Scriptable Object Architecture Pattern

User Guide

OBVIOUS
Game

SOAP:
Scriptable Object Architecture Pattern

Table of Contents

Introduction

Thank you for downloading Soap 😊. The goal of Soap is to help you developing your project. It aims to be easy to understand and to use, and therefore makes it useful to both junior and senior game developers.

Soap leverages the power of **scriptable objects**, which are native objects from the Unity Engine. Scriptable objects are usually used for data storage. However, because they are assets, they can be referenced by other assets and are accessible at runtime and in editor.

Soap is based and build upon the GDC talk of Ryan Hipple: https://www.youtube.com/watch?v=raQ3iHhE_Kk&ab_channel=Unity.

It was developed mostly for and while making Hypercasual games. Because of the nature of these games, the core strength and focus of Soap is **speed** and **simplicity** rather than scalability and complexity. Nevertheless, it can be used in more complex games to solve certain problems, particularly dependencies.

The package contains **6 examples scenes** to show how to use this framework. Each scene has its **own specific documentation** (in the same folder as the scene). I hope you give it a try 😊.

Why use Soap?

Solve dependencies:

Avoid coupling by using a reference to a common Scriptable object as a bridge or using events.

This is particularly true in the Hypercasual market for several reasons:

- Many features are enabled/disabled through independent AB Tests. Therefore, we want to avoid noisy code and hardcoding our features into the core of our game. Having them "hooked" with an independent architecture makes it easy to add or remove them.
- Because you make a lot of games quickly, you want to reuse features as much as you can. Therefore, having features as "drag and drop" can be a huge time saver over the long run for things that generally don't change too much across games.

Speed:

- Avoid creating new classes for simple behaviors
- Gain access to key variables directly with a simple reference
- Modify variable from the editor or code
- Have persistent data across scene or play sessions already handled
- Be able to debug visually and have the game react in real time

Efficiency / Clarity

- Subscribe to what you need and have each class handle itself
- Avoid useless managers
- Reduce code complexity (by avoiding spaghetti code)

Scriptable Variables

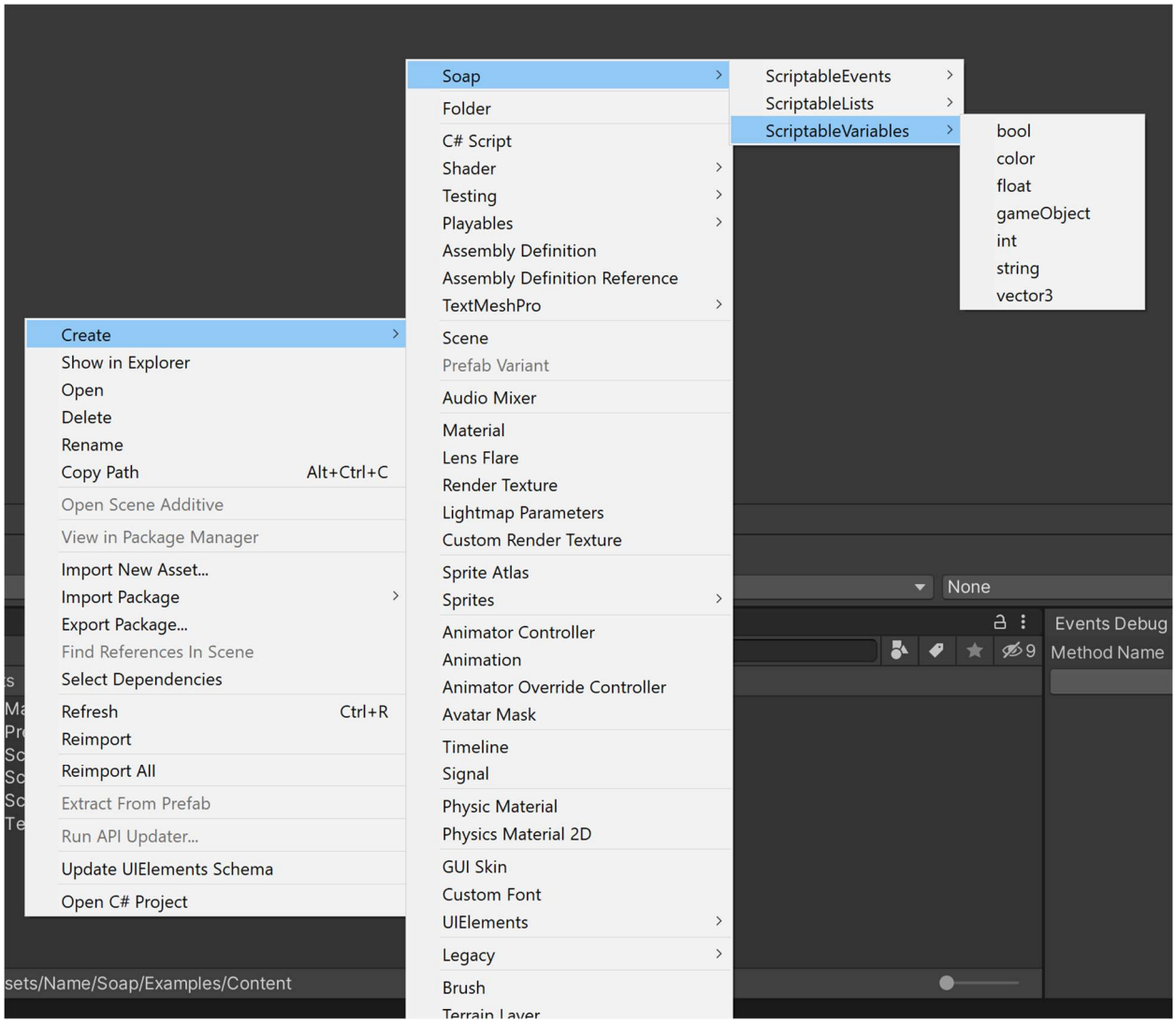
A scriptable variable is a scriptable object of a certain type containing a value. Here is an example of a FloatVariable:

