uFAction

GUIDEBOOK

VFACTION

BY VEXE

INDEX

What is uFAction?

Intro

Class hierarchy

Usage

The editors

Settings and customization

Bonus

What is uFAction?

- It's a Unity extension/package that gives you a full set of serializable delegates that could be easily integrated and made visible in the inspector for editing.

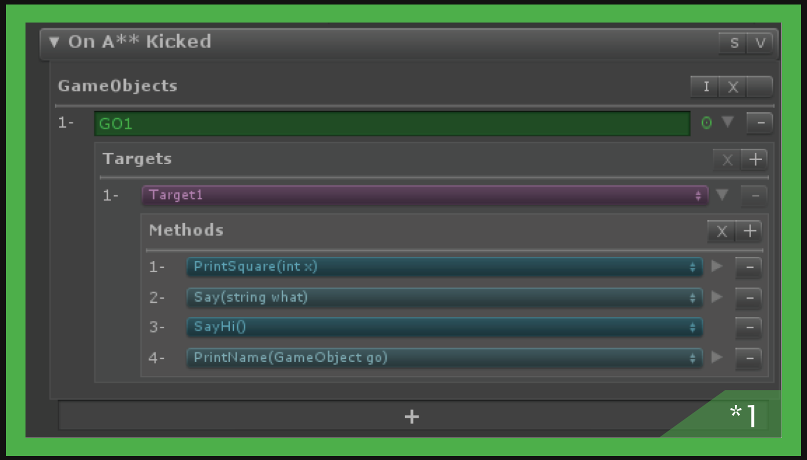
- You can target any UnityEngine.Object with UnityDelegate (Actions and Funcs - all with generic versions)

- You can target any non UnityEngine.Object with SysObjDelegate (Actions and Funcs - with generic versions too)

- You get a KickassDelegate that could accept any method with no return (void) regardless of the parameters' signature! (*1)

- Set your handlers' arguments (if any) from the editor by means of a direct value, or from a source! (*2)

- Support for extension methods! (*3)



▼ On A** Kicked

GameObjects

1- GO1

Targets

1- Target1

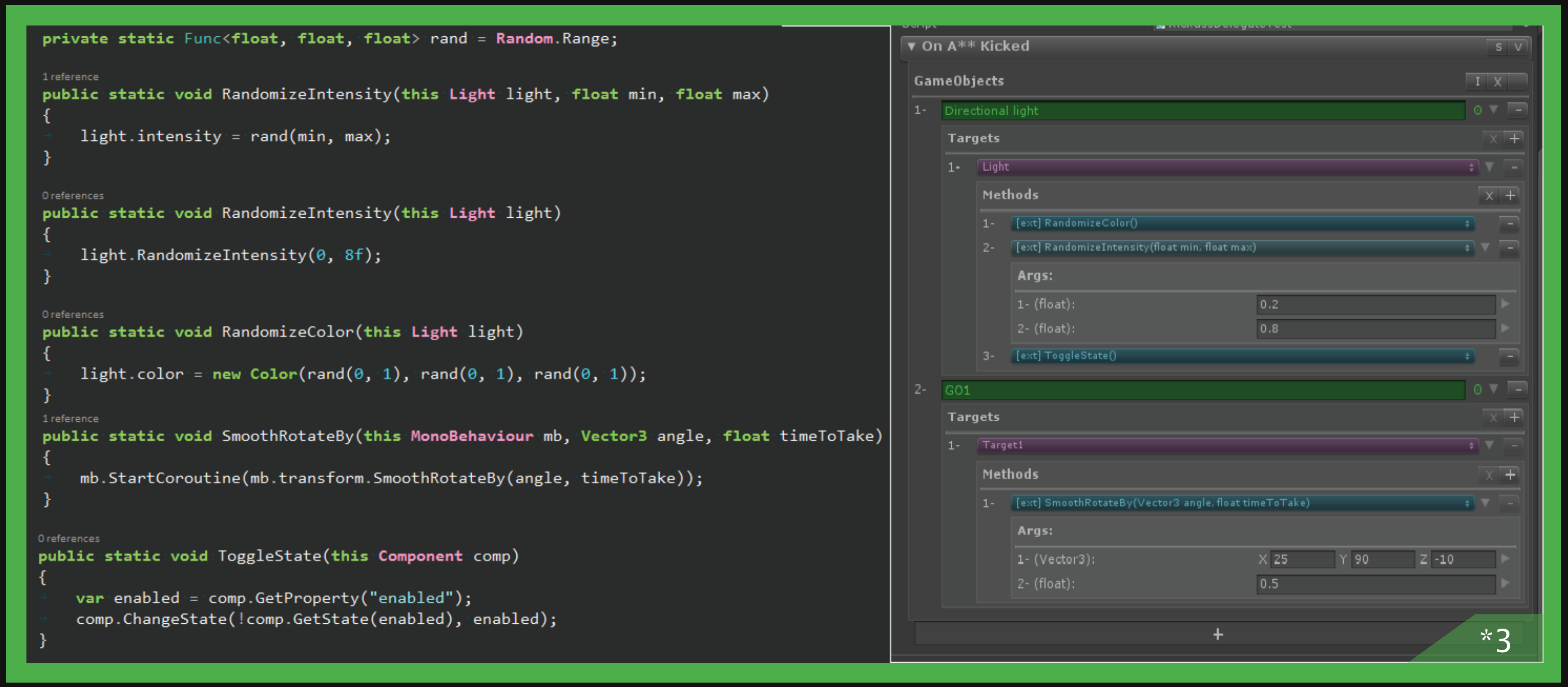
Methods

1- PrintSquare(int x)

2- Say(string what)

3- SayHi()

4- PrintName(GameObject go)



```
private static Func<float, float, float> rand = Random.Range;

1 reference
public static void RandomizeIntensity(this Light light, float min, float max)
{
    light.intensity = rand(min, max);
}

0 references
public static void RandomizeIntensity(this Light light)
{
    light.RandomizeIntensity(0, 8f);
}

0 references
public static void RandomizeColor(this Light light)
{
    light.color = new Color(rand(0, 1), rand(0, 1), rand(0, 1));
}
1 reference
public static void SmoothRotateBy(this MonoBehaviour mb, Vector3 angle, float timeToTake)
{
    mb.StartCoroutine(mb.transform.SmoothRotateBy(angle, timeToTake));
}

0 references
public static void ToggleState(this Component comp)
{
    var enabled = comp.GetProperty("enabled");
    comp.ChangeState(!comp.GetState(enabled), enabled);
}
```

▼ On A** Kicked

GameObjects

1- Directional light

Targets

1- Light

Methods

1- [ext] RandomizeColor()

2- [ext] RandomizeIntensity(float min, float max)

Args:

1- (float): 0.2

2- (float): 0.8

3- [ext] ToggleState()

2- GO1

Targets

1- Target1

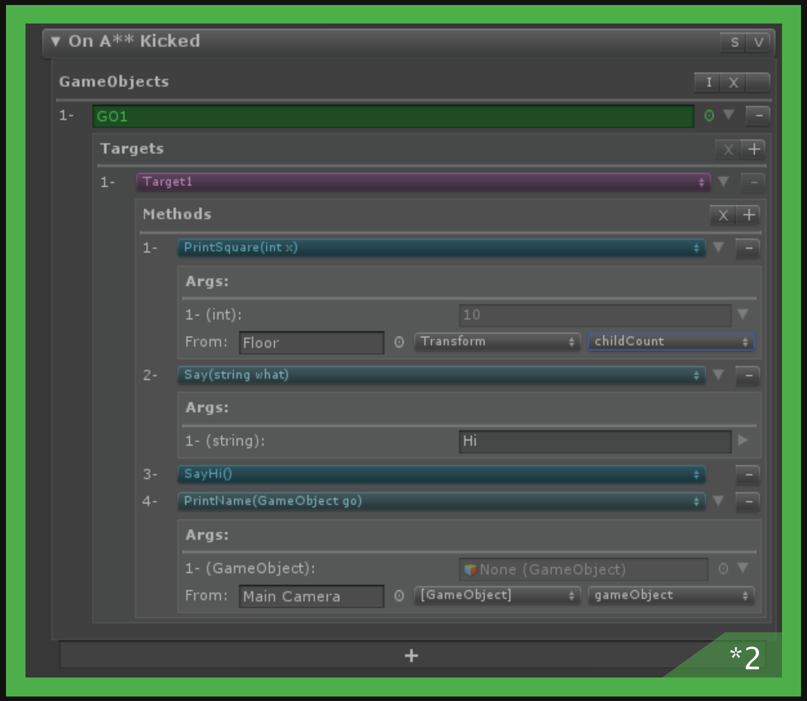
Methods

1- [ext] SmoothRotateBy(Vector3 angle, float timeToTake)

Args:

1- (Vector3): X 25 Y 90 Z -10

2- (float): 0.5



▼ On A** Kicked

GameObjects

1- GO1

Targets

1- Target1

Methods

1- PrintSquare(int x)

Args:

1- (int): 10

From: Floor Transform childCount

2- Say(string what)

Args:

1- (string): Hi

3- SayHi()

4- PrintName(GameObject go)

Args:

1- (GameObject): None (GameObject)

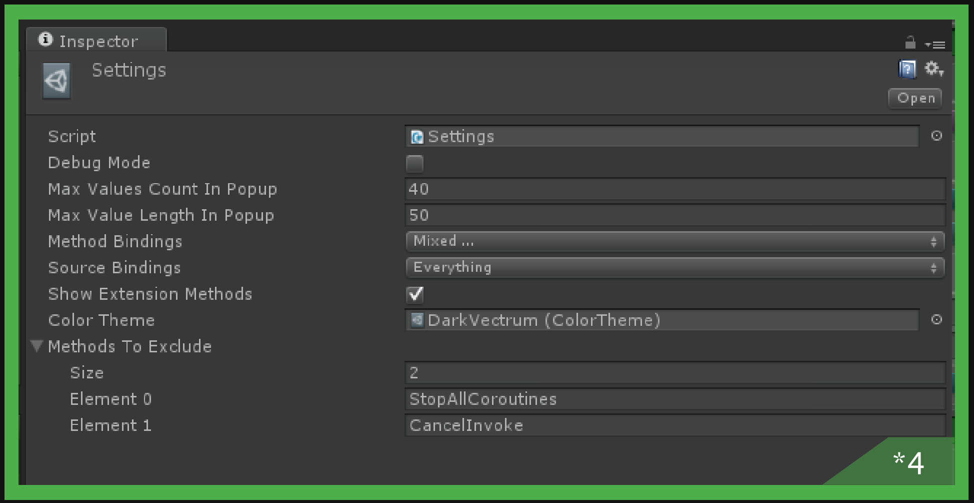
From: Main Camera [GameObject] gameObject

