EVENTS 2.0

FOR

UNITY

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**.

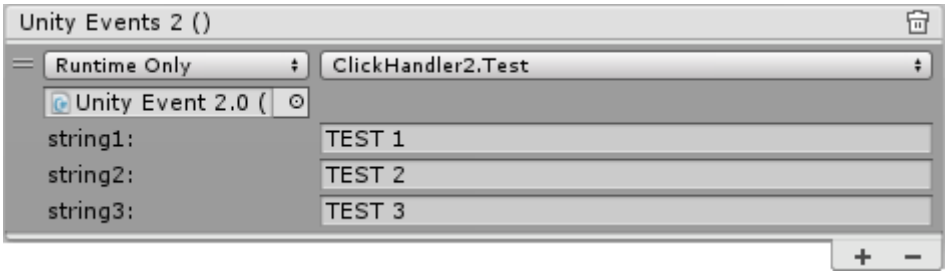
This package was created using Unity **2017.1.0f3**, so please make sure to use this version or later.

2. Events 2.0

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

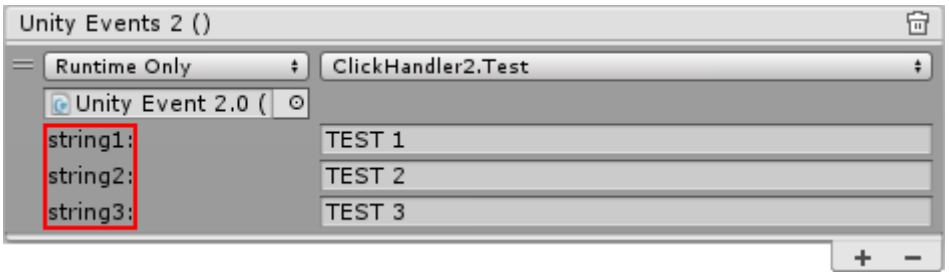
a. Multiple parameters

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



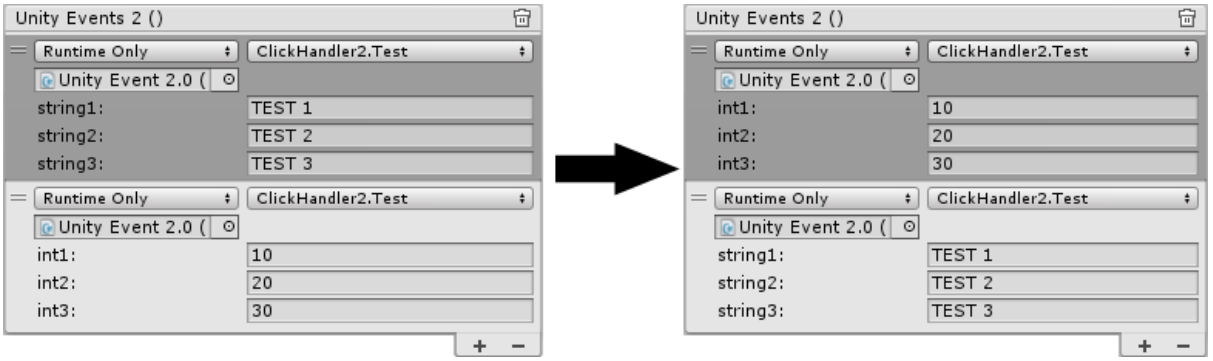
b. Parameter name

Parameter name is visible (see image below).



