Contact: *tobias.froihofer@gmx.at*

GODS – Globally Observable Data Structures

GODS offers a range of Scriptable Objects and fields designed to decouple systems in your project. It provides globally accessible structures that are easy to subscribe to and respond to when their values change.

Content

GODS: Simplifying Development and Enhancing Flexibility

GODS (Global Object Data System) streamlines project development by reducing scene dependencies and improving modularity. Instead of relying on specific scene objects, we can use globally accessible Observables (events, variables, lists and hash sets), making testing and management easier. Observable variables, lists and hash sets are smart data objects which can raise events once their stored values change, allowing listeners to react to these changes quickly and efficiently.

Key Benefits:

1. **Improve code decoupling**: Reduce dependencies between different parts of the system, making it easier to modify and maintain code.
2. **Enhance modularity**: Create more self-contained components that can be easily added, removed, or modified without affecting other parts of the system.
3. **Increase code reusability**: Develop generic components and systems that can be used across multiple projects or different parts of the same project.
4. **Support data-driven design**: Enable more game logic and behavior to be controlled through data rather than hard-coded logic, increasing flexibility and reducing the need for code changes.
5. **Improve scalability**: Make it easier to add new features or expand existing systems without major code refactoring.
6. **Streamline debugging**: Provide a centralized system for monitoring and debugging global state changes, making it easier to identify and fix issues.
7. **Enhance workflow efficiency**: Improve the process of reacting to value changes across the project, reducing development time and effort.
8. **Facilitate rapid prototyping**: Allow quick setup and testing of new ideas without extensive coding or scene setup, speeding up the iteration process.
9. **Enhance collaboration**: Enable designers and programmers to work more independently by reducing direct code dependencies and providing shared data structures.
10. **Increase design flexibility**: Reduce coding requirements for design changes, allowing designers to iterate more freely without constant programmer involvement.
11. **Simplify testing**: Make it easier to test new functionality by reducing scene dependencies and providing globally accessible variables.
12. **Free up developers**: Reduce the amount of repetitive or boilerplate code, allowing developers to focus on more complex or creative tasks.
13. **Improve version control efficiency**: Reduce merge conflicts by centralizing data that multiple team members might need to modify, streamlining the collaborative development process.
14. **Facilitate multi-user save management**: Enable easy creation and management of flexible save slots for multiple users or profiles, enhancing the game's ability to support various play styles, multiple playthroughs, or shared device scenarios.

Practical Example: PlayerHealth Variable

Consider a PlayerHealth variable in GODS. When the player takes damage, the health value changes, triggering an event. This single change can affect multiple systems:

1. UI Components:
   - Health display flashes red to indicate damage
   - Numerical health value updates automatically
   - Health bar updates automatically
2. Audio System:
   - Plays a damage sound effect
3. Visual Effects:
   - Triggers screen shake effect
4. Enemy AI:
   - Adapts behavior based on player's health (e.g., less aggressive when player health is low)

Traditional methods would require setting up object references (e.g. FindObjectOfType, Serialized Fields, …) or custom event systems. GODS simplifies this process by using globally referenceable structures like events, variables, lists and hash sets resulting in:

- Cleaner code
- Enhanced flexibility
- Improved testability

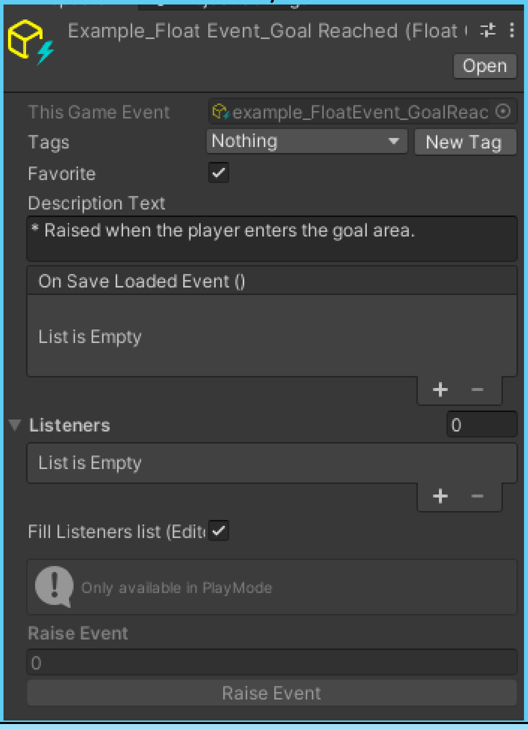
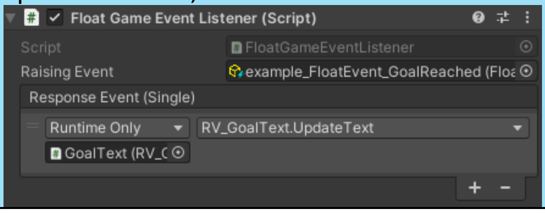
This approach eliminates the need for complex object referencing or custom event systems, making your project more modular and easier to maintain.

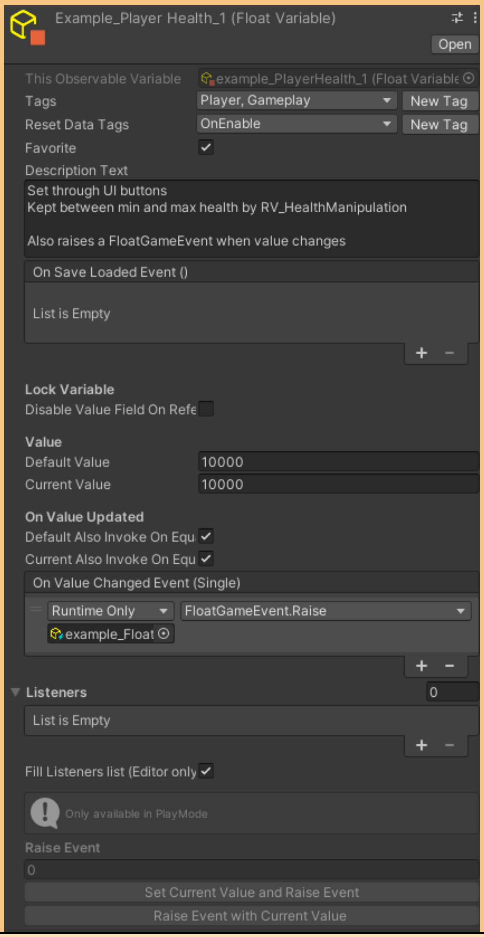
Available Data Structures

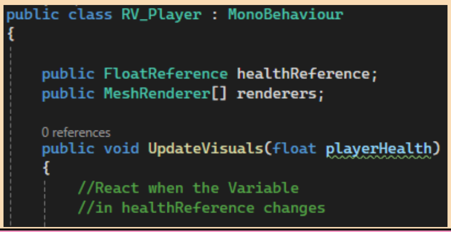
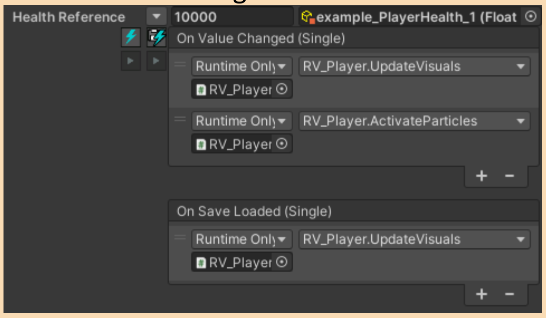
These are the structures managed by GODS:

Scriptable Object (data holder)	**Component/Field (used to interact with the data)**	**Examples**
Observable Game Event <T>	Game Event Listeners <T>	• FloatGameEvent – FloatGameEventListener • TransformGameEvent - TransformGameEventListener
Observable Variable <T>	Referenced Observable Variable <T>	• FloatVariable – FloatReference • StringVariable – StringReference
Observable List <T>	Referenced Observable List <T>	• GameObjectList- GameObjectListReference • TransformList - TransformListReference
Observable HashSet <T>	Referenced Observable HashSet <T>	• GameObjectHashSet – GameObjectHashSetReference • TransformHashSet – TransformHashSetReference

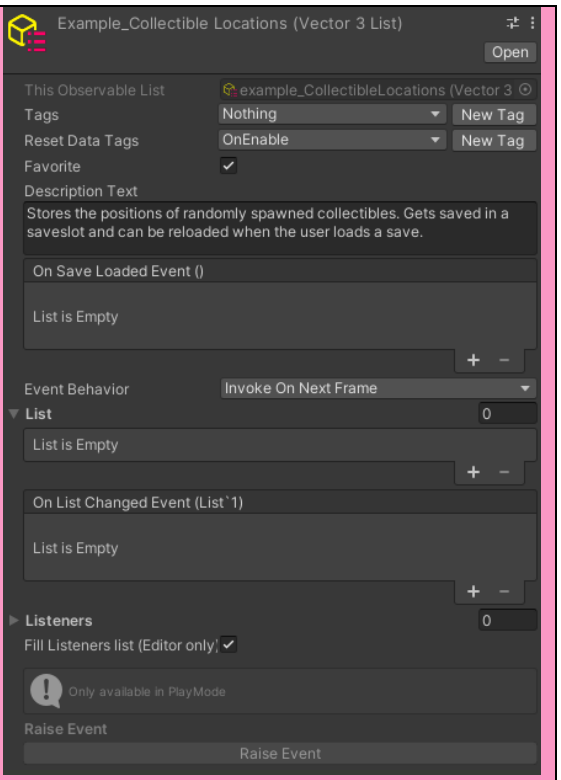
Here in more detail:

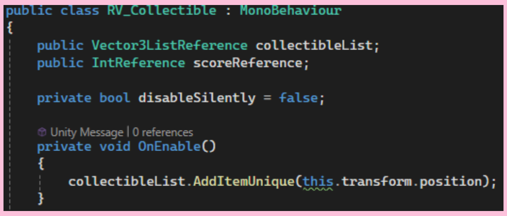
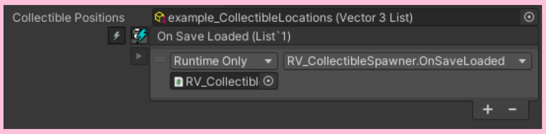
Observable Type	**Design Goal**	**Average Developer Engagement**
Observable Game Events <T> Type: Scriptable Object	Reduce your load on coding with these objects. Instead of writing delegates or UnityEvents, you can create an Observable Game Event. Other objects like Game Event Listeners can subscribe to this event. When the event gets raised, listeners can react with their own event and methods.	**Medium** – Create these assets when needed, reference them in your project and call the Raise method to notify the listeners.  Example_Float Event_Goal Reached (Float Open This Game Event: example_FloatEvent_GoalReac Tags: Nothing · New Tag Favorite: ✓ Description Text * Raised when the player enters the goal area. On Save Loaded Event () List is Empty Listeners: 0 List is Empty Fill Listeners list (Edit: ✓ Only available in PlayMode Raise Event 0 Raise Event
Game Event Listeners <T> Type: Mono Behaviour	These components will automatically subscribe to the referenced Game Event. When the event is raised, this component will react with a response event.	**High** – easily react to events and call suitable methods. E.g. display UI, play particle effects, spawn enemies, …  ✓ Float Game Event Listener (Script) Script: FloatGameEventListener Raising Event: example_FloatEvent_GoalReached (Floa Response Event (Single) Runtime Only · RV_GoalText.UpdateText GoalText (RV_(

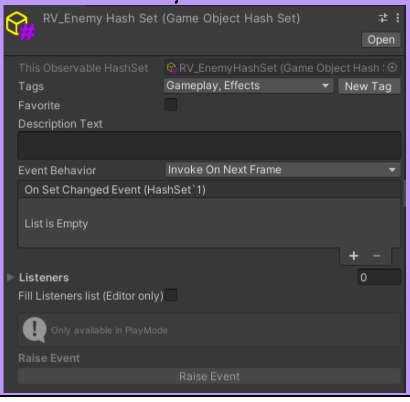
Observable Variable <T> Type: Scriptable Object	Somewhat comparable to **Observable Game Events**, but also **stores a value**. When the value changes, the event gets raised. Listeners (e.g. Observable Variable References) can automatically react to the new value. You can also change the value 'silently', and thereby not raise the event. **Access the stored value from everywhere.** As the Observable Variable is a Scriptable Object, it does not require a specific scene to exist and can keep its value across scene changes. Use **"Reset Data Tags"** to **automatically reset data** at specific points (defaults include: OnEnable, OnSceneLoaded, OnSceneLoadedAdditively, OnSceneUnloaded) This allows to create more modular systems which are not dependent on another. For example: parts of your UI or other systems don't require another object to be active in the scene. (e.g. Player Health 🡐🡒 UI Health)	**High** – Create these assets when needed, react to changes through **Observable Variable References**  Example_Player Health_1 (Float Variable) Open This Observable Variable: example_PlayerHealth_1 (Float Variable Tags: Player, Gameplay — New Tag Reset Data Tags: OnEnable — New Tag Favorite: ✓ Description Text Set through UI buttons Kept between min and max health by RV_HealthManipulation Also raises a FloatGameEvent when value changes On Save Loaded Event () List is Empty **Lock Variable** Disable Value Field On Refe: ☐ **Value** Default Value: 10000 Current Value: 10000 **On Value Updated** Default Also Invoke On Equ: ✓ Current Also Invoke On Equ: ✓ On Value Changed Event (Single) Runtime Only — FloatGameEvent.Raise example_Float **Listeners**: 0 List is Empty Fill Listeners list (Editor only: ✓ Only available in PlayMode Raise Event 0 Set Current Value and Raise Event Raise Event with Current Value

Observable Variable Reference <T>
 	
Type:
Serializable class
	This is the main way to read/write to Observable Variables and to react to value changes on the referenced variable.
	
	The current value can be accessed through **referenceName.Value**. Compare to the previous value with **referenceName.PreviousValue**
	
	
	`public class RV_Player : MonoBehaviour`
	`{`
	`    public FloatReference healthReference;`
	`    public MeshRenderer[] renderers;`
	`    0 references`
	`    public void UpdateVisuals(float playerHealth)`
	`    {`
	`        //React when the Variable`
	`        //in healthReference changes`
		High – You will likely mainly use these fields to interact with Observable Variables and to react to value changes.
		
		Health Reference: 10000 — example_PlayerHealth_1 (Float
		On Value Changed (Single)
		Runtime Only — RV_Player.UpdateVisuals
		RV_Player
		Runtime Only — RV_Player.ActivateParticles
		RV_Player
		On Save Loaded (Single)
		Runtime Only — RV_Player.UpdateVisuals
		RV_Player
		In this example, we react to value changes by activating green/red particles depending on the new health value.

Observable List <T> Type: Scriptable Object	Similar to Observable Variable these objects store data as a list and can invoke an onListChangedEvent when the list changes (when items get added/removed). Listeners can automatically react to the changes this way.	**Low** – Similar to Observable Variables, you can create these assets when needed, but the main interaction will likely be through **Observable List References**.

	For when to react there are several options available. For example you can immediately react when an item gets added to the list. But if you regularly add a bunch of items at once, it is recommended to invoke the event after a short time to save performance. Otherwise you would invoke the event for each item which gets added, causing a lot of overhead. You can also add items 'silently', meaning you add them to the list without invoking any events.	 Example_Collectible Locations (Vector 3 List) Open This Observable List: example_CollectibleLocations (Vector 3 Tags: Nothing New Tag Reset Data Tags: OnEnable New Tag Favorite: ✓ Description Text: Stores the positions of randomly spawned collectibles. Gets saved in a saveslot and can be reloaded when the user loads a save. On Save Loaded Event () List is Empty Event Behavior: Invoke On Next Frame List: 0 List is Empty On List Changed Event (List`1) List is Empty Listeners: 0 Fill Listeners list (Editor only): ✓ Only available in PlayMode Raise Event Raise Event

Observable List Reference <T>
 	
Type:
Serializable class
	This is the main way to manipulate the list of an Observable List. Add/remove items and react to value changes on the referenced list.
	