- Customize all your way through. You can adjust BindingFlags, create new themes, and many more by means of a simple asset file! (*4)

- Three unique editors: ***Readonly, Mini*** and ***Advanced***. Each with its own unique characteristics. Switch between available editors by a single click! (*5)

- Extremely easy and simple to use! Just declare your delegate and annotate/markup with [ShowDelegate("Title")] - No need for any custom editors! (*6)

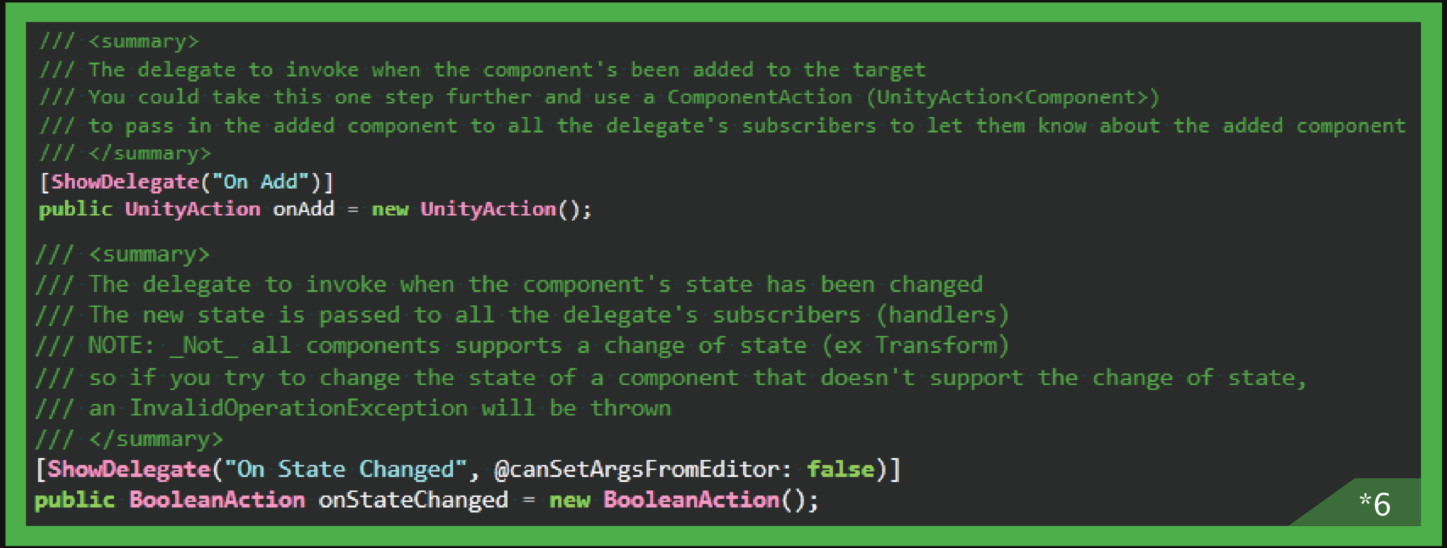
- Support for both Unity free and pro skins. (6.5*)



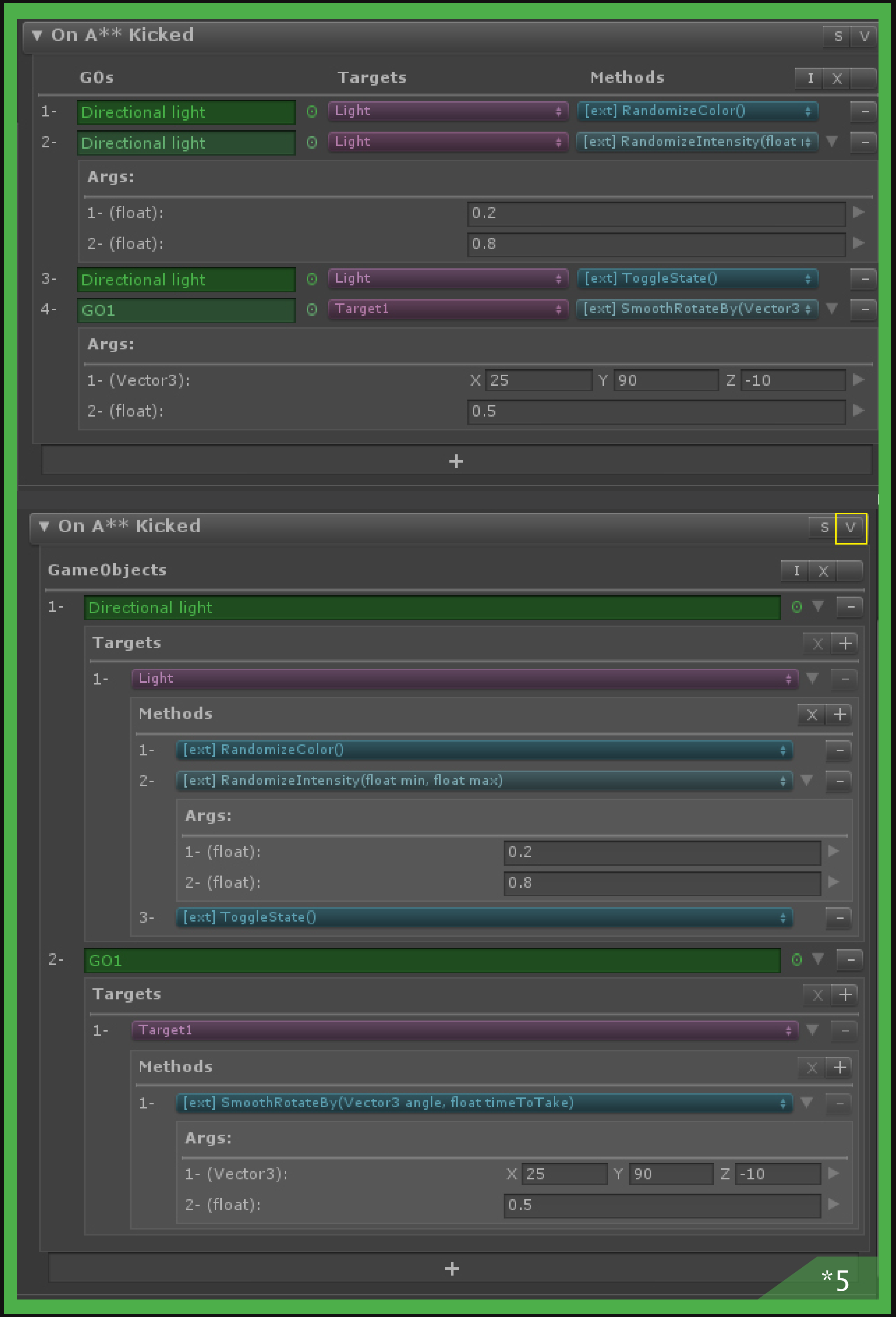
Inspector

Settings

Script	Settings
Debug Mode	□
Max Values Count In Popup	40
Max Value Length In Popup	50
Method Bindings	Mixed ...
Source Bindings	Everything
Show Extension Methods	☑
Color Theme	DarkVectrum (ColorTheme)
▼ Methods To Exclude	
Size	2
Element 0	StopAllCoroutines
Element 1	CancelInvoke



```
/// <summary>
/// The delegate to invoke when the component's been added to the target
/// You could take this one step further and use a ComponentAction (UnityAction<Component>)
/// to pass in the added component to all the delegate's subscribers to let them know about the added component
/// </summary>
[ShowDelegate("On Add")]
public UnityAction onAdd = new UnityAction();

/// <summary>
/// The delegate to invoke when the component's state has been changed
/// The new state is passed to all the delegate's subscribers (handlers)
/// NOTE: _Not_ all components supports a change of state (ex Transform)
/// so if you try to change the state of a component that doesn't support the change of state,
/// an InvalidOperationException will be thrown
/// </summary>
[ShowDelegate("On State Changed", @canSetArgsFromEditor: false)]
public BooleanAction onStateChanged = new BooleanAction();
```



▼ On A** Kicked

	GOs		Targets	Methods
1-	Directional light	⊙	Light	[ext] RandomizeColor()
2-	Directional light	⊙	Light	[ext] RandomizeIntensity(float i

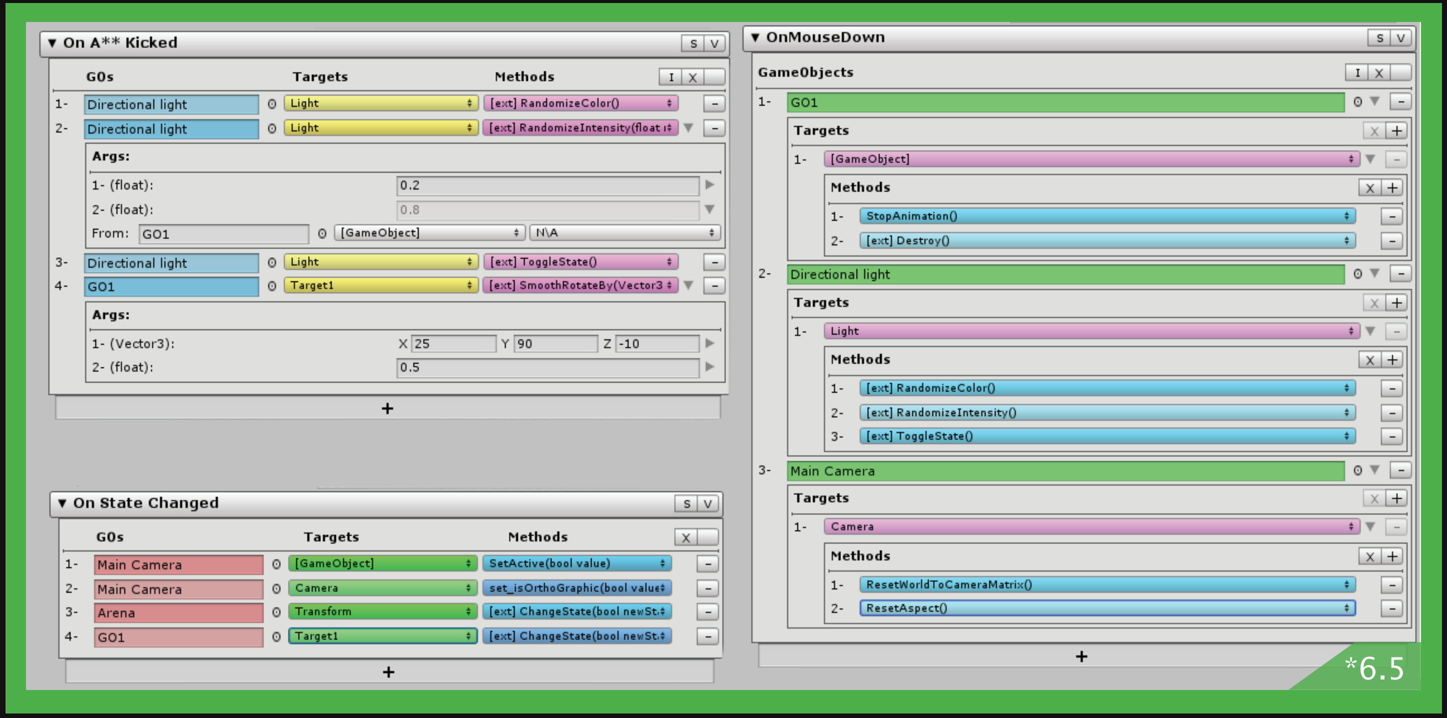
Args:
1- (float): 0.2
2- (float): 0.8

3-	Directional light	⊙	Light	[ext] ToggleState()
4-	GO1	⊙	Target1	[ext] SmoothRotateBy(Vector3

