

DustProductions.Core

Overview Guide

Last updated: 23 Feb 19

Online Documentation

docs.productionsofdust.com

Table of Contents

Online Documentation	1
Table of Contents	1
Frequently Asked Questions	3
Where do I go to get help?	3
So what is DustProductions.Core, exactly?	3
Do you have documentation on the API?	3
How to use DustProductions.Core	4
Attributes and Property Drawers	4
HideFieldAttribute	4
Simple Example	4
Advanced Usage Example	5
QuickButtonAttribute	6
Example	6
StringDropDownAttribute	6
Example	7
ScriptableObjects	7
CursorDefinition	8
TransitionDefinition	10
ValueSetters	13
ValueSetter	13
FloatValueSetter	13
AudioSourceVolumeSetter	14

CanvasGroupAlphaSetter	14
LightIntensitySetter	14
ColorValueSetter	14
GraphicColorSetter	14
LightColorSetter	14
TransformValueSetter	14
CanvasGroupInteractableSetter	15
Utilities	15
MonoBehaviours	16
Comment	16
CursorManager	16
DeveloperInformationMonoBehaviour	18
DontDestroyOnLoad	18
Singleton	18
Libraries	19
CoroutineEssentials	19
CurveInterpolator	19
DiskUtilities	19
ExtensionMethods	19
Interpolate	19
OpenInFileBrowser	19
ReflectionUtilities	19
UIUtilities	19
Other Stuff	20
Cacheable	20
DeveloperInformation	20
IEditorComponentAddedHandler	20
Locker	20
MultiValueDictionary	20
ScriptableObjectFactory	20
SerializableCursorDefinitionStack	20
SerializableStack	21
TransformValues	21

Frequently Asked Questions

Where do I go to get help?

The first thing to do is check out this FAQ and guide or the [online documentation](#) for answers.

If your problem is not answered in those places, the next fastest way to get support is through GitLab. It's possible an issue is already open for your problem.

GitLab: <https://gitlab.com/DustProductions/DustProductions.Core/issues>

If you cannot access GitLab, or you just have a quick question or something, you can email us as well.

Email: support@productionsofdust.com

So what is DustProductions.Core, exactly?

Core is a collection of useful assets I use in all of my Unity projects to make development faster and easier, including Trigger Me Timbers. But it's not just me and Trigger Me Timbers that gets to benefit from Core. As a result of purchasing Trigger Me Timbers, you also get access in your projects.

Inside you will find plenty many things that Trigger Me Timbers leverages to be effective, the ScriptableObject **TransitionDefinition** is a great example. It allows you to define what is essentially an animation that can happen over time. You can base it off of pre-created transitions, or make your own custom transition with a curve editor. The transitions can then be plugged in to something like **GraphicColorSetter**, which then means, instead of a color changing from white to blue instantly, it happens over whatever amount of time and using the animation curve defined in the **TransitionDefinition**. For programmers, there's all sorts of useful attributes and boilerplate saving utilities that we'll go over later.

Do you have documentation on the API?

Yep, we sure do. You can find that on our website at docs.productionsofdust.com

How to use DustProductions.Core

As mentioned, there are plenty of time saving features of DustProductions.Core for use in Unity. First we will talk about the Attributes and Property Drawers, then the ScriptableObjects, then Setters, then the Utilities.

