

EVENTS 2.0

FOR

UNITY

1. Overview	3
2. Events 2.0	3
a. Multiple parameters	3
b. Parameter name	3
c. Reorder events	4
d. Enum	4
e. Vector2, Vector3 and Vector4	4
f. Color	5
g. LayerMask	5
h. Layer	6
i. Quaternion	6
j. Generic Lists and Arrays	7
k. Slider and IntSlider	7
l. Tag	9
m. Delete all events	10
3. Dynamic Arguments	11
4. UI 2.0	13
a. TextMeshPro	17
5. Event Trigger 2.0	19
6. IL2CPP Workaround	22

1. Overview

Thanks for purchasing **Events 2.0 for Unity**. It's almost like the current **Unity Event**, but with a few upgrades. With it, the possibilities of improving your game are tremendous. Check the **Sample Scene** under **Assets/Gabriel Pereira/Events 2.0 for Unity/Examples**.

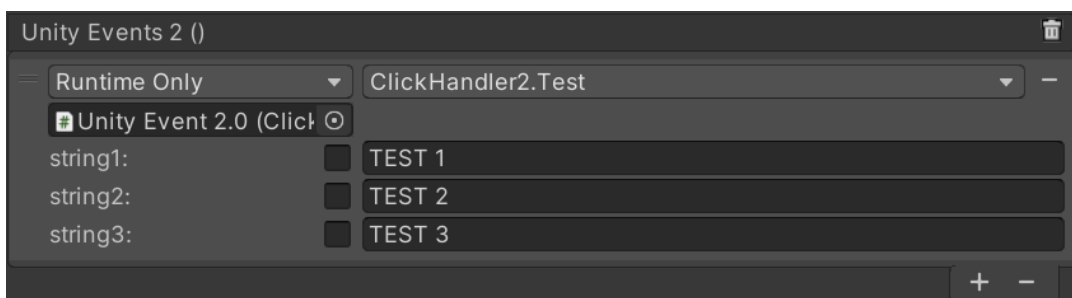
This package was created using Unity **2017.1.0f3**, so please make sure to use this version or later.

2. Events 2.0

This package is based on the current Unity Event and the upgrades are listed below.

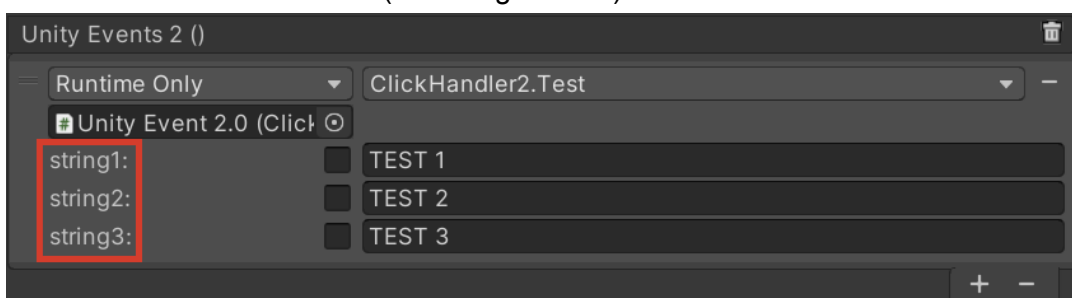
a. Multiple parameters

You can set multiple parameters in a method (see image below).



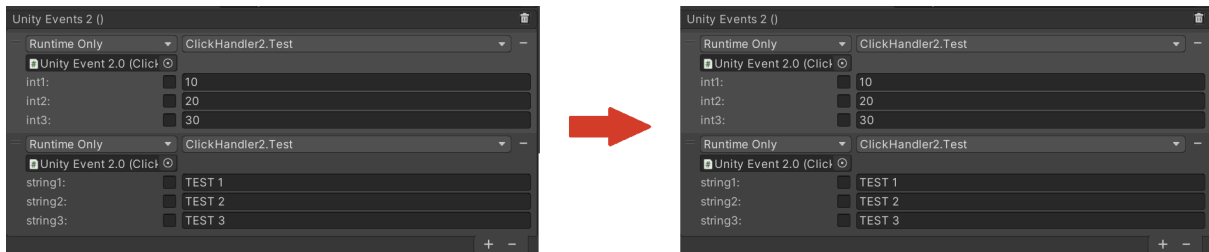
b. Parameter name

Parameter name is visible (see image below).



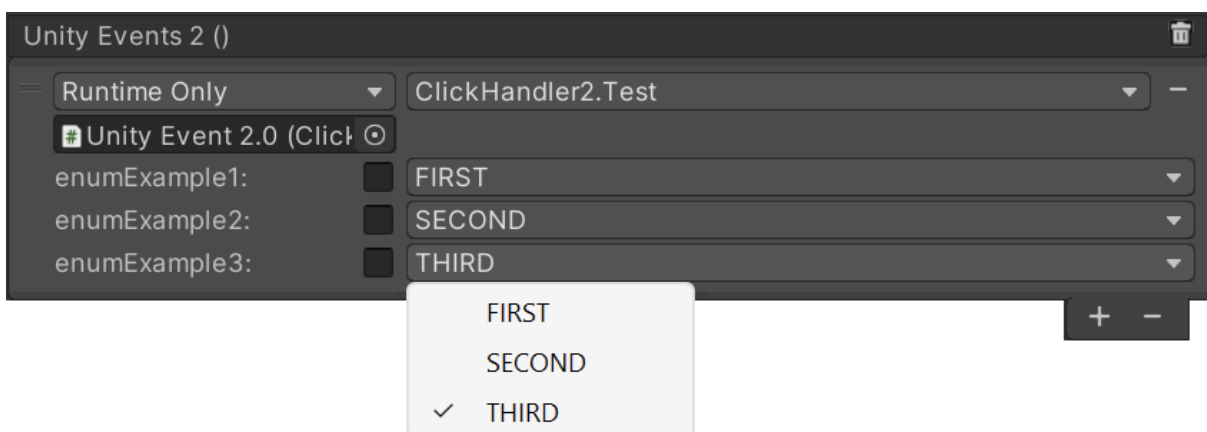
c. Reorder events

You can reorder events. For that, just click and drag to the position you need (see image below)



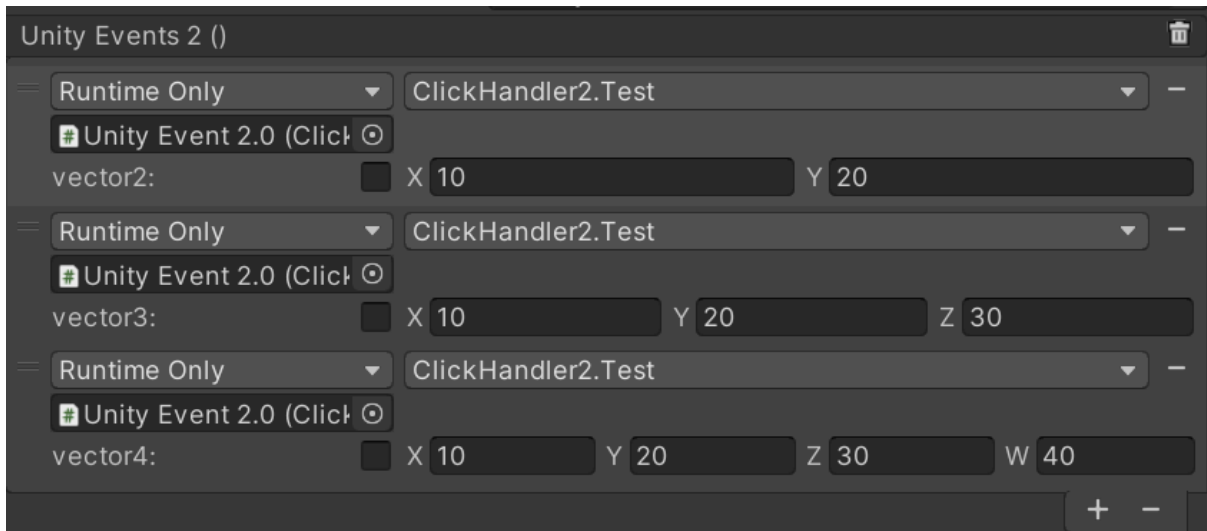
d. Enum

It is possible to use enum parameters (see image below).