	The list can be accessed through **referencedListName.List**
	
	
	`public class RV_Collectible : MonoBehaviour`
	`{`
	`    public Vector3ListReference collectibleList;`
	`    public IntReference scoreReference;`
	
	`    private bool disableSilently = false;`
	
	`    Unity Message | 0 references`
	`    private void OnEnable()`
	`    {`
	`        collectibleList.AddItemUnique(this.transform.position);`
	`    }`
		Medium - You will likely mainly use these fields to interact with Observable Lists.
		
		Collectible Positions: example_CollectibleLocations (Vector 3 List)
		On Save Loaded (List`1)
		Runtime Only
		RV_CollectibleSpawner.OnSaveLoaded
		RV_Collectibl
		
		In this example we store the positions of randomly spawned objects in an Observable List.
		When a SaveSlot gets loaded, the Observable List values get updated with the saved data. Then the CollectibleSpawner clears the current collectible objects and spawns objects on the positions of the Observable List.

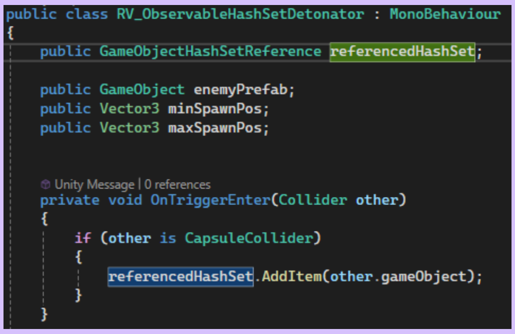
Observable Hash Set <T> Type: Scriptable Object	This is very similar to an Observable List <T> but this guarantees that each element only exists once in the set.	**Low** – Create when needed to manage unique elements in a list easily and to access them from anywhere.  RV_Enemy Hash Set (Game Object Hash Set) Open This Observable HashSet: RV_EnemyHashSet (Game Object Hash : Tags: Gameplay, Effects New Tag Favorite Description Text Event Behavior: Invoke On Next Frame On Set Changed Event (HashSet`1) List is Empty Listeners: 0 Fill Listeners list (Editor only) Only available in PlayMode Raise Event Raise Event

Observable Hash Set Reference <T>

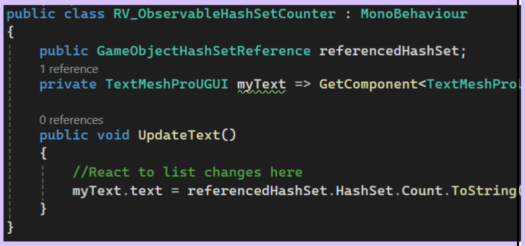
Type:
Serializable class

This is the main way to manipulate the sets of a Observable Hash Set. Add/remove elements and react to value changes on the referenced hash set.

The hash set can be accessed through **referencedHashSetName.HashSet**

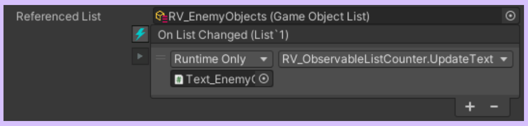


```
public class RV_ObservableHashSetDetonator : MonoBehaviour
{
    public GameObjectHashSetReference referencedHashSet;

    public GameObject enemyPrefab;
    public Vector3 minSpawnPos;
    public Vector3 maxSpawnPos;


    Unity Message | 0 references
    private void OnTriggerEnter(Collider other)
    {
        if (other is CapsuleCollider)
        {
            referencedHashSet.AddItem(other.gameObject);
        }
    }
```



```
public class RV_ObservableHashSetCounter : MonoBehaviour
{
    public GameObjectHashSetReference referencedHashSet;
    1 reference
    private TextMeshProUGUI myText => GetComponent<TextMeshPro

    0 references
    public void UpdateText()
    {
        //React to list changes here
        myText.text = referencedHashSet.HashSet.Count.ToString(
    }
}
```

Medium - You will likely mainly use these fields to interact with Observable Hash Sets.



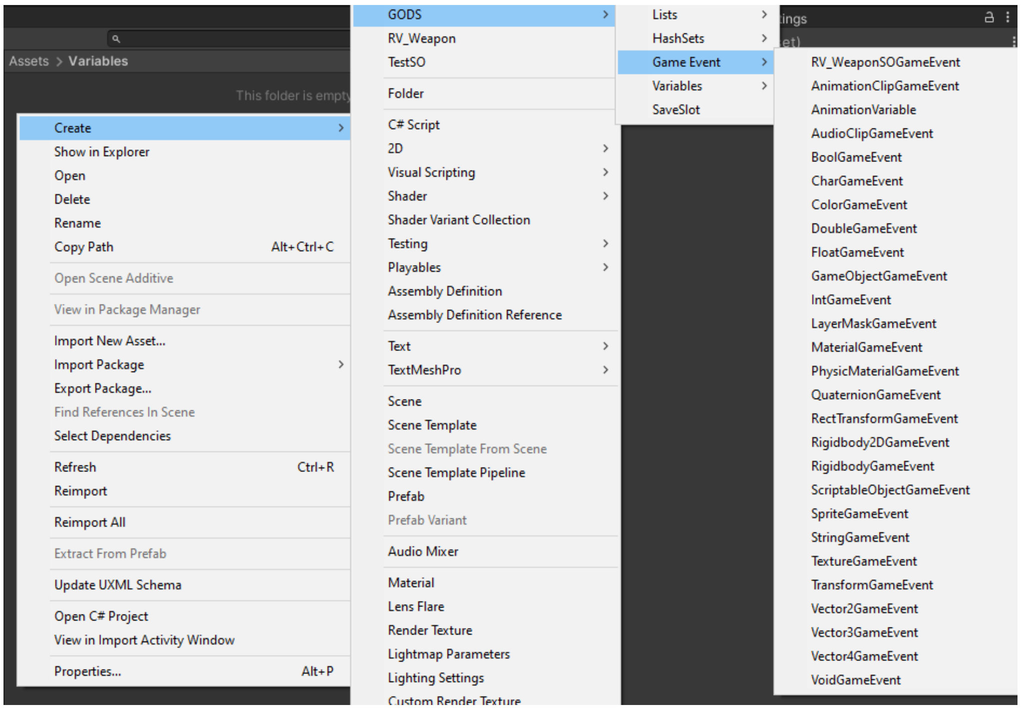
Referenced List — RV_EnemyObjects (Game Object List)
On List Changed (List`1)
Runtime Only — RV_ObservableListCounter.UpdateText
Text_Enemy(

In this example, we count objects within an area. When an object enters/exits a sphere we add/remove the object to/from the Observable Hash Set.
A UI text then displays the current count of objects within the Observable Hash Set.

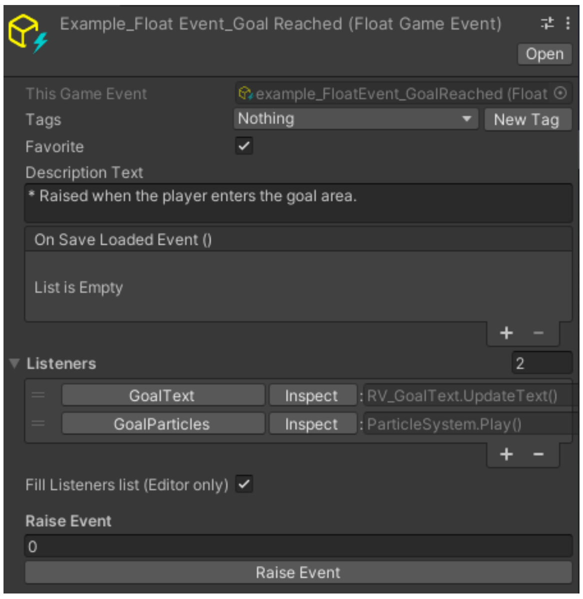
Observable Game Events and Game Event Listeners

Observable Game Events are Scriptable Objects which can be used similar to a UnityEvent. Game Event Listeners can register as listeners and once the Game Event is raised - by calling the Raise method on the Scriptable Object - the Game Event Listeners can respond by invoking their own UnityEvents.

Create an Observable Game Event via the assets menu “**Assets/Create/GODS/Game Event/{Type}GameEvent**”



This will create a Game Event of the chosen type. In this example it is a FloatGameEvent, which means it has a float as argument.

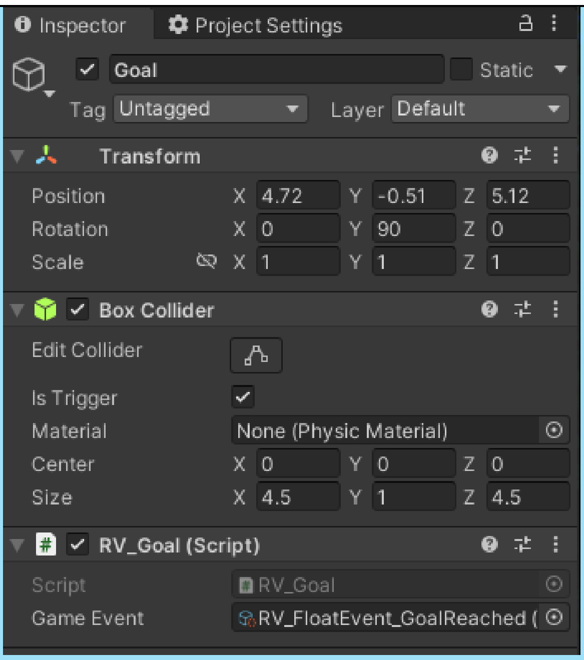
Select Tag	You can optionally assign a tag for searching and filtering purposes.	
Favorite	Tick this field for searching and filtering purposes.	
Description Text	Like Variables, a Game Event also has a Description Text which you can use to make notes about its functionality.	
On Save Loaded Event	Objects can listen for this event and react accordingly when a save slot is loaded. Note: you can not drag and drop in an object which exists in a scene as this is a Scriptable Object living outside any scene. Instead if needed, reference this object and subscribe via code.	
Listeners	At runtime this can display a list of objects which subscribed to the On List Changed Event. The list displays the name of the object which listens for the event and it also displays the methods which will be called as a response to the event.	
Fill Listeners list (Editor only)	This is a global setting for all Game Events/Observable Variables/Observable Lists/…	

Example_Float Event_Goal Reached (Float Game Event)
Open
This Game Event: example_FloatEvent_GoalReached (Float
Tags: Nothing | New Tag
Favorite: ✓
Description Text
Raised when the player enters the goal area.
On Save Loaded Event ()
List is Empty
Listeners: 2
GoalText | Inspect | RV_GoalText.UpdateText()
GoalParticles | Inspect | ParticleSystem.Play()
Fill Listeners list (Editor only) ✓
Raise Event
0
Raise Event

	Tick this to fill the above list at runtime for debugging purposes. This will only work in the editor. **Disabling this option will improve performance.**	
Raise Event button	At runtime this button can be used to trigger the event for testing purposes.	

Example: Goal Reached Event

In this simple example we have a goal area. When the player enters it we want to raise an event and display how much time was needed to reach the goal.

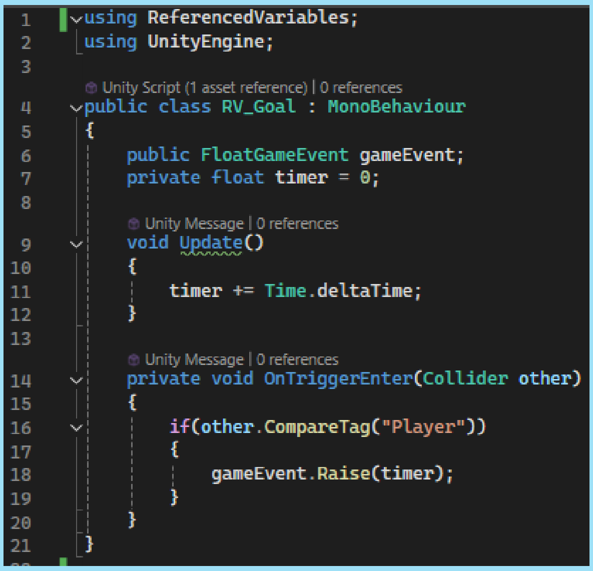

Inspector | Project Settings

Goal ✓ Static
Tag Untagged | Layer Default

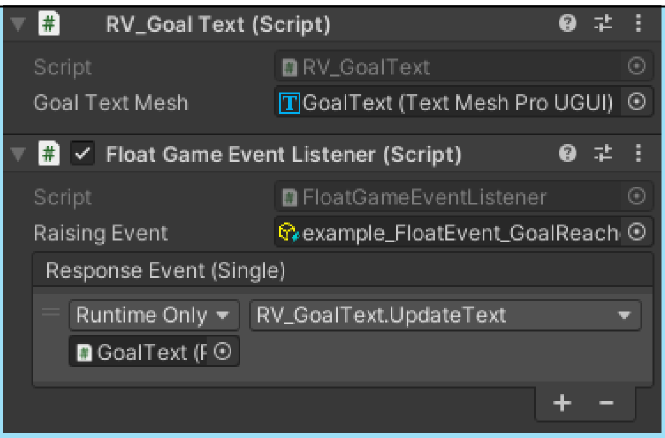
Transform
Position X 4.72 Y -0.51 Z 5.12
Rotation X 0 Y 90 Z 0
Scale X 1 Y 1 Z 1

Box Collider ✓
Edit Collider
Is Trigger ✓
Material None (Physic Material)
Center X 0 Y 0 Z 0
Size X 4.5 Y 1 Z 4.5

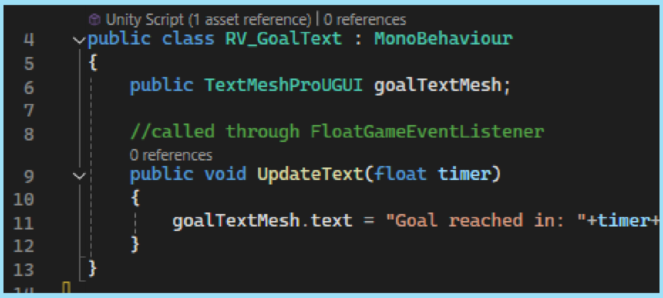
RV_Goal (Script) ✓
Script RV_Goal
Game Event RV_FloatEvent_GoalReached (


```
using ReferencedVariables;
using UnityEngine;

Unity Script (1 asset reference) | 0 references
public class RV_Goal : MonoBehaviour
{
    public FloatGameEvent gameEvent;
    private float timer = 0;

    Unity Message | 0 references
    void Update()
    {
        timer += Time.deltaTime;
    }

    Unity Message | 0 references
    private void OnTriggerEnter(Collider other)
    {
        if(other.CompareTag("Player"))
        {
            gameEvent.Raise(timer);
        }
    }
}
```

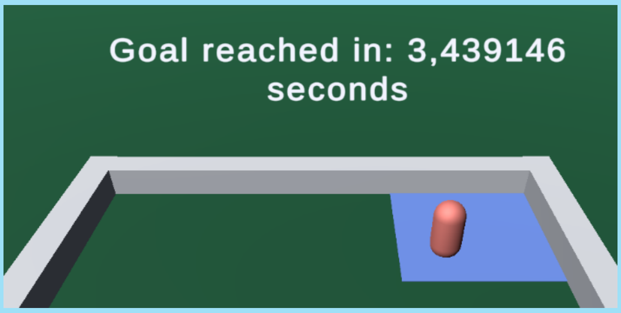
When the player triggers the goal area, we Raise the referenced event with the timer as argument.