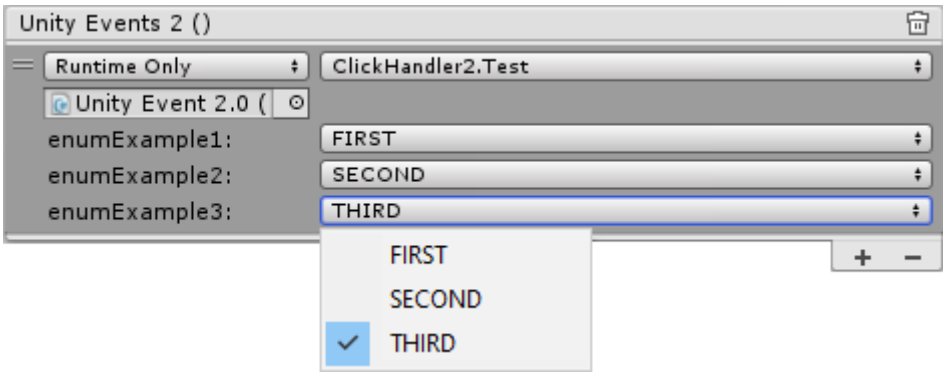
c. Reorder events

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



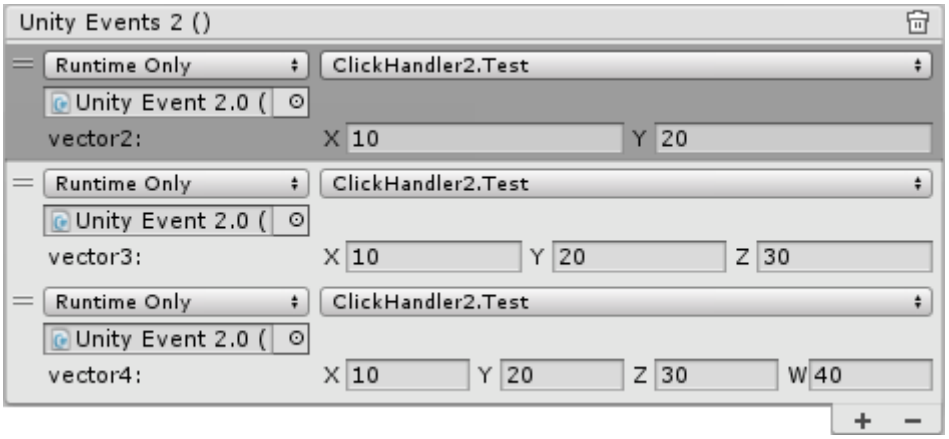
d. Enum

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



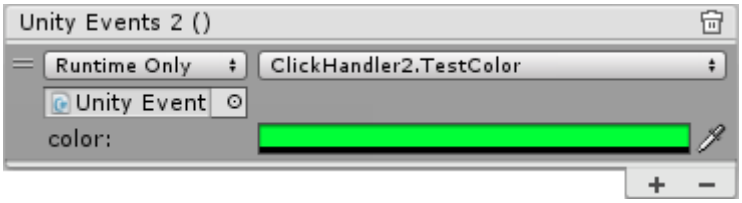
e. Vector2, Vector3 and Vector4

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



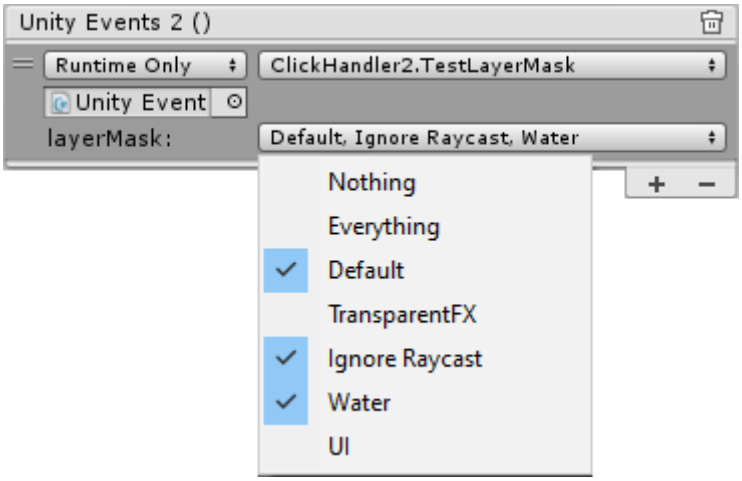
f. Color

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



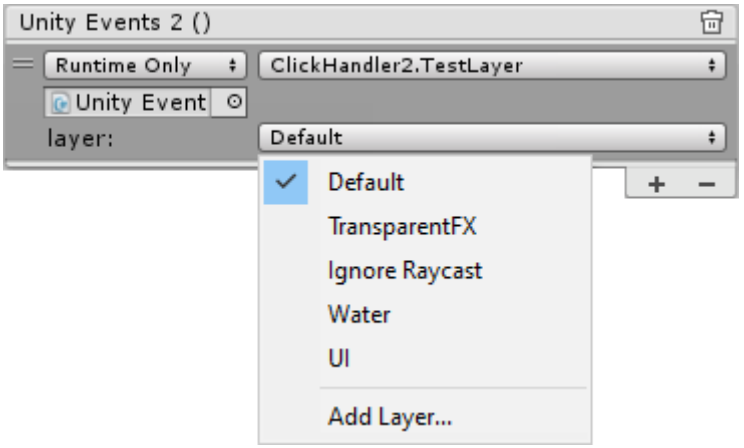
g. LayerMask

You can inform a LayerMask parameter (see image below)