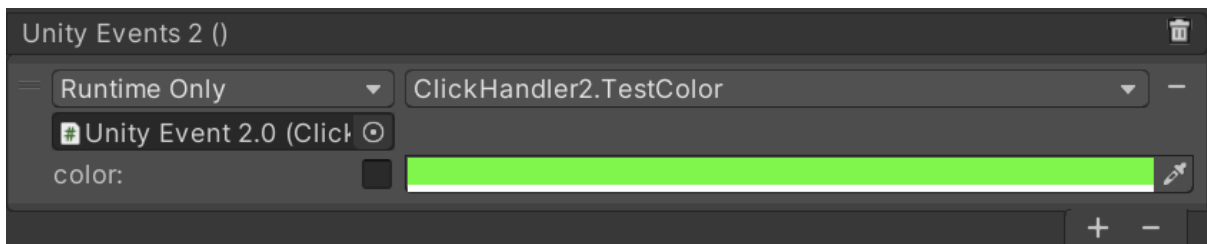
e. Vector2, Vector3 and Vector4

It's possible now to inform a Vector2, Vector3 or Vector4 parameter (see image below)



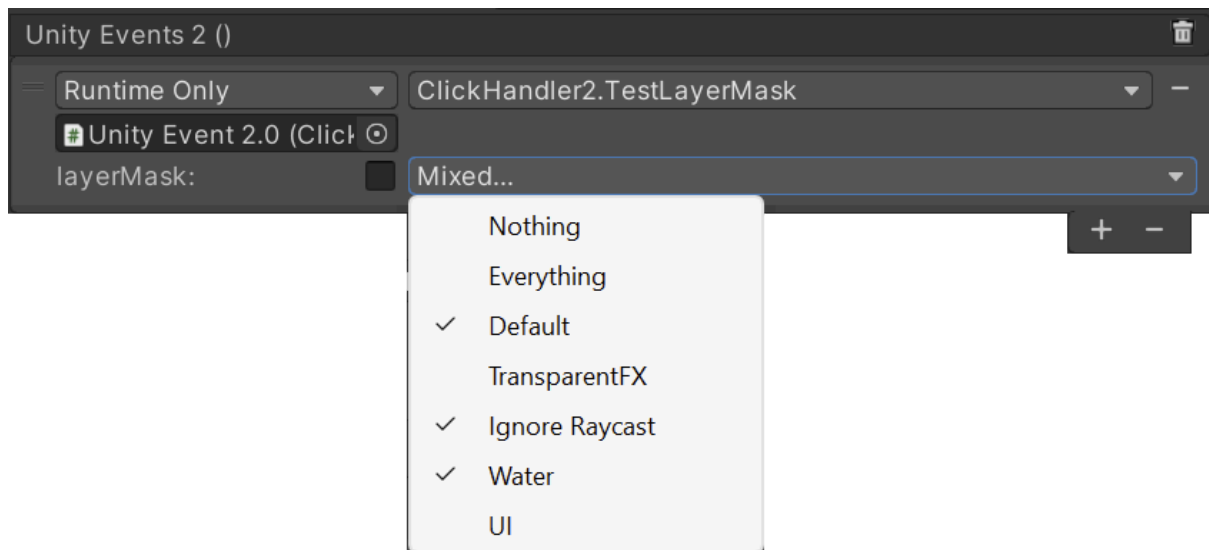
f. Color

Now, you can inform a color through a parameter (see image below)



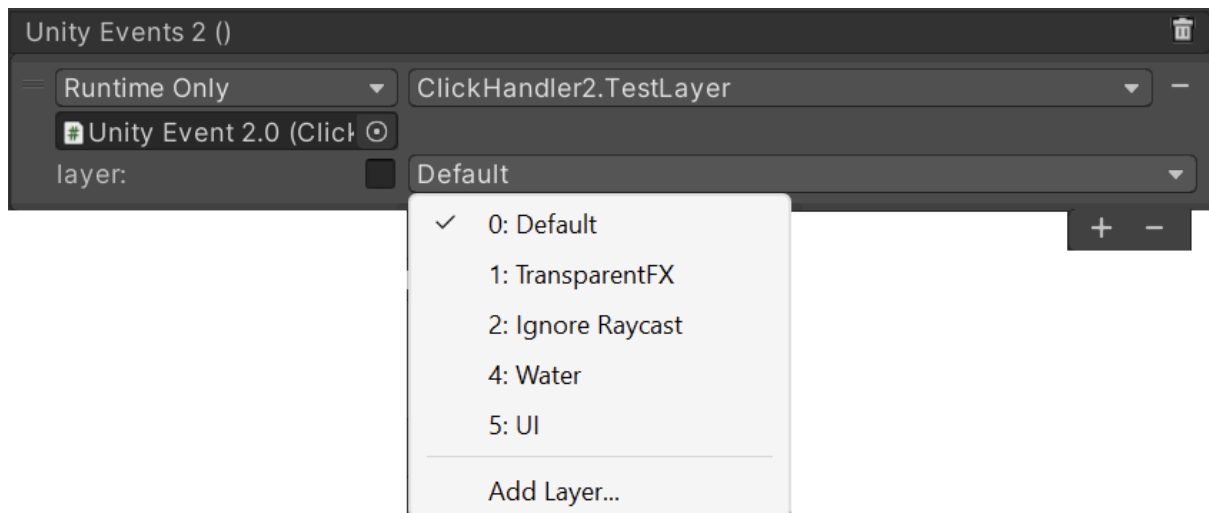
g. LayerMask

You can inform a LayerMask parameter (see image below)



h. Layer

Instead of informing a Layer by its index or name, you can inform via a popup. Check the method TestLayer in ClickHandler2.cs script (see image below)



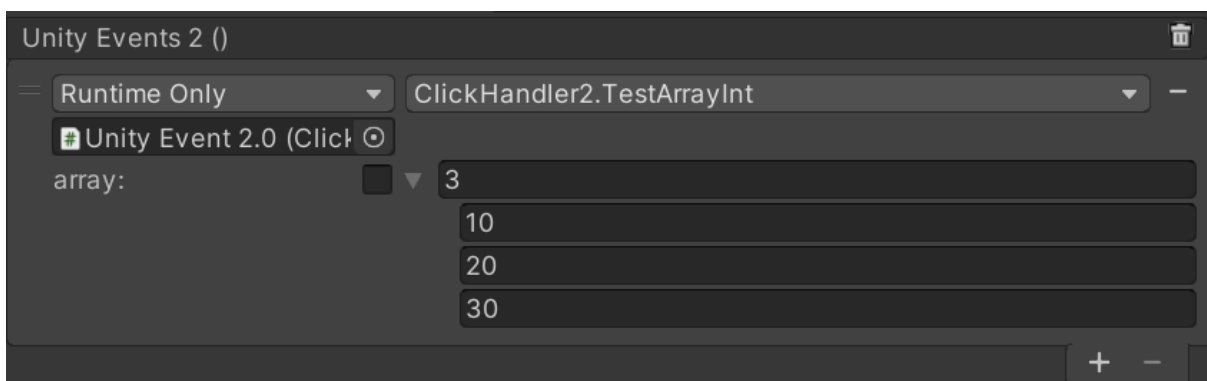
i. Quaternion

You can inform a Quaternion parameter. It'll be presented as a Vector4 (see image below).



j. Generic Lists and Arrays

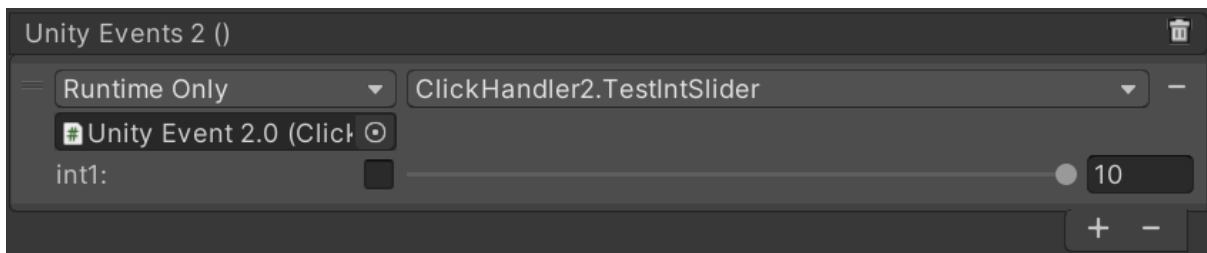
You can inform a Generic List or Array of types above (see image below).