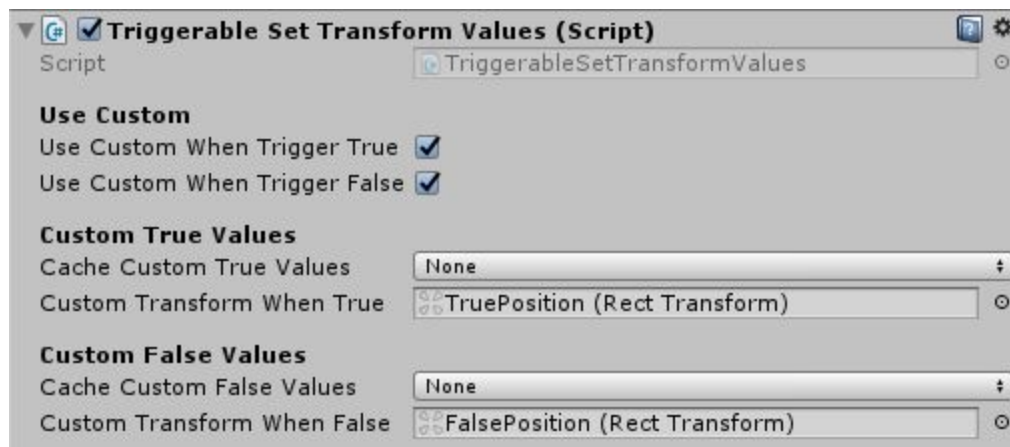
Attributes and Property Drawers

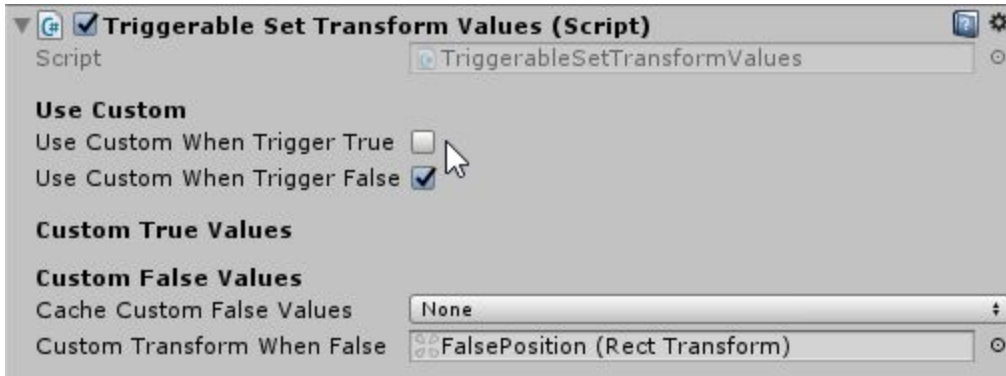
Property Drawers are a nice way to give extra functionality to your inspector fields without having to write Custom Editors. This way, if you want to change the appearance of a field in the inspector, you just tag it with a custom Attribute, instead of having to maintain a bunch of unwieldy editor scripts, often duplicating a bunch of code. You can [read more about Property Drawers in here](#). For now, we'll dive in to the ones Core provides you.

HideFieldAttribute

Allows hiding / disabling of fields without writing fancy custom editors. The simplest usage involves passing in the name of another serialized field that's a boolean. If that boolean is true, the field will be drawn in the inspector, if it's false, it will not be drawn.

Simple Example





```
[SerializeField]
protected bool UseCustomWhenTriggerTrue = true;

[SerializeField]
[HideField("UseCustomWhenTriggerTrue")]
CachePromptType CacheCustomTrueValues;
```

Additional options allow for inverting the output, or choosing not to completely hide fields, just disabling them.

Advanced usage allows you to pass in multiple conditional sources, checking ints and floats against certain values, checking the type of a reference, checking a string against another string, etc.

Advanced Usage Example

```
[SerializeField]
private Transform _TransformToSet;

[SerializeField]
[HideField("_TransformToSet", CompletelyHide = false)]
private bool _ModifyLocalScale;

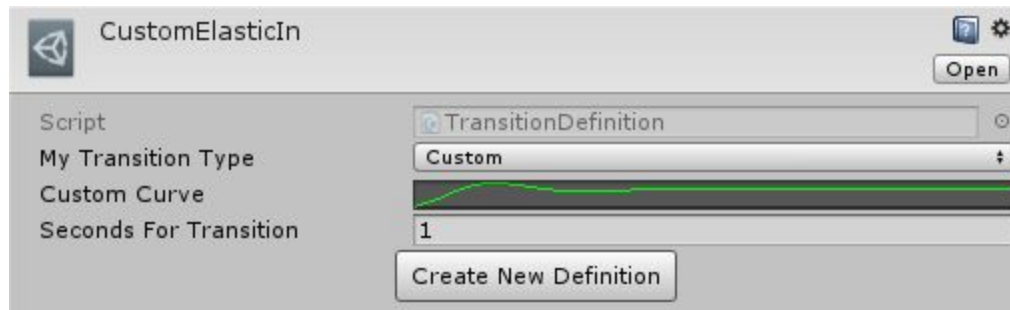
[SerializeField]
[HideField("_TransformToSet", typeof(RectTransform))]
private bool _ModifyAnchors;
```

_ModifyLocalScale will be disabled in the inspector unless **_TransformToSet** is not a null value. **_ModifyAnchors** will not be drawn at all unless **_TransformToSet** is a RectTransform.

