

EasyEditor

Warning : If you move EasyEditor folder to another location that <project root>/Assets, you need to update its location in ResourcesLocation.cs.

Note 1 : If you want to quickly use EasyEditor, reading Quick Start section and opening each scene in EasyEditor/Examples should be enough. If you want a deeper understanding, then reading this documentation may help.

Note 2 : EasyEditor works for both MonoBehaviour and Scriptable Object. For ease of write and read, this documentation may sometime only mention MonoBehaviour when it stands for both MonoBehaviour and Scriptable Object.

1. WHAT IS EASYEDITOR

EasyEditor is a handy framework that allows you to drastically decrease the time you spend to create editor interfaces for your MonoBehaviour or Scriptable Object.

Most of time, programmers prefer to spend time on developing the game instead of creating nit interfaces that would allow artists and game designers to easily visualize game elements and fine tune it. It is a real production killer.

With EasyEditor, programmers can create nice interfaces with a near to zero coding time! Customizing interface is one of the most powerful feature of Unity. EasyEditor pushes this power at a further limit.

1.1. QUICK START

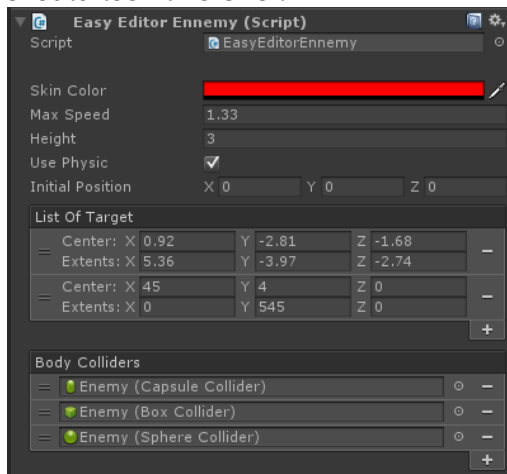
Create a new MonoBehaviour and call it EasyEditorEnemy.cs. Copy the code below:

```
public class EasyEditorEnemy: MonoBehaviour
{
    public Color skinColor;
    public float maxSpeed;
    public float height = 3f;
    public bool usePhysic = true;
    public Vector3 initialPosition;
    public List<Bounds> listOfTarget;
    public List<Collider> BodyColliders;
}
```

Supposing you tuned a bit the values of these fields, Unity editor should by default display



Now, right click on EasyEditorEnemy.cs and click on **Customize Interface**. Your interface should look like this :



Only few changes are noticeable. The two lists are now drawn in a very fancy way: **Elements are interchangeable!** This customization of the array objects is generously provided by the open source project from Rotorz (<https://bitbucket.org/rotorz/reorderable-list-editor-field-for-unity/overview>).

It allows a better integration of the lists in the inspector and also a context menu to insert elements at a specific position in the list.


Now, let say that your enemy under some circumstances can get really angry and get into the fury state. You need a function that triggers this fury state. Here you are:

```
public void GetIntoFuryState()
{
    Debug.Log("Here start the fury state !!!");
}
```

The problem is that in the game you need to collect 100 diamonds before the fury state is activated, and your game designer wants to visualize it while the game is playing, and want to visualize it several times in a roll. Wouldn't be cool to create a button that allows the game designer to trigger this function while the game is playing? Yes, but writing an editor

script for that seems pretty annoying. EasyEditor brings you the fast way. Add `[Inspector]` on top of your function:

```
[Inspector]
public void GetIntoFuryState()
{
    Debug.Log("Here start the fury state !!!");
}
```

This will automatically add the button  in the inspector. The game designer just need to click on it to see the fury state ! But EasyEditor is not only about that. What follows will show to you all the features EasyEditor has.

1.2. WHAT EXACTLY DOES EASYEDITOR

Now, you probably wonder what is behind the hood. When you right click on your mono script and select 'Customize Interface' in the context menu, an editor script is created for your monobehaviour/scriptable object. This editor script inherits from EasyEditorBase which is our main class to draw the inspector. From this new editor script you can enhance your interface in a very easy way.