Args:
1- (Vector3): X 25 Y 90 Z -10
2- (float): 0.5

▼ On A** Kicked

GameObjects
1- Directional light
Targets
1- Light
Methods
1- [ext] RandomizeColor()
2- [ext] RandomizeIntensity(float min, float max)
Args:
1- (float): 0.2
2- (float): 0.8
3- [ext] ToggleState()
2- GO1
Targets
1- Target1
Methods
1- [ext] SmoothRotateBy(Vector3 angle, float timeToTake)
Args:
1- (Vector3): X 25 Y 90 Z -10
2- (float): 0.5



▼ On A** Kicked

	GOs		Targets	Methods
1-	Directional light	⊙	Light	[ext] RandomizeColor()
2-	Directional light	⊙	Light	[ext] RandomizeIntensity(float i

Args:
1- (float): 0.2
2- (float): 0.8
From: GO1 ⊙ [GameObject] N\A

3-	Directional light	⊙	Light	[ext] ToggleState()
4-	GO1	⊙	Target1	[ext] SmoothRotateBy(Vector3

Args:
1- (Vector3): X 25 Y 90 Z -10
2- (float): 0.5

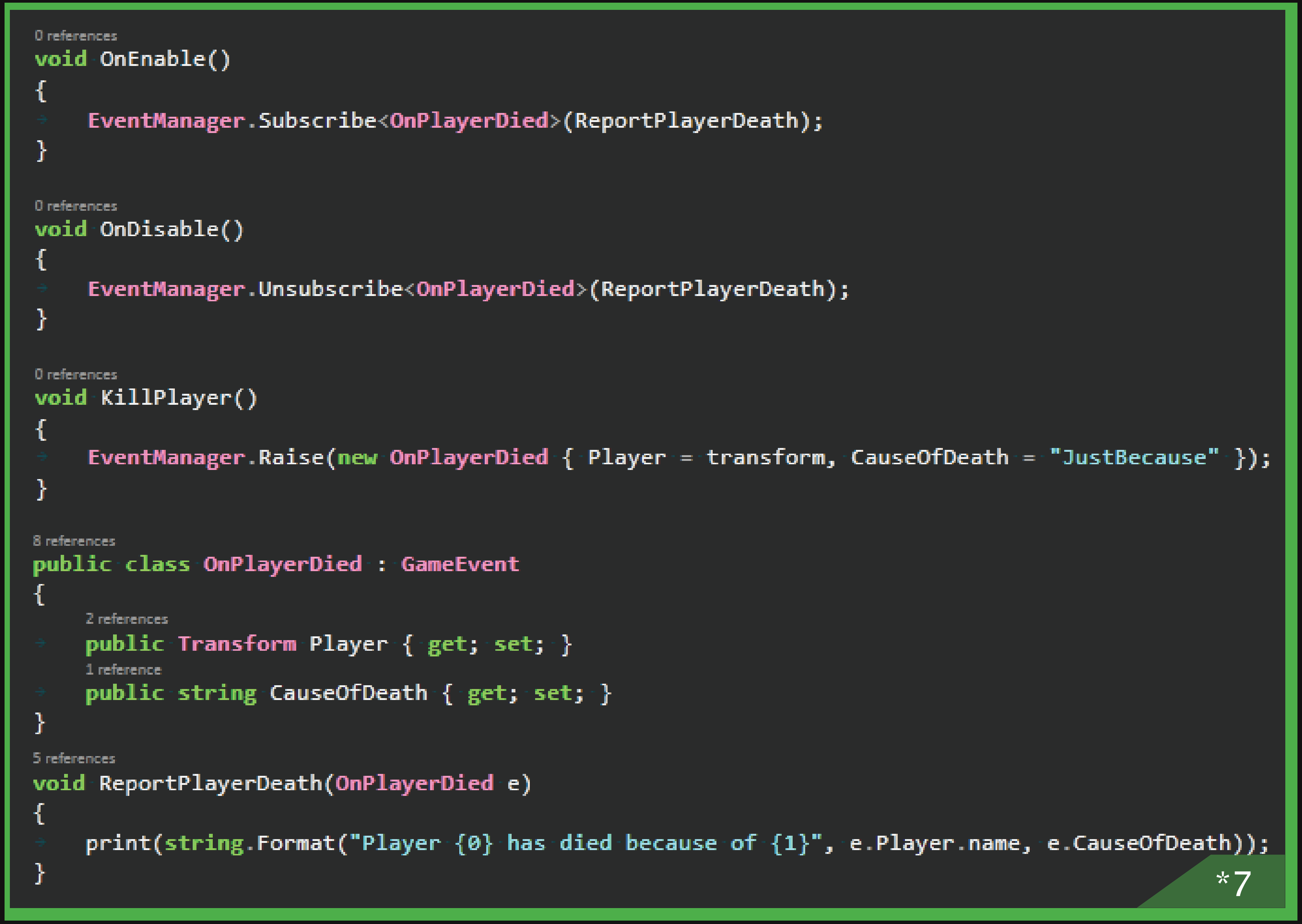
▼ On State Changed

	GOs		Targets	Methods
1-	Main Camera	⊙	[GameObject]	SetActive(bool value)
2-	Main Camera	⊙	Camera	set_isOrthoGraphic(bool value
3-	Arena	⊙	Transform	[ext] ChangeState(bool newSt
4-	GO1	⊙	Target1	[ext] ChangeState(bool newSt

▼ OnMouseDown

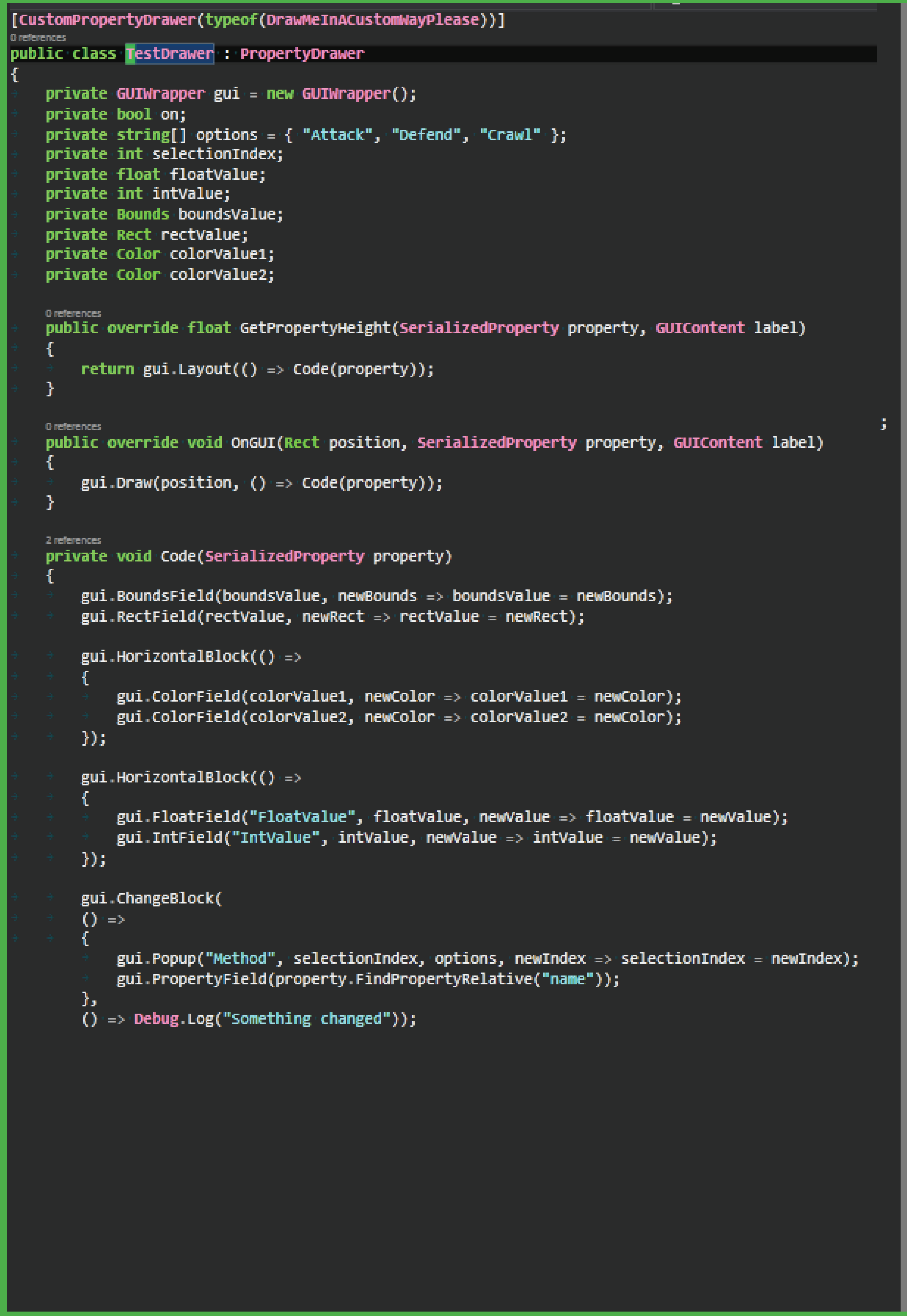
GameObjects
1- GO1
Targets
1- [GameObject]
Methods
1- StopAnimation()
2- [ext] Destroy()
2- Directional light
Targets
1- Light
Methods
1- [ext] RandomizeColor()
2- [ext] RandomizeIntensity()
3- [ext] ToggleState()
3- Main Camera
Targets
1- Camera
Methods
1- ResetWorldToCameraMatrix()
2- ResetAspect()