Let's go through its properties:

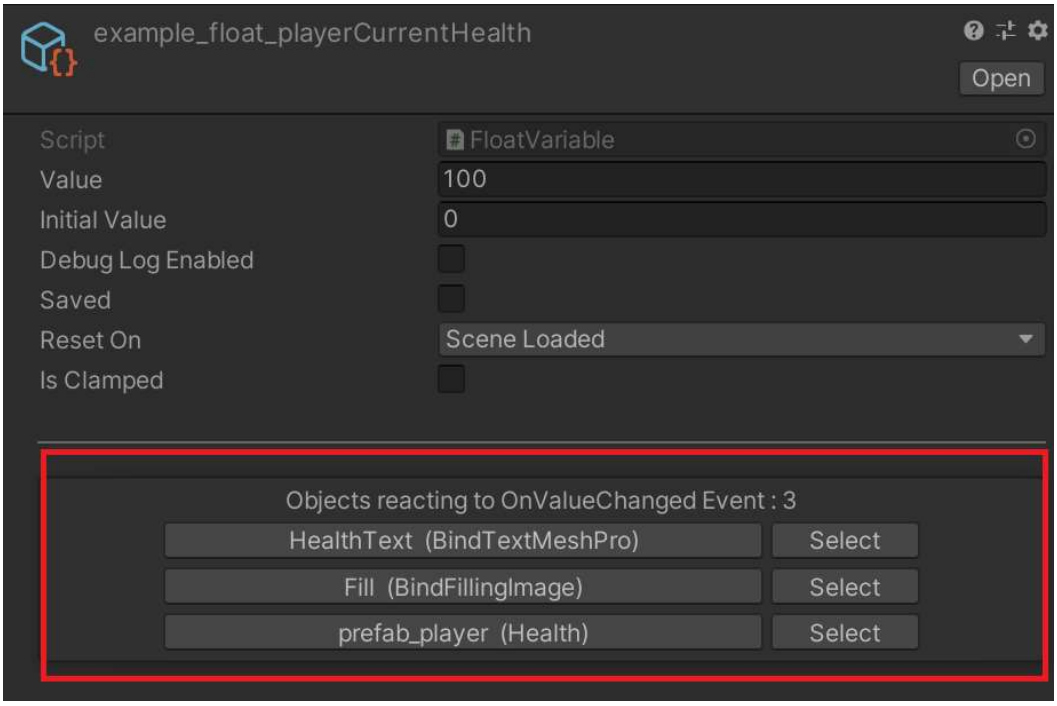
- **Value**: the current value of the variable. Can be changed in inspector, at runtime, by code or in unity events. Changing the value will trigger an event "OnValueChanged" that can be subscribed to by code. See examples for practical usage.
- **Initial Value**: the value it gets reset to.
- **Debug Log Enabled**: if true, will log in the console whenever this value is changed.
- **Saved**: If true, the value of the variable will be saved to Player Prefs when it changes. (More information in the 5_Save_Documentation).
- **Reset On:** when is this variable reset (or loaded, if saved)?
    - *Scene Loaded* : whenever a scene is loaded. Ignores Additive scene loading. Use this if you want the variable to be reset if it is only used in a single scene.
    - *Application Start*: reset once when the game starts. Useful if you want changes made to the variable to persist across scenes. (Stats of a character between levels for example)

To create a new variable, simply right click in the project window and find the scriptable variable you want:

Create/Soap/ScriptableVariables/



When you are in **play mode**, the objects (and their component) that have registered to the **OnValueChanged** Event of a scriptable variable are visible in the inspector.



As you can see in the screenshot above, 3 objects are registered to this variable.

If we inspect the first element that react to this variable, we can find the GameObject named “HealthText” in the hierarchy and, more specially, its component “BindTextMeshPro”.