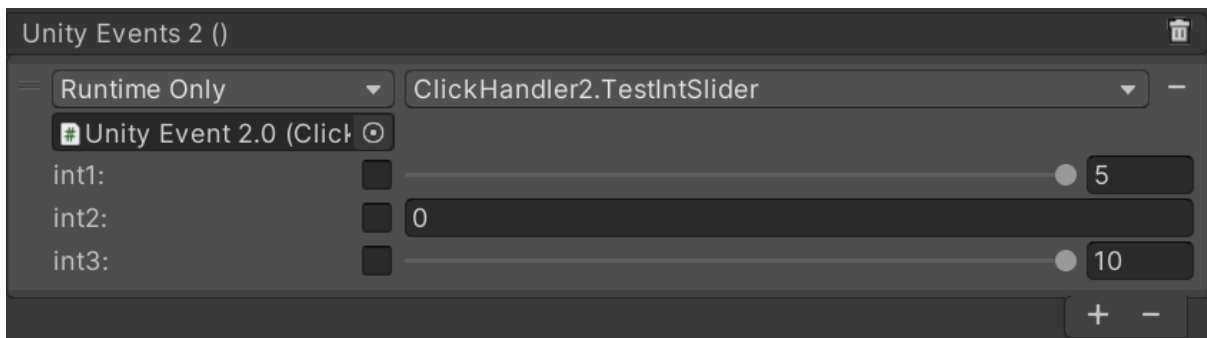
k. Slider and IntSlider

You can turn a float or integer parameter into a slider. All you have to do is add the Slider (for floats) or IntSlider (for integers) at the method or parameter.

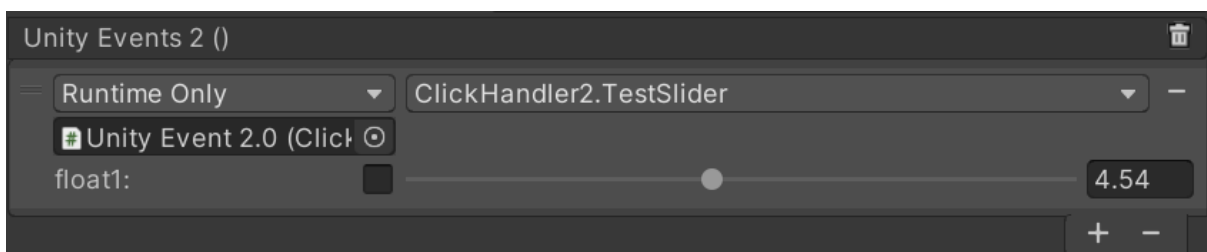
```
[IntSlider(0, 10, 5)]
public void TestIntSlider(int int1)
{
    Debug.LogFormat("TestIntSlider method. int param:{0}", int1);
}
```



```
public void TestIntSlider([IntSlider(0, 5, 2)] int int1, int int2, [IntSlider(0, 10, 3)] int int3)
{
    Debug.LogFormat("TestIntSlider method. int param1:{0} - int param2:{1} - int param2:{2}", int1, int2, int3);
}
```



```
[Slider(0f, 10f, 4.54f)]
public void TestSlider(float float1)
{
    Debug.LogFormat("TestSlider method. float param:{0}", float1);
}
```



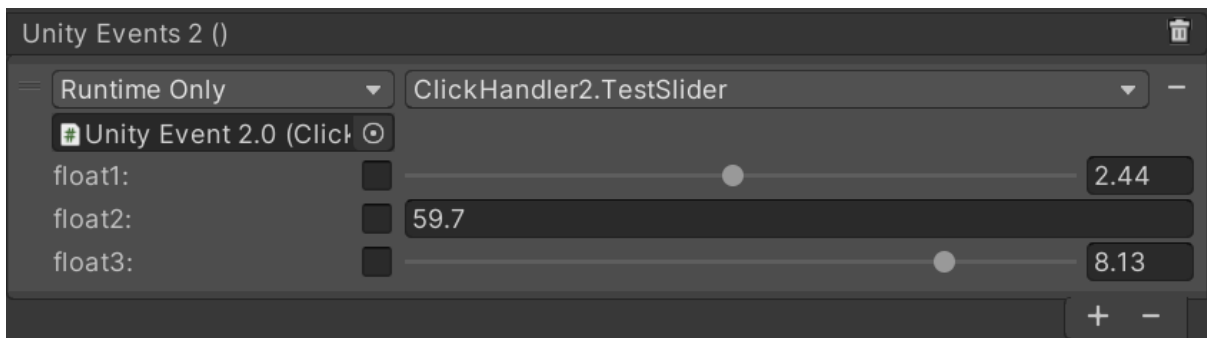
```
public void TestSlider([Slider(0f, 5f, 2.44f)] float float1, float float2, [Slider(0f, 10f, 8.13f)] float float3)
{
}
```



```

        Debug.LogFormat("TestSlider method. float param1:{0} - float param2:{1} - float param2:{2}", float1, float2, float3);
    }