RV_Goal Text (Script)
Script RV_GoalText
Goal Text Mesh GoalText (Text Mesh Pro UGUI)

Float Game Event Listener (Script) ✓
Script FloatGameEventListener
Raising Event example_FloatEvent_GoalReach
Response Event (Single)
Runtime Only | RV_GoalText.UpdateText
GoalText (F
+ −


```
Unity Script (1 asset reference) | 0 references
public class RV_GoalText : MonoBehaviour
{
    public TextMeshProUGUI goalTextMesh;

    //called through FloatGameEventListener
    0 references
    public void UpdateText(float timer)
    {
        goalTextMesh.text = "Goal reached in: "+timer+
    }
}
```

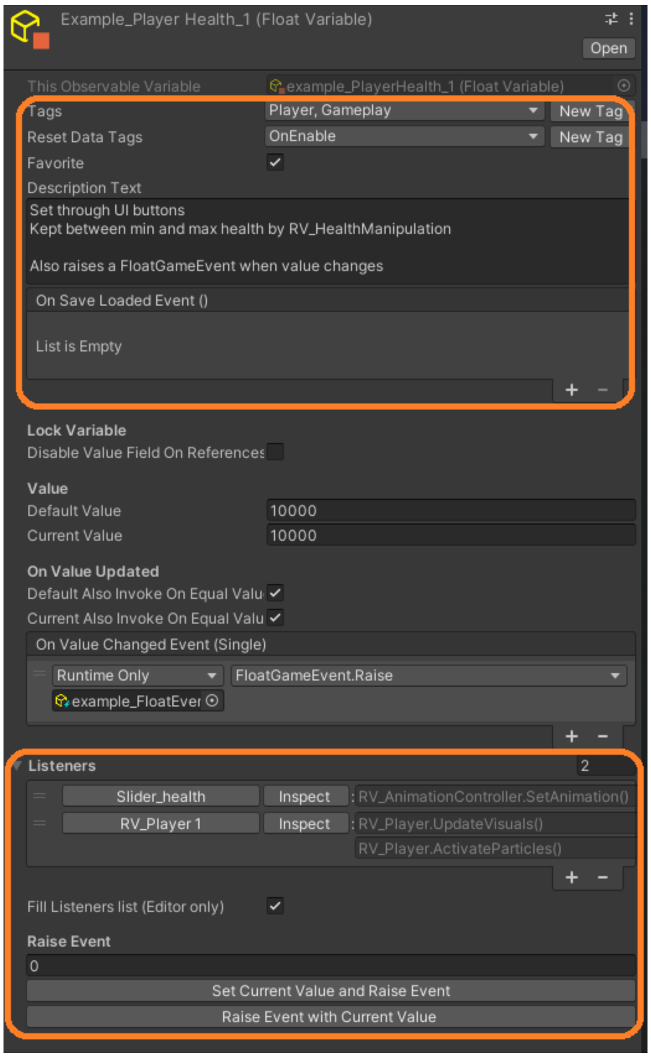
The Goal Text object has a Float Game Event Listener attached which references the same Game Event as the RV_Goal script.
Once the event is raised, the Response Event invokes the Update Text method and thereby setting the goal text in the scene.



Shared Fields in Observable Asset Inspectors

Here are some fields which Observable-Variables, -Lists, and -HashSets all have in common. Some are used for management (e.g. filtering options), others are settings for gameplay behavior.

Tags	You can optionally assign tags for searching and filtering purposes.
Reset Data Tags	Assign tags to tell the Variable when to set the Current Value to Default Value. OnEnable will be assigned by default to reset the value once the game starts/once the Variable activates.
Favorite	Tick this field for searching and filtering purposes.
Description Text	All Variables have a Description Text. It provides no game functionality, but can help in keeping a better overview of the functionality of a Variable. Type in from where the Value gets changed for example.
On Save Loaded Event	Objects can listen for this event and react accordingly when a save slot is loaded. Note: you can not drag and drop in an object which exists in a scene as this is a Scriptable Object living outside any scene. Instead if needed, reference this object and subscribe via code.
Listeners	At runtime this will display a list of objects which subscribed to the On List Changed Event. The list displays the name of the object which listens for the event and it also displays the methods which will be called as a response to the event.
Fill Listeners list (Editor only)	This is a global setting for all Game Events/Observable Variables/Observable Lists/… Tick this to fill the above list at runtime for debugging purposes. This will only work in the editor. **Disabling this option will improve performance.**
Raise Event	At runtime, you can use test buttons to raise the value changed method. This way you can see what would happen if the event was raised by code.


Example_Player Health_1 (Float Variable)

Open

This Observable Variable: example_PlayerHealth_1 (Float Variable)

Tags: Player, Gameplay — New Tag

Reset Data Tags: OnEnable — New Tag

Favorite: ✓

Description Text

Set through UI buttons
Kept between min and max health by RV_HealthManipulation

Also raises a FloatGameEvent when value changes

On Save Loaded Event ()

List is Empty

Lock Variable

Disable Value Field On References: ☐

Value

Default Value: 10000

Current Value: 10000

On Value Updated

Default Also Invoke On Equal Valu: ✓

Current Also Invoke On Equal Valu: ✓

On Value Changed Event (Single)

Runtime Only — FloatGameEvent.Raise

example_FloatEver

Listeners 2

Slider_health	Inspect	RV_AnimationController.SetAnimation()
RV_Player 1	Inspect	RV_Player.UpdateVisuals()
		RV_Player.ActivateParticles()

Fill Listeners list (Editor only): ✓

Raise Event

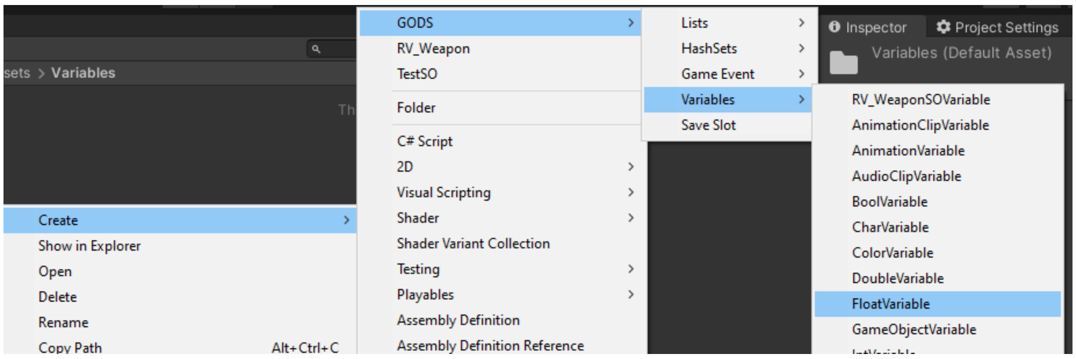
0

Set Current Value and Raise Event

Raise Event with Current Value

Observable Variables

Observable Variables are **Scriptable Objects** of a generic type, holding a Value which gets reset to a default value when **OnEnable** (or another selected event) is called. This means we can have float-, bool-, Vector2-, Vector3-, Transform-, GameObject-Variables and many many more (also custom types). To create a Variable use "**Assets/Create/GODS/Variables/**" and select the type of variable you need.

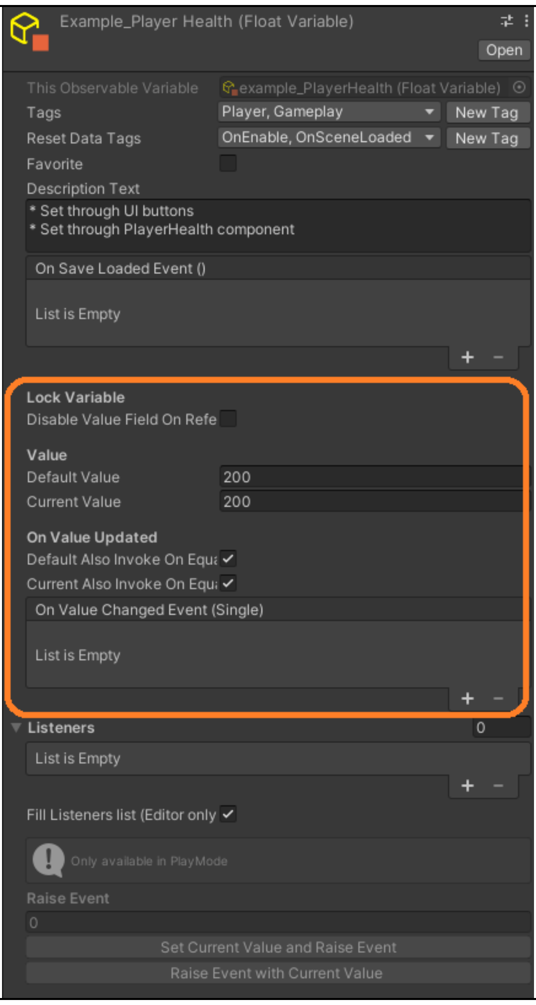


How more types can be created will be discussed in the next chapter.

Let's take a (resettable) FloatVariable as an example. It is a Scriptable Object which can be referenced by your scripts. Once you change the **Current Value** with '**nameOfReference.Variable = aNewValue**' it will fire an event to all active objects which reference this FloatVariable.

Here is the inspector of an Observable Variable described in more detail.

Shared Fields in Observable Asset Inspectors	
Default Value	This is the default value for the next field and will be applied when the OnEnable or Reset method is called.
Current Value	The value used at runtime.
Default Also Invoke On Equal Value	This is the default value for the next field.
Also Invoke On Equal Value	Controls if the OnValueChanged event is also called if the newly applied value to "Current Value" is equal to the current one.
On Value Changed Event	This event gets raised when the "Current Value" gets updated.



Example_Player Health (Float Variable)

Open

This Observable Variable: example_PlayerHealth (Float Variable)

Tags: Player, Gameplay — New Tag

Reset Data Tags: OnEnable, OnSceneLoaded — New Tag

Favorite

Description Text

* Set through UI buttons
* Set through PlayerHealth component

On Save Loaded Event ()

List is Empty

Lock Variable

Disable Value Field On Refe

Value

Default Value: 200

Current Value: 200

On Value Updated

Default Also Invoke On Equa ✓

Current Also Invoke On Equa ✓

On Value Changed Event (Single)

List is Empty

Listeners: 0

List is Empty

Fill Listeners list (Editor only ✓

Only available in PlayMode

Raise Event

0

Set Current Value and Raise Event

Raise Event with Current Value

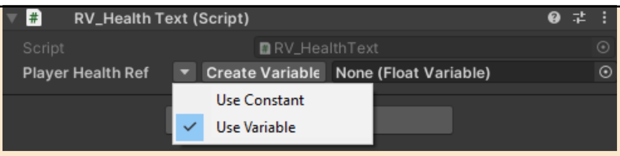
Observable Variable References

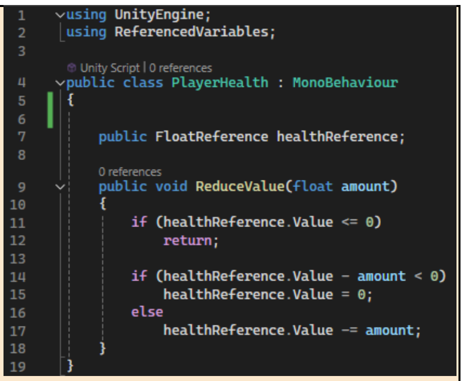
Variable References are designed to be used within your scripts to reference your Observable Variables in order to:

- Read the current value for easy global access
- Change the Current Value of the Variable
- Listen to when the Current Value changed
- React with a response event to the new value.
- Respond when a save slot got loaded

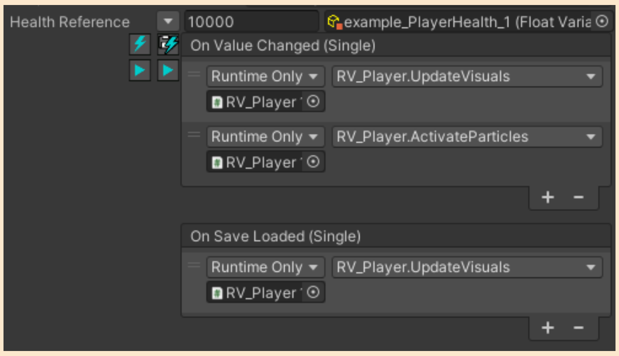
Variable References derive from the BaseReference class and can act in one of two modes. Here is an example of a **FloatReference** (derives from ObservableVariableReferenceBase).

Use Variable


RV_Health Text (Script)
Script: RV_HealthText
Player Health Ref: Create Variable | None (Float Variable)
Use Constant
✓ Use Variable



```
using UnityEngine;
using ReferencedVariables;

public class PlayerHealth : MonoBehaviour
{

    public FloatReference healthReference;

    public void ReduceValue(float amount)
    {
        if (healthReference.Value <= 0)
            return;

        if (healthReference.Value - amount < 0)
            healthReference.Value = 0;
        else
            healthReference.Value -= amount;
    }
}
```


Health Reference: 10000 | example_PlayerHealth_1 (Float Varia
On Value Changed (Single)
Runtime Only | RV_Player.UpdateVisuals | RV_Player
Runtime Only | RV_Player.ActivateParticles | RV_Player
On Save Loaded (Single)
Runtime Only | RV_Player.UpdateVisuals | RV_Player

This is the usual mode to use. After dragging in a (Float-) Variable, the **Default Value** of the Variable will be displayed when the Unity Editor is **in Edit Mode**.

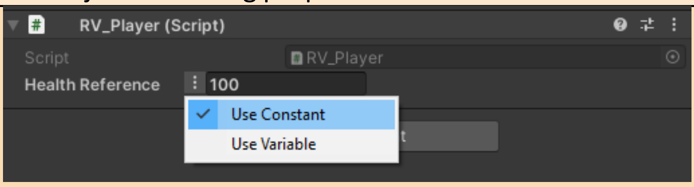
Once we switch to **Play Mode**, the **Current Value** will be displayed instead to make debugging easier.

We can listen for the **On Value Changed event** by activating the **lightning bolt symbol**. This event gets raised once we update the Current Value of the Observable Variable.

We can listen for the **On Save Loaded event** by activating the **save disc + lightning bolt symbol**. This event gets raised after a save slot which holds the referenced Variable is loaded. More about Save Slots: ***SaveSlots: Saving/Loading Observables***

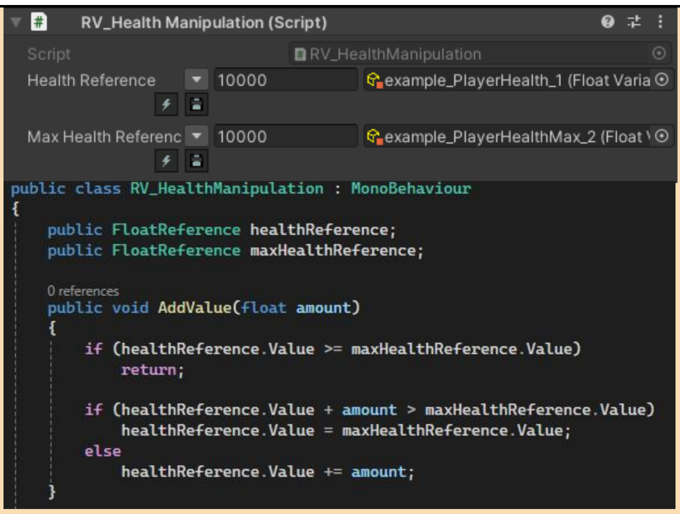
At runtime you can use the buttons below (with the arrow symbol) to invoke the events on this object for testing purposes.

Use Constant


RV_Player (Script)
Script: RV_Player
Health Reference: 100
✓ Use Constant
Use Variable

In this mode the FloatReference acts just like a float. This can be useful if you decide at some point that a float is enough and you don't need to reference the value and don't need to react to the events of the Variable.