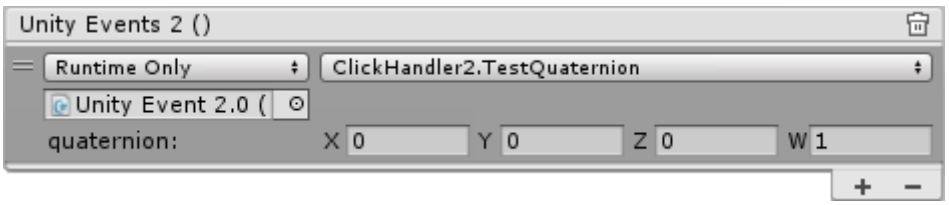
h. Layer

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)



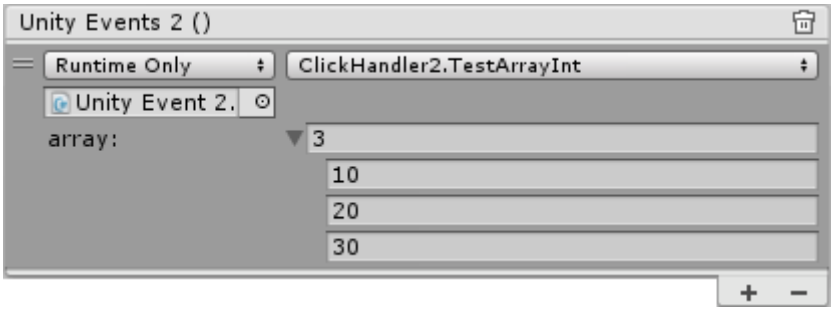
i. Quaternion

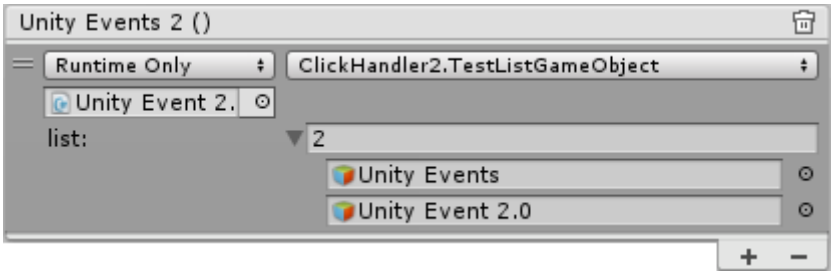
You can inform a Quaternion parameter. It'll be presented as a Vector4 (see image below).



j. Generic Lists and Arrays

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

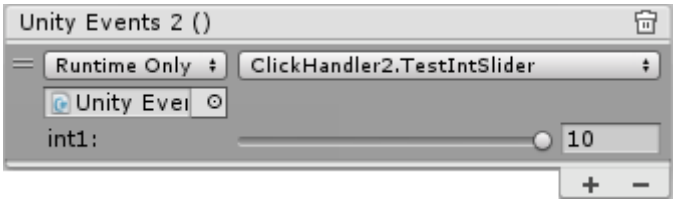




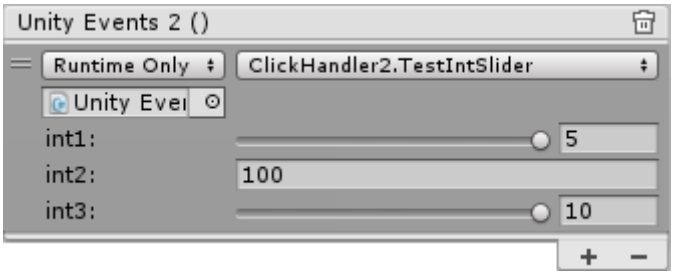
k. Slider and IntSlider

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

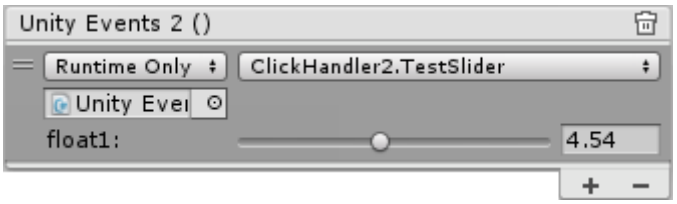



```
public void TestIntSlider([IntSlider(0, 5)] int int1, int int2, [IntSlider(0, 10)] int int3)
{
    Debug.LogFormat("TestIntSlider method. int param1:{0} - int param2:{1} - int param2:{2}", int1, int2, int3);
}
```

