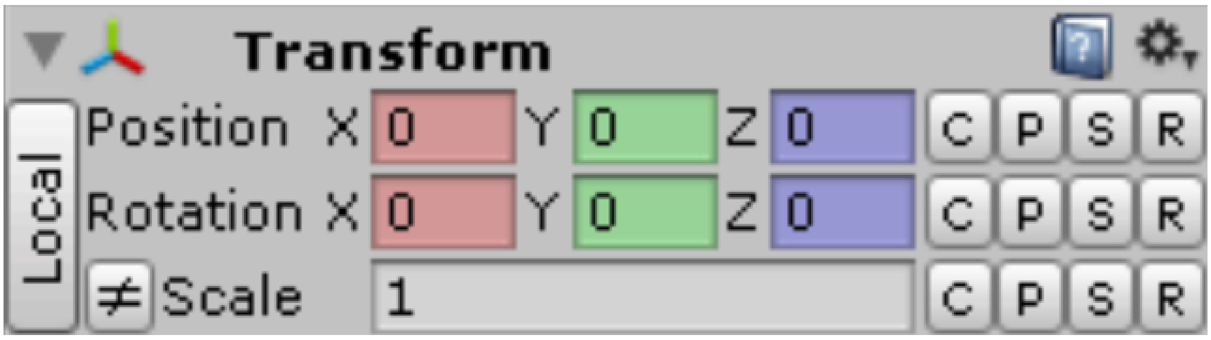
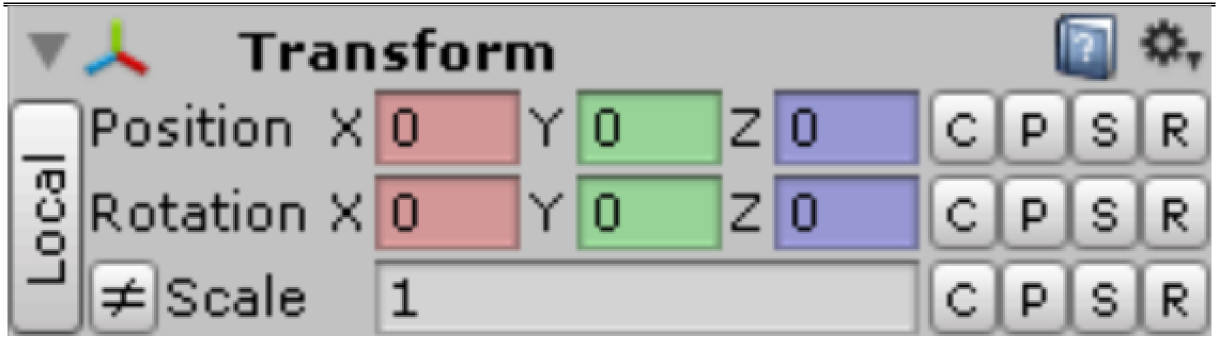
Inspector Gadgets

v2.2 Created by Kybernetik



Contents

Transform Inspector



Utility Buttons

You can use the buttons on the right to [**Copy**], [**Paste**], [**Snap**], and [**Reset**] individual transform elements (position, rotation, scale).

- These buttons are greyed out when they would do nothing. I.E. Copy and Paste are greyed out if the clipboard contains the same values as the selected object, Snap is disabled if the object is already snapped, and Reset is disabled if the object is already at the default values.
- Copied values are retained until Unity reloads the Assemblies, such as when you modify a script and Unity recompiles it or when you enter play mode.
- The snap buttons use Unity's own snap settings located in *Edit/Snap Settings*.
- The *Edit/Selection/New Locked Inspector (Ctrl + Alt + S)* menu item opens an inspector window locked to the current selection so you can easily compare and copy values between different selection sets.

Uniform Scale

Most of the time when you scale an object you want to do so on all axes so that its proportions remain the same. When the object's scale is uniform (same value on all axes) it will be shown as a single value so you don't need to enter the same value multiple times.

- You can toggle between uniform scale mode and the regular vector mode using the [**≠**] button on the left.
- If the scale isn't already uniform when you enter uniform scale mode, it will be set to the average of the current values.

World Space

The [**Local**] button on the left allows you to view and edit the transform in world space instead of local space. It will automatically revert to local space when you select a different object.

Pro-Only Features

Inspector Gadgets Pro includes various additional features which are not available in the Lite version.

- Multi-object editing (the Lite version reverts to the default Transform inspector when multiple objects are selected).
- You can middle click on any Transform field to reset it.
- Any Transform field which isn't a multiple of the snap value will be shown in italics.
- You can disable any of these features in the Transform Context Menu (using the cog icon on the right).

MonoBehaviour Constructors [Experimental]

Unity does not allow you to pass parameters into a constructor when you create a component as you could with a regular class. There are legitimate reasons for this limitation and it is unlikely to ever change, so this plugin provides two different workarounds: the `[RequireInitialisation]` attribute and the `ConstructableBehaviour<T>` class (both in the `InspectorGadgets` namespace), each with different advantages as outlined in the following table.