- **Some extra bonus stuff:**
    1- A globally accessed generic event system that you could use to globally (un)subscribe event handlers and fire game events (*7)
    2- An awesome GUIWrapper that makes it possible to use GUILayout-like methods in GUILayout-restricted areas (PropertyDrawer.OnGUI for example (This is what I used to draw the delegate editors! It can draw Buttons, Labels, Blocks)Horizontal/Vertical, (fields) IntField, ColorField, ObjectField, etc), Popups, Foldouts, Boxes, DragDropAreas, GetLastRect, ColorBlock, ChangedBlock, EnabledBlock and many more! (*8)

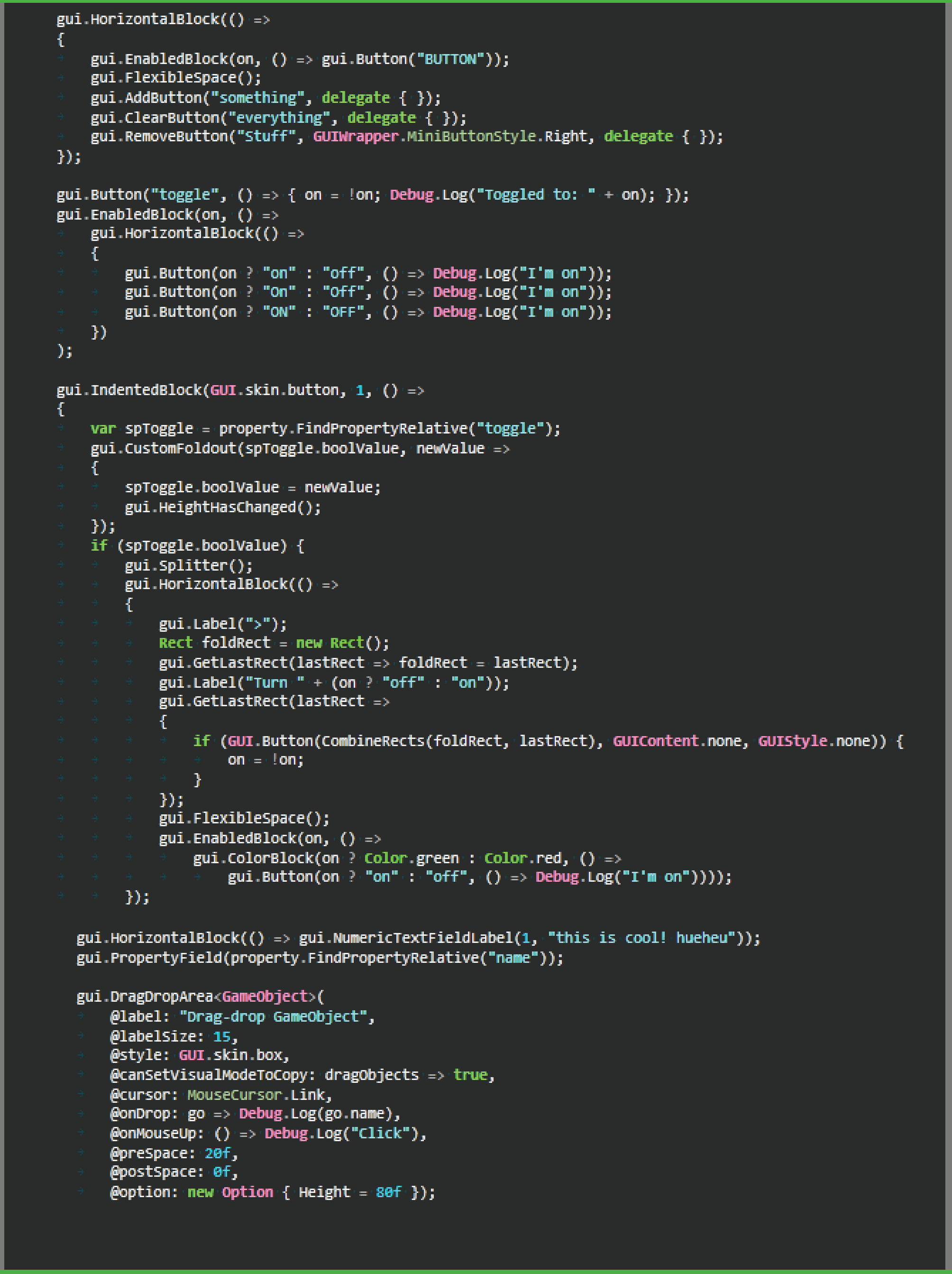


```
0 references
void OnEnable()
{
    EventManager.Subscribe<OnPlayerDied>(ReportPlayerDeath);
}

0 references
void OnDisable()
{
    EventManager.Unsubscribe<OnPlayerDied>(ReportPlayerDeath);
}

0 references
void KillPlayer()
{
    EventManager.Raise(new OnPlayerDied { Player = transform, CauseOfDeath = "JustBecause" });
}

8 references
public class OnPlayerDied : GameEvent
{
    2 references
    public Transform Player { get; set; }
    1 reference
    public string CauseOfDeath { get; set; }
}
5 references
void ReportPlayerDeath(OnPlayerDied e)
{
    print(string.Format("Player {0} has died because of {1}", e.Player.name, e.CauseOfDeath));
}
```

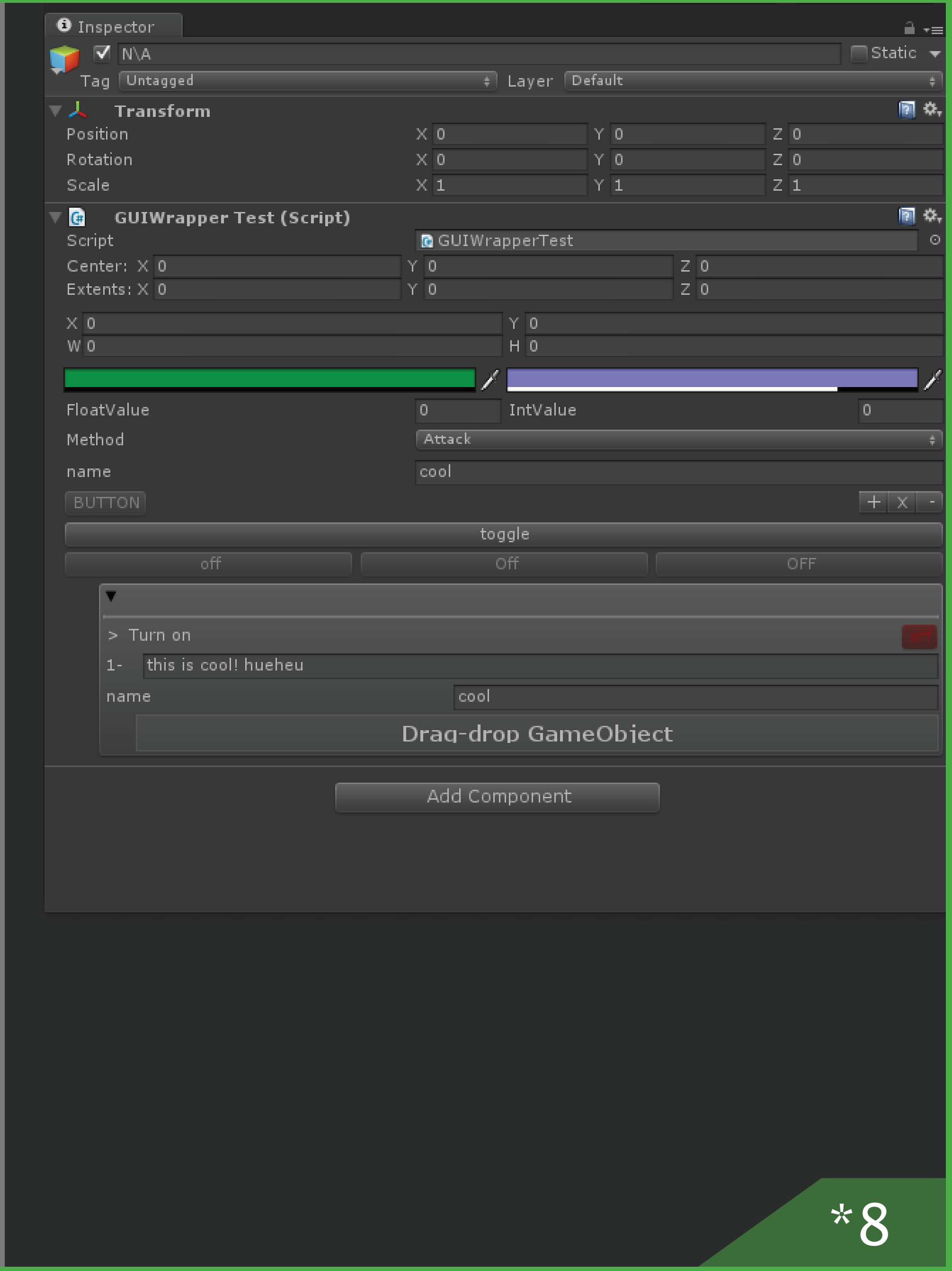
*7



```
[CustomPropertyDrawer(typeof(DrawMeInACustomWayPlease))]
0 references
public class TestDrawer : PropertyDrawer
{
    private GUIWrapper gui = new GUIWrapper();
    private bool on;
    private string[] options = { "Attack", "Defend", "Crawl" };
    private int selectionIndex;
    private float floatValue;
    private int intValue;
    private Bounds boundsValue;
    private Rect rectValue;
    private Color colorValue1;
    private Color colorValue2;

    0 references
    public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
    {
        return gui.Layout(() => Code(property));
    }

    0 references
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        gui.Draw(position, () => Code(property));
    }

    2 references
    private void Code(SerializedProperty property)
    {
        gui.BoundsField(boundsValue, newBounds => boundsValue = newBounds);
        gui.RectField(rectValue, newRect => rectValue = newRect);

        gui.HorizontalBlock(() =>
        {
            gui.ColorField(colorValue1, newColor => colorValue1 = newColor);
            gui.ColorField(colorValue2, newColor => colorValue2 = newColor);
        });

        gui.HorizontalBlock(() =>
        {
            gui.FloatField("FloatValue", floatValue, newValue => floatValue = newValue);
            gui.IntField("IntValue", intValue, newValue => intValue = newValue);
        });

        gui.ChangeBlock(
        () =>
        {
            gui.Popup("Method", selectionIndex, options, newIndex => selectionIndex = newIndex);
            gui.PropertyField(property.FindPropertyRelative("name"));
        },
        () => Debug.Log("Something changed"));
```



```
        gui.HorizontalBlock(() =>
        {
            gui.EnabledBlock(on, () => gui.Button("BUTTON"));
            gui.FlexibleSpace();
            gui.AddButton("something", delegate { });
            gui.ClearButton("everything", delegate { });
            gui.RemoveButton("Stuff", GUIWrapper.MiniButtonStyle.Right, delegate { });
        });

        gui.Button("toggle", () => { on = !on; Debug.Log("Toggled to: " + on); });
        gui.EnabledBlock(on, () =>
            gui.HorizontalBlock(() =>
            {
                gui.Button(on ? "on" : "off", () => Debug.Log("I'm on"));
                gui.Button(on ? "On" : "Off", () => Debug.Log("I'm on"));
                gui.Button(on ? "ON" : "OFF", () => Debug.Log("I'm on"));
            })
        );

        gui.IndentedBlock(GUI.skin.button, 1, () =>
        {
            var spToggle = property.FindPropertyRelative("toggle");
            gui.CustomFoldout(spToggle.boolValue, newValue =>
            {
                spToggle.boolValue = newValue;
                gui.HeightHasChanged();
            });
            if (spToggle.boolValue) {
                gui.Splitter();
                gui.HorizontalBlock(() =>
                {
                    gui.Label(">");
                    Rect foldRect = new Rect();
                    gui.GetLastRect(lastRect => foldRect = lastRect);
                    gui.Label("Turn " + (on ? "off" : "on"));
                    gui.GetLastRect(lastRect =>
                    {
                        if (GUI.Button(CombineRects(foldRect, lastRect), GUIContent.none, GUIStyle.none)) {
                            on = !on;
                        }
                    });
                    gui.FlexibleSpace();
                    gui.EnabledBlock(on, () =>
                        gui.ColorBlock(on ? Color.green : Color.red, () =>
                            gui.Button(on ? "on" : "off", () => Debug.Log("I'm on"))));
                });

          gui.HorizontalBlock(() => gui.NumericTextFieldLabel(1, "this is cool! hueheu"));
          gui.PropertyField(property.FindPropertyRelative("name"));

          gui.DragDropArea<GameObject>(
              @label: "Drag-drop GameObject",
              @labelSize: 15,
              @style: GUI.skin.box,
              @canSetVisualModeToCopy: dragObjects => true,
              @cursor: MouseCursor.Link,
              @onDrop: go => Debug.Log(go.name),
              @onMouseUp: () => Debug.Log("Click"),
              @preSpace: 20f,
              @postSpace: 0f,
              @option: new Option { Height = 80f });
```