```



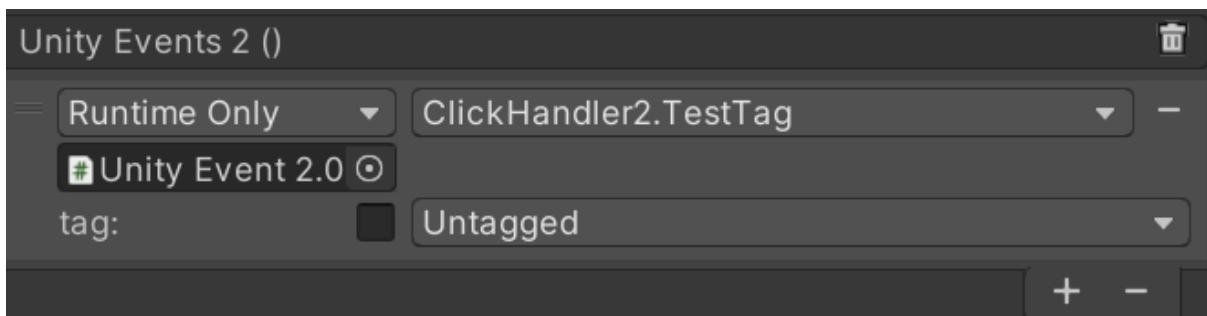
I. Tag

You can turn a string parameter into a tag field. All you have to do is add the Tag attribute at the method or parameter.

```

[Tag] // If no tag is specified, by default the "Untagged" value is selected
public void TestTag(string tag)
{
    Debug.LogFormat("TestTag method. tag param:{0}", tag);
}

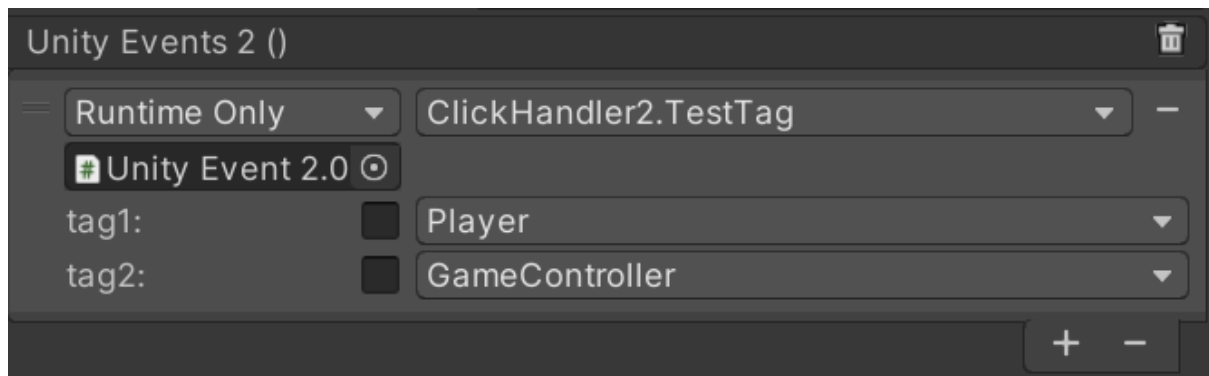
```



```

[Tag("Player")]
public void TestTag(string tag1, [Tag("GameController")] string tag2)
{
    Debug.LogFormat("TestTag method. tag1 param:{0}, tag2 param:{1}", tag1, tag2);
}

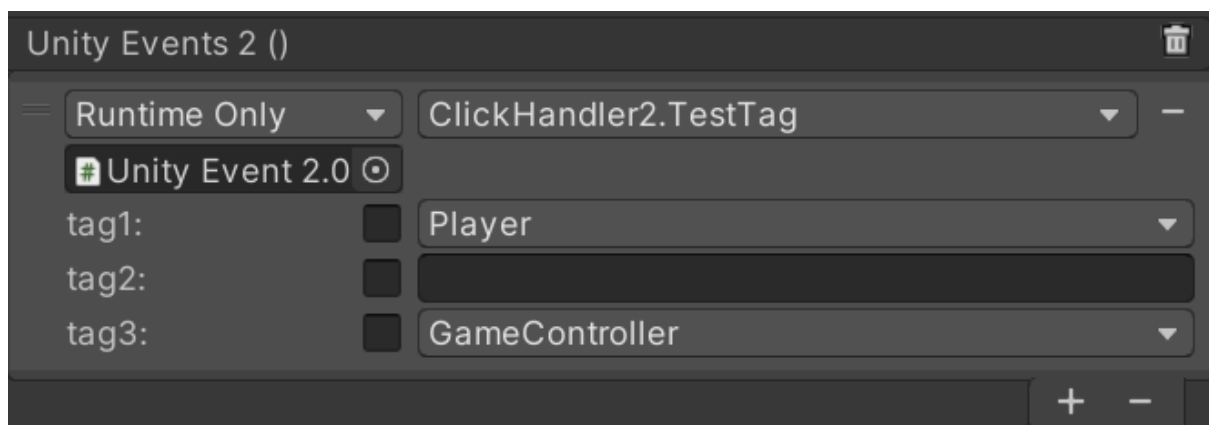
```



```

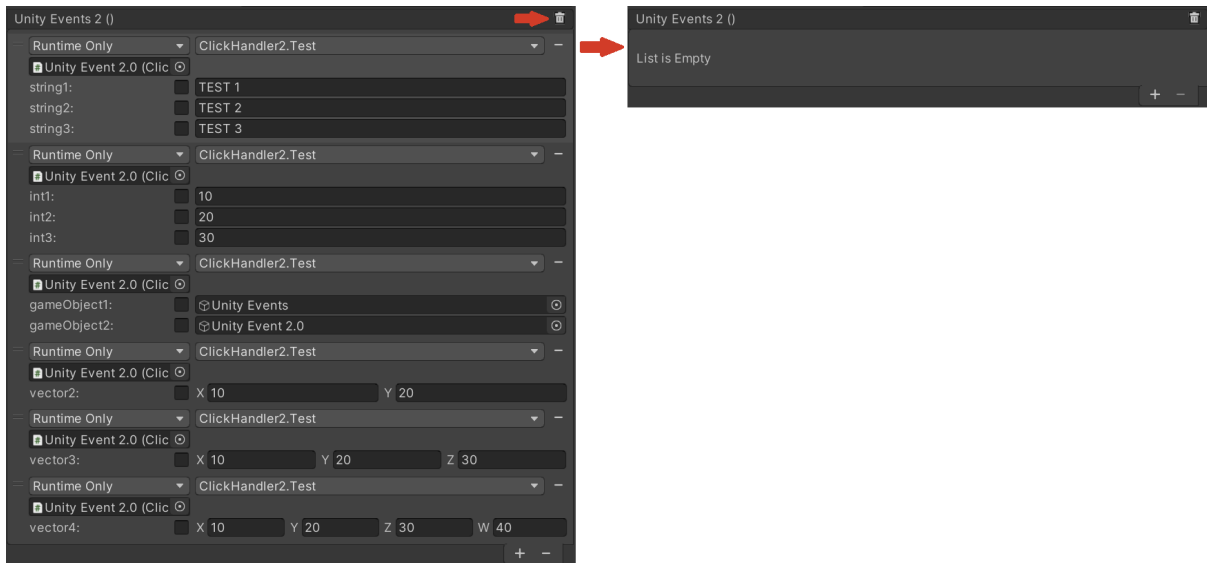
public void TestTag([Tag("Player")] string tag1, string tag2, [Tag("GameController")]
string tag3)
{
    Debug.LogFormat("TestTag method. tag1:{0}, tag2:{1}, tag3:{2}", tag1, tag2,
tag3);
}

```



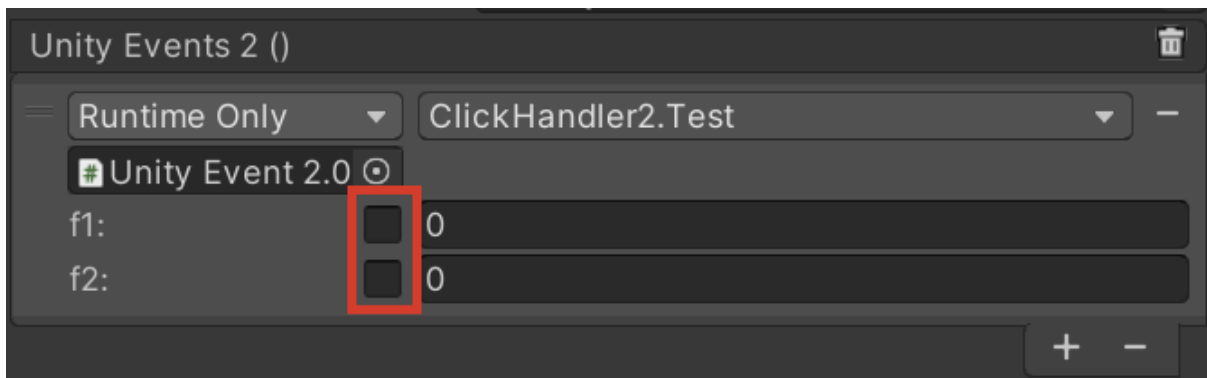
m. Delete all events

In case you have a lot of events and want to delete them all, just click on the trash can to the upper right (see Image below)

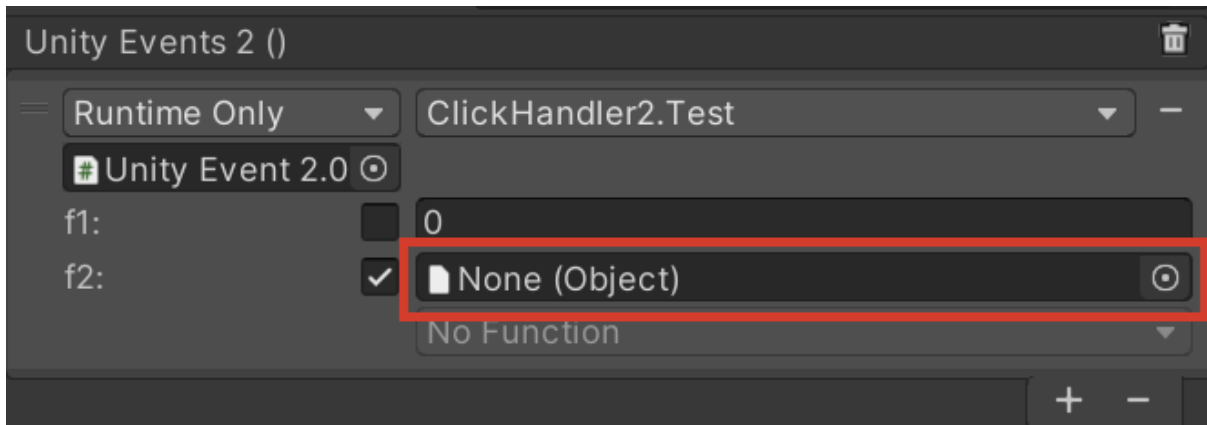


3.Dynamic Arguments

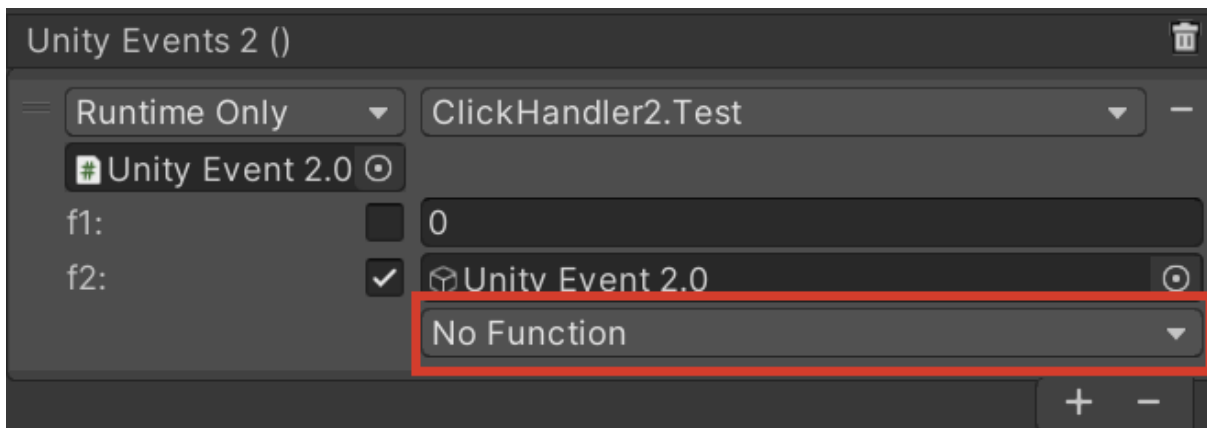
It's possible to specify a dynamic argument for a method by checking the checkbox next to an argument's value field (see image below).