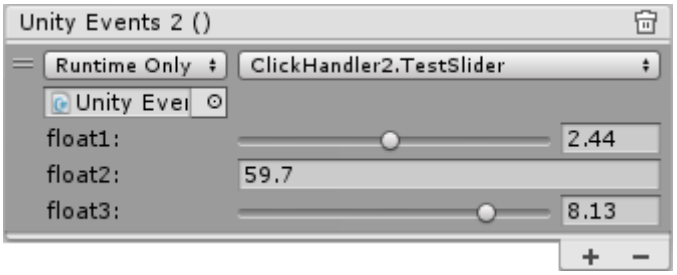



```
[Slider(0f, 10f)]
```

```
public void TestSlider(float float1)
{
        Debug.LogFormat("TestSlider method. float param:{0}", float1);
}
```




```
public void TestSlider([Slider(0, 5)] float float1, float float2, [Slider(0f, 10f)] float float3)
{
        Debug.LogFormat("TestSlider method. float param1:{0} - float param2:{1} - float param2:{2}", float1, float2, float3);
}
```



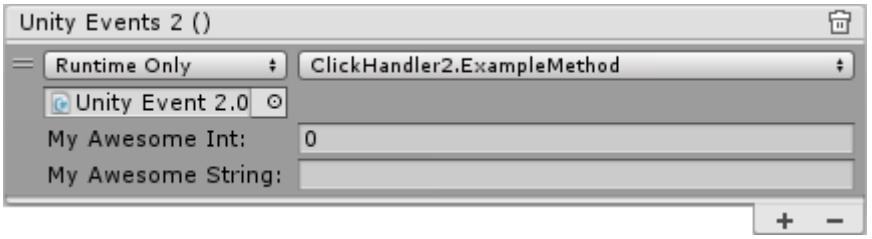
I. Custom Inspector

You can create a custom inspector for your method. Let's take the **ExampleMethod** below.

```
[CustomInspector("ExampleMethodCustomInspector")]
public void ExampleMethod(int intParam, string stringParam)
{
        Debug.LogFormat("Int:{0} - String{1}", intParam, stringParam);
}

public void ExampleMethodCustomInspector(SerializedProperty arguments, Rect argNameRect, Rect argRect)
{
```

```
            // Here, we get the first argument (aka intParam) from ExampleMethod
            var intParam = arguments.GetArrayElementAtIndex(0);

            // Since intParam is an integer,
            // we need to get the value through m_IntArgument field
            intParam = intParam.FindPropertyRelative("m_IntArgument");

            EditorGUI.LabelField(argNameRect, new GUIContent("My Awesome Int:",
"My awesome integer argument"));
            EditorGUI.PropertyField(argRect, intParam, GUIContent.none);

            // DON'T FORGET TO INCREASE THE Y AXIS OF BOTH RECTS
            argNameRect.y += EditorGUIUtility.singleLineHeight +
EditorGUIUtility.standardVerticalSpacing;
            argRect.y += EditorGUIUtility.singleLineHeight +
EditorGUIUtility.standardVerticalSpacing;

            // Here, we get the second argument (aka stringParam) from ExampleMethod
            var stringParam = arguments.GetArrayElementAtIndex(1);

            // Since stringParam is a string,
            // we need to get the value through m_StringArgument field
            stringParam = stringParam.FindPropertyRelative(“m_StringArgument”);

            EditorGUI.LabelField(argNameRect, new GUIContent(“My Awesome String:”,
“My awesome string argument”));
            EditorGUI.PropertyField(argRect, stringParam, GUIContent.none);
}
```

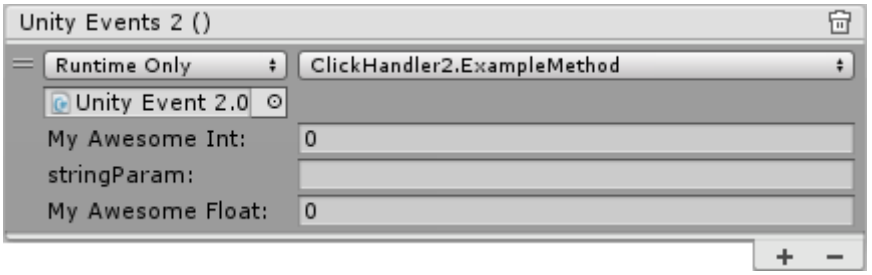
The result will be like the image below.



The CustomInspector attribute is also acceptable on parameters. See example below:

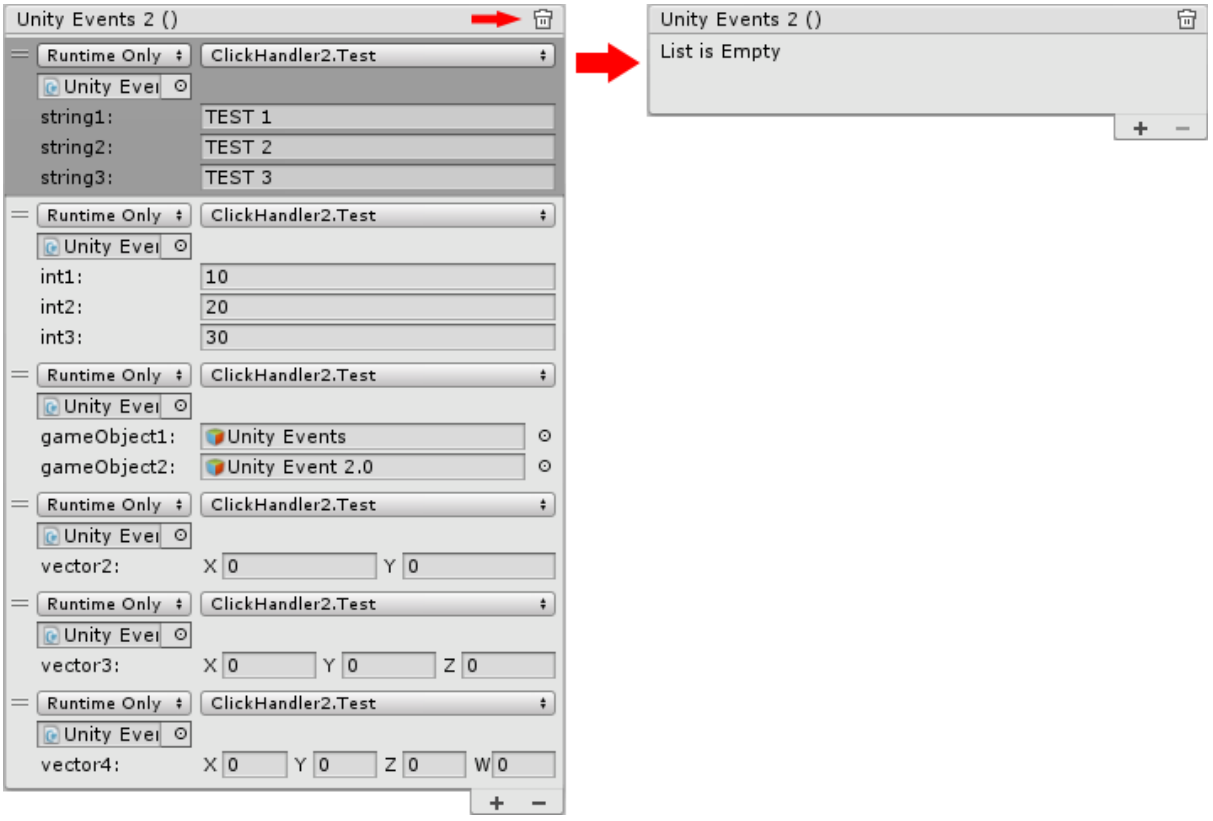
```
public void ExampleMethod([CustomInspector("ExampleMethodCustomInspectorForInteger")] int intParam, string stringParam, [CustomInspector("ExampleMethodCustomInspectorForFloat")] float floatParam)
{
    Debug.LogFormat("Int:{0} - String{1} - Float:{2}", intParam, stringParam, floatParam);
}

public void ExampleMethodCustomInspectorForInteger(SerializedProperty argument, Rect argNameRect, Rect argRect)
{
    EditorGUI.LabelField(argNameRect, new GUIContent("My Awesome Int:", "My awesome integer argument"));
    EditorGUI.PropertyField(argRect, argument, GUIContent.none);
}

public void ExampleMethodCustomInspectorForFloat(SerializedProperty argument, Rect argNameRect, Rect argRect)
{
    EditorGUI.LabelField(argNameRect, new GUIContent("My Awesome Float:", "My awesome float argument"));
    EditorGUI.PropertyField(argRect, argument, GUIContent.none);
}
```

The result will be like the image below.



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)