Inspector

☑ N\A □ Static

Tag Untagged Layer Default

Transform

Position X 0 Y 0 Z 0

Rotation X 0 Y 0 Z 0

Scale X 1 Y 1 Z 1

GUIWrapper Test (Script)

Script GUIWrapperTest

Center: X 0 Y 0 Z 0

Extents: X 0 Y 0 Z 0

X 0 Y 0

W 0 H 0

🟩 🟪

FloatValue 0 IntValue 0

Method Attack

name cool

BUTTON + X -

toggle

off Off OFF

▼

> Turn on off

1- this is cool! hueheu

name cool

Drag-drop GameObject

Add Component

*8

Intro

- Awesomeness:

Delegates in C# are really powerful.
With delegates you could pass in code between methods as arguments which gives high flexibility, solve many design problems (circular depedency for instance), allow your modules to indirectly communicate with each other, promote loose coupling, increase cohesiveness and they give us the power of event-based programming.
Not to mention all LINQ methods are based around delegates. In short, delegates are pure awesomeness!

- Problem using awesomeness:

The problem with using delegates in Unity, is that Unity doesn't know how to serialize them by default. So if you set up a delegate with a list of subscribers at edit-time and enter playmode, the delegate value will revert to null (because it didn't serialize, it didn't make it to the C++ unmanaged side for it to return safely to the C# managed side when an assembly reload happens)
See Tim Cooper's video on the subject of serialization for more info (https://www.youtube.com/watch?v=MmUT0ljrHNc)

- Clarification:

Before going into the solution, there's something I want to make clear.
When talking delegates, you usually hear the following:

1- delegate
2- event
3- handler
4- subscriber
5- delegate method
6- Action
7- Func
8- the delegate's target object

Now I'm going to do a delegates 101 tutorial, for that there are many resources, like Jamie King for example
(highly recommended) https://www.youtube.com/playlist?list=PLAE7FECFFFCBE1A54

1- As you might know, a delegate is type that could reference methods with a particular signature
(you can think of them as managed and elegant function pointers).
A delegate from the inside has an invocation list,
methods are added to this list, invoking the delegate will invoke each method on this list.
For more information: http://msdn.microsoft.com/en-us/library/ms173171.aspx

2- Technically speaking, An event is a delegate - only difference is there is an added layer of abstraction and security that makes it impossible to set the delegate value directly, instead you're only allowed to add/remove methods.
In other words, with delegates it is possible to do:
do **myDel = value;** and **myDel X= value;** (where X: - or +)
But with events only **myEvent X= value;** is allowed (where X: - or +)
A lot of times, the terms "event" and "delegate" are used interchangeably

3, 4, 5- They're all the same. They all refer to a method that's hooked to the delegate - a method that the delegate will execute upon its invocation. So:
onClick += inventory.ReactToClick;
ReactToClick is the method/handler/subscriber that will be hooked into this delegate upon the execution of the previous statement.

6- Action is a built-in delegate that could hook up methods of the signature:
void MethodName()
There are 16 generic versions of Action to make it possible to hook up to methods with parameters.
So Action<int, string> is a delegate that accepts methods of the signature: void MethodName(int param1, string param2) and so on.

7- Func is like Action, except it has a return value (as opposed to Action which has no return (void))

Just like Action, it takes generic arguments. The last argument is always the return type.

So Func<Vector3, float, int> is a delegate that accepts methods of the signature: int MethodName(Vector3 param1, float param2);

8- The delegate's target object, is the object to invoke the handler method upon. So in the following statement:

myDel += target.handler;

myDel: obviously, the delegate

handler: the method to execute upon the delegate's invocation

target: the object to invoke the handler on

Cause as you might know, you need an object to invoke a member method on (unless if the method was static)

- Solution:

You might say, well why don't you just always hook the delegate up in OnEnable so that when an assembly reload happens, OnEnable gets called and the delegate is rewired? - Well that's not always the case. What if you wanted to wire a delegate at edit-time and have it persist after you enter playmode? - So we have to serialize the delegate.

The solution is to find a way to somehow serialize a delegate. There are two ways:

1- Serialize the delegate instance itself (which will in turn serialize its targets and methods).

2- Serialize its targets and methods independently and then later rebuild/recreate/rewire the delegate.

In uFAction, I did both of these approaches. The first for delegates that target any non UnityEngine.Object, and the second for delegates that target any UnityEngine.Object

The reason for this seperation is: when you target a System.Object, you can't recreate the delegate because like we said to recreate it, you'd have to have the targets and methods available to you, the problem with this is that the targets in this case are pure System.Objects which Unity can't serialize. So we use a serializer ourselves to serialize the whole delegate.

But when you target a UnityEngine.Object, it's a different story. If we try and apply the previous approach, we'll soon find out that it won't work because most serializers require you to have custom attributes/annotations on your classes for them to be serialized. BinaryFormatter for example requires [System.Serializable] to be present. And since we can't modify Unity's code to be able to add these attributes, we're not going to be able to use any serializer that requires annotations. And even if we use an annotation-free serializer, it's not safe to assume that the object will deserialize safely. Because, as you know you can't instantiate components via the 'new' operator, and there's a good reason for that. Unity does some home cooking/internal prepping to build the component which high chances are the serializer doesn't know about.

So we take the second approach, we serialize the targets and methods, and rebuild the delegate. But didn't I just say that it's hard to serialize UnityEngine.Objects? well yes, but we're not going to do it ourselves, we're gonna let Unity do it for us. Serializing the methods is easy, we could just use a strings to store the methods names, or a serialized MethodInfo (See SerializedMethodInfo class). Once we have the targets and methods, all that's left to do is a call to Delegate.Create

Class hierarchy

- Not going into details, but just to show you the whole picture.
  - At the top we have an abstract BaseDelegate from which we have our main two abstract branches, BaseUnityDelegate and SysObjDelegate

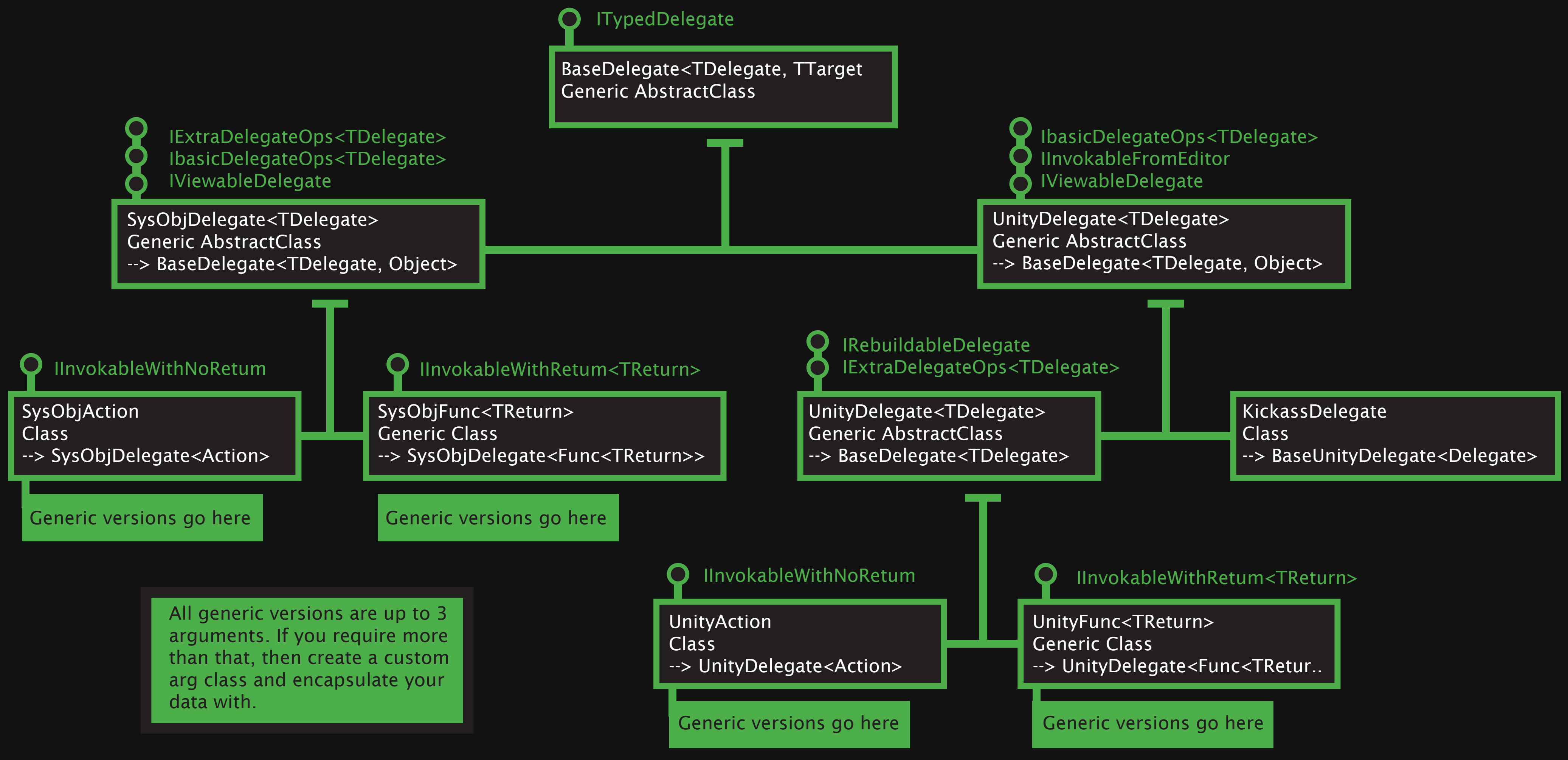
  - From SysObjDelegate we have SysObjAction and SysObjFunc. These are concrete delegates that you could instantiate. With these two, you could target any object that is **not** a UnityEngine.Object **nor contain any.**

  - From BaseUnityDelegate we have two branches, one abstract UnityDelegate and the other is a concrete KickassDelegate.