Example: Player Health and Health Text



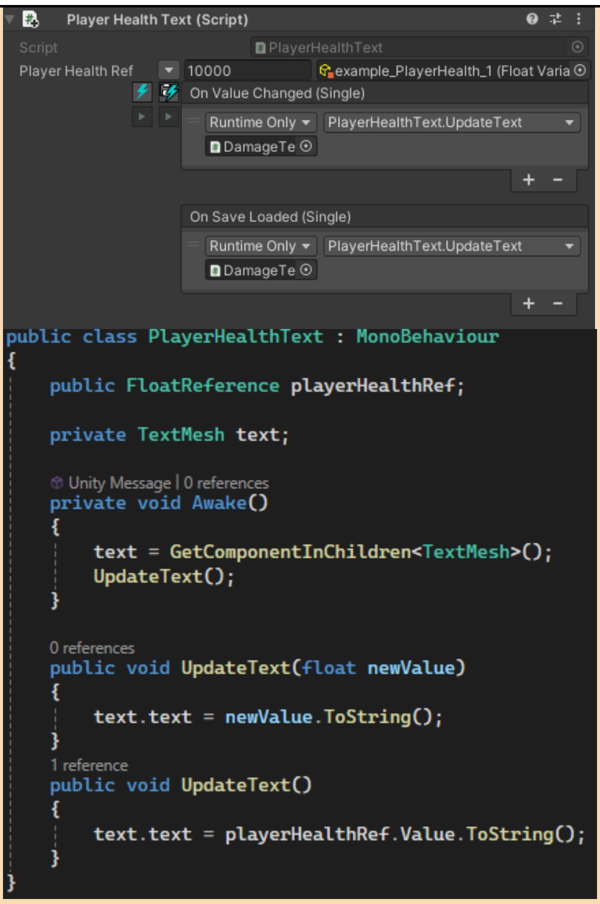
RV_Health Manipulation (Script)

Script: RV_HealthManipulation
Health Reference: 10000 — example_PlayerHealth_1 (Float Varia
Max Health Referenc: 10000 — example_PlayerHealthMax_2 (Float \

```
public class RV_HealthManipulation : MonoBehaviour
{
    public FloatReference healthReference;
    public FloatReference maxHealthReference;

    0 references
    public void AddValue(float amount)
    {
        if (healthReference.Value >= maxHealthReference.Value)
            return;

        if (healthReference.Value + amount > maxHealthReference.Value)
            healthReference.Value = maxHealthReference.Value;
        else
            healthReference.Value += amount;
    }
```

We set the Current Value in the HealthManipulation component by calling:
healthReference.Value = amount;

Setting the Current Value this way will invoke the OnValueChanged event of the referenced Variable.



Player Health Text (Script)

Script: PlayerHealthText
Player Health Ref: 10000 — example_PlayerHealth_1 (Float Varia
On Value Changed (Single): Runtime Only — PlayerHealthText.UpdateText — DamageTe
On Save Loaded (Single): Runtime Only — PlayerHealthText.UpdateText — DamageTe

```
public class PlayerHealthText : MonoBehaviour
{
    public FloatReference playerHealthRef;

    private TextMesh text;

    Unity Message | 0 references
    private void Awake()
    {
        text = GetComponentInChildren<TextMesh>();
        UpdateText();
    }

    0 references
    public void UpdateText(float newValue)
    {
        text.text = newValue.ToString();
    }
    1 reference
    public void UpdateText()
    {
        text.text = playerHealthRef.Value.ToString();
    }
}
```

Player Health Text references the same FloatVariable as Player Health. Once the event triggers, it calls the UpdateText method which sets the text to the Current Value.

Note that in this example we have defined two UpdateText methods. We can use either one of them. The difference is that the first method uses the dynamic argument of the OnValueChanged method to set the text while the second method reads the value from the referenced Variable directly.

CAUTION! Be careful when reacting to the OnChangedEvent. If you set the Value again in a response method, you will again trigger the event and might end up in an infinite loop in which the event gets triggered over and over. If you need to set the value again in a response event, consider using **healthReference.SetValueSilent(value)**. This sets the value without triggering the event.

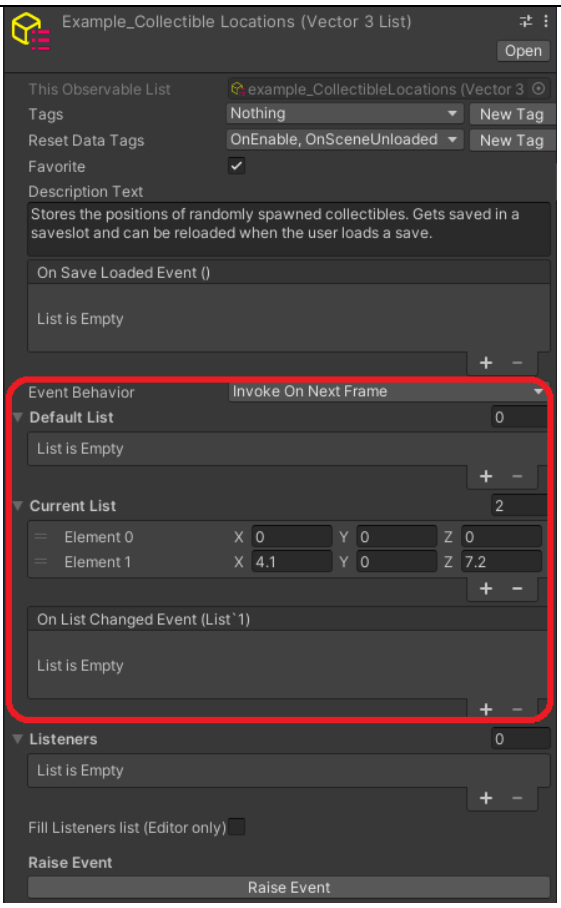
HINT. If you need to set the value every frame, setting it with **healthReference.SetValueSilent(value)** instead of **healthReference.Value = value** will improve performance as you will not trigger the event every time.

Observable List

Observable Lists are somewhat similar to Observable Variables, but instead of storing a single value they store a list of values. A GameObjectList for example can be referenced by some object to store all objects in a given area. This can be useful to easily apply operations on only these objects and to quickly react when the list changes.

We can create a new List of some type under **"Assets/Create/GODS/Lists/…"**

An Observable List asset has some similar fields as an Observable Variable, but there are some differences:

Shared Fields in Observable Asset Inspectors		
Event Behavior	This controls when the OnListChangedEvent gets invoked when the list of items changes. It has these options: • Invoke On Next Frame • Invoke After X Seconds • Invoke Immediately Since it is possible to add many items to a list in a single frame, it can make sense to invoke the event after a while and not immediately. If we select Invoke Immediately and we add 20 objects to the list, the event will be invoked 20 times, which might not be what we want. To save performance we can select Invoke On The Next Frame for example.	
Seconds To Wait For Invoking	If we select Invoke After X Seconds as the Event Behavior, we can choose how long to wait until we actually invoke the event and inform listeners that the list changed	
Default List	The default list. When the ResetData method is called, the List will clear and add the items in the Default List.	
Current List	This is the list with all our items used at runtime. The list gets cleared at certain points (depending on the Reset Data Tags above)	
On List Changed Event	The event to which other objects can subscribe to.	

Example_Collectible Locations (Vector 3 List)
Open

This Observable List	example_CollectibleLocations (Vector 3
Tags	Nothing
Reset Data Tags	OnEnable, OnSceneUnloaded
Favorite	✓

Description Text
Stores the positions of randomly spawned collectibles. Gets saved in a saveslot and can be reloaded when the user loads a save.

On Save Loaded Event ()
List is Empty

Event Behavior: Invoke On Next Frame
Default List: 0
List is Empty
Current List: 2

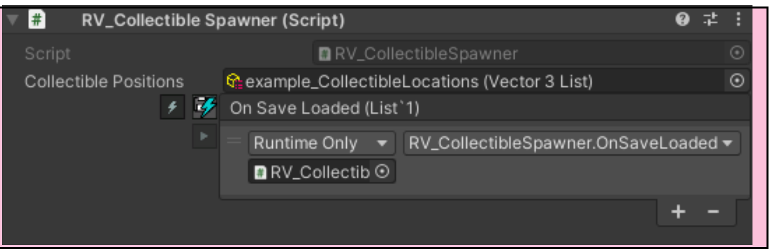
Element 0	X 0	Y 0	Z 0
Element 1	X 4.1	Y 0	Z 7.2

On List Changed Event (List`1)
List is Empty

Listeners: 0
List is Empty
Fill Listeners list (Editor only)
Raise Event

Observable List Reference

Observable List References are fields you can add to your MonoBehaviours. They share many similarities with ***Observable Variable References***. The difference is they reference an Observable List and have no "Use Constant" mode. Instead an Observable List is required as a reference.

The **On List Changed** event gets raised once we update the list of the referenced Observable List. The **On Save Loaded** event gets raised when a SaveSlot loads a save which includes the	

RV_Collectible Spawner (Script)

Script	RV_CollectibleSpawner
Collectible Positions	example_CollectibleLocations (Vector 3 List)

On Save Loaded (List`1)
Runtime Only — RV_CollectibleSpawner.OnSaveLoaded
RV_Collectib

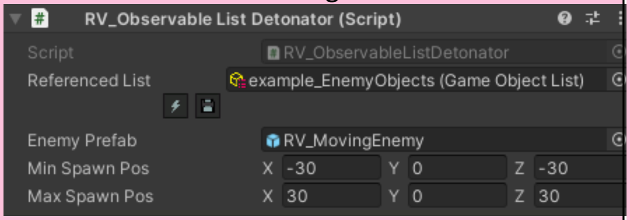
referenced Observable List. More about Save Slots: ***SaveSlots: Saving/Loading Observables***

Just like Observable Variable References you can use the lightning bolt and the lightning bolt + save disc symbol to subscribe to events.

Example: Tracking Objects in an Area

The Observable List Detonator component references an Observable List. The object has a Sphere Collider attached and tracks objects entering and leaving the sphere. In the OnTriggerEnter and OnTriggerExit methods, it updates the referenced list. By calling AddItem and RemoveItem the OnListChanged event of the Observable List gets invoked. Other objects which are listening for this event can respond to the changes.

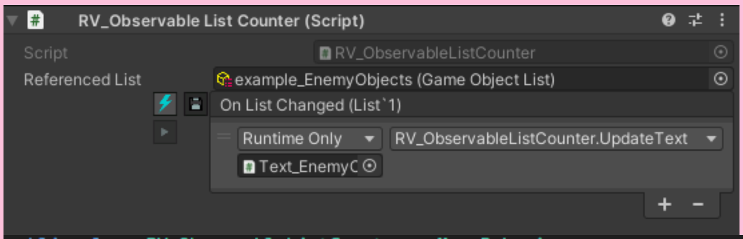
This script also has a method KillEnemiesInsideSphere which will destroy all GameObjects stored in the Observable List. Then it clears the list with referencedList.Clear(). This will also invoke the OnListChanged event.



RV_Observable List Detonator (Script)

Script	RV_ObservableListDetonator
Referenced List	example_EnemyObjects (Game Object List)
Enemy Prefab	RV_MovingEnemy
Min Spawn Pos	X -30 Y 0 Z -30
Max Spawn Pos	X 30 Y 0 Z 30

The Observable List Counter references the same Observable List as the Observable List Detonator. It listens for the OnListChanged event (the lightning bolt symbol is activated). Once the event is fired, this object updates a text and displays the current count of objects within the referenced Observable List.



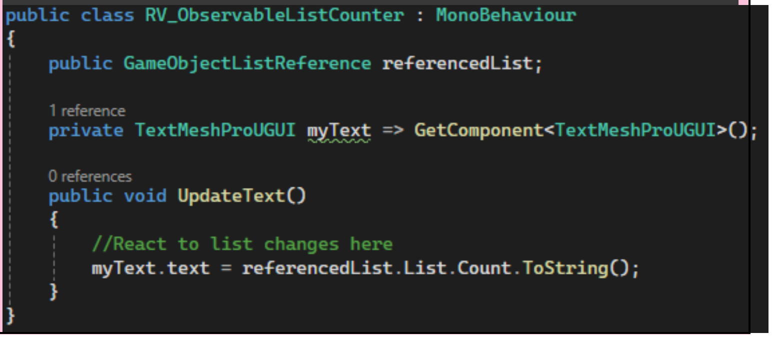
RV_Observable List Counter (Script)

Script	RV_ObservableListCounter
Referenced List	example_EnemyObjects (Game Object List)

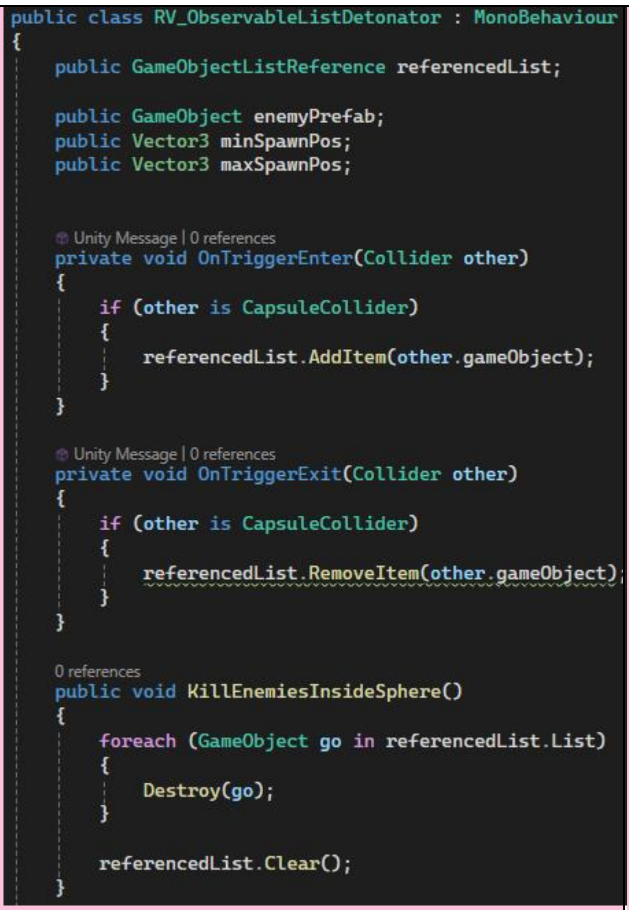
On List Changed (List`1)

Runtime Only | RV_ObservableListCounter.UpdateText

Text_EnemyC



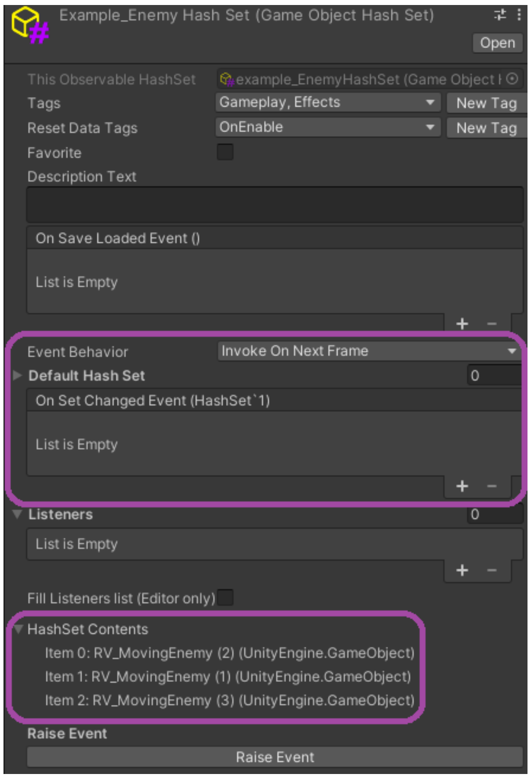
```
public class RV_ObservableListCounter : MonoBehaviour
{
    public GameObjectListReference referencedList;

    1 reference
    private TextMeshProUGUI myText => GetComponent<TextMeshProUGUI>();

    0 references
    public void UpdateText()
    {
        //React to list changes here
        myText.text = referencedList.List.Count.ToString();
    }
}
```



```
public class RV_ObservableListDetonator : MonoBehaviour
{
    public GameObjectListReference referencedList;

    public GameObject enemyPrefab;
    public Vector3 minSpawnPos;
    public Vector3 maxSpawnPos;


    Unity Message | 0 references
    private void OnTriggerEnter(Collider other)
    {
        if (other is CapsuleCollider)
        {
            referencedList.AddItem(other.gameObject);
        }
    }

    Unity Message | 0 references
    private void OnTriggerExit(Collider other)
    {
        if (other is CapsuleCollider)
        {
            referencedList.RemoveItem(other.gameObject);
        }
    }

    0 references
    public void KillEnemiesInsideSphere()
    {
        foreach (GameObject go in referencedList.List)
        {
            Destroy(go);
        }

        referencedList.Clear();
    }
```

Observable Hash Set

Observable Hash Sets very similar to Observable Lists, but it guarantees that the elements in the set are unique.