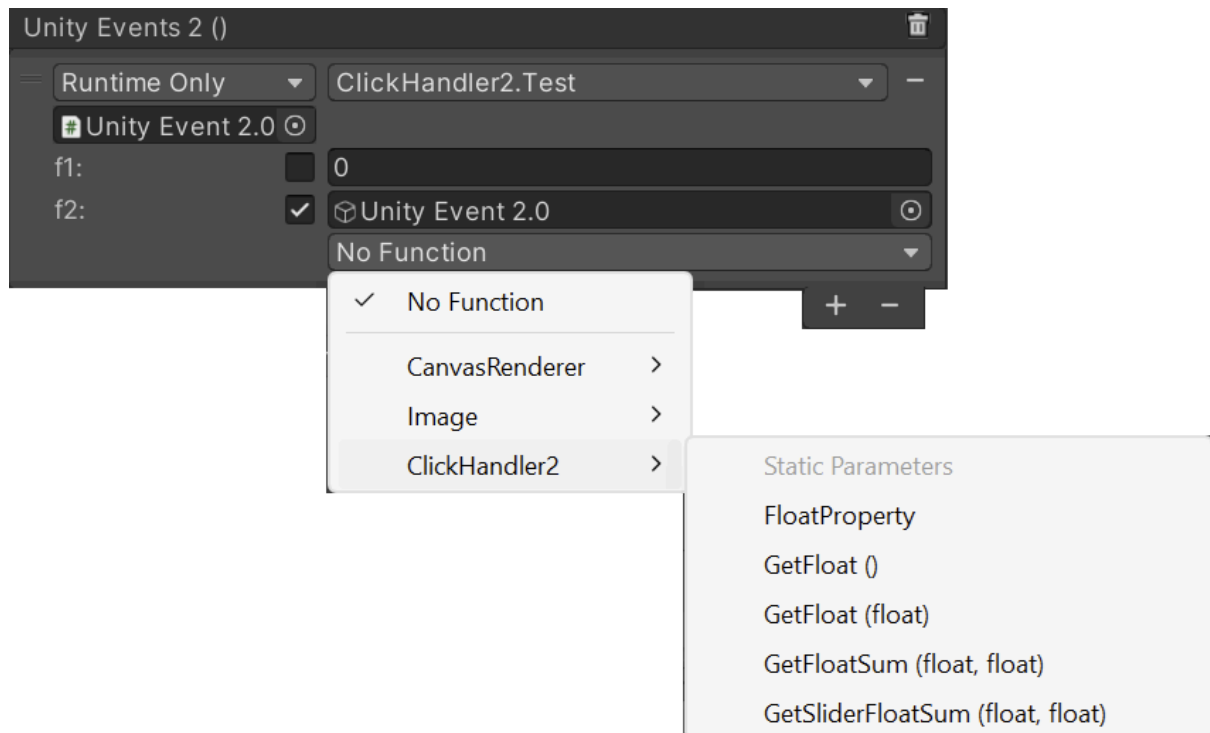
After checking the checkbox, an Object field will be displayed (see image below).



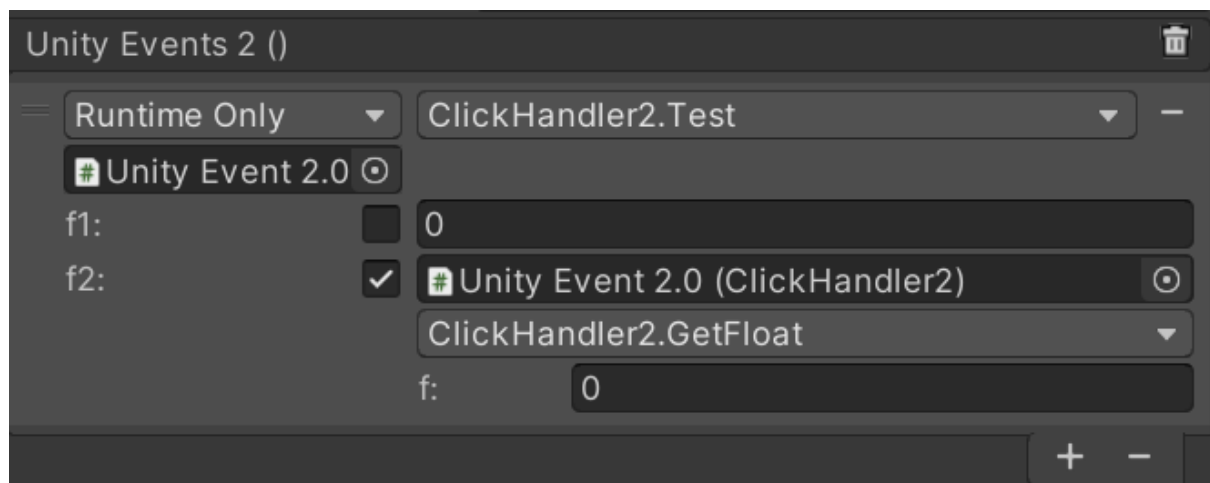
After dragging a GameObject from the Hierarchy panel to the field displayed above, the function will be enabled for selection.



All functions and properties available for selection will be dependent on the parameter type (e.g. The **f2** parameter on image above is a **float** type, so the available functions are the ones with return type of **float**).



After selecting a function, here's how it'll look.



Now, every time the **Test** function from **ClickHandler2** class is triggered, the **GetFloat** function will be triggered in order to pass its return value to **f2** parameter.