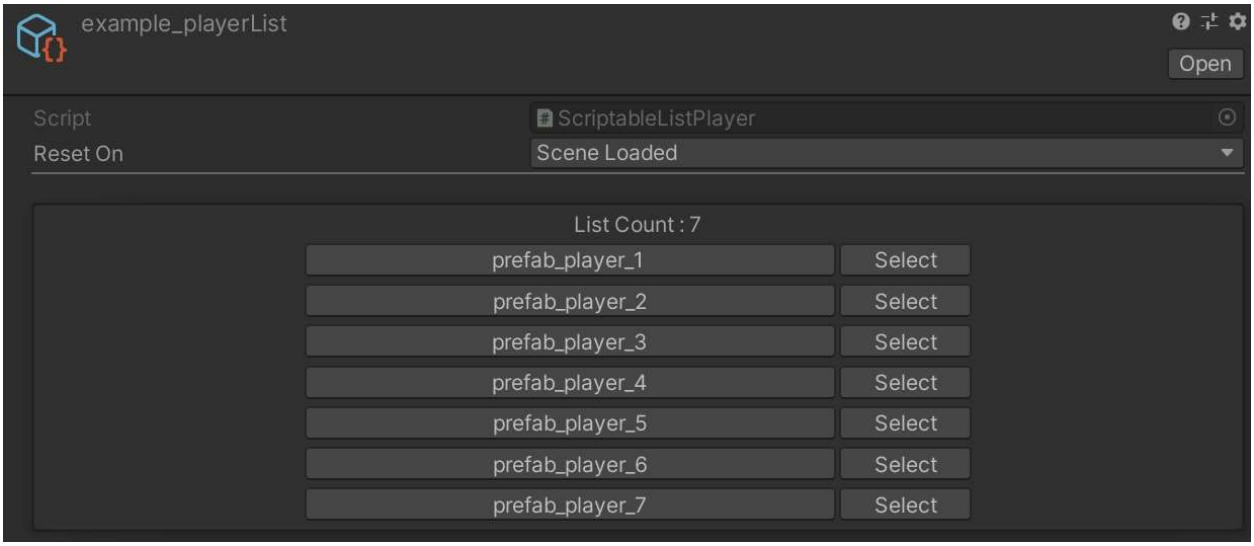
By clicking on the first button, we ping the object in the hierarchy. By clicking on the “Select” button, it will select the object in the hierarchy.

This visual debug can be useful to keep track of what is reacting to the changes of your variable.

Please check the example scene 1_ScriptableVariables to understand concrete use of scriptable variables.

Scriptable Lists

Lists are useful to solve dependencies by avoiding the need to have a "manager" that holds that list.



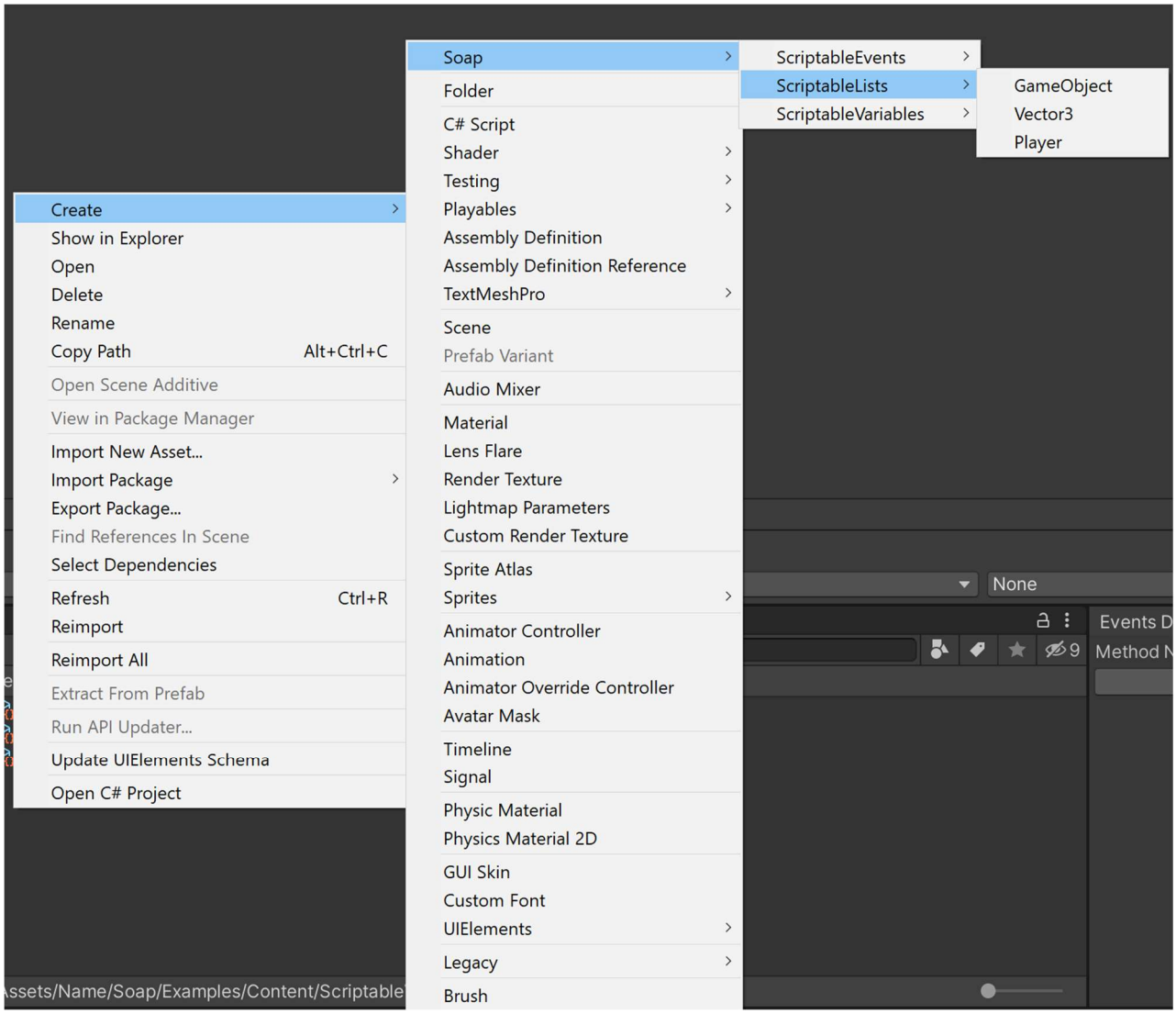
Let's go through its properties:

- **Reset On:** when is this list cleared?
  - *Scene Loaded:* whenever a scene is loaded. Ignores Additive scene loading.
  - *Application Start:* reset once when the game starts.

- **List content**: in play mode, you can see all the elements that populate the list. By clicking on the first button (with the name of the object), it pings the object in the hierarchy. By clicking on the "Select" button, it will select the object in the hierarchy.

To create a new list, simply right click in the project window and find the scriptable list:

Create/Soap/ScriptableLists/

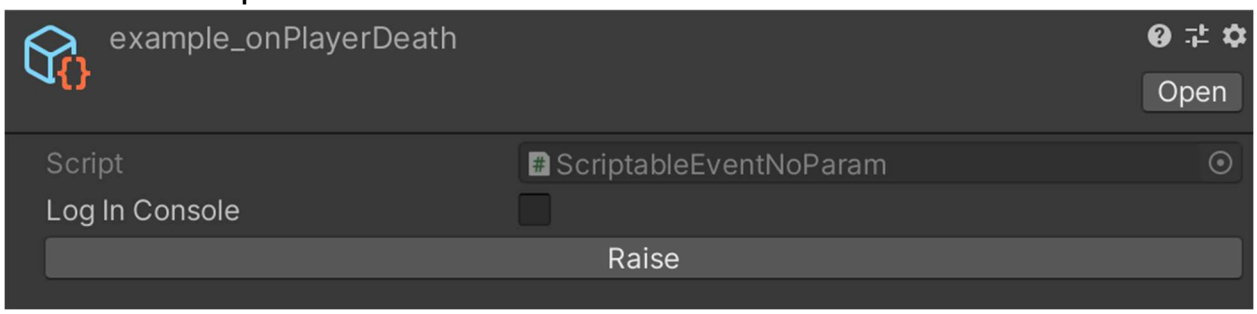


Please check the example scene 3_Lists to understand concrete use of scriptable lists.

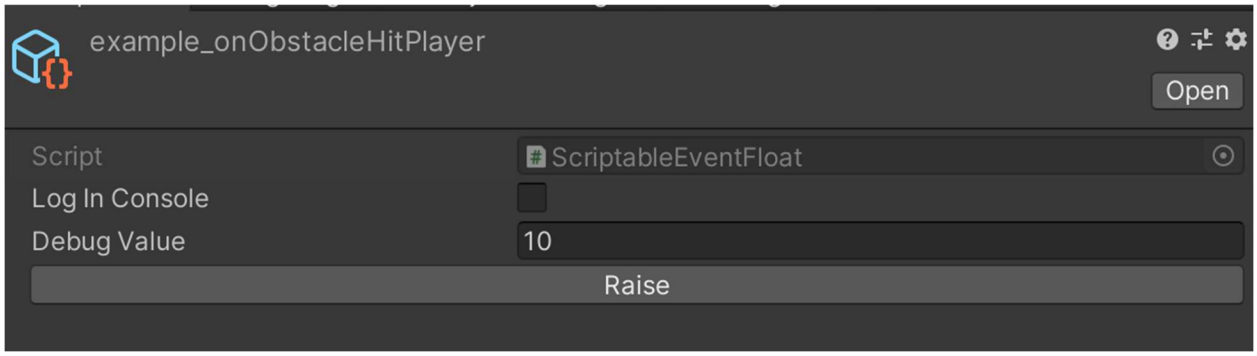
Scriptable Events

There are two types of scriptable events:

- Event without parameters



- Event with parameter (int, bool, float, string, Vector2, Vector3, Color)

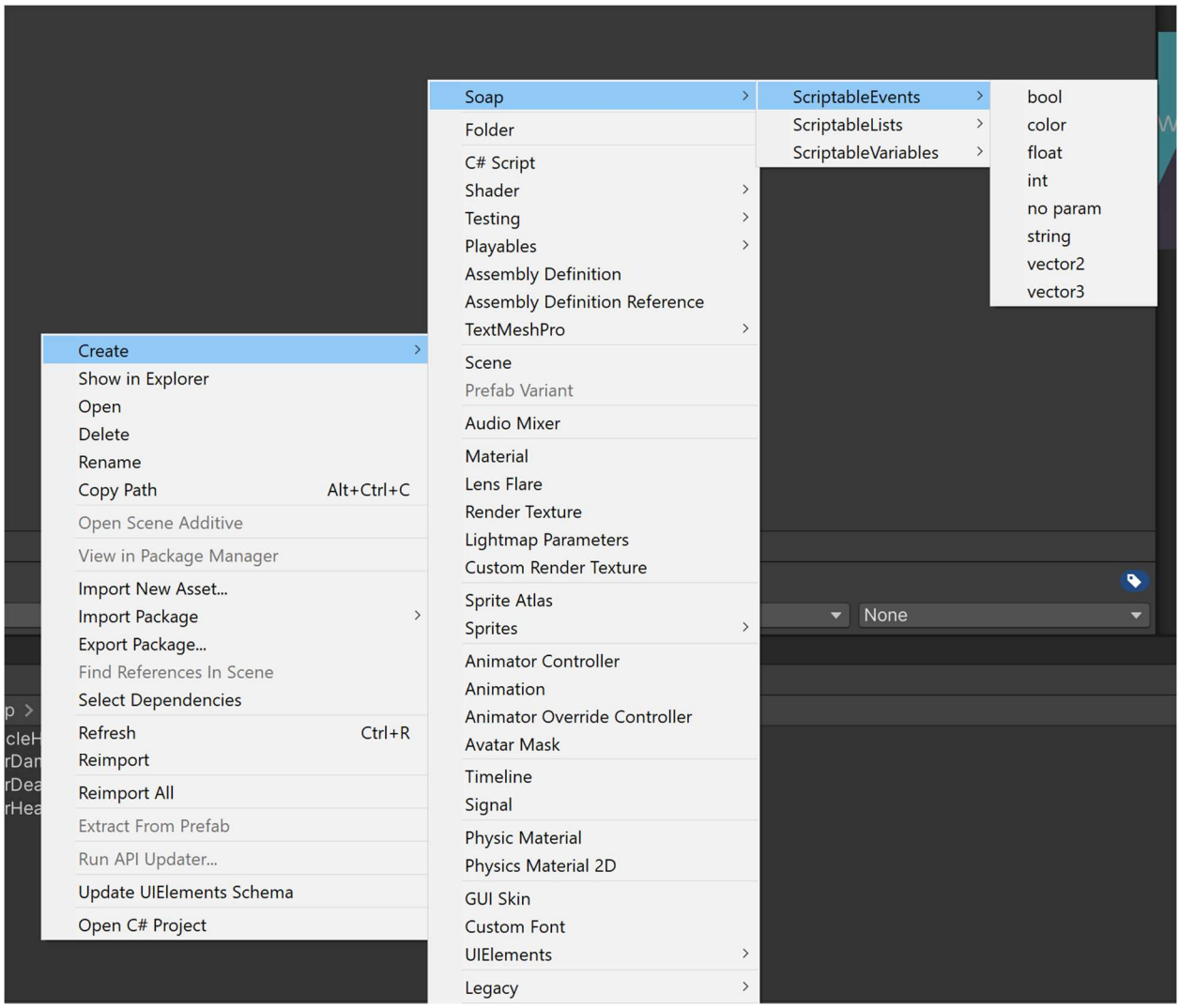


Events can be triggered by code, unity actions or the inspector (via the raise button).