Example_Enemy Hash Set (Game Object Hash Set)

Open

This Observable HashSet	example_EnemyHashSet (Game Object I
Tags	Gameplay, Effects
Reset Data Tags	OnEnable
Favorite	
Description Text	

On Save Loaded Event ()

List is Empty

Event Behavior	Invoke On Next Frame
Default Hash Set	0

On Set Changed Event (HashSet`1)

List is Empty

Listeners 0

List is Empty

Fill Listeners list (Editor only)

HashSet Contents

Item 0: RV_MovingEnemy (2) (UnityEngine.GameObject)

Item 1: RV_MovingEnemy (1) (UnityEngine.GameObject)

Item 2: RV_MovingEnemy (3) (UnityEngine.GameObject)

Raise Event

We can create a new Observable Hash Set of some type under **"Assets/Create/GODS/Hash Sets/…"**

The fields of an Observable Hash Set asset are so similar to an Observable List that there should be no need to list them here.

The main differences to the Observable List are:

- Default Hash Set is actually a List. Duplicate values will be removed when the data gets reset and before the values get added to the Current Hash Set.
- The contents of a Current Hash Set are not serializable so in order to display them, a custom list "HashSet Contents" gets generated by a custom inspector.

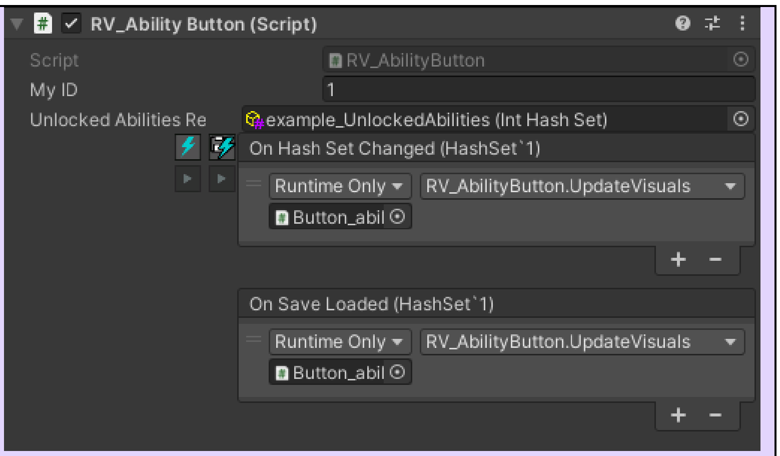
Observable Hash Set Reference

Observable Hash Set References are fields you can add to your MonoBehaviours. They are very similar to ***Observable List References***. An Observable Hash Set is required as a reference.

The **On Hash Set Changed** event gets raised once we update the Current Hash Set of the Observable Hash Set.

The **On Save Loaded** event gets raised when a SaveSlot loads a save which includes the referenced Observable Hash Set. More about Save Slots: ***SaveSlots: Saving/Loading Observables***

Just like with other Observable References you can use the lightning bolt and the lightning bolt + save disc symbol to subscribe to events.



RV_Ability Button (Script)

Script RV_AbilityButton

My ID 1

Unlocked Abilities Re example_UnlockedAbilities (Int Hash Set)

On Hash Set Changed (HashSet`1)

Runtime Only RV_AbilityButton.UpdateVisuals

Button_abil

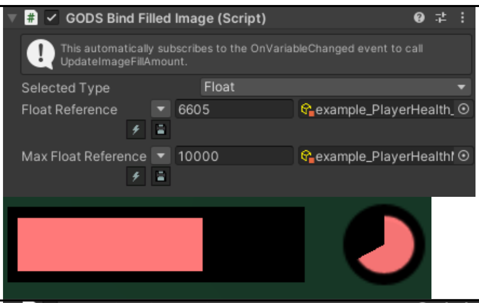
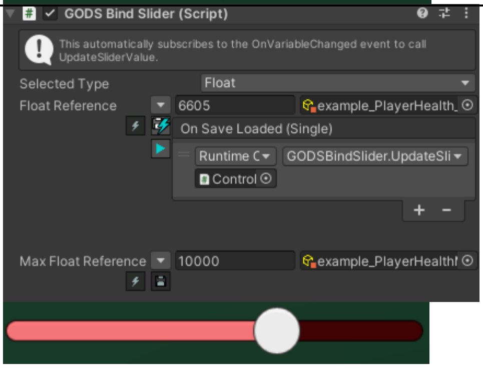
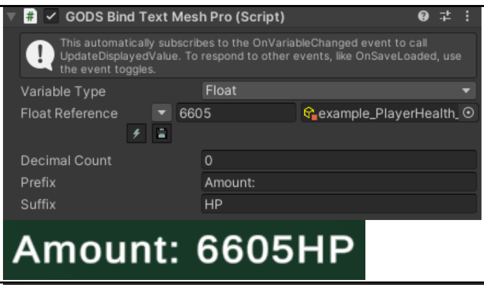
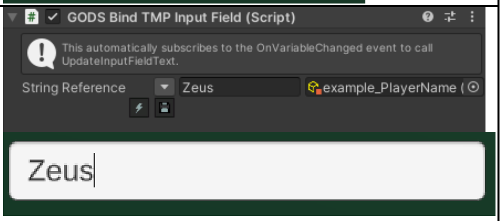
On Save Loaded (HashSet`1)

Runtime Only RV_AbilityButton.UpdateVisuals

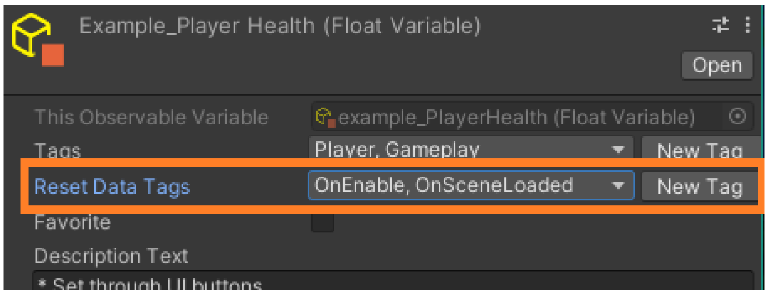
Button_abil

Display Variable Data via Binding Components

Some components are already available to make displaying certain data types in UI more easy. You can make use of them instead of creating your own custom scripts. The scripts are:

GODSBindFilledImage Available types are: Float, Double, Int This allows you to reference a maximum and a minimum value. The component will automatically set the Fill Amount of the (required) image component.	 GODS Bind Filled Image (Script) This automatically subscribes to the OnVariableChanged event to call UpdateImageFillAmount. Selected Type: Float Float Reference: 6605 example_PlayerHealth_ Max Float Reference: 10000 example_PlayerHealthl
GODSBindSlider Available types are: Float, Double, Int Similarly this displays the current value ratio of two referenced Observables. It subscribes to the onValueChanged events on both Observables to update the visuals when needed. This component is useful for something like health bars, experience bars, audio sliders, … Keep the Handle Area enabled to let the user control the value at any time.	 GODS Bind Slider (Script) This automatically subscribes to the OnVariableChanged event to call UpdateSliderValue. Selected Type: Float Float Reference: 6605 example_PlayerHealth_ On Save Loaded (Single) Runtime C GODSBindSlider.UpdateSli Control Max Float Reference: 10000 example_PlayerHealthl
GODSBindTextMeshPro Available types are: Float, Double, Int, String, Char This allows you to display values in TextMeshPro components. You can add a Prefix and a Suffix to the displayed value. For float and double values you can define a decimal count.	 GODS Bind Text Mesh Pro (Script) This automatically subscribes to the OnVariableChanged event to call UpdateDisplayedValue. To respond to other events, like OnSaveLoaded, use the event toggles. Variable Type: Float Float Reference: 6605 example_PlayerHealth_ Decimal Count: 0 Prefix: Amount: Suffix: HP Amount: 6605HP
GODSBindTMPInputField Available types are: String Allows you to change the value of a StringVariable. This automatically subscribes to the onValueChanged event of the referenced Observable and updates the displayed text on a change. This means if you have two of the same GODSBindTMPInputFields in the scene, both will update their text when you edit one.	 GODS Bind TMP Input Field (Script) This automatically subscribes to the OnVariableChanged event to call UpdateInputFieldText. String Reference: Zeus example_PlayerName (Zeus
There are more binders available which work in a similar fashion. • **GODSBindImageColor** • **GODSBindMaterial** • **GODSBindMaterialColor** • **GODSBindRendererColor** • **GODSBindSpriteColor** • **GODSBindTexture** • **GODSBindToggle**	

Reset Data Tags



Example_Player Health (Float Variable)
Open

This Observable Variable: example_PlayerHealth (Float Variable)
Tags: Player, Gameplay — New Tag
Reset Data Tags: OnEnable, OnSceneLoaded — New Tag
Favorite
Description Text
* Set through UI buttons

Reset Data Tags are available on Observable-Variables, -Lists and -HashSets. They control when the data on these Observables will be reset. The following data will be reset:

- Variables
  - Current Value gets set to Default Value
  - Current Also Invoke On Equal Value gets set to Default Also Invoke On Equal Value
- Lists
  - The List gets cleared
- HashSets
  - The HashSet gets cleared

When you create a new Observable it will have the OnEnable tag applied by default. This means it will reset its data when the game starts. There are more tags available like OnSceneLoaded, OnSceneLoadedAdditively and OnSceneUnloaded which can be used out of the box.

Adding Reset Data Tags

You can add more tags when you need them, but you will have to add the functionality for them on your own. For example, you could add a tag called “OnLevelCompleted” to reset some Observables when the user completed a section. Tags can be added in the inspector of an Observable with the “New Tag” button or in the ***GODS Hub: Observable Management Window*** by clicking the Pen-Button next to the Tags filter options.

Let’s stick with the “OnLevelCompleted” example. After creating the new tag, we need to go through the Observables which we want to reset and add that tag to the Reset Data Tags.

Finally in our code where we detect that the player has completed the level we need to call the static method **GODSManagedObservable.HandleResetDataEventAllObservables(string eventName);**

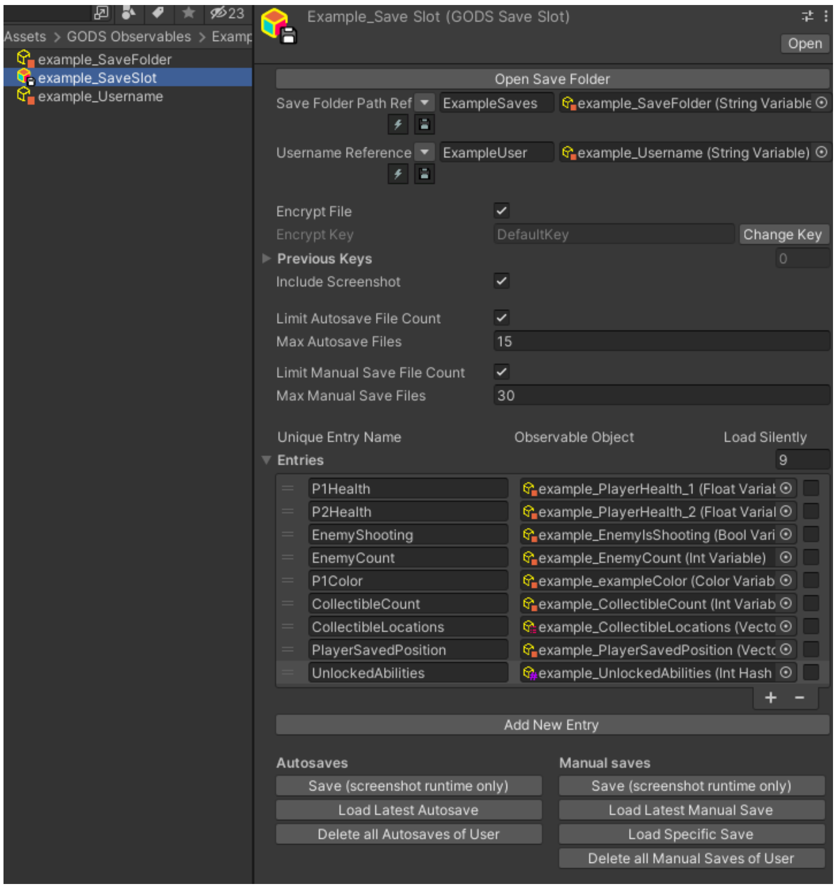
In our case this would be: **GODSManagedObservable.HandleResetDataEventAllObservables(“OnLevelCompleted”);**

This will reset the data for any Observable Variable/List/HashSet which has “OnLevelCompleted” in its Reset Data Tags.

SaveSlots: Saving/Loading Observables

Overview

The GODSSaveSlot is a scriptable object designed to manage the saving and loading of game data for Observable Variables, Lists, and HashSets. This system supports both manual and automatic saving, with options for encryption and screenshot inclusion. It's highly customizable and can easily be integrated into your game's saving and loading logic.



Save Folder Path Ref	ExampleSaves	example_SaveFolder (String Variable
Username Reference	ExampleUser	example_Username (String Variable)
Encrypt File	✓	
Encrypt Key	DefaultKey	Change Key
Previous Keys		0
Include Screenshot	✓	
Limit Autosave File Count	✓	
Max Autosave Files	15	
Limit Manual Save File Count	✓	
Max Manual Save Files	30	

Entries 9

Unique Entry Name	Observable Object	Load Silently
P1Health	example_PlayerHealth_1 (Float Varial	
P2Health	example_PlayerHealth_2 (Float Varial	
EnemyShooting	example_EnemyIsShooting (Bool Vari	
EnemyCount	example_EnemyCount (Int Variable)	
P1Color	example_exampleColor (Color Variab	
CollectibleCount	example_CollectibleCount (Int Variab	
CollectibleLocations	example_CollectibleLocations (Vecto	
PlayerSavedPosition	example_PlayerSavedPosition (Vectc	
UnlockedAbilities	example_UnlockedAbilities (Int Hash	

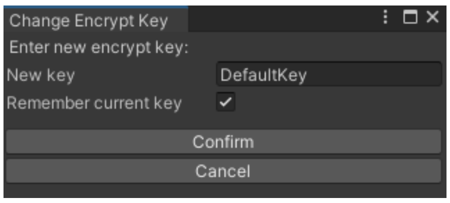
+ −
Add New Entry
Autosaves
Save (screenshot runtime only)
Load Latest Autosave
Delete all Autosaves of User
Manual saves
Save (screenshot runtime only)
Load Latest Manual Save
Load Specific Save
Delete all Manual Saves of User

Step-by-Step Guide

1. Configuring Save Settings

- **Save Folder Path Reference**: In the GODSSaveSlot inspector, assign a StringReference to define where the save files will be stored. This can be a relative path under Application.persistentDataPath.
- **Username Reference**: Set a StringReference for the username, which will be used as a prefix in save file names. This allows you to manage multiple users or profiles.
- **Encryption**: Enable encryptFile to encrypt your save files. You can customize the encryptKey by using the "Change Key" button – this will open a "Change Encrypt Key" window where you can change the value. Encrypting the save files adds a layer of security, making them unreadable outside your game.