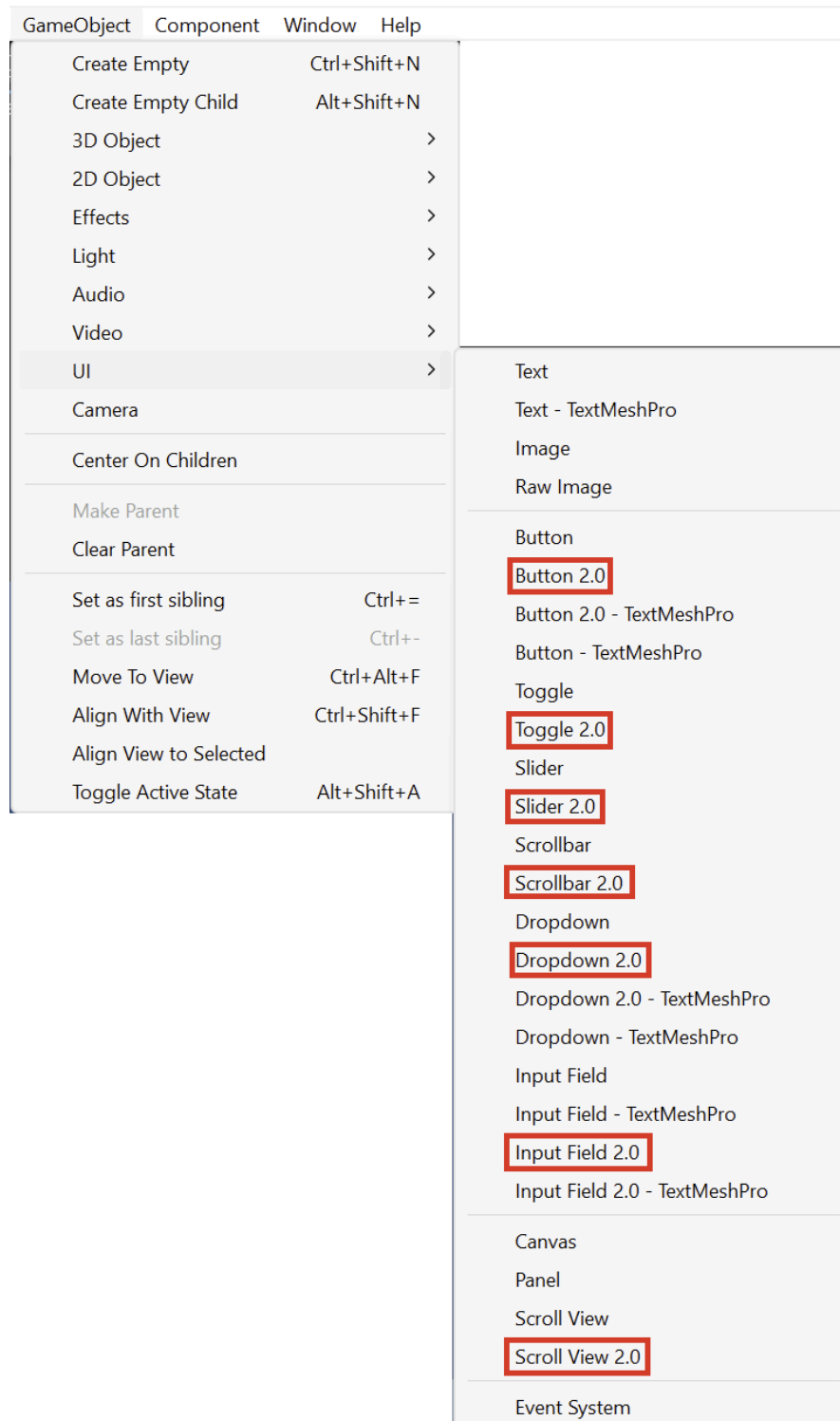
4.UI 2.0

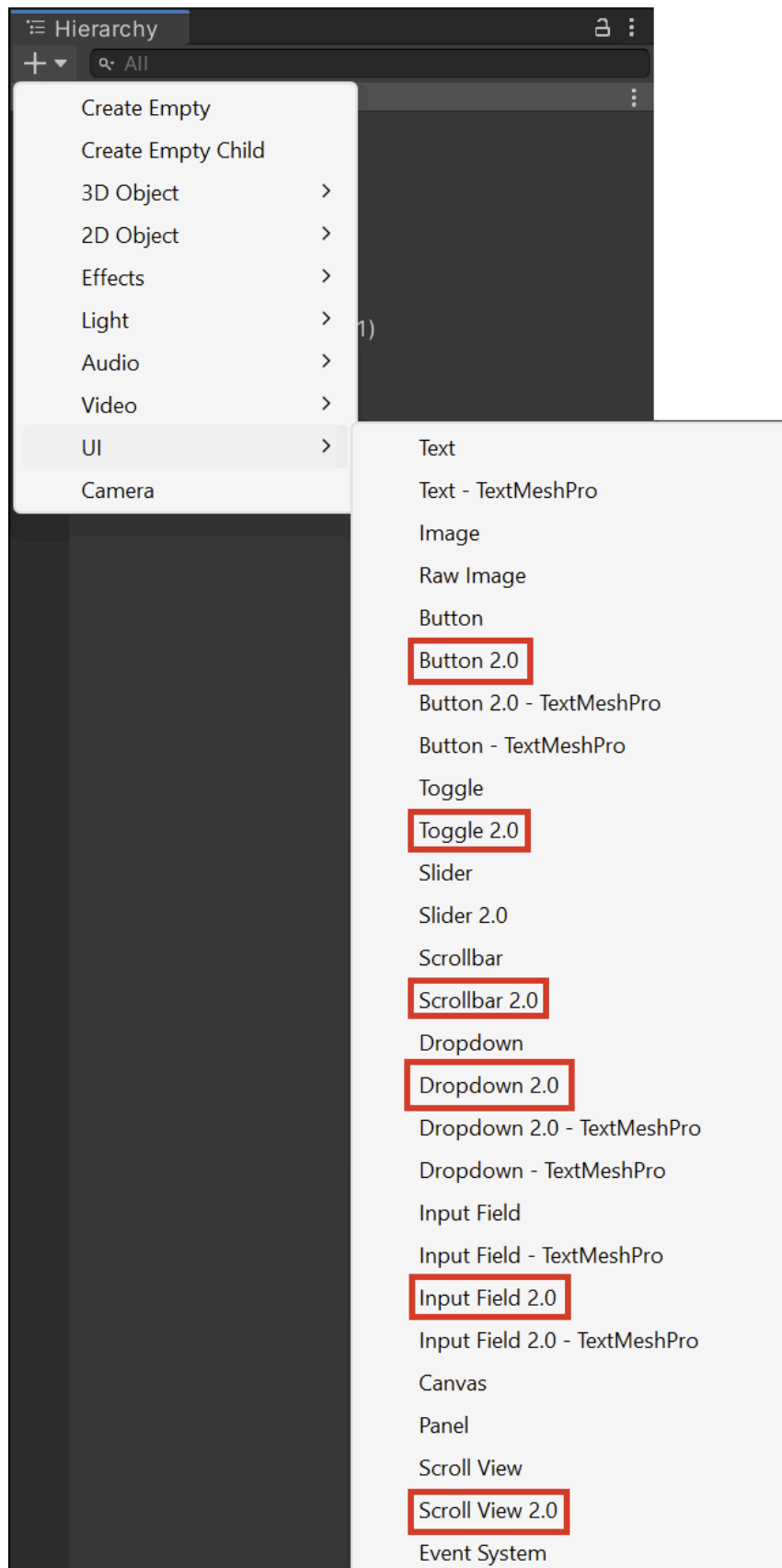
The Unity's UI components were upgraded as well to use **Events 2.0**. Every component has its "component 2.0", such as:

- Button > Button 2.0

- Toggle > Toggle 2.0
- Slider > Slider 2.0
- Etc

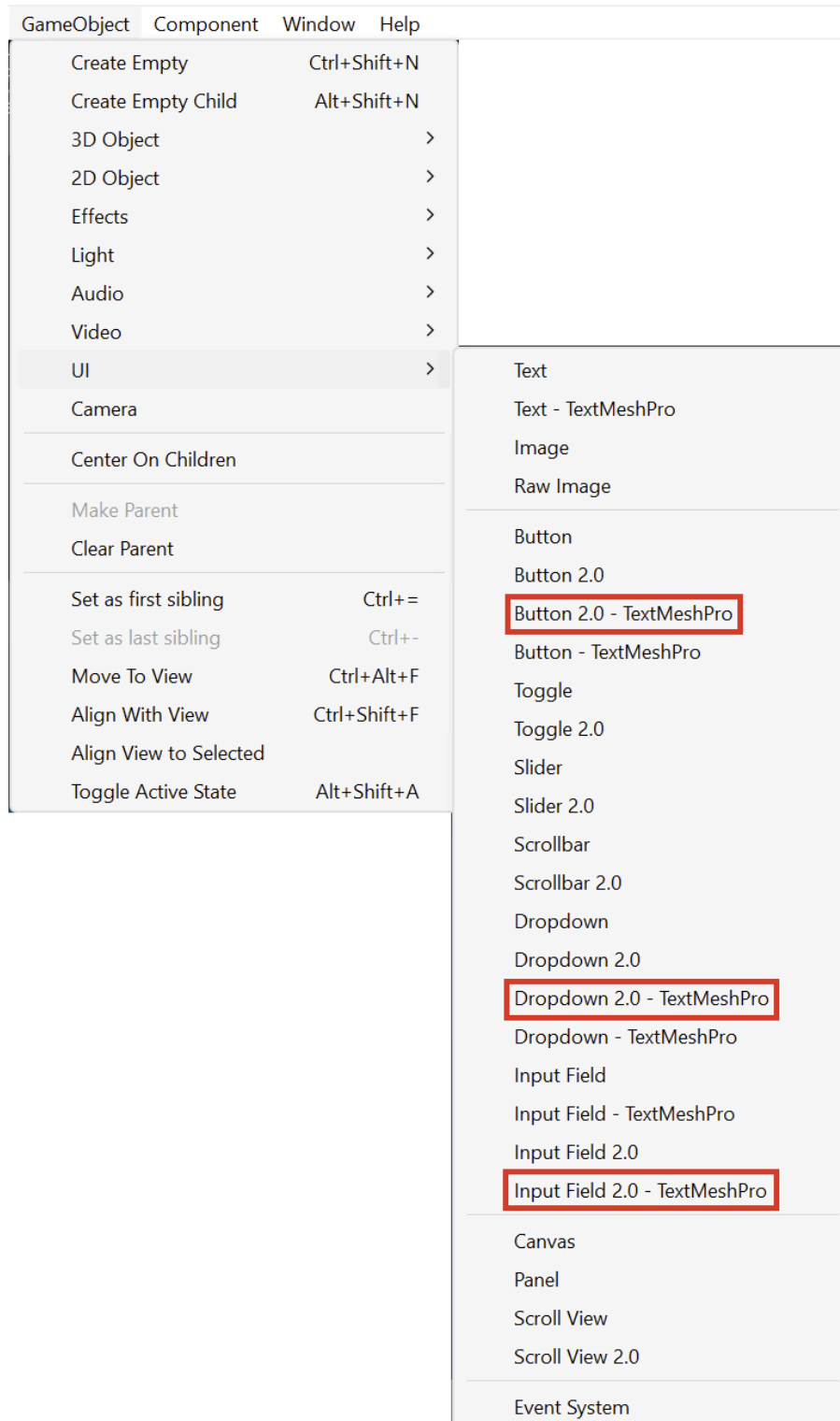
You can access these components in the **GameObject > UI** menu or just go to **Create > UI** menu in the Hierarchy window (see images below). For more information on these components, check the **Sample UI Scene** under **Assets/Gabriel Pereira/Events 2.0 for Unity/Examples**.