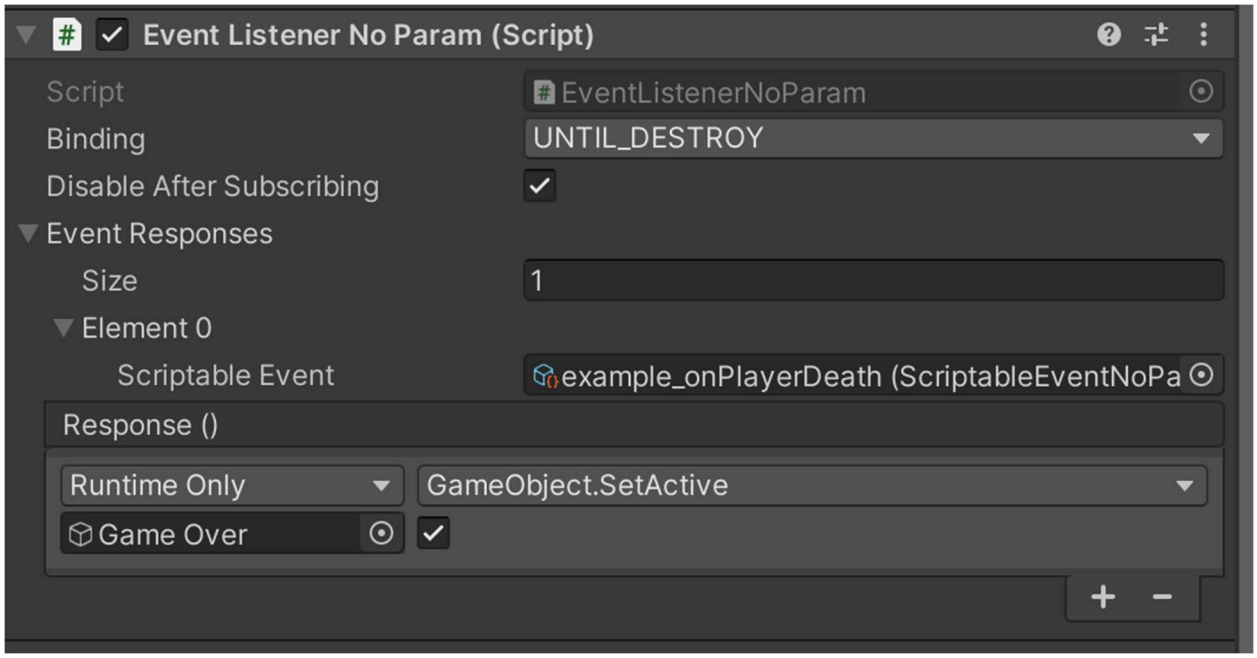
Raising events in the inspector can be useful to quickly debug your game.

To create an event:

Create/Soap/ScriptableEvents/



To listen to these events when they are fired, you need to attach an **Event Listener** component (of the same type) to your GameObjects.



- **Binding**:
  - Until_Destroy: will register in Awake() and unsubscribe OnDestroy().
  - Until_Disable: will register OnEnable() and unsubscribe OnDisable().

- **Disable after subscribing**: if true, will deactivate the GameObject after registering to the event. Useful for UI elements.

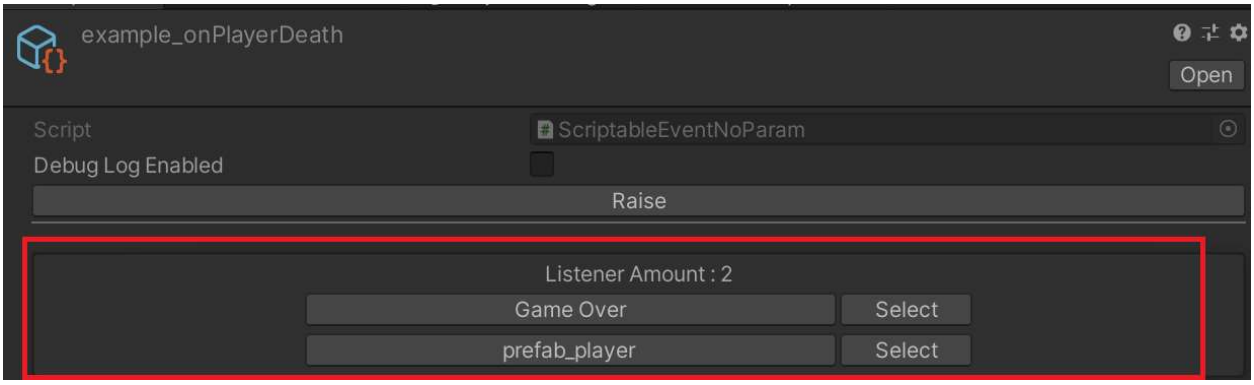
- **Event responses:** here you can add multiple events and trigger things with unity events when they are fired.

You can also register to events directly from code like this:

```
[SerializeField] private FloatVariable _currentHealth = null;  example_float_playerCurrentHealth.asset
[SerializeField] private ScriptableEventInt _onObstacleHitPlayer = null;  Unchanged

Event function  Pierre *
private void Start()
{
    _onObstacleHitPlayer.OnRaised += TakeDamage;
}

Event function  Pierre *
private void OnDestroy()
{
    _onObstacleHitPlayer.OnRaised -= TakeDamage;
}

2 usages  new *
private void TakeDamage(int amount)
{
    _currentHealth.Add(-amount);
}
```