CAUTION!
Changing the encryptKey at a later time will cause earlier save files to be unreadable. If you need to change the key during production and want to be able to read in older save files, you can tick "**Remember current key**". Future files will be encrypted with the new key.



Enter new encrypt key:

New key	DefaultKey
Remember current key	✓

- **Include Screenshot**: If enabled, the system will capture and save a screenshot alongside each save file, providing a visual preview in your save/load menu.
- **Maximum Save File Limits**: You can set a limit on the number of manual and autosave files a user can create. When this limit is reached and a new save is made, the oldest save(s) will be automatically deleted to stay within the defined limit.

2. Adding Save Entries

- **Save Entry**: Each entry represents an observable object (Variable, List, or HashSet) that you want to save. Add a new entry by clicking "Add New Entry" in the inspector.
- **Entry Name**: Provide a unique name for each entry. The system will automatically ensure names are unique, but it's good practice to use meaningful names for easier debugging and reference.
- **Observable Object**: Select the observable object you wish to save. The system will serialize this object when saving.
- **Load Silently**: If enabled, the OnSaveLoaded event will not be triggered for this entry when loading. This can be useful if you want to load data without updating the UI or triggering other game events.

3. Saving Data

- **Manual Save**: Trigger a manual save by calling SaveEntries(GODSSaveSlot.SaveType.Manual) or by using the provided UI button in the GODSSaveLoadManager. Manual saves are typically used when the player explicitly saves the game.
- **Auto Save**: Automatically save the game by calling SaveEntries(GODSSaveSlot.SaveType.Auto). Autosaves are usually triggered by events like level completion or at regular intervals.

Note: When saving, the system serializes the observable objects into JSON and stores them in the specified save path. If encryption is enabled, the JSON data is encrypted before being written to the file.

4. Loading Data

- **Load Latest Autosave**: Call LoadLatestAutoSave() to load the most recent autosave for the current user. This is useful for resuming gameplay from where the player left off.
- **Load Specific Save**: Select a save file from the UI list and load it by calling LoadSaveFile(selectedSaveFile). This allows players to choose a specific save to load.

Behind the Scenes:

- The system uses reflection to deserialize the saved JSON data back into the observable objects.
- It automatically updates the objects with the loaded values, ensuring your game state is restored accurately.
- After all data is loaded, the system triggers the OnSaveLoaded event on the Observables in the list, allowing you to respond to the loaded data (e.g., updating UI elements or triggering game events). If you don't want to trigger the OnSaveLoaded event, you can enable the loadSilently check box per Observable in the list.

5. Managing Save Files

- **Save File List**: The system can generate a list of all save files for a user, allowing you to populate a UI list where players can select files to load or delete.
- **Deleting Save Files**: Use the delete functionality to remove unwanted save files and their associated screenshots, keeping your save directory clean.

6. Customizing and Extending the System

- **Custom Save Logic**: You can customize the GODSSaveSlot class to include additional data or alter the save process to fit specific needs.
- **Observable Objects**: The system is designed to work with any observable object inheriting from ObservableVariableBase, ObservableListBase, or ObservableHashSetBase. You can extend these classes to support additional data types or complex objects.
- **Events and Hooks**: Utilize the OnSaveLoaded event to trigger custom logic when data is loaded. This makes the system flexible and easily integrated with other game mechanics.

Code Explanations

Entry Name Uniqueness

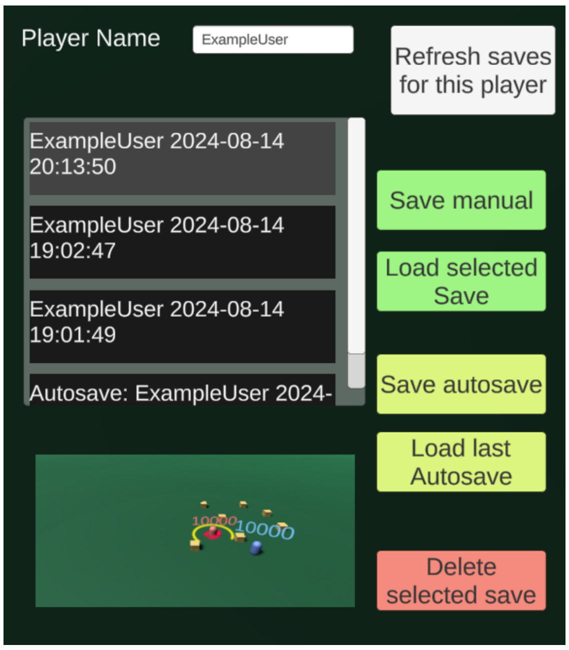
The system ensures that each entry name is unique by automatically appending numbers to duplicate names. This prevents conflicts during saving and loading.

```
if (entriesDictionary.ContainsKey(entries[i].entryName))
{
    int counter = 1;
    string newName = $"{entries[i].entryName}_{counter}";
    while (entriesDictionary.ContainsKey(newName))
    {
        counter++;
        newName = $"{entries[i].entryName}_{counter}";
    }
    entries[i].entryName = newName;
}
```

Screenshot Capturing

The system includes a feature to capture and save screenshots alongside the save file. This can be useful for providing visual context to save files in a UI.

Example: Saving/Loading with GODSSaveLoadManager


Player Name
ExampleUser
Refresh saves for this player
ExampleUser 2024-08-14 20:13:50
ExampleUser 2024-08-14 19:02:47
ExampleUser 2024-08-14 19:01:49
Autosave: ExampleUser 2024-
Save manual
Load selected Save
Save autosave
Load last Autosave
Delete selected save

The **GODSSaveLoadManager** class is an example implementation of how to integrate saving and loading functionality into a Unity project. This class handles user interactions for saving, loading, and managing saved game files, leveraging the **GODSSaveSlot** class for data serialization, encryption, and file management.

Fields of GODSSaveLoadManager component

1. **UI Elements**:
   - **TMP_InputField playerNameInput**: Allows the player to input their name, which is used to personalize and organize save files.
   - **Transform saveListContent**: A container for dynamically generated save file entries.
   - **GameObject saveEntryPrefab**: A prefab for creating individual save file entries in the UI.
   - **Image screenshotPreview**: Displays a preview of the save file's screenshot (if available).
   - **Buttons**:
     - **refreshSavedFilesButton**: Refreshes the list of available save files.
     - **saveManualButton**: Triggers saving of a new manual save file.
     - **saveAutosaveButton**: Triggers saving of a new autosave file.
     - **loadSaveButton**: Loads the selected save file.
     - **loadLatestAutoSaveButton**: Loads the latest autosave.
     - **deleteSelectedSaveButton**: Deletes the selected save file.
2. **Internal Data**:
   - **List<string> currentSaveFiles**: Stores the paths of all save files associated with the current player.
   - **string selectedSaveFile**: Stores the path of the currently selected save file.

Initialization: Start()

The Start method initializes the GODSSaveLoadManager:

- **Validation**: It checks whether the saveSlot reference is correctly assigned.
- **UI Setup**: Configures buttons and input fields to respond to user actions.
- **Load Existing Saves**: Calls LoadSavesForPlayer to load any existing save files for the player whose name is currently in the playerNameInput.

Example Usage in Unity

To use GODSSaveLoadManager in your Unity project:

1. Attach the GODSSaveLoadManager script to a GameObject in your scene.

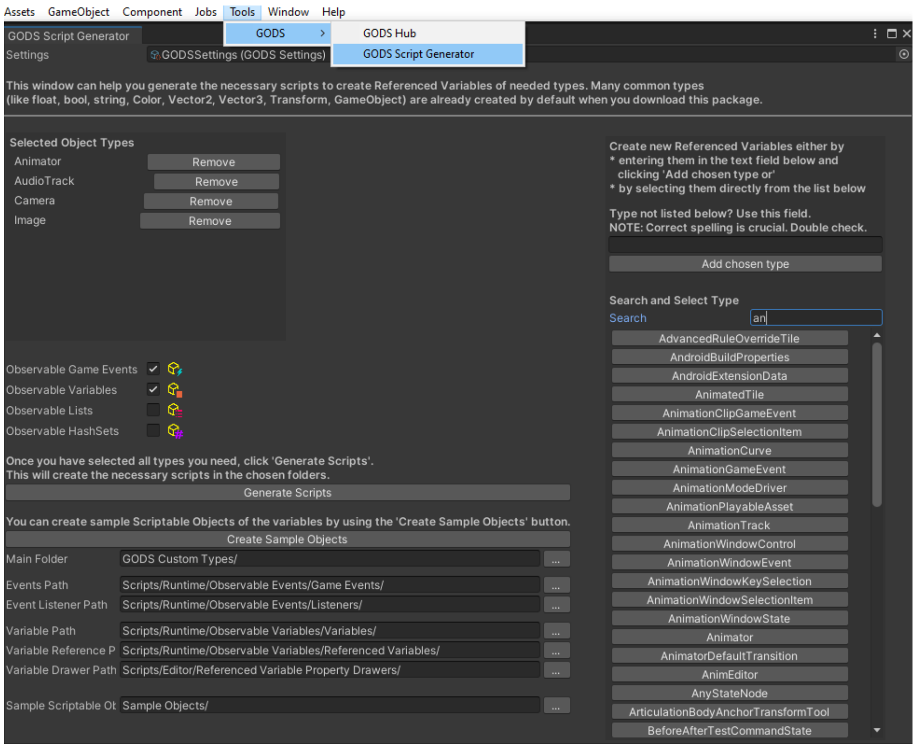
2. Assign references to the UI elements in the Inspector.
3. Ensure that the GODSSaveSlot reference is correctly set up.
4. Customize the save/load functionality as needed to fit your game's logic.

Also have a look at the **SaveAndLoadData** example scene for more information.

Remember: The **GODSSaveLoadManager** is an example class. You can make use of it or create your own saving/loading class to provide the user with the necessary functionality you need.

Creating More Observable Types

The generator window makes it easy to create more variable types, should you need them. Open the Variable Generator window by clicking on **Tools → GODS → GODS Script Generator**



GODS Script Generator

Settings: GODSSettings (GODS Settings)

This window can help you generate the necessary scripts to create Referenced Variables of needed types. Many common types (like float, bool, string, Color, Vector2, Vector3, Transform, GameObject) are already created by default when you download this package.

Selected Object Types

- Animator
- AudioTrack
- Camera
- Image

Observable Game Events ✓
Observable Variables ✓
Observable Lists
Observable HashSets

Once you have selected all types you need, click 'Generate Scripts'.
This will create the necessary scripts in the chosen folders.

You can create sample Scriptable Objects of the variables by using the 'Create Sample Objects' button.

Main Folder	GODS Custom Types/
Events Path	Scripts/Runtime/Observable Events/Game Events/
Event Listener Path	Scripts/Runtime/Observable Events/Listeners/
Variable Path	Scripts/Runtime/Observable Variables/Variables/
Variable Reference P	Scripts/Runtime/Observable Variables/Referenced Variables/
Variable Drawer Path	Scripts/Editor/Referenced Variable Property Drawers/
Sample Scriptable Ot	Sample Objects/

Create new Referenced Variables either by
* entering them in the text field below and clicking 'Add chosen type or'
* by selecting them directly from the list below

Type not listed below? Use this field.
NOTE: Correct spelling is crucial. Double check.

Search and Select Type

Search: an

AdvancedRuleOverrideTile
AndroidBuildProperties
AndroidExtensionData
AnimatedTile
AnimationClipGameEvent
AnimationClipSelectionItem
AnimationCurve
AnimationGameEvent
AnimationModeDriver
AnimationPlayableAsset
AnimationTrack
AnimationWindowControl
AnimationWindowEvent
AnimationWindowKeySelection
AnimationWindowSelectionItem
AnimationWindowState
Animator
AnimatorDefaultTransition
AnimEditor
AnyStateNode
ArticulationBodyAnchorTransformTool
BeforeAfterTestCommandState

To get started, select the types you need from the list on the right side. There is a search field above the list to speed up that process. Should you not be able to find your desired type, you can enter the type name into the text field above and then click “Add chosen type”. Chosen types will appear on the left side under “Selected Object Types”. Once you have selected the types you need, click on the “Generate Scripts” button. Depending on what boxes you have checked, this will create the following scripts.

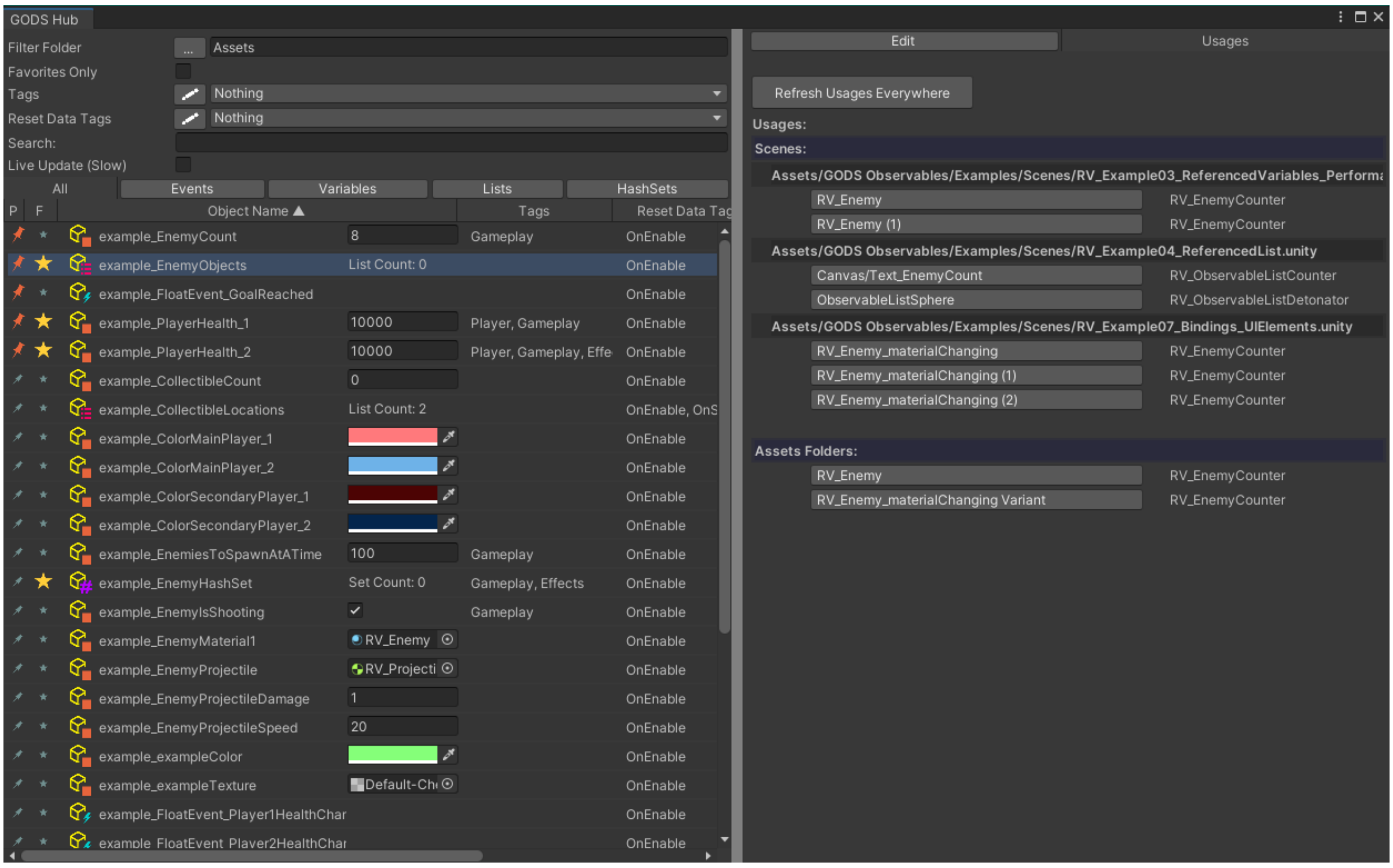
- Observable Game Events
  - {type name}GameEvent
  - {type name}GameEventListener
- Observable Variables
  - {type name}Variable
  - {type name}Reference
  - { type name}ReferencePropertyDrawer
- Observable Lists
  - {type name}List
  - {type name}ListReference
  - { type name}ListReferencePropertyDrawer
- Observable HashSets
  - {type name}HashSet
  - {type name}HashSetReference
  - { type name}HashSetReferencePropertyDrawer

If this creates compile errors after creating the new scripts, we might have to rewrite some of the automatically created code. Have a look at the

Troubleshooting section for more information.