  - Like SysObjDelegate, from UnityDelegate we have UnityAction and UnityFunc. These are what you're going to be instantiating to target UnityEngine.Objects

  - KickassDelegate is a special delegate that could target UnityEngine.Objects too, what makes it special is that it accepts methods with no return (void) regardless of their parameters signature!

  - So you'll mostly be interested in: UnityAction, UnityFunc, KickassDelegate, SysObjAction and SysObjFunc.



ITypedDelegate

BaseDelegate<TDelegate, TTarget
Generic AbstractClass

IExtraDelegateOps<TDelegate>
IbasicDelegateOps<TDelegate>
IViewableDelegate

SysObjDelegate<TDelegate>
Generic AbstractClass
--> BaseDelegate<TDelegate, Object>

IbasicDelegateOps<TDelegate>
IInvokableFromEditor
IViewableDelegate

UnityDelegate<TDelegate>
Generic AbstractClass
--> BaseDelegate<TDelegate, Object>

IInvokableWithNoRetum

SysObjAction
Class
--> SysObjDelegate<Action>

Generic versions go here

IInvokableWithRetum<TReturn>

SysObjFunc<TReturn>
Generic Class
--> SysObjDelegate<Func<TReturn>>

Generic versions go here

IRebuildableDelegate
IExtraDelegateOps<TDelegate>

UnityDelegate<TDelegate>
Generic AbstractClass
--> BaseDelegate<TDelegate>

KickassDelegate
Class
--> BaseUnityDelegate<Delegate>

All generic versions are up to 3 arguments. If you require more than that, then create a custom arg class and encapsulate your data with.

IInvokableWithNoRetum

UnityAction
Class
--> UnityDelegate<Action>

Generic versions go here

IInvokableWithRetum<TReturn>

UnityFunc<TReturn>
Generic Class
--> UnityDelegate<Func<TRetur..

Generic versions go here

Usage

1- For non-generic delegates, the setup is very simple:

```
UnityAction onClick = new UnityAction();
KickassDelegate kickass = new KickassDelegate();
SysObjAction sysAction = new SysObjAction();
```

However, if you place one of these in a script you won't see anything visible in the inspector. To make a delegate visible you have to:

1- Make it public (or marked-up/annotated with [SerializeField] if it's not)
2- Annotate with [ShowDelegate("DelegateTitle")]

So:

```
[ShowDelegate("On Click")]
public UnityAction onClick = new UnityAction();
```

2- For generic delegates, there's one extra step. You'd first have to create a child class. (This is due to the limitation in Unity's serialization system not being able to serialize generic types) So:

```
[System.Serializable]
public class TransformAction : UnityAction<Transform> { }
```

And then just like before:

```
[ShowDelegate("On Me Action")]
public TransformAction meAction = new TransformAction();
```

Same thing for funcs:

```
[System.Serializable]
public class MyAwesomeFunc : UnityFunc<int, float, string, GameObject> { }

[ShowDelegate("On Awesome")]
public MyAwesomeFunc myFunc = new MyAwesomeFunc();
```

And that's it, easy and simple. No need to create any custom editors!

3- Delegate ops - the following applies to all delegates but KickassDelegate:

A) Adding:

You could add handlers to a delegate via myDel.Add(handler);
If you're using a non-generic delegate, you could do
myDel += handler;
You could still use the + sign in generic delegates, it's just that you have to do: myDel = (MyCustomDelType)(myDel + handler);
That's cause the + sign is overloaded in the generic class and so if you want to be able to do a += you'd have to overload the plus operator in your derived delegate (MyCustomDelType)

B) Removing:

Similar to adding: myDel.Remove(handler); - same deal with non-generics: myDel -= handler;

C) Invoking:

- For SysObjDelegates myDel.Invoke(prams_if_neccessary);
- For UnityDelegates, you can do the same of course, as well as myDel.InvokeWithEditorArgs(); which will invoke the delegate using whatever arguments set by you in the editor. This is relatively slower than invoking a delegate directly of course because invoking a delegate with selective arguments being passed to its methods, requires extra work.
I use MethodInfo.Invoke for each method passing it the right arguments.
For benchmarks about means of invocation, see:
http://byterot.blogspot.com/2012/05/performance-comparison-of-code.html
(My method pretty much sits in the middle of the benchmark chart ^ "Reflected after binding")
- It's worth mentioning that calling a regular myDel.Invoke(args) will ignore whatever arguments set in the editor and invoke the delegate with the passed arguments (if any)

D) Clearing:

- You could clear a delegate (which will wipe out its whole invocation list) by calling Clear - myDel.Clear();

E) Setting:

- You could set the delegate to a specific value by: myDel.Set(handler);
You could think of this as clearing the delegate and adding the specified handler.

F) Checking if a handler is contained within the delegate's invocation list:

- For that, you use myDel.Contains(handler);

- For more usage examples, see UnityActionTest, UnityFuncTest, SysObjActionTest and SysObjFuncTest.

4- More on ShowDelegate:

- ShowDelegate (or ShowDelegateAttribute) is a PropertyAttribute used to draw the delegates to make them visible in the inspector.
- ShowDelegate is not required for the delegate to serialize properly.
- It takes the delegate's title as its first argument, and could take 2 more optional arguments:

A) canSetArgsFromEditor:

By default it's true. If you pass it false, you can't manipulate the handlers' arguments (if they have any) from the editor.
Even though you like to have your delegate be visible in the inspector, sometimes it makes sense not to be able to have the ability to set the handers' arguments selectively and use those args when invoking the delegate.
An example of this is ComponentStateChanger (in the Examples folder).
With ComponentStateChanger you target a specific component on an arbitrary gameObject to change its state (enabled/disabled if it supports that). When the component's state changes, an onStateChanged delegate gets invoked, passing in the new state of the component. In this case, you don't want anybody to mess with the handlers' arguments in the inspector, because you'll always pass the new state and not some arbitrary boolean values.

B) forceExpand:

By default it's false. If you pass it true, the delegate's title header will always be unfolded and can't be folded.
Useful when you want to control the delegate's folding/unfolding by an external foldout from a custom editor.

5- More on KickassDelegate:

- Like we mentioned earlier, it accepts methods with no return (void) and any parameter signature. It uses reflection to achieve that
- Due to that, delegate ops are a bit different:

A) Invoking:

- Since it could have methods of arbitrary signatures hooked to it, it's not safe to assume a single Invoke signature.
That's why the only invocation available is InvokeWIthEditorArgs - which, as the name tells invokes the delegate with whatever arguments set from the editor.

B) Adding:

- All of the previous makes adding a bit tricky - I first didn't want to give the ability to add from code, but hey this is a delegate right? To add, you first pass in the handler (as usual) and whatever arguments the handler needs to be invoked (if any). There are two ways you could pass this information:

A) In the form of a direct value:

```
public void AddUsingValues(Delegate handler, params object[] directValues)
```

Example:

```
myDel.AddUsingValues((Action<int, float string>)handler, 5, 1.3f, "Hello");
```

The above Action cast is needed because the handler in the signature is in the form of Delegate.

B) In the form of a source value:

```
public void AddUsingSource(Delegate handler, Component source, string field)
```

Example:

```
myDel.AddUsingSource((Action<Vector3>)handler, transform, "position");
```

- This will use whatever value that was in the position property of transform when invoking this handler.
- Use source values instead of direct values when you want to defer the evaluation of the value till the moment you invoke the handler. Otherwise, if it's something trivial to compute, just use direct values.
- If you have more than one argument, use the other source overload:

```
public void AddUsingSource(Delegate handler, params SourceSet[] sets)
```

Example:

```
myDel.AddUsingSource(
            (Action<Vector3, int>)MethodThatTakesVector3AndInt,
            new SourceSet(transform, "position"),
            new SourceSet(transform, "childCount")
);
```

- Lastly, if the handler doesn't take any arguments, you could just use:

```
myDel.Add((Action)myHandler);
```

- For more usage examples, see KickassDelegateTest.
Also read the KickassDelegate code documentation to see what exceptions might get thrown

C) Removing:

- It's easy you just have to pass the handler:

```
myDel.Remove((Action)handler);
```

D) Clearing:

- Just call Clear:

```
myDel.Clear();
```

The editors

- You won't see any editor if you don't annotate with [ShowDelegate].
There are 3 types of editors available:

1) SysObjDelegates (Actions/Funcs) only have a Readonly editor which will allow you to see what targets and methods are hooked.

2) UnityDelegates (pretty much all that inherit BaseUnityDelegate) have two type of editors: Mini and Advanced.

A) Mini is more compact, takes less space, a bit easier to use.

1) The title header foldout.

2) The GoToSettings Button - takes you to the settings asset file.

3) The SwitchViewStyle Button - switches editor views (between Mini and Advanced)

4) Used to invoke the delegate. If the delegate takes arguments, it will use whatever arguments set in the editor.
This button is not available if canSetArgsFromEditor was false.

5) Clears all entries (**This is undoable, so it's safe)**

6) Click to toggle advanced mode - Allows you to re-order handlers (which will actually change the order they're invoked by)

7) The gameObject field. Left click to ping, double click to actively select the gameObject.
Click on the thumb to browse the scene for a gameObject. You can drag/drop a gameObject to it.
And you can also drag the field around by holding left click and dragging the mouse.