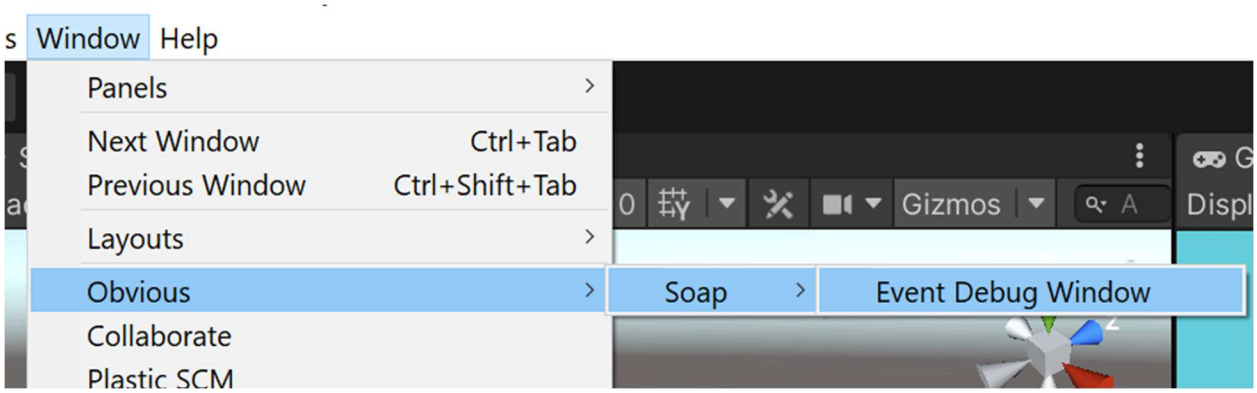
Scriptable events also have their own play mode custom inspector.

When you hit play, you can see all objects that have registered to that event:



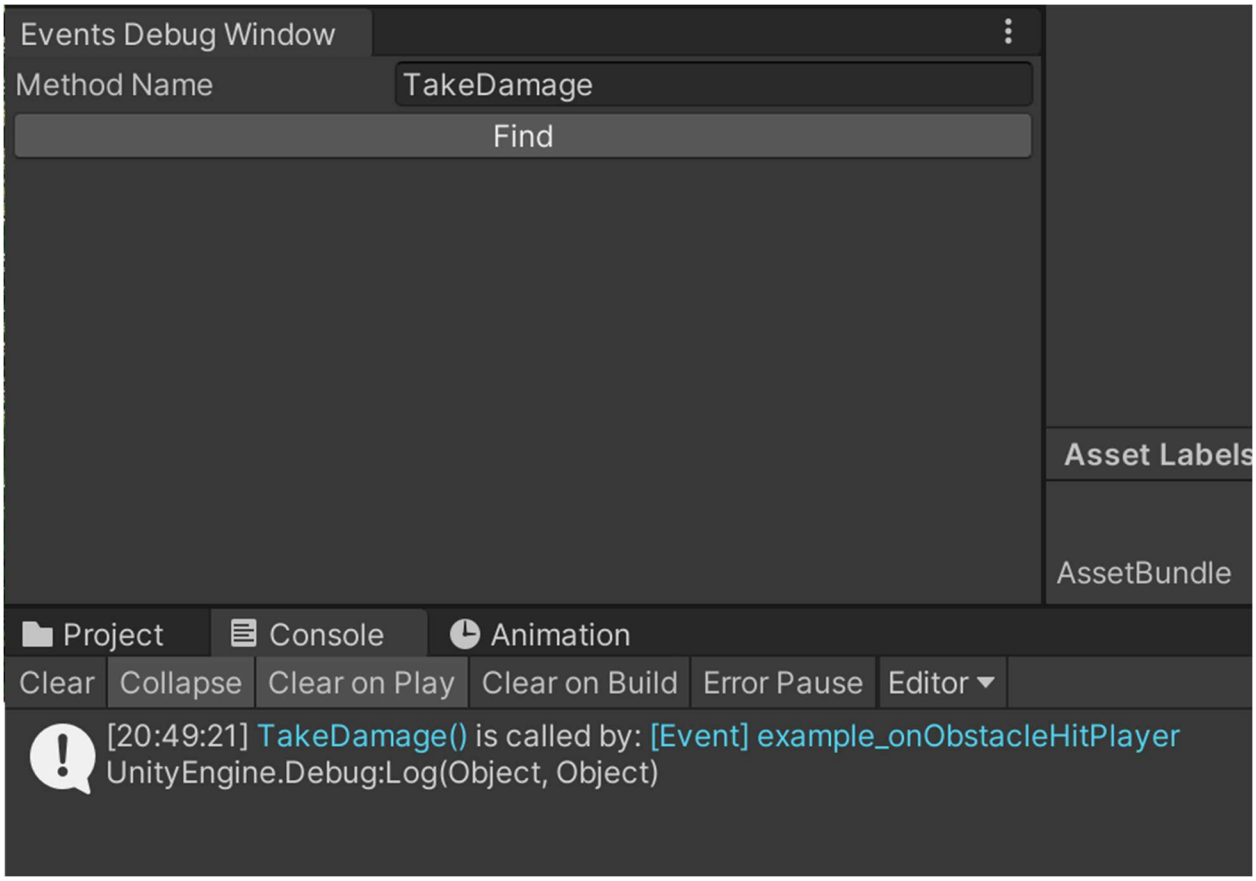
If you have a method in one of your scripts, but you don't remember which event calls that method, you can use the **Event Debug Window.** You can access it through the menu:

Window/Obvious/Soap/Event Debug Window



By typing the name of the method in the event debug window, it will search all objects in your scene and find which event calls that method.