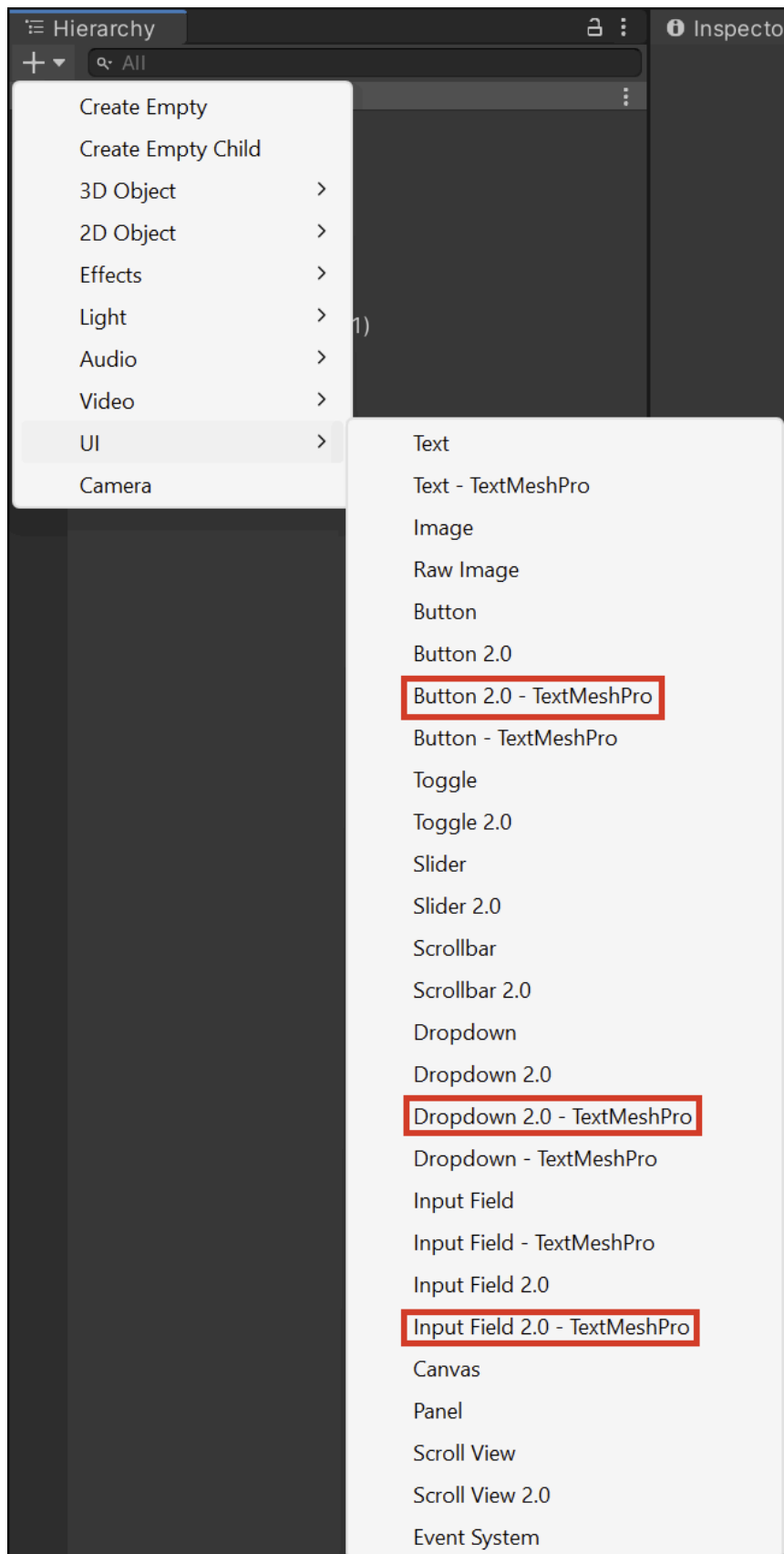




a. TextMeshPro

It's possible to add UI 2.0 elements with TextMeshPro. You can access these components the same way as regular UI 2.0 elements.

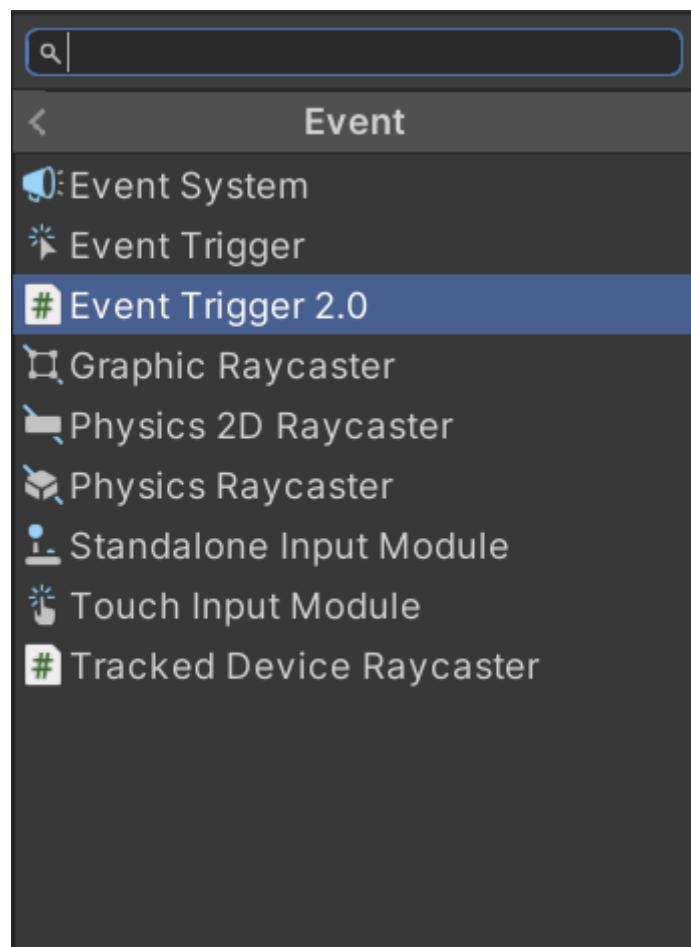




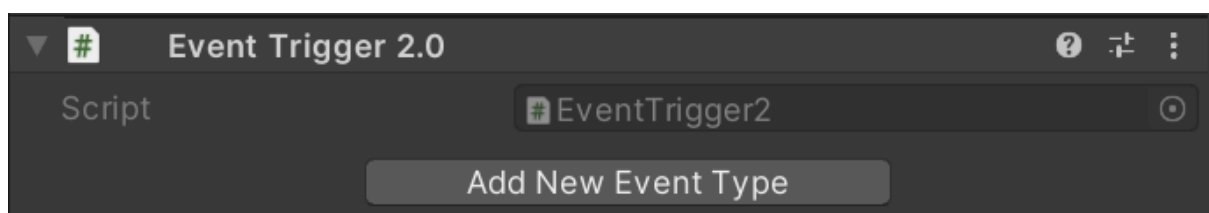
5.Event Trigger 2.0

With Event Trigger 2.0, you can add some predetermined events, like **PointerDown**, **PointerClick**, **Select**, etc.

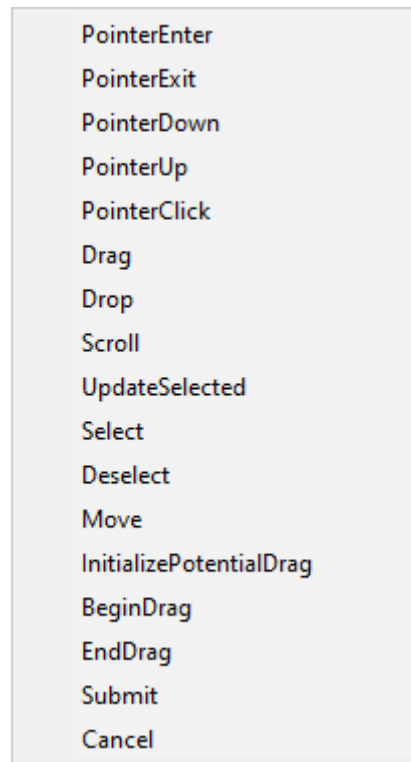
To do that, select the desired component, click on the **Add Component** button, go to **Event** and click on the **Event Trigger 2.0** component (see image below).