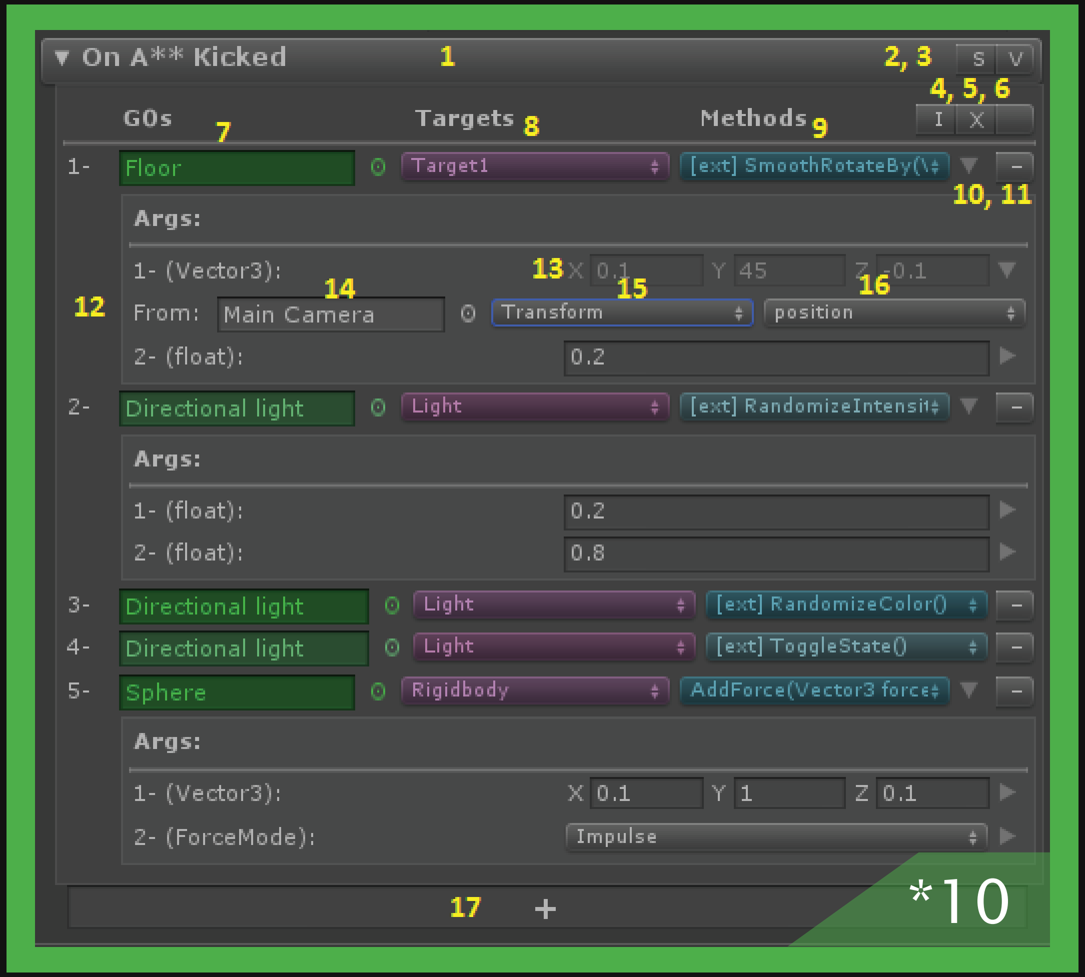
8) The targets field. Right click to ping, double right click to actively select the target.
This field is a popup if you're targeting components from a gameObject, and a static label field if you're targeting a UnityEngine.Object that is not a Component.

9) The methods popup - to change what methods to show, adjust the options in the settings.

10) The arguments foldout - only availble if a method has arguments and canSetArgsFromEditor is true.

11) The remove button - removes the current entry

12) The arguments area - set what arguments to pass to your handlers here -
There are two ways
of setting them, directly or from a source.

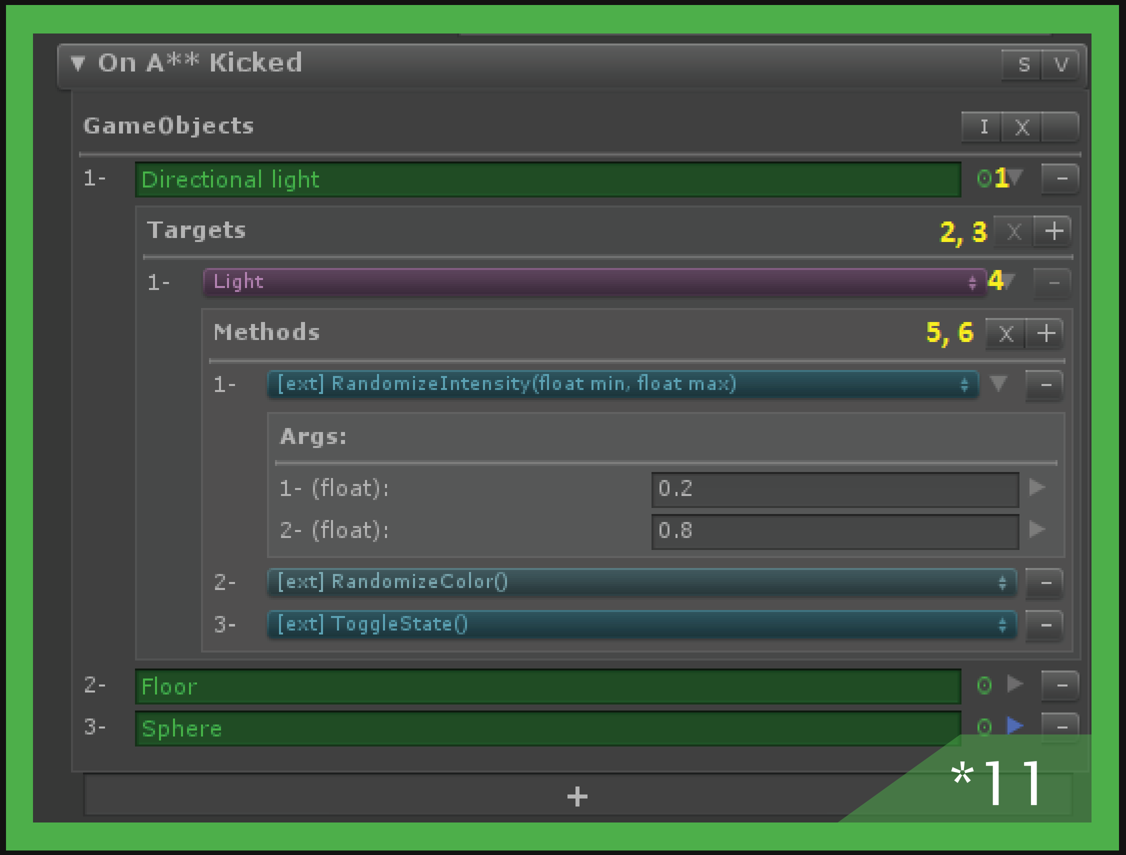


13) This is where you can directly set your argument value
14) Or you can choose a value from a source. This is the source gameObject
15) The source component
16) And the source value: Anything from a variable, property or a parameterless method with the appropriate return type can be selected from here. To adjust the bindings for these values, adjust the SourceBindings option in the settings.
17) The adding area:

Click to show a selection window from which you can add a certain gameObject.
Or drag/drop a UnityEngine.Object to add it.

B) Advanced editor has more features, you can add/remove targets/methods from a gameObject/target with ease. So if you wanted to add another method from the same target, you don't have to drag/drop the target's gameObject again.
Advanced editor also has better performance, this is due to the fact that less controls are drawn. Imagine you had 5 entries, in Mini editor a total of 15 controls will be draw (each entry a gameObject field, a target and a method field)
in Advanced editor, it could be much less depending on the gameObjects/targets.
You won't get any duplicate entries, all entries (gameObjects and targets) are unique.
So targets of the same gameObject will be grouped under one gameObject,
and methods of the same target will be grouped under one target. (*11)

1) gameObject entry foldout - shows/hides the target entries
2) Clear targets in current gameObject entry (can be undone safely)
3) Add a target entry **(can be undone safely)**
4) target entry foldout - shows/hides the method entries
5) Clear methods
6) Add a new method **(can be undone safely).**



▼ On A** Kicked

GameObjects

1- Directional light

Targets

1- Light

Methods

1- [ext] RandomizeIntensity(float min, float max)

Args:

1- (float): 0.2

2- (float): 0.8

2- [ext] RandomizeColor()

3- [ext] ToggleState()

2- Floor

3- Sphere

3) A couple of notes:

A) When changing editor views, the delegate data remains the same. It's just that it gets represented differently by each editor.

B) There's a data integration that happens for instance when you switch views, or when the delegate drawer gets active (when you click on a gameObject that has the delegate) during this integration, a filtering occurs, which will remove any null gameObject entry, target entry or method entry. So if lets say you manually deleted a gameObject that was an entry in your delegate, when you go back to your delegate editor you won't see it because it's now null thus filtered.

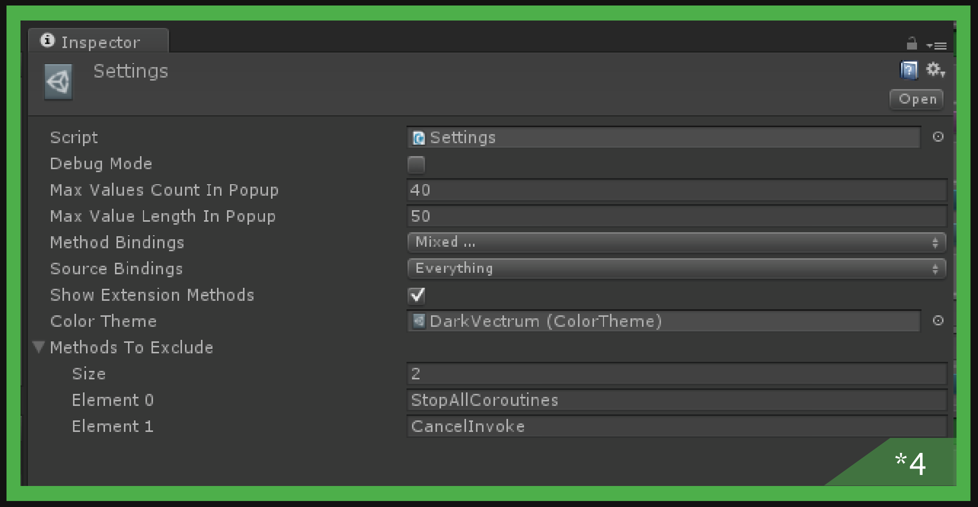
Settings and customization

- To get to the settings asset, click the "S" button in any of the delegate's editor view styles, or go "**Component | uFAction | Settings**"

- There, you will see the following settings (*4):

1- DebugMode:

Useful at development time to help find bugs via console logs.