Note: it is case sensitive.



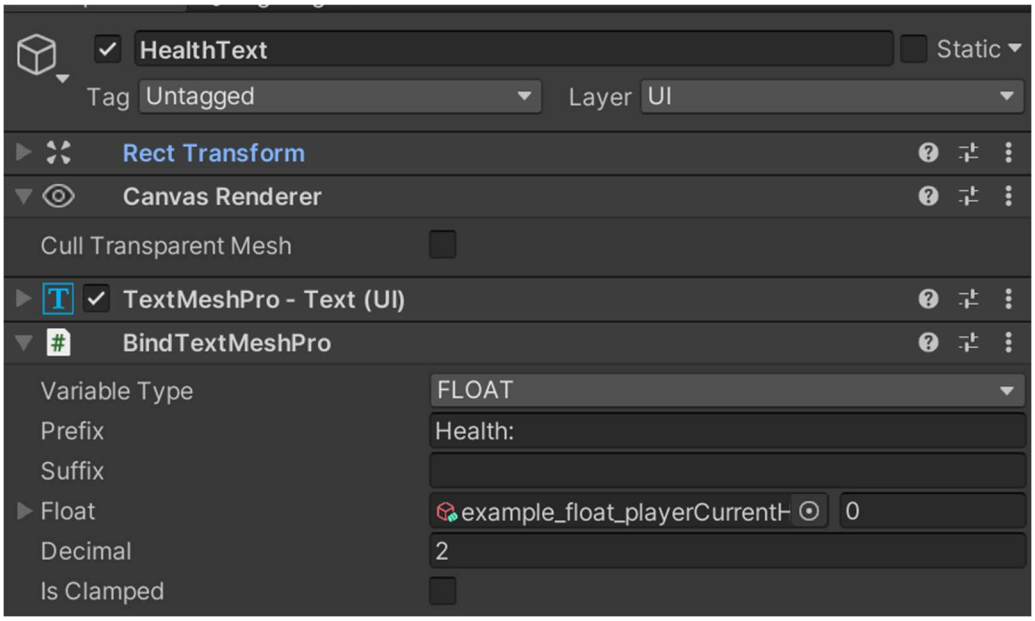
Once the method if found (or not), a message is logged in the console and if you click on it, it will ping the corresponding object in the hierarchy. You can also click on the debug message and check the stack trace for additional information.

Please check the example scene to understand concrete use of scriptable events and how to raise them.

Bindings

Bindings are components that you attach to GameObjects to bind them to variable and execute a simple behavior when that variables changes. They were created to save time and to not have to create a script for small things like changing the color or text / image and others.

Please check out the Example scene “2_Bindings_Examples_Scene” and its documentation, as it is the best way to understand how to use Bindings and how useful they are.



How to extend this SDK?

You can extend this SDK by creating your own scriptable variables, lists and events.

Simply inherit from these classes and replace T with your type:

- ScriptableVariable<T>
- ScriptableList<T>
- ScriptableEvent<T>
- EventListener<T>
- VariableReferences<V,T>

Check the classes in the Examples folder to see few examples.

You can also create new “Binding” components. I have made a few, but you can be creative and make new bindings that will be useful in your games.

Examples of extending the SDK can be found in the various example’s scripts.

Content

ScriptableVariables
BoolVariable
IntVariable
FloatVariable
StringVariable
Vector2Variable
Vector3Variable
ColorVariable
GameObjectVariable
PlayerVariable (Example)

VariableReferences
BoolReference
IntReference
FloatReference
StringReference
Vector2Reference
Vector3Reference
ColorReference

ScriptableEvents
ScriptableEventNoParam
ScriptableEventInt
ScriptableEventFloat
ScriptableEventString
ScriptableEventVector2
ScriptableEventvector3
ScriptableEventColor
ScriptableEventGameObject

ScriptableLists
ScriptableListVector3
ScriptableListGameObject
ScriptableListPlayer (Example)

Bindings
BindText
BindGraphicColor
BindRendererColor
BindToggle
BindSlider
BindComparisonToUnityEvent
BindFillingImage
CacheComponent

And a few editor scripts.

Contact

Any questions, suggestions, or feedback?

Feel free to reach me at: obviousgame.contact@gmail.com

Or join the discord server: https://discord.gg/CVhCNDbxF5

If you have 5 minutes, please leave a review on the asset store, as it really helps to get feedback from developers for us to improve or create new packages.