable Variables have a built-in debug option that logs value changes to the console.
- **Inspector Integration**: Values of Scriptable Variables can be viewed and modified directly in the Unity Inspector, making it easier to debug and test changes during development.
- **Persistence**: Scriptable Variables can maintain their values across different scenes and play sessions, supporting persistent game states and smoother development workflows.

> **NOTE**
> In the Unity Editor, the values of Scriptable Variables (like any other ScriptableObject implementation) are persisted across play sessions. However, in the final build, these values are reset to their original state when the game is closed. **To ensure the persistence of Scriptable Variable values across play sessions in the final build, use the VariableRepository.**

Implementations in the Framework

The SODD Framework offers several implementations of Scriptable Variables to cover various data types and use cases. Below is a table detailing the core variable types included in the framework:

Variable Type	Description
Int Variable	A ScriptableObject that holds an integer value.
Float Variable	A ScriptableObject that holds a float value.
String Variable	A ScriptableObject that holds a string value.
Bool Variable	A ScriptableObject that holds a boolean value.
Vector2 Variable	A ScriptableObject that holds a 2D vector value.
Vector3 Variable	A ScriptableObject that holds a 3D vector value.
GameObject Variable	A ScriptableObject that holds a GameObject reference.
LayerMask Variable	A ScriptableObject that holds a LayerMask value.

Practical Example

Let's walk through a practical example of managing a player's health using a `FloatVariable`.

Step 1: Creating the Scriptable Variable

First, we need to create a Scriptable Variable that will store the player's health.

1. Right-click in the Project window of Unity.
2. Navigate to `Create > SODD > Variables > Float`.
3. Name the newly created event `PlayerHealth`.

This Scriptable Variable will now hold the player's health value and can be accessed by various game components.

Step 2: Creating the Health Manager Script

Next, we'll create a script that handles increasing and decreasing the player's health when the player takes damage or heals. This script will reference the `PlayerHealth` variable directly.

Here’s the code for the `PlayerHealthManager` script:

```
using UnityEngine;
using SODD.Variables;

public class PlayerHealthManager : MonoBehaviour
{
    public Variable<float> playerHealth; // Reference to the Scriptable Variable

    public void TakeDamage(float damage)
    {
        playerHealth.Value -= damage; // Decrease the player's health
    }

    public void Heal(float amount)
    {
        playerHealth.Value += amount; // Increase the player's health
    }
}
```

Once the script is created:

1. Create a new GameObject in the scene (e.g., `Player`).
2. Attach the new `PlayerHealthManager` script to this GameObject.
3. Add the reference to the `PlayerHealth` Scriptable Variable in the `PlayerHealthManager` script.

Step 3: Creating the Health Display Script

We will create another script dedicated to displaying the health value on the screen. This script will also reference the `PlayerHealth` variable but will not interact directly with the health management logic.

Here’s the code for the `HealthDisplay` script:

```
using UnityEngine;
using TMPro;
using SODD.Variables;

public class HealthDisplay : MonoBehaviour
{
    public Variable<float> playerHealth; // Reference to the Scriptable Variable
    public TMP_Text healthText; // Reference to the UI text component that displays
the health

    private void OnEnable()
    {
        playerHealth.AddListener(UpdateHealthDisplay); // Subscribe to value changes
        UpdateHealthDisplay(playerHealth.Value); // Initialize the health display
with the current health value
    }

    private void OnDisable()
    {
        playerHealth.RemoveListener(UpdateHealthDisplay); // Unsubscribe from
value changes
    }

    private void UpdateHealthDisplay(float value)
    {
        healthText.text = value.ToString(); // Update the health display text
    }
}
```

1. Create a new GameObject in the scene (e.g., `UI`).
2. Attach the new `HealthDisplay` script.
3. Add the reference to the `PlayerHealth` scriptable event in the `HealthDisplay` script.

Step 4: Triggering Health Changes

We need a way to trigger changes to the player's health. For this example, let's assume the player can take damage from enemies and heal by collecting health packs.

> **NOTE**
>
> We are using the new Send method provided by the SODD framework to trigger health changes.

Here’s the code for the DamageDealer script:

```
using UnityEngine;
using SODD.Core;

public class DamageDealer : MonoBehaviour
{
    public float damageAmount = 10f; // Amount of damage to deal

    private void OnCollisionEnter(Collision collision)
    {
        if (collision.gameObject.CompareTag("Player")) // Check if the collider
belongs to the player
        {
            collision.gameObject.Send<PlayerHealthManager>(manager =>
manager.TakeDamage(damageAmount)); // Use Send method to deal damage to the player
        }
    }
}
```

Here’s the code for the HealthPack script:

```
using UnityEngine;
using SODD.Core;

public class HealthPack : MonoBehaviour
{
    public float healAmount = 20f; // Amount of health to restore

    private void OnTriggerEnter(Collider other)
    {
        if (other.gameObject.CompareTag("Player")) // Check if the collider belongs
to the player
        {
            collision.gameObject.Send<PlayerHealthManager>(manager =>
manager.Heal(healAmount)); // Use Send method to heal the player
            Destroy(gameObject); // Destroy the health pack
        }
    }
}
```

Step 6: Testing the Setup

Now, let's test the setup to ensure everything works as expected.

1. Run the game in the Unity Editor.
2. When the player collides with a DamageDealer, the PlayerHealth value decreases.
3. When the player collides with a HealthPack, the PlayerHealth value increases.
4. Observe the health value updating in the UI as the player takes damage or collects health packs.
5. Modify the health value directly in the Inspector during play mode to test the dynamic updating of the health display.