GODS Hub: Observable Management Window

The GODS Hub window assists in managing numerous Observable assets in larger projects. It features a search/filter section on the left and a manipulation/debug section on the right. The right side includes the default inspector, bulk operations, and usage lists for selected Observables. This tool complements the debug information provided by individual inspectors.



GODS Hub

Filter Folder: Assets
Favorites Only: ☐
Tags: Nothing
Reset Data Tags: Nothing
Search:
Live Update (Slow): ☐

All | Events | Variables | Lists | HashSets

P	F	Object Name ▲		Tags	Reset Data Tag
📌		example_EnemyCount	8	Gameplay	OnEnable
📌	★	example_EnemyObjects	List Count: 0		OnEnable
📌		example_FloatEvent_GoalReached			OnEnable
📌	★	example_PlayerHealth_1	10000	Player, Gameplay	OnEnable
📌	★	example_PlayerHealth_2	10000	Player, Gameplay, Effe	OnEnable
		example_CollectibleCount	0		OnEnable
		example_CollectibleLocations	List Count: 2		OnEnable, OnS
		example_ColorMainPlayer_1	🟥		OnEnable
		example_ColorMainPlayer_2	🟦		OnEnable
		example_ColorSecondaryPlayer_1	🟫		OnEnable
		example_ColorSecondaryPlayer_2	🟦		OnEnable
		example_EnemiesToSpawnAtATime	100	Gameplay	OnEnable
	★	example_EnemyHashSet	Set Count: 0	Gameplay, Effects	OnEnable
		example_EnemyIsShooting	✓	Gameplay	OnEnable
		example_EnemyMaterial1	RV_Enemy		OnEnable
		example_EnemyProjectile	RV_Projecti		OnEnable
		example_EnemyProjectileDamage	1		OnEnable
		example_EnemyProjectileSpeed	20		OnEnable
		example_exampleColor	🟩		OnEnable
		example_exampleTexture	Default-Ch		OnEnable
		example_FloatEvent_Player1HealthChar			OnEnable
		example_FloatEvent_Player2HealthChar			OnEnable

Edit | Usages

Refresh Usages Everywhere

Usages:

Scenes:

Assets/GODS Observables/Examples/Scenes/RV_Example03_ReferencedVariables_Performa

RV_Enemy	RV_EnemyCounter
RV_Enemy (1)	RV_EnemyCounter

Assets/GODS Observables/Examples/Scenes/RV_Example04_ReferencedList.unity

Canvas/Text_EnemyCount	RV_ObservableListCounter
ObservableListSphere	RV_ObservableListDetonator

Assets/GODS Observables/Examples/Scenes/RV_Example07_Bindings_UIElements.unity

RV_Enemy_materialChanging	RV_EnemyCounter
RV_Enemy_materialChanging (1)	RV_EnemyCounter
RV_Enemy_materialChanging (2)	RV_EnemyCounter

Assets Folders:

RV_Enemy	RV_EnemyCounter
RV_Enemy_materialChanging Variant	RV_EnemyCounter

Left side: Filter and Search	
Filter Options	You can filter the list below by: • Folder (and subfolders) • Favorites (Observables marked as Favorite) • Tags • Reset Data Tags (when data gets reset, default Tags include: OnEnable, OnSceneLoaded, OnSceneLoadedAdditive, , OnSceneUnloaded) • Search (filter by asset name) • Filter Observable Types (All, Events, Variables, Lists, HashSets)
Live Update (Slow)	When enabled, the window will repaint ever frame at runtime to keep the displayed values

Right side: edit and debug	
Mode Options	• **Edit**: Allows you to edit the selected Observable(s) • **Usages**: Lists usages of the selected Observable across your project. Usages for only one Observable can be displayed at a time.
Bulk Operations	On the top we have operations which can be applied to multiple selected Observables at once. These are: • Mark as pinned • Mark as favorite • Change Tags Rename, Duplicate and Delete
Inspector	Displayed below the bulk operations is the default inspector of the selected Observable object.

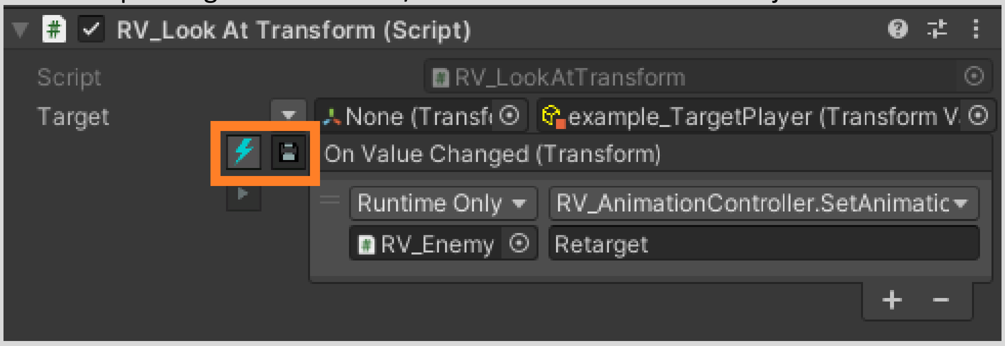
	updated. This option is performance intensive but can be useful for debugging purposes.
Sorting Options	By clicking on the headers (P, F, Object Name, Tags, …) you can sort the list below.
List details	Each displayed Observable shows additional details, these are: • Pinned • Favorite • Object Name + Value or List/HashSet count • Tags • Reset Data Tags • System Type • Last Modified If the type is supported, each Variable displays the default value in edit mode and the current value in play mode. If you want to edit enemy health values or ability values, etc. this might be a good place to do so as it gives a nice overview if filtered correctly.

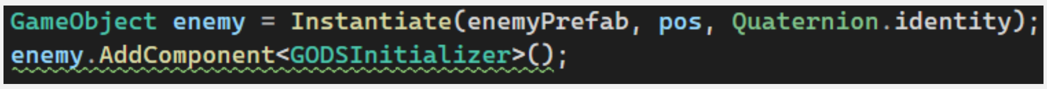
Usages	Finally you can update a list of usages all Observables by clicking the “Refresh Usages Everywhere” button. Below will appear a list of every object which references the selected Observable. The list is separated into a Scenes- and Assets Folder section. Scenes: Only scenes get displayed which are added to the build in Build Settings. Asset Folders: These are assets and prefabs which reference the selected Observable.

About Performance

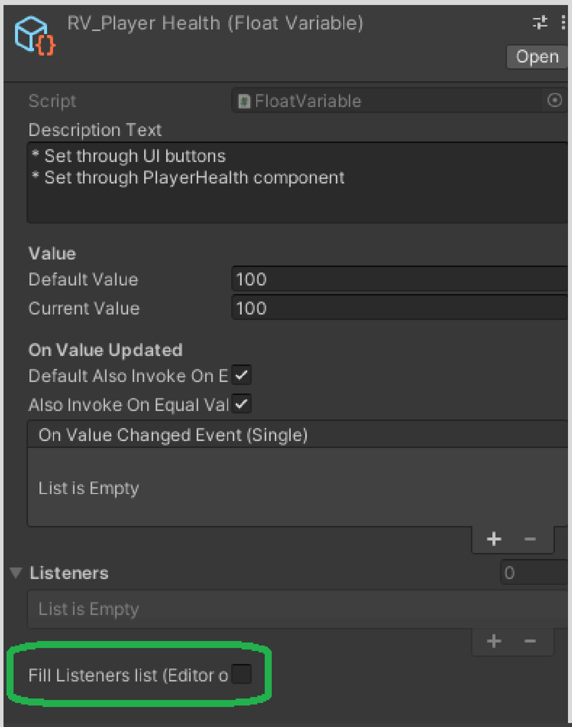
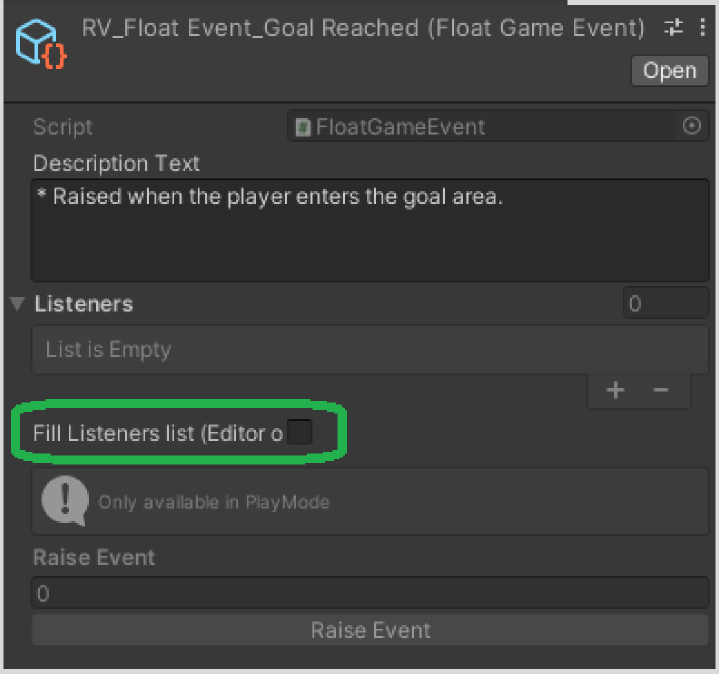
While the described Observables can be very handy to plug into many places in the project, please be also aware that using them instead of regular property fields can have an impact on performance, especially if you have a huge amount of objects which use the event systems (OnValueChanged and OnSaveLoaded). In most scenarios this should not be noticeable, so don't worry too much.

Here are some things to keep in mind.

Only use response events where necessary.	Spawning objects with Referenced Variable fields can be more performance intensive than using regular fields, especially when you have the **useEvents (thunderbolt icon) checkbox** enabled. Having this enabled means the **GODSRuntimeManager** will add a **GODSStateDetector** component to this GameObject to ensure the variable listens for events depending on the enabled/disabled state of the GameObject.  RV_Look At Transform (Script) Script: RV_LookAtTransform Target: None (Transf… / example_TargetPlayer (Transform V… On Value Changed (Transform) Runtime Only / RV_AnimationController.SetAnimatic… RV_Enemy / Retarget If you disable these checkboxes, the Referenced Variable will not subscribe to the OnValueChanged event and will not react, saving performance. HINT: **Keep event toggles disabled when you don't need to respond to events.** HINT: **If you need to spawn huge amounts of objects, consider using object pools. As spawning and despawning is costly.**

Add a GODSInitializer component if you create and spawn an Observable Reference from scratch.	
	`GameObject enemy = Instantiate(enemyPrefab, pos, Quaternion.identity);`
	`enemy.AddComponent<GODSInitializer>();`
	
	Observables have a OwningGameObject field which is hidden in the inspector. This field will automatically get set when you interact with the GameObject through the inspector. However, if you create a GameObject from scratch via code, this will not get set (for Prefabs this should already be set as you interacted with the object previously).
	
	If you spawn a GameObject from scratch which has Observable(s), the GODSRuntimeManager will have to search through all existing GameObjects in the Scene(s) to find the Observable. This can impact performance a lot when spawning many GameObjects. To improve performance, add a GODSInitializer to the new GameObject. This script goes through the property fields of all components of the GameObject and all children and sets the OwningGameObject. So instead of going through all objects in the scene, it only needs to check the new GameObject.

In the Editor: Disable "Fill Listeners List (Editor Only)" when not needed to improve performance.	In the Editor you can improve performance when spawning objects by disabling the "Fill Listeners list (Editor only)" checkbox on Game Event Scriptable Objects and/or Referenced Variable Scriptable Objects. Both settings control the same global Boolean variable. When disabled, the Listeners list in Game Events and Referenced Variables

	won’t be filled. You only need to enable this setting when you need help debugging your scenes.  RV_Player Health (Float Variable) Open Script FloatVariable Description Text * Set through UI buttons * Set through PlayerHealth component **Value** Default Value 100 Current Value 100 **On Value Updated** Default Also Invoke On E ✓ Also Invoke On Equal Val ✓ On Value Changed Event (Single) List is Empty **Listeners** 0 List is Empty Fill Listeners list (Editor o ☐  RV_Float Event_Goal Reached (Float Game Event) Open Script FloatGameEvent Description Text * Raised when the player enters the goal area. **Listeners** 0 List is Empty Fill Listeners list (Editor o ☐ Only available in PlayMode Raise Event 0 Raise Event

Code Documentation

The following documentation might not be complete, but gives you a good overview of the most important properties and methods.

GODSManagedObservable

This class derives from ScriptableObject and is the base class of all Observable GameEvents/Variables/Lists/HashSets.

Properties

isPinned	Boolean used for filtering in the GODS Hub Window
favorite	Boolean used for filtering in the GODS Hub Window
selectedTags	A list of strings used for filtering in the GODS Hub Window
selectedResetDataTags	A list of strings used to control when the data of Observables gets reset
descriptionText	A text area for the development team to provide information about the event, such as where it is raised from and which objects listen for the event.
`resetEvent`	An event which gets invoked when the data of the Observable gets reset
`onSaveLoadedEvent`	An event which gets invoked when the data gets loaded from disk via a GODSSaveSlot
allManagedObservables	A HashSet which stores all GODSManagedObservables

Methods

HandleResetDataEventAllObservables(string resetDataTag)	Static method which goes through all managed Observables and calls the ResetData() method for any Observable which has the resetDataTag string in its **selectedResetDataTags** list.
HandleResetDataEvent(string resetDataTag)	Checks if the Observable has the resetDataTag in its **selectedResetDataTags** list and if true, resets the stored data.
ResetData	Abstract method used to reset the stored data.
RaiseEventWithCurrentValue	Abstract method used to raise the OnValueChanged event with the currently stored Value/List/HashSet (also for testing purposes)
RaiseSaveLoadedEventWithCurrentValue	Abstract method used to raise the OnSaveLoaded event on Observables
OnEnable()	Unity lifecycle method called when the scriptable object is enabled. Resets the variable to its default value and subscribes to the `resetEvent`.
OnDisable()	Unity lifecycle method called when the scriptable object is disabled. Unsubscribes from the `resetEvent`.

ObservableGameEvent

Properties

gameEventListeners	A list of listeners that are subscribed to this event.
listeners	A list of listener data used for debugging in the editor.

Methods

Raise(T item)	Raises the event, invoking the `OnEventRaised` method on all subscribed listeners with the provided item.

RegisterListener(GameEventListener<T> listener)	Registers a listener to the event.
UnregisterListener(GameEventListener<T> listener)	Unregisters a listener from the event.
ResetData()	Clears both listeners lists.

ObservableGameEventListener

Properties

raisingEvent	The event that this listener is subscribed to.
responseEvent	The response event that will be invoked when the `GameEvent` is raised.

Methods

OnEventRaised(T item)	Called through the referenced `GameEvent ScriptableObject` once the `Raise` method is called on it. Invokes the `responseEvent` with the provided item.
OnEnable()	Registers the listener to the `raisingEvent` when the component is enabled.
OnDisable()	Unregisters the listener from the `raisingEvent` when the component is disabled.

ObservableVariable

Description

An abstract base class for referenced variables. This class provides a mechanism to reset and manage variable values within the game, ensuring that changes to these values can trigger events to notify other components.