QuickButtonAttribute

Use this on a serialized field to quickly add a button to the inspector without having to write custom inspector code. This is an abuse of the PropertyDrawer system to make it so instead of drawing a field, it will draw a button that can run a method in your script instead.

Example



```
[SerializeField]
[QuickButton("EDITORONLY_CreateNewDefinition", false)]
private bool CreateNewDefinition;

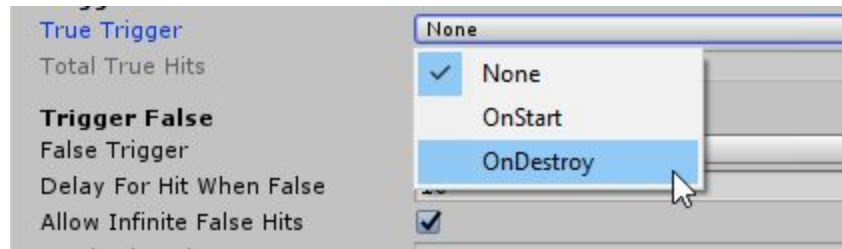
private void EDITORONLY_CreateNewDefinition()
{
    // Make a new Transition Definition Code
}
```

Additional options include the ability to define a minimum width of the button, label the button (by default it will take the name of the field), and choose whether to display the value of the field on the button itself (the above example passes in **false** for this because we don't care about the boolean value of **CreateNewDefinition**).

StringDropDownAttribute

Allows drawing a dropdown for strings in the editor, similar to how enums are drawn. Although I'd probably recommend using enums in most situations you might use this, there are some good reasons to use strings as well. This allows you to pick a string from a set of strings for a field, instead of having to type the string manually.

Example



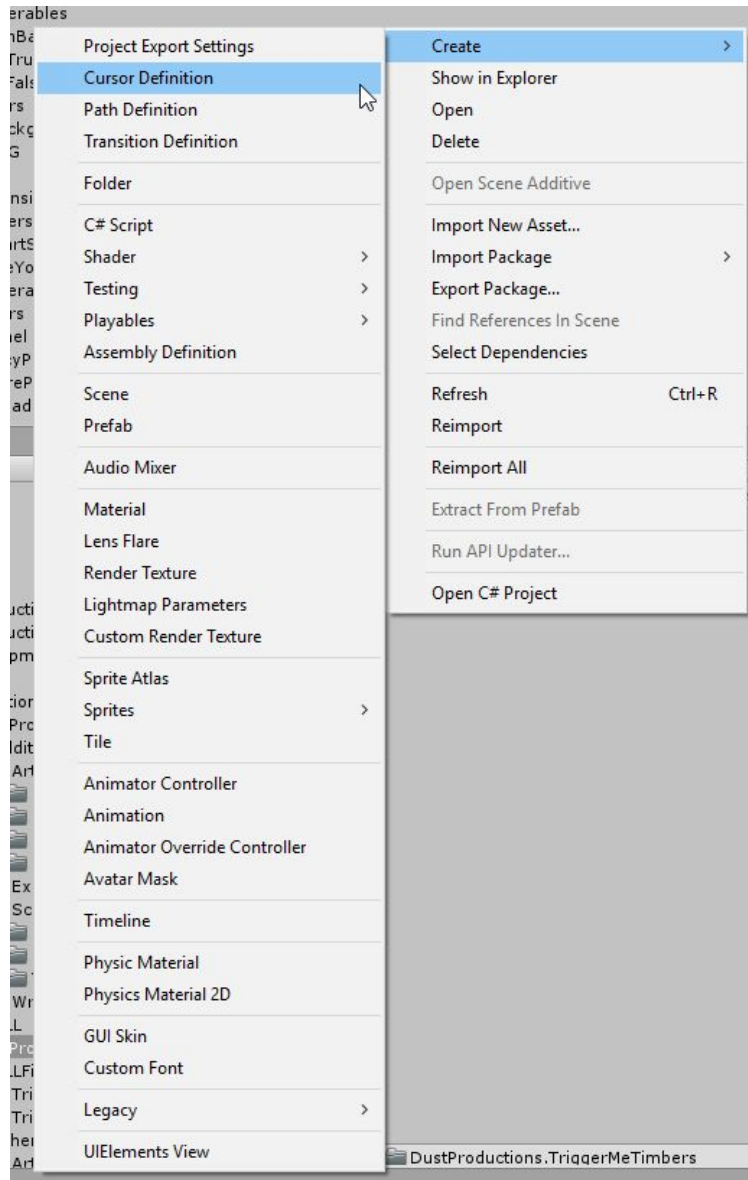
```
[SerializeField]
protected string[] TriggerDropDownOptions = new string[] { "OnStart",
"OnDestroy" };

[SerializeField]
[StringDropDown("TriggerDropDownOptions", "None")]
protected string TrueTrigger = "None";
```