	ConstructableBehaviour	**RequireInitialisation**
Runtime performance cost	Very small	None
Specify "constructor" parameters before `Awake` is called.	✓	
Allows you to specify when the restriction applies	✓	✓
Allows any number of parameters or any other complex initialisation		✓
Supports the `[RequireAssignment]` attribute		✓

A simple example of both features can be found in *Assets/Plugins/Inspector Gadgets/Examples*.

Note that both of these features are fairly new and may not be the best way of doing things. Any feedback would be greatly appreciated.

Inspector Gadgets Lite disables these features after 5 instances are created, allowing you to try them out but not use them properly. To remove this limitation, please purchase Inspector Gadgets Pro.

Constructable Behaviour

Inherit from this class instead of `MonoBehaviour` and override the `OnConstructor` method. The `T` type is the type of parameter you want.

- This means that you will not be able to create instances of your script using the regular `GameObject.AddComponent` method (doing so will throw an exception). Note that this verification is only performed in the Unity Editor, not at runtime.
- Instead, you will need to use `InspectorGadgets.Utils.AddComponent` (an extension method). This forces you to provide a parameter of type `T` which is passed into the `OnConstructor` method.
- The `OnConstructor` method is called before the `Awake` event, allowing you to get the required parameter beforehand. Note that this means the call occurs before the component has been initialised by Unity, so none of the `MonoBehaviour` properties such as `name` and `gameObject` can be used. It may also be called from a non-main thread, preventing access to many of the other Unity Engine features.
- If you pass `false` into the constructor (`MyScript(allowSerialization = false)`), then it will also throw an exception whenever an instance is created by any method other than `Utils.AddComponent` (such as if the user manually adds the script to an object or if it is deserialized after being saved in a prefab or scene).

Require Initialisation Attribute

- When this attribute is applied to a `MonoBehaviour` script, all instances of that script must have `InspectorGadgets.Utils.MarkAsInitialised()` called on them as soon as they are created, otherwise a warning will be logged.
- The `MarkAsInitialised` method can be used as an extension method, I.E. `this.MarkAsInitialised();` from within the script itself (as long as you have `using InspectorGadgets;` at the top of the script.
- This system operates in the Unity Editor only, and has no effect at runtime.

For example, you might have a `Projectile` script which always needs to be given a position, velocity, damage value, needs to know who fired it, etc. If you want to make sure that a `Projectile` is never created without being given those details, you can give it a `[RequireInitialisation]` attribute and give it a `Fire` method which takes all the required parameters, applies them, and then calls `MarkAsInitialised` on it. Then whenever you add the Projectile component to an object or instantiate a prefab with one on it, you will get a warning if you don't call that method.

Note that the system won't stop you from making an instance and calling `MarkAsInitialised` manually instead of calling `Fire`, but you would need to know the system is in place and do this intentionally, so there doesn't seem to be any point in trying to prevent it.

Initialisation Types

The `InitialisationType` enum is used by `[RequireInitialisation]` to determine when the requirement should actually be enforced. By default, the enforcement will take place when an instance is created using `GameObject.AddComponent` or as part of an `Object.Instantiate` call, but not when an instance is deserialized (such as when loading a scene or prefab). You can change this behaviour by passing a different combination into the attribute's constructor.

Require Assignment

The `[InspectorGadgets.RequireAssignment]` attribute has two aspects to it. The first is that any field with the attribute will be highlighted in red in the inspector if that field still has its default value (false, 0, null, etc. depending on the field type). Note that this feature does not currently work with nested serializable classes.

The second aspect requires the class containing the attributed field to have a `[RequireInitialisation]` attribute and works as part of that system. If such a class has any fields with `[RequireAssignment]` attributes, it will be considered to be "initialised" to avoid the error message if all those fields are assigned non-default values or if `MarkAsInitialised` is called on it.

Script Inspector [Pro-Only]

This section applies to `MonoBehaviour`, `ScriptableObject`, and `StateMachineBehaviour`.

Decorator Attributes

Unity has several attributes which can be used to easily customise the appearance of the inspector without needing to write a custom inspector class: Header, HideInInspector, Multiline, Range, SerializeField, Space, TextArea, Tooltip. The `[RequireAssignment]` attribute mentioned above is also a decorator attribute.

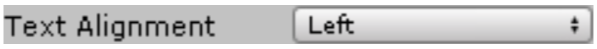
The `InspectorGadgets` namespace contains some additional attributes to allow for even greater customisation without the need to write your own custom inspector.

Toolbar

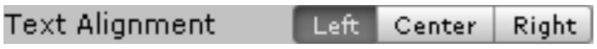
Normally Unity draws enums as dropdown fields. Unfortunately this isn't always what you want because you can't see all the options without opening the list, you always need to open the list to change the value, and it doesn't properly support enums which use bitwise flags.