Inspector

Settings

Script	Settings
Debug Mode	□
Max Values Count In Popup	40
Max Value Length In Popup	50
Method Bindings	Mixed ...
Source Bindings	Everything
Show Extension Methods	☑
Color Theme	DarkVectrum (ColorTheme)
Methods To Exclude	
Size	2
Element 0	StopAllCoroutines
Element 1	CancelInvoke

*4

2- MaxValuesCountInPopup:

The maximum number of options/values/entries allowed in a popup. If the number of options was higher than this max, a selection window is used instead.

3- MaxValueLengthInPopup:

The maximum length (number of characters) a popup option/value/entry is allowed to have. If an entry had a higher length, a selection window is used instead.

4- MethodBindings:

The BindingFlags that's used when reflecting a target's methods.
Public: Show public methods
DeclaredOnly: Show the declared-only methods in a target (inherited methods won't be shown)
NonPublic: Show private/protected methods
NOTE: you have to play nice with these settings when manipulating the delegate from code. So if you try to add a private method and NonPublic is not ticked, or an inherited method but DeclaredOnly is ticked or a Public method but Public is not ticked, you'll get an InvalidOperationException. For more info see AssertHandlerAndSettingsArePlayingNice in BaseDelegate.

5- SourceBindings:

The BindingFlags that's used when using a source value when setting a handler's arguments

6- ShowExtensionMethods:

Whether or not to show Extension methods

7- ColorTheme:

The color theme used in the delegate editors. You can create and customize your own: Component | uFAction | CreateTheme

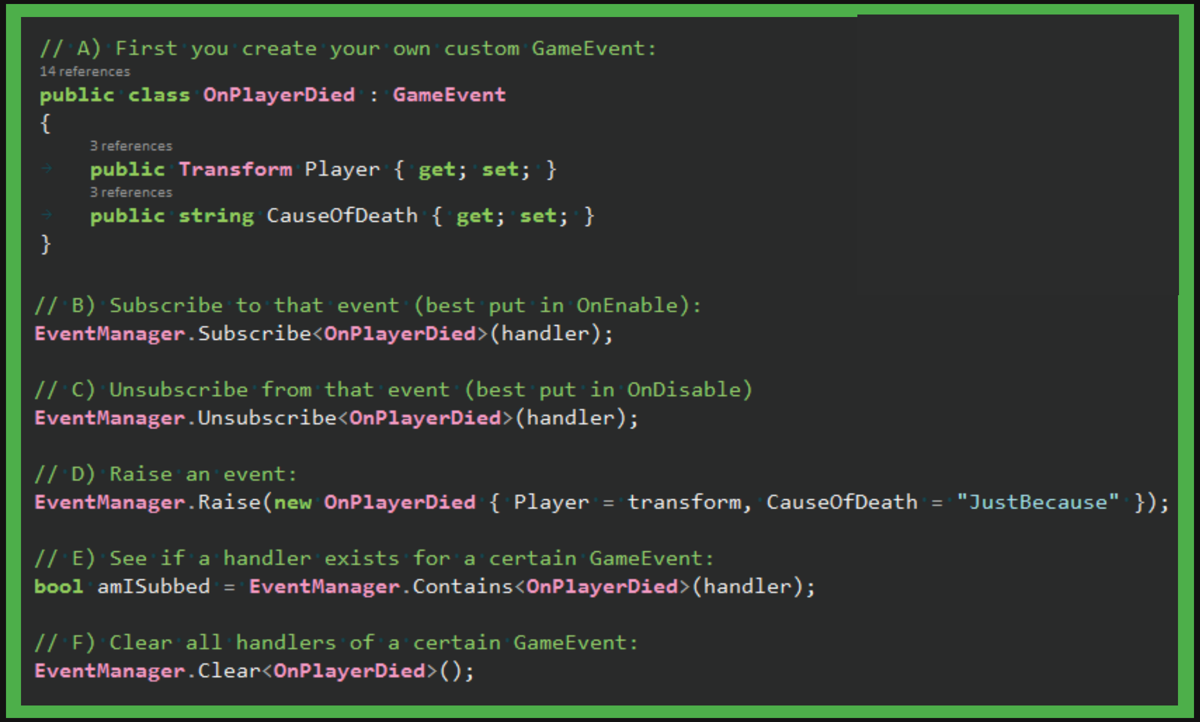
8- MethodsToExclude:

Put whatever methods you don't want to see in a target's methods popup in this list.

- When you're done adjusting the settings, you could press Ctrl+Shift+- to go back to the previously selected object. So if you came to the settings from the "S" button, going back will take you to the delegate.

Bonus

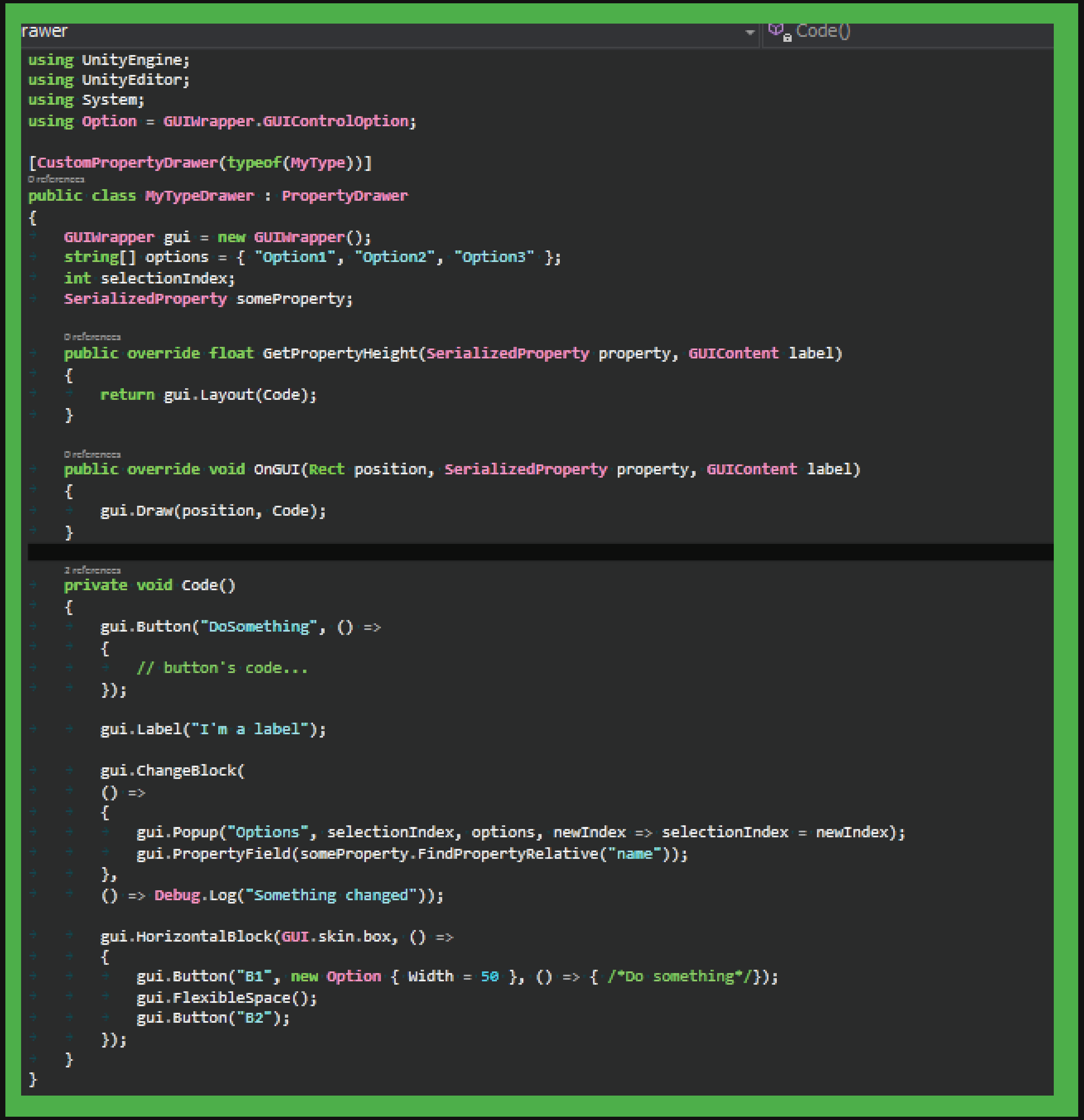
1- First is the EventManager through which you could globally (un)subscribe



```
// A) First you create your own custom GameEvent:
14 references
public class OnPlayerDied : GameEvent
{
    3 references
    public Transform Player { get; set; }
    3 references
    public string CauseOfDeath { get; set; }
}

// B) Subscribe to that event (best put in OnEnable):
EventManager.Subscribe<OnPlayerDied>(handler);

// C) Unsubscribe from that event (best put in OnDisable)
EventManager.Unsubscribe<OnPlayerDied>(handler);

// D) Raise an event:
EventManager.Raise(new OnPlayerDied { Player = transform, CauseOfDeath = "JustBecause" });

// E) See if a handler exists for a certain GameEvent:
bool amISubbed = EventManager.Contains<OnPlayerDied>(handler);

// F) Clear all handlers of a certain GameEvent:
EventManager.Clear<OnPlayerDied>();
```

2- Second is a really cool GUIWrapper that lets you use GUILayout-like methods in GUILayout-restricted areas (like PropertyDrawer.OnGUI)
(See the TestDrawer in the examples)

Example:



```
using UnityEngine;
using UnityEditor;
using System;
using Option = GUIWrapper.GUIControlOption;

[CustomPropertyDrawer(typeof(MyType))]
0 references
public class MyTypeDrawer : PropertyDrawer
{
    GUIWrapper gui = new GUIWrapper();
    string[] options = { "Option1", "Option2", "Option3" };
    int selectionIndex;
    SerializedProperty someProperty;

    0 references
    public override float GetPropertyHeight(SerializedProperty property, GUIContent label)
    {
        return gui.Layout(Code);
    }

    0 references
    public override void OnGUI(Rect position, SerializedProperty property, GUIContent label)
    {
        gui.Draw(position, Code);
    }

    2 references
    private void Code()
    {
        gui.Button("DoSomething", () =>
        {
            // button's code...
        });

        gui.Label("I'm a label");

        gui.ChangeBlock(
        () =>
        {
            gui.Popup("Options", selectionIndex, options, newIndex => selectionIndex = newIndex);
            gui.PropertyField(someProperty.FindPropertyRelative("name"));
        },
        () => Debug.Log("Something changed"));

        gui.HorizontalBlock(GUI.skin.box, () =>
        {
            gui.Button("B1", new Option { Width = 50 }, () => { /*Do something*/});
            gui.FlexibleSpace();
            gui.Button("B2");
        });
    }
}
```

THANK YOU FOR READING

uFAction