Conclusion

By using Scriptable Variables, we have successfully decoupled health management from health display, ensuring that the `PlayerHealthManager` handles health changes while the `HealthDisplay` script updates the UI independently. This decoupling enhances modularity, allowing each script to be modified, replaced, or reused without impacting others. New game logic regarding player health, such as effects when taking damage or healing, or restarting the level when health decreases to zero, can be added by creating new scripts that reference the variable without modifying existing scripts.

Scriptable Collections

Concept

At its core, a Scriptable Collection 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 Scriptable Collection is their ability to function as central repositories for specific types of objects, such as enemies, collectible items, or interactive game elements.

How It Works

- **Collection Creation**: A Scriptable Collection is created as a ScriptableObject, maintaining a list of specific types of objects. There are two ways to create these collections:
  - From the **Create menu**: Right-click in the Project window, navigate to `Create > SODD > Collections`, and select the desired event type (e.g., `GameObjectCollection`).
  - From the **SODD menu**: In the main menu, go to `Tools > SODD > Colelctions`, and choose the desired collection type from the provided options.
- **Collection Management**: Items can be dynamically added to or removed from the collection during runtime.
- **Inspector Integration**: The current state of the collection can be viewed and modified directly in the Unity Inspector, aiding in debugging and testing.
- **Listeners**: Scriptable Collections provide events that are triggered when items are added or removed, allowing other components to react to these changes.
- **Persistence**: Collections maintain their state across different scenes within the editor. However, they reset to their initial state when the game is restarted in a build.

Implementations in the Framework

The SODD Framework offers several implementations of Scriptable Collections to cover various use cases. Below is a table detailing the core collection types included in the framework:

Collection Type	Description
GameObject Collection	A collection that holds references to GameObjects.
Transform Collection	A collection that holds references to Transforms.
String Collection	A collection that holds a set of strings.
Int Collection	A collection that holds a set of integers.

Collection Type	Description
Float Collection	A collection that holds a set of floats.
Vector2 Collection	A collection that holds a set of Vector2s.
Vector3 Collection	A collection that holds a set of Vector3s.

Practical Example

To illustrate the functionality of Scriptable Collections, let's explore a practical use case termed "lock and key." 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.

Step 1: Creating the Player Inventory Collection

First, we need to create a GameObject Collection named "Player Inventory" to represent the player's inventory throughout the level.

1. Right-click in the Project window of Unity.
2. Navigate to `Create > SODD > Collections > GameObject`.
3. Name the newly created collection `PlayerInventory`.

This collection will now be used to manage the player's inventory items.

Step 2: Adding Keys to the Inventory

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

Here’s the code for the `Key` script:

```
using UnityEngine;

public class Key : MonoBehaviour
{
    public GameObjectCollection playerInventory;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            playerInventory.Add(gameObject);
            gameObject.SetActive(false); // Optionally deactivate the key GameObject
```

```
        }
    }
}
```

1. Attach the `Key` script to key GameObjects in the scene.
2. Assign the `PlayerInventory` collection to the `playerInventory` field in the Key script.

Step 3: Unlocking Doors

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.

Here’s the code for the `Door` script:

```
using UnityEngine;
using SODD.Collections;

public class Door : MonoBehaviour
{
    public Collection<GameObject> playerInventory;
    public GameObject requiredKey;

    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player") && playerInventory.Contains(requiredKey))
        {
            UnlockDoor();
        }
    }

    private void UnlockDoor()
    {
        Debug.Log("Door unlocked");
    }
}
```

1. Attach the `Door` script to door GameObjects in the scene.
2. Assign the `PlayerInventory` collection to the `playerInventory` field in the Door script.
3. Assign the specific key GameObject to the `requiredKey` field in the Door script.

Step 4: Displaying Key Count

An additional script can be 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.

Here’s the code for the KeyCountDisplay script:

```
using UnityEngine;
using TMPro;
using SODD.Collections;

public class KeyCountDisplay : MonoBehaviour
{
    public Collection<GameObject> playerInventory;
    public TMP_Text keyCountText;

    private void OnEnable()
    {
        playerInventory.OnItemAdded += UpdateKeyCount;
        playerInventory.OnItemRemoved += UpdateKeyCount;
        UpdateKeyCount();
    }

    private void OnDisable()
    {
        playerInventory.OnItemAdded -= UpdateKeyCount;
        playerInventory.OnItemRemoved -= UpdateKeyCount;
    }

    private void UpdateKeyCount()
    {
        keyCountText.text = playerInventory.Count.ToString();
    }
}
```

1. Attach the `KeyCountDisplay` script to a UI GameObject.
2. Assign the `PlayerInventory` collection to the `playerInventory` field in the `KeyCountDisplay` script.
3. Assign a `TMP_Text` component to the `keyCountText` field in the `KeyCountDisplay` script.

Step 5: Testing the Setup

Now, let's test the setup to ensure everything works as expected.

1. Run the game in the Unity Editor.

2. When the player collides with a key, the key is added to the inventory, and the key count display updates.
3. When the player collides with a door, the script checks for the required key in the inventory and unlocks the door if the key is present.

Conclusion

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. The PlayerInventory collection acts as a central repository for keys, allowing the Key, Door, and KeyCountDisplay scripts to interact with it independently.