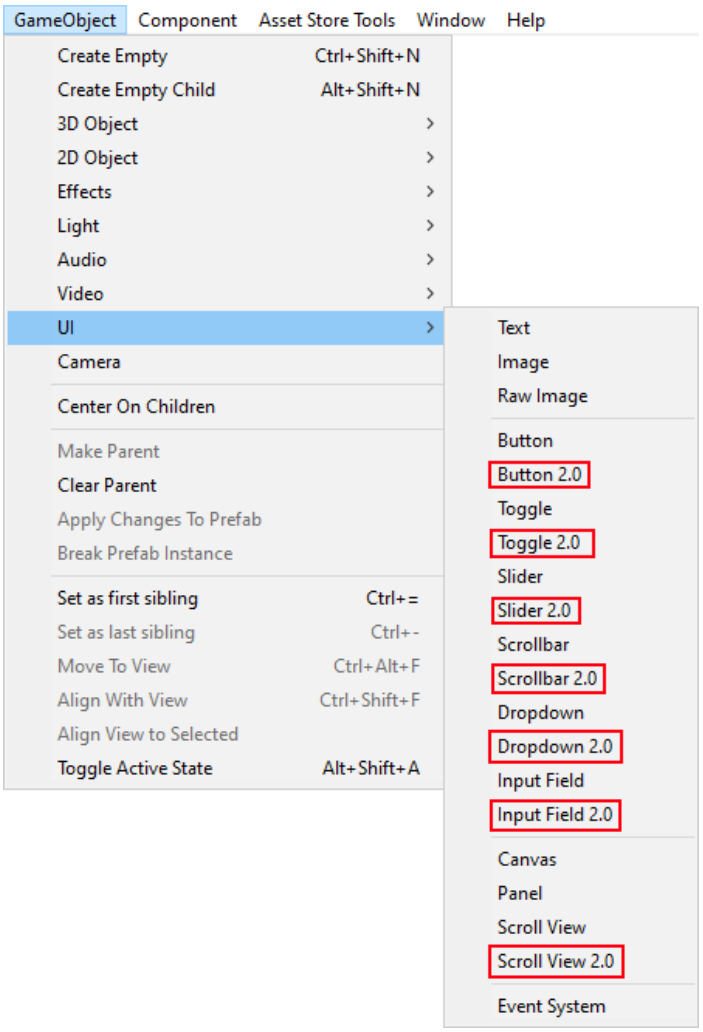


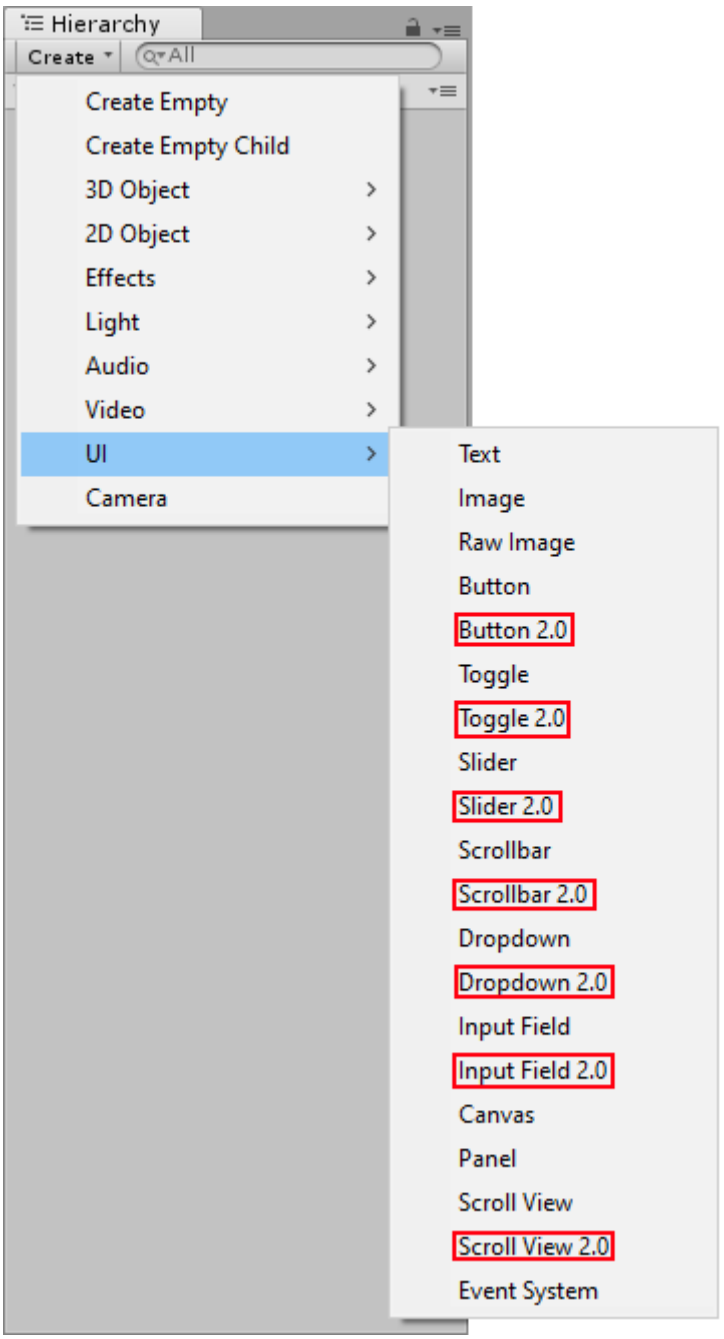
3. UI 2.0

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**.