Put the tag on a serialized string field and pass in the name of a string array. Optional parameters to pass in a string that will be in the list before all of the items in the string array (in this example “None” will always be present in the dropdown). You can also pass in another array (**Note**: this array must have an object reference in order to work), and it will be added to the dropdown (after the single string parameter, but before the other values).

ScriptableObjects

ScriptableObjects act like blueprints to create a variety of assets for use in your game. You can assign them to values in the inspector, but they cannot be added onto GameObjects like MonoBehaviour Components can. So, what’s the point? Well, the point is that they are like plug and play ways to modify behaviour on an object without having to write or compile code to do so. You can [read more about ScriptableObjects in Unity’s documentation here](#). The easiest way to create a new ScriptableObject asset is by right clicking in the Project view and going to the Create menu.

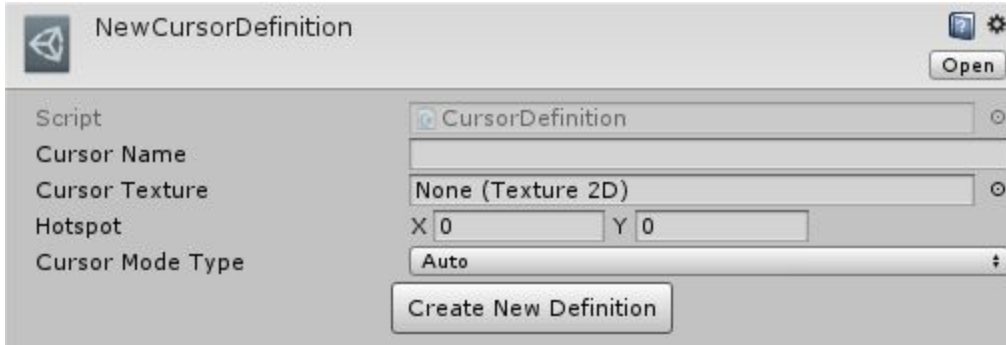


There you will see the ScriptableObjects at the top of the list. Let's talk about the ScriptableObjects you get access to with Core.

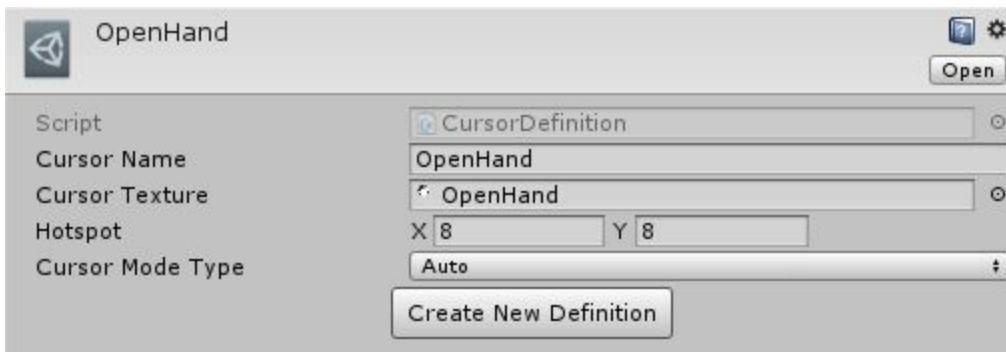
CursorDefinition

This is an asset to hold a cursor to use in game. You might want to polish your game by using a custom cursor instead of the default arrow that comes with the operating system. If so, this is how you would do it. First create a **CursorDefinition** asset. I'm going to create one for this

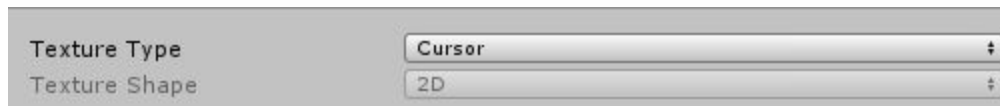
texture:  Here's what a newly created definition looks like.