Note that EasyEditor changes only the appearance of your fields and allows designing some interfaces that would require time to implement in an editor script. Also, the whole Unity3 serialization system is still in use which means:

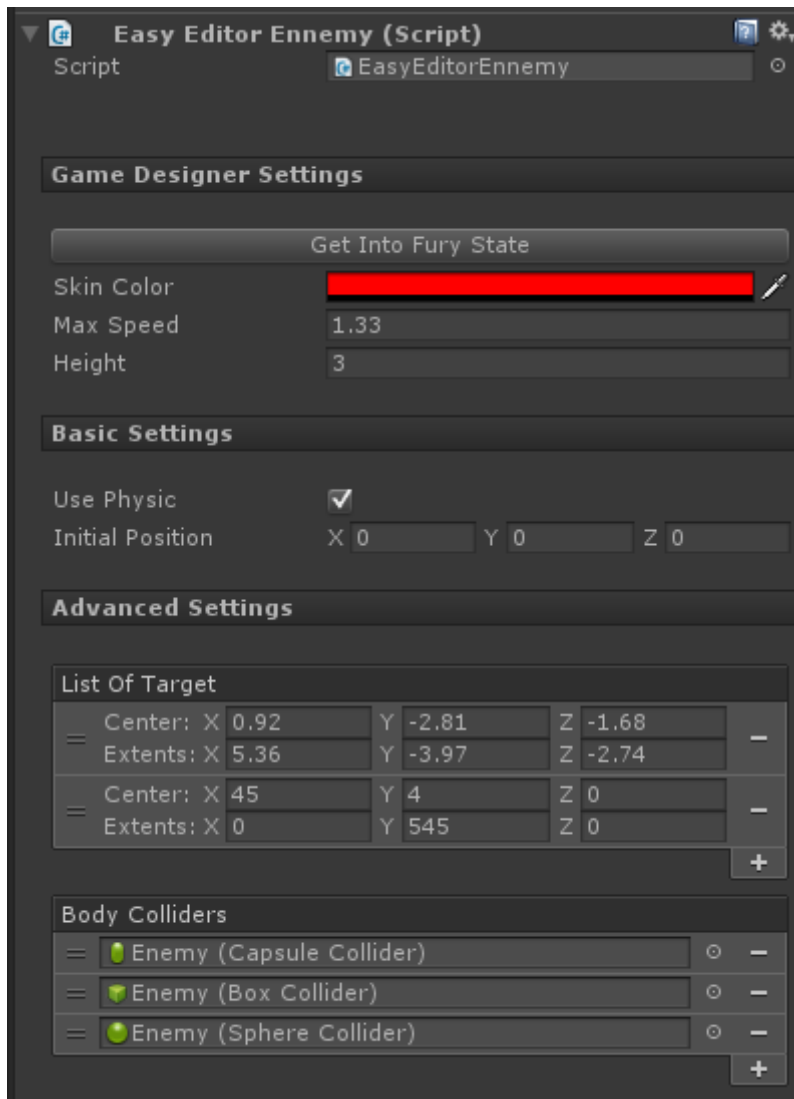
- The performance of your game will not change since EasyEditor code is executed only in the editor.
- All the attributes `[SerializeField]`, `[HideInInspector]`, `[Tooltip]`, `[Range(0f, 1f)]`, `[System.NonSerialized]` can still be used with EasyEditor.

Note that Easy Editor is far from being limitative since you can also integrate your own editor code anywhere in the inspector.

Now, let see the other possibilities.

2. GROUPS AND ORDER

Some scripts can contain a lot of settings in the inspector. There are some settings you want the game designers to play with, and others you don't want. How would it look like if you could organize these settings in groups? Like this:



To add this group, you need first to define the groups in EasyEditorEnemyEditor.cs in the order you want them to appear:

```
[EasyEditor.Groups("Game Designer Settings", "Basic Settings", "Advanced Settings")]
[CustomEditor(typeof(EasyEditorEnemy))]
public class EasyEditorEnemyEditor : EasyEditorBase
{
}
```

Then on top of each group of fields you want to display add `[Inspector(group = "Name of the group")]`.

```
public class EasyEditorEnemy : MonoBehaviour
{
    [Inspector(group = "Game Designer Settings")]
    public Color skinColor;
    public float maxSpeed;
    public float height = 3f;

    [Inspector(group = "Basic Settings")]
```

```

public bool usePhysic = true;
public Vector3 initialPosition;

[Inspector(group = "Advanced Settings")]
public List<Bounds> listOfTarget;
public List<Collider> BodyColliders;

[Inspector(group = "Game Designer Settings")]
public void GetIntoFuryState()
{
    Debug.Log("Here starts the fury state !!!");
    GetComponent<Animation>().Play();
}
}

```

Every field below a field with an Inspector attribute and a specified group will belong to the same group. You always can re-specify a new group on top of any field to move it to a different group. Each field that does not belong to any group will be displayed on top.