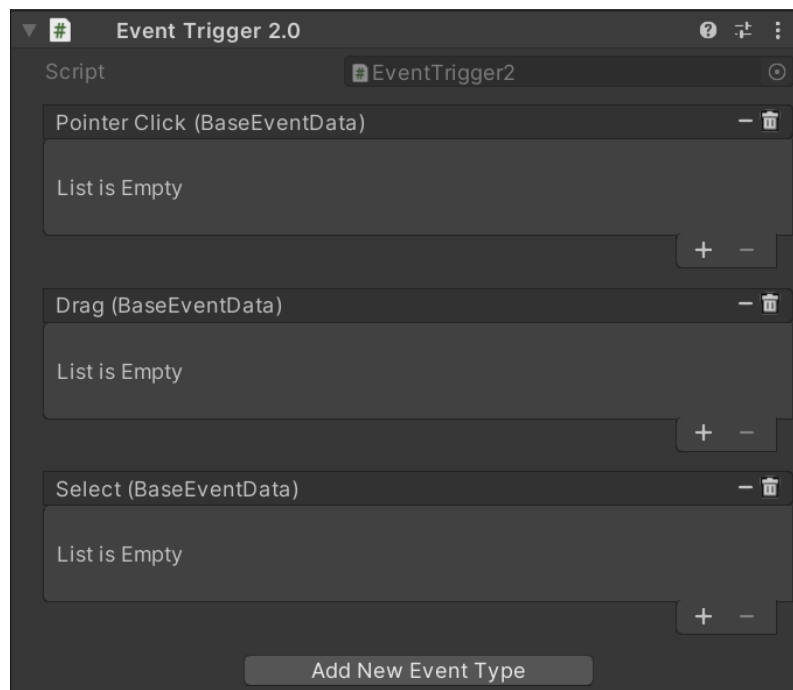
After adding, it'll look like the image below (see image below).



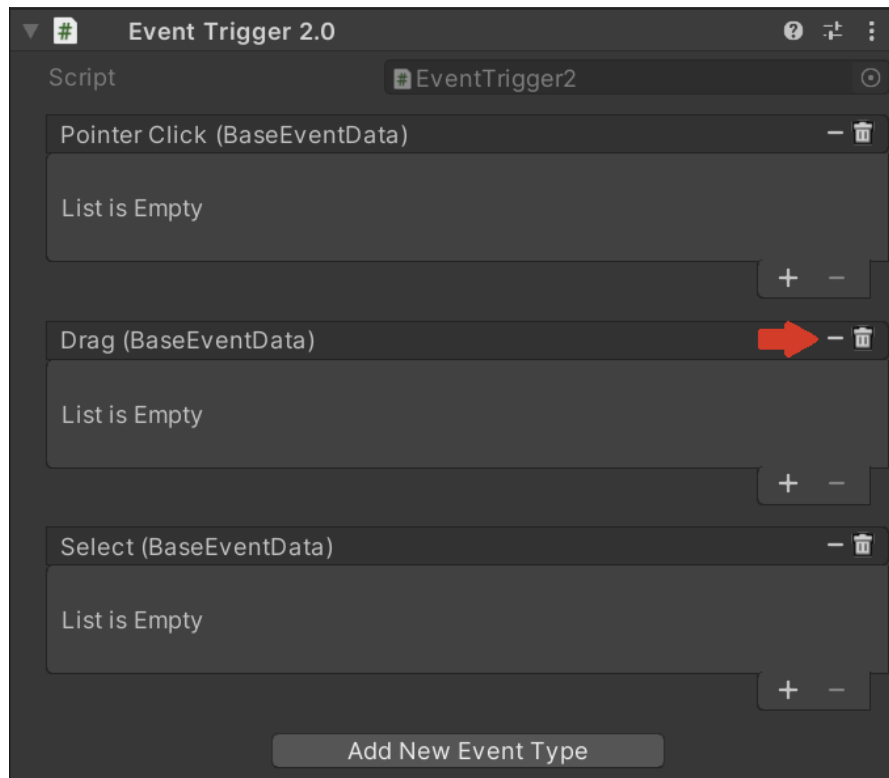
Click on the **Add New Event Type** button to visualize all the possible events that can be added to the GameObject (See image below).



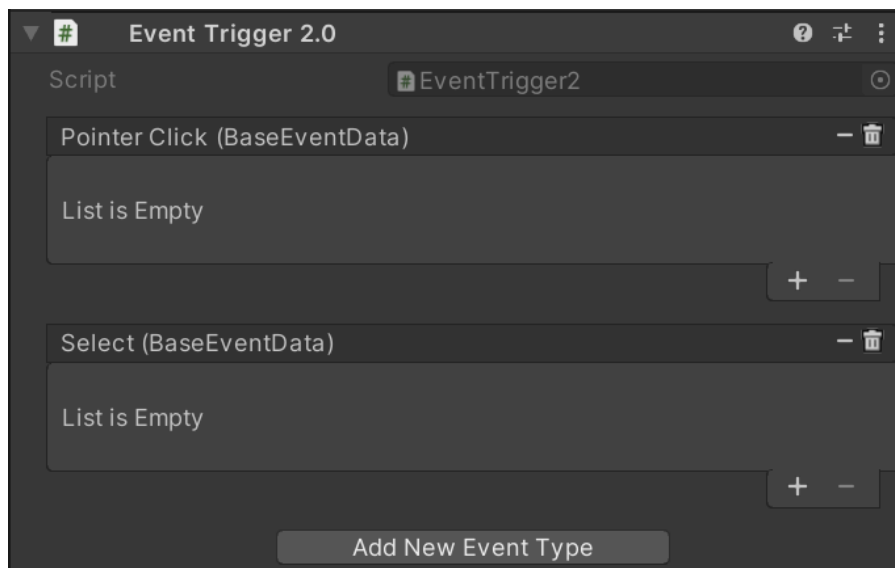
After selecting the desired events (e.g. **PointerClick**, **Drag** and **Select**), your GameObject will show those events on the Inspector (see image below).



When you click on the **Minus** icon, the event will be deleted (see images below).



Before delete



After delete

6. IL2CPP Workaround

In order to use the **IL2CPP Scripting Backend**, there are a few more steps to take. Let's take the **ExampleClass** below as an example.

```
using UnityEngine;
public class ExampleClass : MonoBehaviour
{
    public void ExampleMethod(int i, string s, float f, GameObject go)
    {
        Debug.LogFormat("Int:{0}", i);
        Debug.LogFormat("String:{0}", s);
        Debug.LogFormat("Float:{0}", f);
        Debug.LogFormat("GameObject:{0}", go);
    }
}
```

In order to trigger **ExampleMethod** from an **UnityEvent2** object, you'll have to create an **unused** attribute of type like below:

```
private  UnityEngine.Events.UpdatableInvokableCall<int,  string,  float,  GameObject>
_unused;
```

So, after creating the attribute, here's how the class will look like:

```

using UnityEngine;
using UnityEngine.Events;
public class ExampleClass : MonoBehaviour
{
    private UpdatableInvokableCall<int, string, float, GameObject> _unused;
    public void ExampleMethod(int i, string s, float f, GameObject go)
    {
        Debug.LogFormat("Int:{0}", i);
        Debug.LogFormat("String:{0}", s);
        Debug.LogFormat("Float:{0}", f);
        Debug.LogFormat("GameObject:{0}", go);
    }
}

```

If you already created an **unused** attribute with your desired generic types, you don't need to create another with the same generic types.

To help maintaining the unused attributes, an **UnusedClass** can be created to store them, like example below:

```

using UnityEngine;
using UnityEngine.Events;
public class UnusedClass
{
    private UpdatableInvokableCall<int, string, float, GameObject> _unused0;
    private UpdatableInvokableCall<int, int> _unused1;
    private UpdatableInvokableCall<string, GameObject> _unused2;

    // Don't need to create attribute below, since _unused1 attribute already has the
    // same generic types
    private UpdatableInvokableCall<int, int> _unused3;
}

```