Once you have created your “NewCursorDefiniton”, you should give it a name (in this case, mine will be named **OpenHand**). Here’s what it looks like all filled out.



Then you assign the texture to be used as a cursor. It’s important this texture be imported as a Cursor, by clicking on the texture in the Project view and setting it in the inspector.



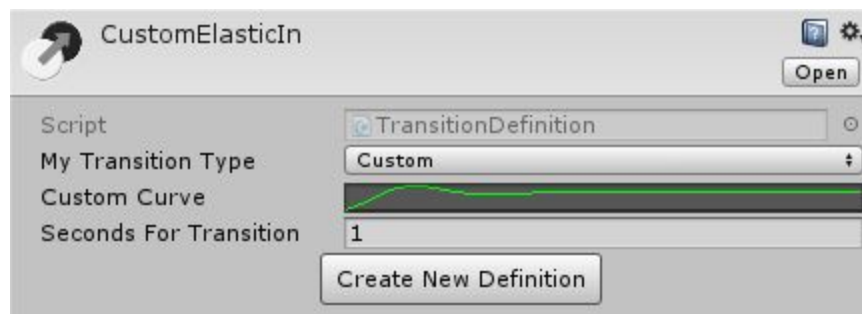
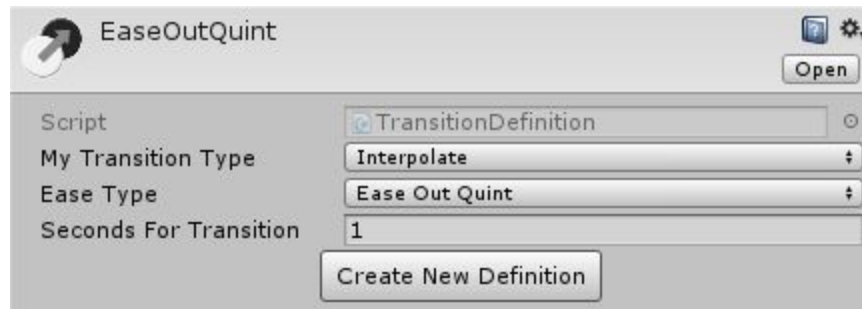
Then you have to assign the hotspot. The hotspot is the pixel on the texture that is actually where the clicking will happen. Measurements start in the top left, and are positive numbers on the x and y axis. My hotspot is right in the middle of the hand, 8 to the right, and 8 down from the top left of the image. After that, you can leave the Cursor mode type as **Auto** unless it doesn’t show up for some reason, then you’ll want to switch it.

So now you have a bunch of **CursorDefinitons** created, and you want to know how to use them, right? Well, the easiest way to use them is by using the **CursorManager** Singleton.

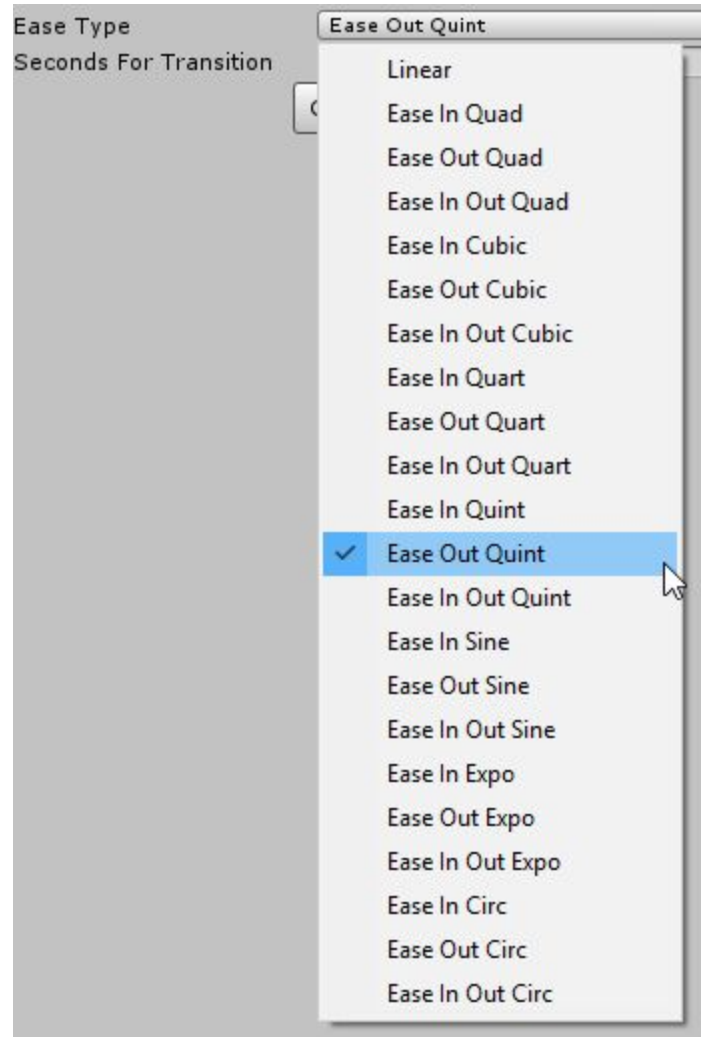
Now, before we get into it, I know there are many reasons why the Singleton (anti-)pattern is bad. The good news is, you don’t have to use it if you don’t want to, you are free to use **CursorDefinitions** without using **CursorManager**. You can read more about **CursorManager** later on in this documentation.