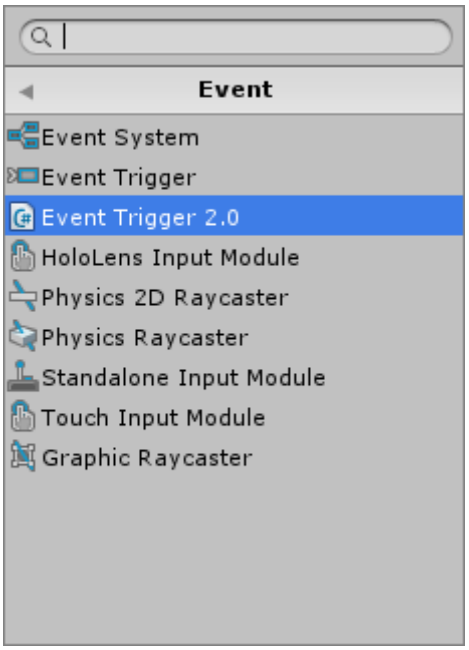



Hierarchy
Create
All
Create Empty
Create Empty Child
3D Object
2D Object
Effects
Light
Audio
Video
UI
Camera
Text
Image
Raw Image
Button
Button 2.0
Toggle
Toggle 2.0
Slider
Slider 2.0
Scrollbar
Scrollbar 2.0
Dropdown
Dropdown 2.0
Input Field
Input Field 2.0
Canvas
Panel
Scroll View
Scroll View 2.0
Event System

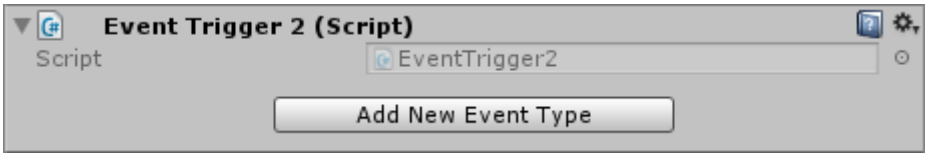
4. Event Trigger 2.0

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

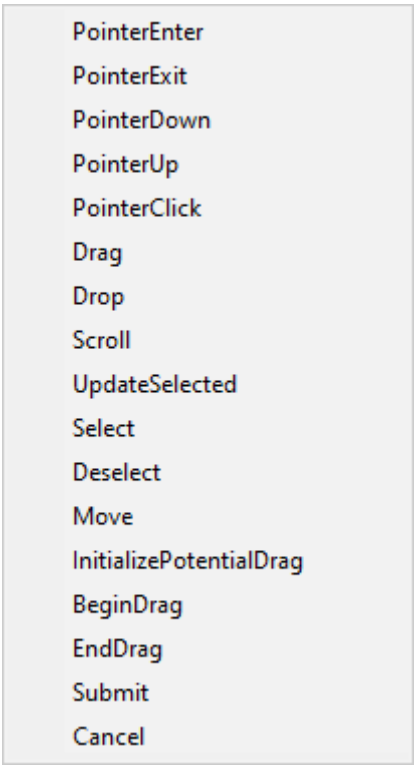
To do that, select the desired component, click on the **Add Component** button, go to **Event** and click on the **Event Trigger 2.0** component (see image below).



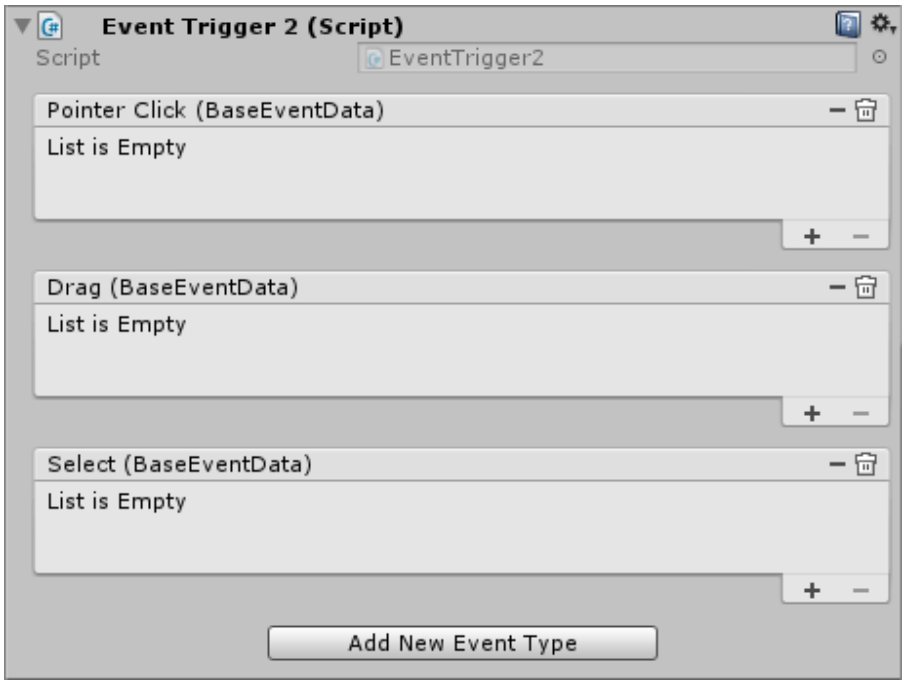
After adding, it'll look like the image below (see image below).