- Placing the `[Toolbar]` attribute on an enum field causes it to instead be drawn as a series of buttons.
- If the enum has too many values to fit into the available space, you can scroll the toolbar across using your scroll wheel while the mouse cursor is pointing at it or you can drag the field label (just like a number field).
- Toolbars for enums marked with the `[System.Flags]` attribute allow you to select and deselect values individually.
- The `[Toolbar]` attribute can also be used for `string` fields. You simply specify the allowed values in the attribute's constructor and they will be displayed as a toolbar.

```
public TextAlignment textAlignment;
```



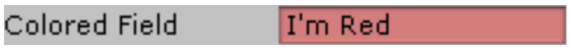
```
[InspectorGadgets.Toolbar]
public TextAlignment textAlignment;
```



Color

The `[Color]` attribute simply changes the color of a field, allowing you to highlight it or group multiple fields together visually.

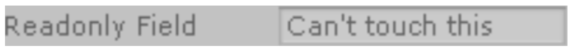
```
[InspectorGadgets.Color(1, 0.5f, 0.5f)]
public string coloredField = "I'm Red";
```



Readonly

The `[Readonly]` attribute causes a field to be greyed out in the inspector so the user can't edit it.

```
[InspectorGadgets.Readonly]
public string readonlyField = "Can't touch this";
```



Inspectable Attributes

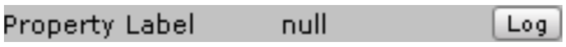
Unlike the above attributes which inherit from `InspectableAttribute` and alter the way Unity shows a field in the inspector, the following attributes inherit from `InspectableAttribute` and add extra elements at the bottom of the inspector.

These attributes have a `When` property which can be set in the constructor to determine when they are drawn: always (default), in play mode, or in edit mode.

Label

The `[Label]` attribute adds a read-only label to display the value of any field, property (with a getter), or method (with no parameters), even static ones. The label also has a button to log the current value to the console.

```
[InspectorGadgets.Label]
string PropertyLabel { get; set; }
```



Button

The `[Button]` attribute adds a button in the inspector which the user can click to invoke the attributed method.

- The will automatically be given the method's name (with spaces between words) unless you specify a different `Label` in its constructor.
- If it is an instance method and multiple objects are selected, the method will be called once for each selected object.
- If the method has a non-void return type, the returned value will be logged.
- If you assign `SetDirty = true` in the attribute's constructor, it will automatically call `UnityEditor.EditorUtility.SetDirty` on the target after invoking the method to make sure changes are saved properly.

```
[InspectorGadgets.Button]
void InspectorButton() { }
```

Inspector Events

If Decorator and Inspectable Attributes aren't able to achieve the level of customisation you want, you can simply add a method called `AfterInspectorGUI` to your script and Inspector Gadgets will automatically call it after drawing the script inspector (so you can draw additional GUI elements below it). Or you can call the method On`InspectorGUI` to replace the regular inspector entirely.

Custom Inspectors

If you require more customisation than you can achieve using the available events and decorator attributes, you will need to create your own custom editor class. You can do this quickly using *Ctrl + Middle Click* in the target script's inspector or using the *Create Editor Script* context menu command.

By inheriting from `InspectorGadgets.Editor<T>` instead of `UnityEditor.Editor` your custom editor will inherit the features described above (hide Script field, Middle Click to open script, support for Inspectable Attributes and Inspector Events). You will also be able to access the inspected objects directly using the `Target` and `Targets` properties rather than using the regular `target` and `targets` properties which you always need to cast to the script type yourself.

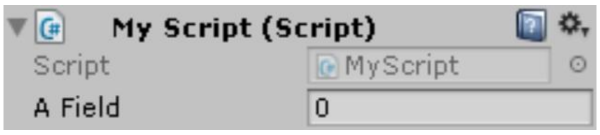
Script Field

The *Script* field at the top of Unity's script inspector is useful because you can click on it to find the script asset or double click to open it in your script editor application, but most of the time it just wastes space and adds to the clutter of the inspector. Inspector Gadgets hides the *Script* field and allows you to do those tasks other ways:

- Open a script by *Middle Clicking* anywhere in its inspector. You can also *Ctrl + Middle Click* to open its custom inspector script (or create a one if it doesn't exist, see Custom Inspectors).
- Find a script asset using the *Ping Script Asset* command in its context menu.

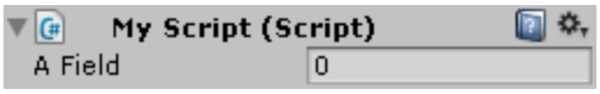
```
// Example.
class MyScript : MonoBehaviour
{
    public int aField;
}
```

Unity Default



My Script (Script)
Script: MyScript
A Field: 0

Inspector Gadgets



My Script (Script)
A Field: 0

Questions, feedback, feature requests, etc: kybernetikgames@gmail.com.