If you wish to change the order in which fields are rendered, you can specify the order value:

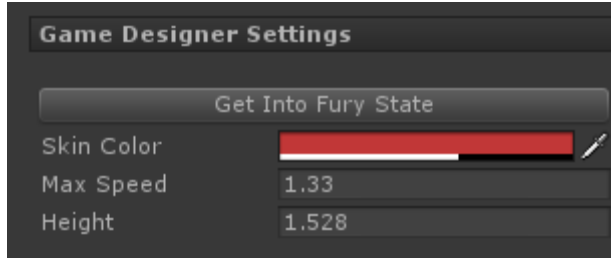
```
[Inspector(group = "", order = 1)];
```

By default, the order of each element you want to render is 100.

For example to place the button 'Get Into Fury State' on top of the Game Designer Settings group you can write on top of it :

```
[Inspector(group = "Game Designer Settings", order = 1)]
```

You will get:



You can also add a boxed description of the group that will appear below it by specifying the groupDescription member of Inspector attribute :

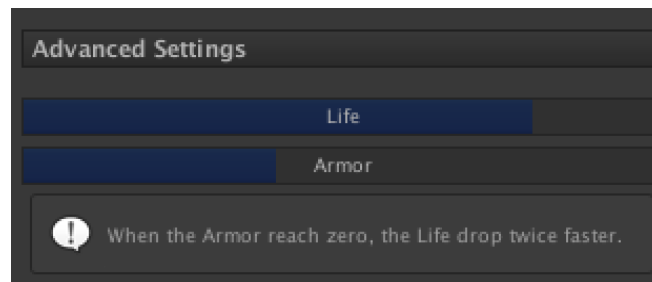
```
[Inspector(group = "Game Designer Settings", groupDescription = "These settings allows you to fine-tune some of the enemy characteristics")]
```

3. COMMENTS, TOOLTIP, LAYOUT

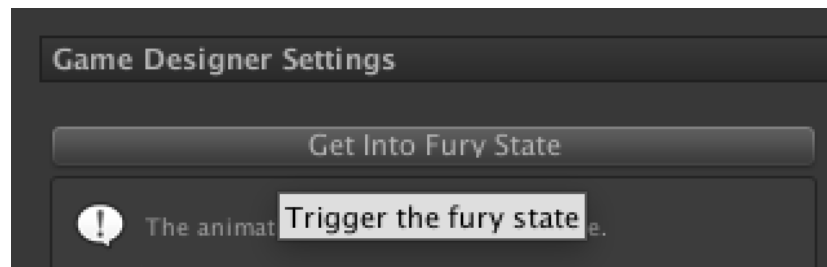
- Comments can be added below a property by using the attribute

```
[Comment("your comment")]
```

It can also be used with custom editor script (see section : Integrating Unity classic editor code).



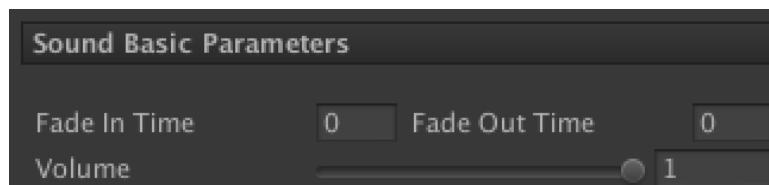
- The default `[Tooltip("your tooltip")]` in Unity only works with fields. You can also display a tooltip on buttons rendered by EasyEditor by the using the attribute `[EETooltip("your tooltip")]`



- Consecutive fields can be laid in an horizontal or vertical fashion with the attributes `[BeginHorizontal]`, `[EndHorizontal]`, `[BeginVertical]`, `[EndVertical]`.