TransitionDefinition

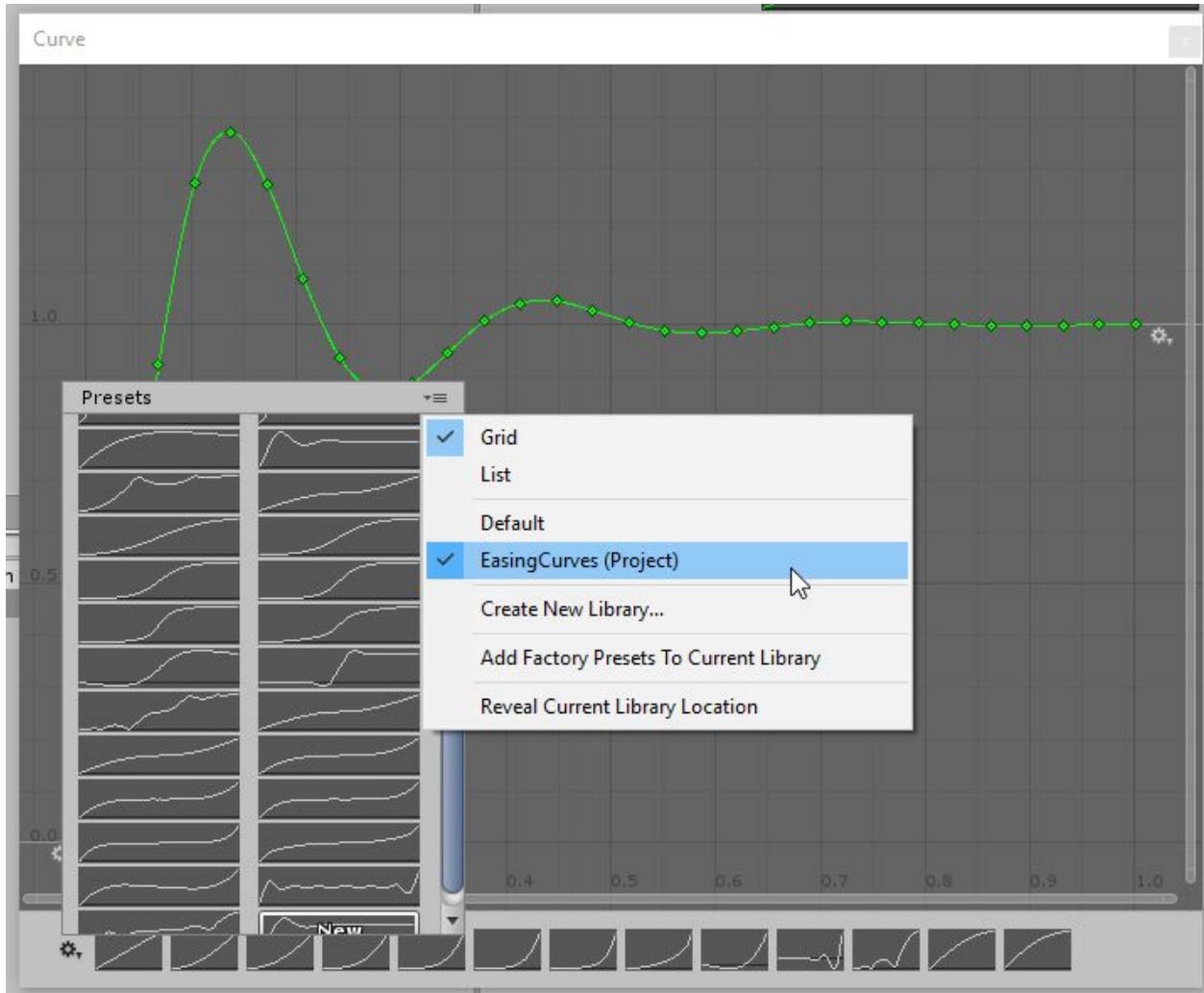
This is an asset used to create transitions that can be shared across various GameObjects. This way you only need to create a transition once, and you can reuse it as many times as you like. Here's an example of **EaseOutQuint** and **CustomElasticIn** which are included with Core:





The main difference between these two is that **EaseOutQuint** is using a mathematical equation to determine what the transition looks like, and **CustomElasticIn** is using an **AnimationCurve**. You can choose between these two types of transitions using the **My Transition Type** dropdown. Choose **Interpolate** if you want a large list of predefined **Ease Types** to quickly choose from.



If you want more control, you can choose a **Custom** in the **Transition Type** dropdown. This will allow you to define your own curves.



Core also comes preloaded with a bunch of custom curves for you to build off of and use. To access them, click the  icon in the bottom left of the curve editor panel, then the  icon in the top right of that menu. Then you want to select the **“EasingCurves (Project)”** option instead of “Default” The EasingCurves Project is licensed under the [2-clause BSD License](https://github.com/nobutaka/EasingCurvePresets), which can be found below.

Note: This license has also been called the "Simplified BSD License" and the "FreeBSD License". See also the 3-clause BSD License.

Copyright 2018 Dust Productions, LLC (originally from <https://github.com/nobutaka/EasingCurvePresets>)

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright

```
notice, this list of conditions and the following disclaimer in the
documentation and/or other materials provided with the distribution.
THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
```

ValueSetters

Setters are scripts used to interact with commonly modified variables. They provide the ability to change values over time, as well as lock and unlock objects (more on what that means later).

ValueSetter

A **ValueSetter** is a Component that's used to set a struct to a new value. A float and `UnityEngine.Color` are examples of structs. There are many more types of structs, but we won't go into much more depth here. The thing to keep in mind is that you can make a **ValueSetter** out of any struct, and we have already created a few for you. **FloatValueSetter** already exists, but here's how it is defined:

```
public abstract class FloatValueSetter : ValueSetter<float>
```

You would do similarly to define your own **ValueSetter**. Another useful feature is these can automatically be added to `GameObjects` depending on context. A dictionary is built every time the code is compiled by calling the static method **GetAffectedType()** on a **ValueSetter**. This dictionary can be accessed only from editor code by calling the static method **SetterHelpers.EDITORONLY_TryGetAutoAddSettersFromType()**. Chances are, you will never have to call this yourself.

FloatValueSetter

By itself **FloatValueSetter** isn't very useful, all it does is tell **ValueSetter** how to transition floats from one value to another. The power comes when we inherit from this and can start actually setting values.

AudioSourceVolumeSetter

This does the work of changing the volume of an AudioSource for you.

CanvasGroupAlphaSetter

This does the work of setting alpha values on CanvasGroups, can also set the blocksRaycasts value when the transparency changes. In case you are unfamiliar with CanvasGroups, I recommend [reading about them in the Unity documentation here](#).

Block Raycasts When Partially Transparent is a nice convenience. If you leave this as **false**, whenever the CanvasGroup is not at full opacity, you will be able to click through it. Setting this to **true** will block the raycasts, and you will not be able to click things that are directly behind the CanvasGroup. This setting does NOT know about other things interacting with the blocksRaycasts value, it only knows when it goes to set the value itself (in other words, it isn't constantly polling that value to see if it has changed, it only checks when **CanvasGroupAlphaSetter** does the work).

LightIntensitySetter

This controls how intense a light's brightness is. It's nice for special effects and things like that.