Properties

disableValueFieldOnReferences	Boolean value. When enabled this will prevent developers from accidentally changing the default value on a ObservableVariableReference field. Can be useful when the value is already finalized.
defaultValue	The default value of the variable. The currentValue gets set to defaultValue when ResetData() is called (e.g. when the game starts, a scene unloads, …)
currentValue	The value to be used at runtime.
previousValue	The previous value of currentValue. Gets updated before currentValue gets reassigned.
resetEvent	Static event that triggers a reset of all resettable variables.
defaultAlsoInvokeOnEqualValue	Determines whether to invoke the event even if the new value is the same as the old value when the scriptable object is first accessed.
alsoInvokeOnEqualValue	Determines whether to invoke the event even if the new value is the same as the old value at runtime.
listeners	A list of listeners that are currently subscribed to this variable. Used for debugging purposes in editor only.
CurrentValue	Property to get or set the current value. Setting this property will raise the `onValueChangedEvent` unless the value is set using `SetValueSilent`.
PreviousValue	Property to get or set the previous value. Can be used to calculate how the current value has changed. E.g. if a player health variable got increased or decreased.

Methods

SetValue(T value)	Sets the `currentValue` through `CurrentValue`. (will try to raise the `onValueChangedEvent` afterwards)

SetValueSilent(T value)	Sets the `currentValue` without invoking the `onValueChangedEvent`.
RaiseEventWithCurrentValue()	Invokes the `onValueChangedEvent` with `currentValue` as the argument
SubscribeListener(UnityAction<T> listener)	Subscribes a listener to the `onValueChangedEvent`.
UnsubscribeListener(UnityAction<T> listener)	Unsubscribes a listener from the `onValueChangedEvent`.
ResetData()	Resets the `currentValue` to the `defaultValue` and clears the `onValueChangedEvent` listeners. Also sets `alsoInvokeOnEqualValue` to `defaultAlsoInvokeOnEqualValue`

ObservableVariableReferenceBase (classes like FloatReference, StringReference, …)

Properties

useConstant	Determines if a constant value should be used instead of a referenced variable.
constantValue	Holds the constant value if useConstant is true.
previousConstantValue	Holds the previous value if useConstant is true.
variable	The referenced variable. Used if useConstant is false.
onValueChanged	Event triggered when the value of the referenced variable changes.
useEvent	If disabled, BaseReference will not subscribe to the referenced variable event. Hint: keep disabled if not needed to save performance when spawning many objects.
Value	Property to get or set the value. If `useConstant` is true, it returns or sets `constantValue`. Otherwise, it interacts with variable.
PreviousValue	Property to get or set the previous value.

Methods

SetValueSilent (T Value)	Sets the value without invoking the `onValueChanged` event. Uses `constantVariable` if `useConstant` is true, otherwise sets the value on variable.
RaiseVariableEventWithCurrentValue()	Raises the `onValueChangedEvent` with the `currentValue` of the variable as argument. Can be used to raise the event without updating the value or for testing purposes.
SetVariable(V newVariable)	Sets a new referenced variable. Unsubscribes the previous one if there is one and subscribes to the new events on the new variable.
Init()	Starts listening for changes on the referenced variable if `useConstant` is false and variable is not null.
Cleanup()	Stops listening for changes on the referenced variable if it was previously subscribed.

ObservableList

A generic class for observable lists. It provides mechanisms to manage a list of items, with events that can be triggered when the list changes.

Properties

defaultList	A List<T>. The currentList takes the items in this list when ResetData() is called. E.g. when the game starts, a scene unloads, …
currentList	The actual List<T> that holds the items. Used at runtime.
onListChangedEvent	UnityEvent that's invoked when the list changes.

eventBehavior	Enum that determines how and when the list change event is invoked (Immediately, Next Frame, or After X Seconds).
secondsToWaitForInvoking	Float value used when eventBehavior is set to InvokeAfterXSeconds.
listeners	A list of listeners subscribed to this observable list (for debugging in editor).

Methods

Add(T item, bool silent = false)	Adds an item to the list. If not silent, it raises the list changed event.
AddUnique(T item, bool silent = false)	Adds an item to the list if it's not already present. If not silent and the item was added, it raises the list changed event.
public void AddRange(List<T> items, bool silent = false)	Adds a list of items to the list. If not silent and the item was added, it raises the list changed event.
Insert(int index, T item, bool silent = false)	Adds an item to the list at the index position. If not silent and the item was added, it raised the list changed event.
Remove(T item, bool silent = false)	Removes an item from the list. If not silent and the item was removed, it raises the list changed event.
RemoveAt(int index, bool silent = false)	Removes an item at index. If the item was removed and not silent, it will raise the list changed event.
Clear(bool silent = false)	Clears the list. If not silent it raises the list changed event.
RaiseEventWithCurrentValue()	Invokes the onListChangedEvent based on the eventBehavior setting.
SubscribeListener(UnityAction<List<T>> listener, UnityEvent<List<T>> listenerEvent, GameObject owningGameObject)	Subscribes a listener to the list changed event. Additional parameters are used for debugging purposes.
UnsubscribeListener(UnityAction<List<T>> listener)	Unsubscribes a listener from the list changed event.
ResetData()	Clears the list, removes all listeners, and clears the debug listeners list.

ObservableListReferenceBase (classes like GameObjectList, TransformList, …)

Description: A generic base class for referencing ObservableLists.

Properties

listVariable	The referenced ObservableList<T>.
onListChanged	UnityEvent that's invoked when the referenced list changes.
List	Getter for the actual List<T> in the referenced ObservableList.
useEvent	Boolean to determine if the reference should subscribe to the list's change events.
OwningGameObject	The GameObject that owns this reference.

Methods

AddItem(T item, bool silent = false)	Adds an item to the referenced list. If silent is true, no events will be invoked.
AddItemUnique(T item, bool silent = false))	Adds an item to the referenced list if the item is not already in the list. If silent is true, no events will be invoked.
InsertItem(int index, T item, bool silent = false)	Inserts an item at index. If not silent and the item was added, this will invoke the on list changed event.
Remove(T item, bool silent = false)	Removes an item from the list. If not silent and the item was removed, it raises the list changed event.
RemoveItemAt(int index, bool silent = false)	Removes an item at index. If not silent and the item was removed, it raises the list changed event.

Clear(bool silent = false)	Clears the list. If not silent it raises the list changed event.
RaiseVariableEventWithCurrentValue()	Raises the list changed event with the current list value.
SetList(V newList)	Sets a new ObservableList as the reference, unsubscribing from the old one and subscribing to the new one
Init()	Initializes the reference, subscribing to the list's events if useEvent is true.
Cleanup()	Unsubscribes from the list's events.

ObservableHashSet

A generic class for observable hash sets. It provides mechanisms to manage a set of items, with events that can be triggered when the hash set changes.

Properties

hashSet	The actual HashSet<T> that holds the items.
defaultHashSet	Is actually a List<T>. The currentHashSet takes the items in this list when ResetData() is called. E.g. when the game starts, a scene unloads, …
currentHashSet	The actual HashSet<T> that holds the items. Used at runtime.
eventBehavior	Enum that determines how and when the set change event is invoked (Immediately, Next Frame, or After X Seconds).
secondsToWaitForInvoking	Float value used when eventBehavior is set to InvokeAfterXSeconds.
listeners	A list of listeners subscribed to this observable list (for debugging in editor).

Methods

Add(T item, bool silent = false)	Adds an item to the list. If not silent, it raises the list changed event.
AddUnique(T item, bool silent = false)	Adds an item to the set if it's not already present. If not silent and the item was added, it raises the set changed event.
Remove(T item, bool silent = false)	Removes an item from the set. If not silent and the item was removed, it raises the set changed event.
Clear(bool silent = false)	Removes all items from the set. If not silent it raises the set changed event
RaiseEventWithCurrentValue()	Invokes the onSetChangedEvent based on the eventBehavior setting.
SubscribeListener(UnityAction<List<T>> listener, UnityEvent<List<T>> listenerEvent, GameObject owningGameObject)	Subscribes a listener to the set changed event. Additional parameters are used for debugging purposes.
UnsubscribeListener(UnityAction<List<T>> listener)	Unsubscribes a listener from the set changed event.
ResetData()	Clears the set, removes all listeners, and clears the debug listeners list.

ObservableHashSetReferenceBase (classes like GameObjectHashSet, TransformHashSet, …)

Description: A generic base class for referencing a ObservableHashSet.

Properties

hashSetVariable	The referenced ObservableHashSet<T>.
onHashSetChanged	UnityEvent that's invoked when the referenced hashSetVariable changes.
HashSet	Getter for the actual HashSet<T> in the referenced ObservableHashSet.

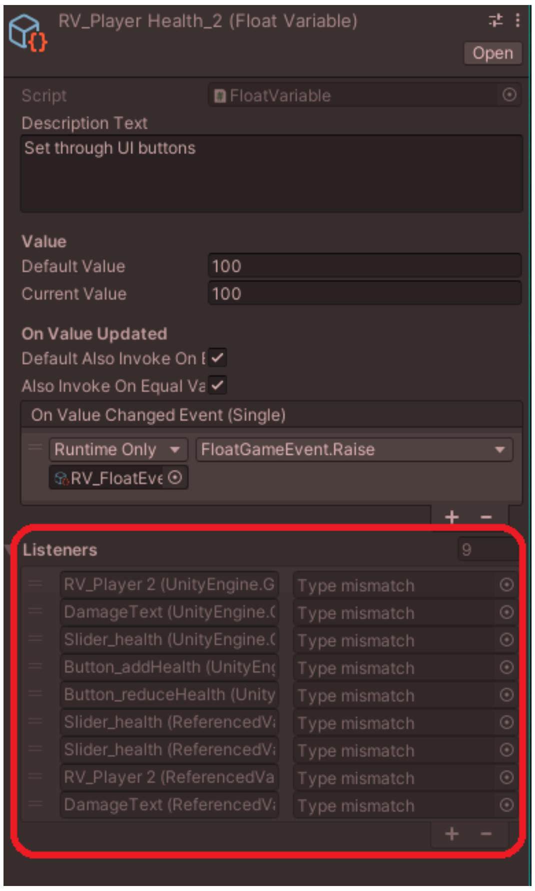
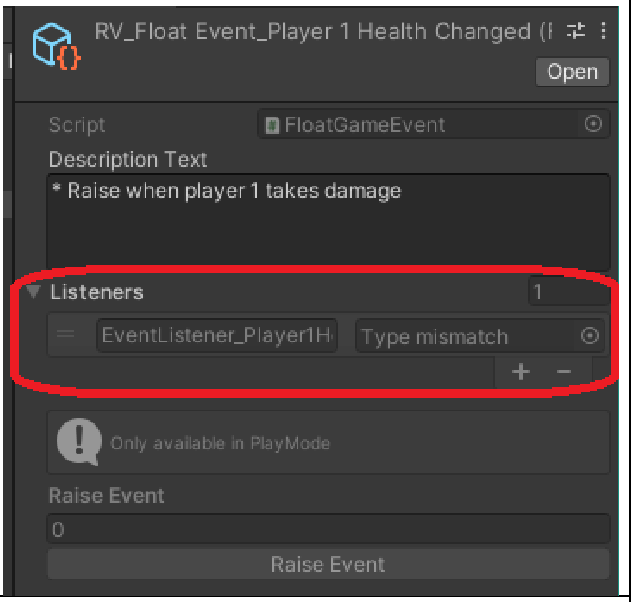
useEvent	Boolean to determine if the reference should subscribe to the ObservableHashSet 's change events.
OwningGameObject	The GameObject that owns this reference.

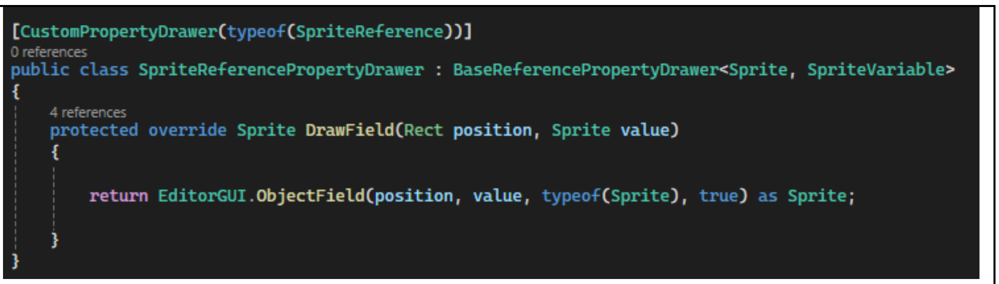
Methods

AddItem(T item, bool silent = false)	Adds an item to the referenced hash set. If silent is true, no events will be invoked.
Remove(T item, bool silent = false)	Removes an item from the set. If not silent and the item was removed, it raises the list changed event.
Clear(bool silent = false)	Clears the list. If not silent it raises the list changed event.
RaiseVariableEventWithCurrentValue()	Raises the list changed event with the current list value.
SetHashSet(V newSet)	Sets a new ObservableHashSet as the reference, unsubscribing from the old one and subscribing to the new one
Init()	Initializes the reference, subscribing to the set's events if useEvent is true.
Cleanup()	Unsubscribes from the set's events.

Troubleshooting

This asset is quite new and problems can occur. Here are some points that can help you.

Issue	**Possible Solutions**
The program freezes when a Referenced Value is set.	You might have created an infinite loop through the event systems. When you set a value of a Variable with reference.Value = X, the event triggers and causes other objects which reference the same Variable to fire their response events. When you set the value of the Variable in a response event again, you created an infinite loop of events. Have a look at your Variables. They have a list of current Listeners. Go through all of them and make sure none of them sets the Value in a response event. Also check your Game Events. They could also be creating infinite loops. The Listeners list on them can help you find such loops within listening objects.  RV_Player Health_2 (Float Variable) Open Script: FloatVariable Description Text: Set through UI buttons Value Default Value: 100 Current Value: 100 On Value Updated Default Also Invoke On E ✓ Also Invoke On Equal Va ✓ On Value Changed Event (Single) Runtime Only — FloatGameEvent.Raise RV_FloatEve Listeners: 9 RV_Player 2 (UnityEngine.G — Type mismatch DamageText (UnityEngine.(— Type mismatch Slider_health (UnityEngine.(— Type mismatch Button_addHealth (UnityEn(— Type mismatch Button_reduceHealth (Unity — Type mismatch Slider_health (ReferencedV: — Type mismatch Slider_health (ReferencedV: — Type mismatch RV_Player 2 (ReferencedVa — Type mismatch DamageText (ReferencedV: — Type mismatch  RV_Float Event_Player 1 Health Changed (Open Script: FloatGameEvent Description Text: * Raise when player 1 takes damage Listeners: 1 EventListener_Player1H — Type mismatch Only available in PlayMode Raise Event: 0 Raise Event
After creating new Variables through the Variable Generator Window I get compile errors in "{variableType} ReferencePropertyDrawer"	It is difficult to generate these property drawers automatically for all possible types. Most property drawers will use a EditorGUI.ObjectField like a Sprite for example:



```
[CustomPropertyDrawer(typeof(SpriteReference))]
0 references
public class SpriteReferencePropertyDrawer : BaseReferencePropertyDrawer<Sprite, SpriteVariable>
{
    4 references
    protected override Sprite DrawField(Rect position, Sprite value)
    {

        return EditorGUI.ObjectField(position, value, typeof(Sprite), true) as Sprite;

    }
}
```

However, depending on the type this field can also be something else and it can happen that we need to use some other field for our custom types.

Going through the already existing Property Drawers in "Assets/ReferencedVariables/Scripts/Editor/VariablePropertyDrawers/" can help you find the field you need. Otherwise it is best to have look at the Unity Documentation: ***https://docs.unity3d.com/ScriptReference/EditorGUI.html***

Contact

You can also always reach out to me per Email should you face troubles: ***tobias.froihofer@gmx.at***

Please consider leaving a review on the Unity Asset Store. This helps the tool to gain visibility and in return allows me to further improve the asset.