Here an example :

```
[Inspector(group = "Sound Basic Parameters")]
[BeginHorizontal]
public float fadeInTime = 0f;
[EndHorizontal]
public float fadeOutTime = 0f;
```

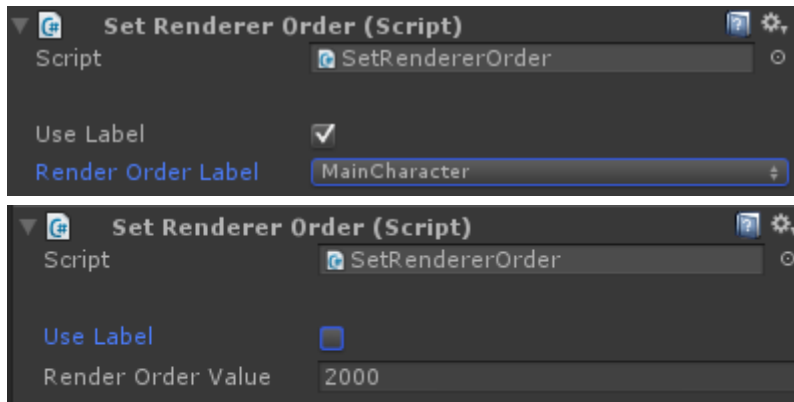


4. HIDING GROUPS AND FIELDS

Groups and fields can be hidden in the editor script. This can be useful if you want to hide a group to some users, or if you want to display some fields under some condition.

Let's take an example.

You want to write a script that allows setting when the object should be render in the render queue. Several objects use this script, and you want to give the user the possibility to set this order based on an absolute int value, or an enum value that would tell you if the object is render at the same time as the main character, or after shadows, or after the terrain (this can be useful for effects like water riddles). Here what you want to get in the editor.



There are two ways of achieving this functionality. Let's start with the most straightforward.

HIDING WITH ATTRIBUTE

You can use the attribute `[Visibility(string id, object value)]` on top of a field.

If the value of the field holding the id `id` is equal to `value`, then the field with attribute `Visibility` will be visible, otherwise it will not. Note that the default id of a field is its name. But you can change the id of a field by specifying it with the attribute :

```
[Inspector(id = "your custom id")]
```

Here is the code :

```
using UnityEngine;
using System.Collections;

public class SetRendererOrder : MonoBehaviour {

    public enum RenderOrderLabel
    {
        Shadow = 1998,
        Terrain = 1999,
        MainCharacter = 2001
    }

    public bool useLabel = true;
    [Visibility("useLabel", true)]
    [SerializeField] private RenderOrderLabel renderOrderLabel =
RenderOrderLabel.MainCharacter;
    [Visibility("useLabel", false)]
    [SerializeField] private int renderOrderValue = 2000;
}
```

Now, in your project window, right-click on `SetRenderOrder.cs` and select *Customize Interface*. That's it.

HIDING FROM THE EDITOR SCRIPT

EasyEditor is not only displaying your monobehaviour/scriptable object script in the inspector. It can render some piece of custom editor script, or a button activating a method. You may want also to hide these based on some specific conditions. You can do it directly from the editor script generated when you right-click on your script and select *Customize Interface* in the context menu.

You can use the function `HideRenderer(string id)` or `ShowRenderer(string id)`. The id is the name of a field in your monobehaviour/scriptable object, or the name of a function if it is rendering a button or some custom editor scripts (see following sections).

Here the code that solves our previous case :

```
public class SetRenderOrderEditor : EasyEditorBase
{
    public override void OnInspectorGUI ()
    {
        SetRenderOrder monobehaviour = (SetRenderOrder) target;

        if(monobehaviour.useLabel)
        {
            HideRenderer("renderOrderValue");
            ShowRenderer("renderOrderLabel");
        }
        else
        {
            HideRenderer("renderOrderLabel");
            ShowRenderer("renderOrderValue");
        }

        base.OnInspectorGUI ();
    }
}
```

Hiding a group is pretty similar to hiding a field in the editor script, but instead of `HideRenderer/ShowRenderer` you need to call `HideGroup/ShowGroup` with the string name of the group as argument. It is illustrated in the following section.

5. ADDING BUTTONS TO THE EDITOR SCRIPT

Buttons can be added from the monobehaviour/scriptable object script by writing `[Inspector]` on top of any function. This works also in the editor script! Let's say you want a button that hides or shows the advanced settings group.