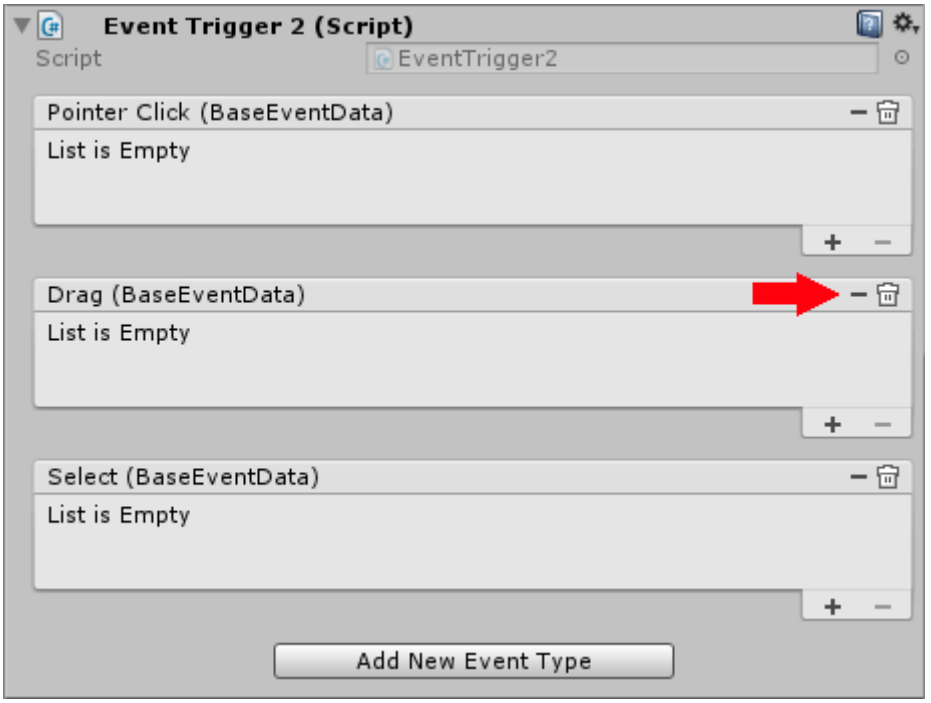
Click on the **Add New Event Type** button to visualize all the possible events that can be added to the GameObject (See image below).



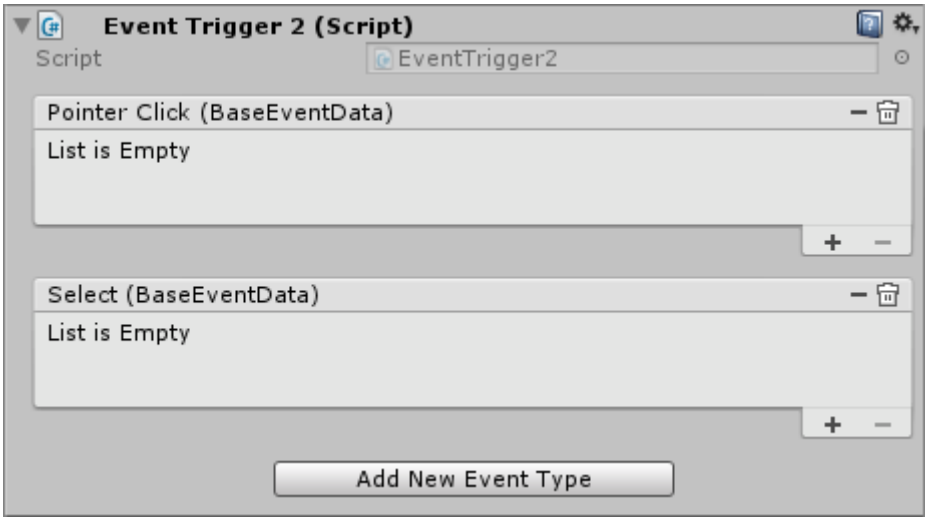
After selecting the desired events (eg. **PointerClick**, **Drag** and **Select**), your GameObject will show those events on the Inspector (see image below).



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



Before delete



After delete

5. 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;
}
```