You first need to declare a serialized field *showAdvancedSettings* in your monobehaviour to save the state show/hidden of the advanced setting group. Let's do it in our *EasyEditorEnemy.cs* script :

```
[HideInInspector]
public bool showAdvancedSetting = false;

[Inspector(group = "Advanced Settings")]
public List<Bounds> listOfTarget;
public List<Collider> BodyColliders;

[Inspector(group = "Game Designer Settings", order = 1)]
public void GetIntoFuryState()
{
    Debug.Log("Here start the fury state !!!");
}
```

Now, in our editor script, we can write a function that will change the state of *showAdvancedSettings* and display it at the end of Basic Settings group, just on top of Advanced Settings group.

```
[EasyEditor.Groups("Game Designer Settings", "Basic Settings", "Advanced Settings")]
[CustomEditor(typeof(HideGroupEnemy))]
public class HideGroupEnemyEditor : EasyEditorBase
{
    [Inspector(group = "Basic Settings")]
    private void ToggleDisplayAdvancedSettings()
    {
        HideGroupEnemy enemy = (HideGroupEnemy)target;

        enemy.showAdvancedSetting = !enemy.showAdvancedSetting;

        if (enemy.showAdvancedSetting)
        {
            ShowGroup("Advanced Settings");
        }
        else
        {
            HideGroup("Advanced Settings");
        }
    }
}
```

Et Voila !

6. INTEGRATING UNITY CLASSIC EDITOR CODE

6.1. PROPERTYDRAWER

EasyEditor uses Unity property default drawer to render every field serialized by default. It means that if you define a property drawer, Easy Editor will render it as you specified it in the class inheriting from PropertyDrawer.

6.2. CUSTOM EDITOR CODE

Easy Editor helps you to render quickly and in an organized way every UI elements you mostly use. But you may wish adding your own custom piece of editor script to render some of your custom properties in one group at a specific position. To do so, you simply need to implement a function in your editor script and to add the attribute `[Inspector(group = "Name of the group", rendererType = "CustomRenderer")]`.

Let's take an example. In part 2 we saw how to define groups for the fields of EasyEditorEnemy.cs. Now, in the group "Advanced Settings" we want to add a progress bar indicating the level of the enemy armor. You just need to open EasyEditorEnemyEditor.cs and add :

```
[Inspector(group = "Advanced Settings", rendererType = "CustomRenderer", order = 1)]
private void RenderProgressBar()
{
    Rect r = EditorGUILayout.BeginVertical();
    EditorGUI.ProgressBar(r, 0.8f, "Life");
    GUILayout.Space(18);
}
```

```

    EditorGUILayout.EndVertical();
}

```



Note that the name of the function is not important.

7. INLINE RENDERING OF CUSTOM CLASSES AND STRUCTURE

EasyEditor allows you use EasyEditor attributes directly into a custom class/struct and to render it in the inspector of the MonoBehaviour which hold it. To achieve it, you can add the attribute `[Inspector(rendererType = "InlineClassRenderer")]`. This feature works also with nested classes.

Let's illustrate it.

Here a custom class Weapon:

```

[System.Serializable]
public class Weapon
{
    [BeginHorizontal]
    public string name = "";
    [EndHorizontal]
    public float strength = 0f;
}

```

and here a custom class Bag that holds the class weapon.

```

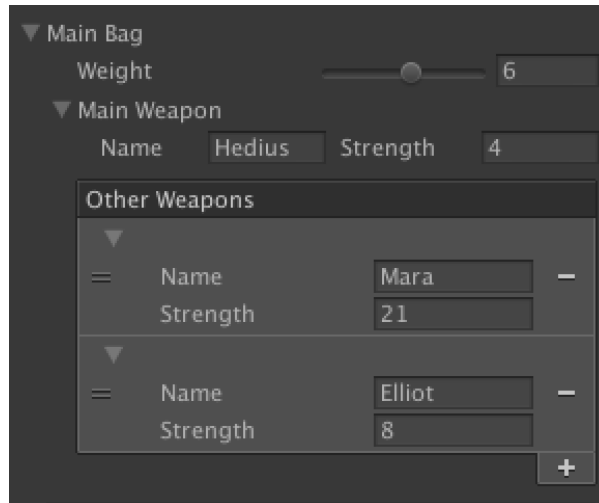
[System.Serializable]
public class Bag
{
    [Range(1, 10)]
    public int weight;

    [Inspector(rendererType = "InlineClassRenderer")]
    public Weapon mainWeapon;

    public List<Weapon> otherWeapons;
}

```

Any monobehaviour with a member of type Bag with the attribute `[Inspector(rendererType = "InlineClassRenderer")]` will see in its inspector :



Please, note that as you can see in the previous screenshot this feature does not work inside of lists. The custom class will be rendered as Unity render it by default in a list.