ColorValueSetter

By itself **ColorValueSetter** isn't very useful, all it does is tell **ValueSetter** how to transition UnityEngine.Color from one value to another. The power comes when we inherit from this and can start actually setting values.

GraphicColorSetter

Use this whenever trying to set a Graphic's color, it will allow you to set transitions and fancy things. It works basically the same as **CanvasGroupAlphaSetter**, but it's for color on a Graphic instead of alpha on a CanvasGroup.

LightColorSetter

Use this to change the Color of a Light easily.

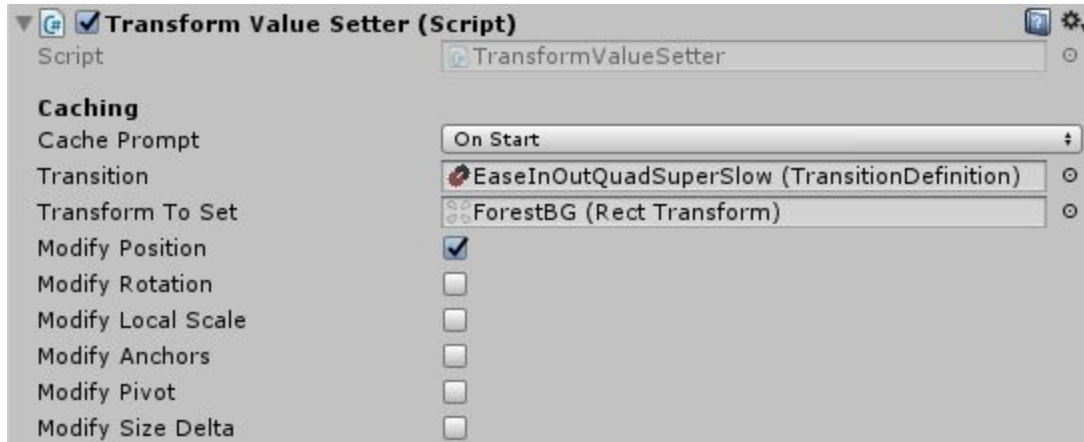
TransformValueSetter

Use this to modify a Transform's position, rotation, and / or scale. With a RectTransform, you can also modify the anchors, pivot, and sizeDelta values. This Component can also set values over time using **TransitionDefinitions**.

The main idea is that you can pass in a **TransformValues** struct, and the Transform linked in the inspector will also be set to those values. In Unity you can't make a Transform without

attaching it to a GameObject, so **TransformValues** solves that problem by saving off the values before we modify them.

Here's an example in the inspector that moves a background image.



As with other Setters, this also has the option to cache the values in case you want to reset them later. Another nice thing about this Component is that the inspector has options to only allow modification of certain parts of the Transform. It might be the case that you want a Transform to move to the same position as another Transform, but not necessarily the same rotation or scale as the other Transform. You can select only the things about the Transform you want to modify.

CanvasGroupInteractableSetter

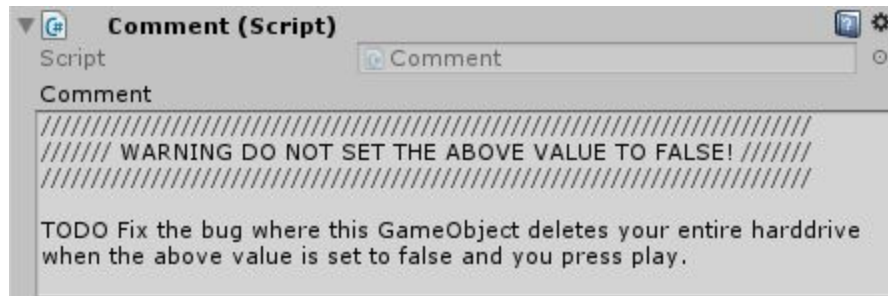
Unlike the others, this does not inherit from **ValueSetter**. This Component's job is to keep track of what objects have requested a CanvasGroup's **Interactable** value to be set to true or false. If any objects are requesting **Interactable** to be **false**, the CanvasGroup's status will be **false**. It's not until all objects have said the status should be **true** until the actual CanvasGroup's **Interactable** value will be **true**. You modify this by calling the **SetInteractable()** method. The **requestingObject** you pass in can be any C# object. The same object must be passed in to lock and unlock. In other words, Object1 can pass in **false**, and Object2 can pass in **true**, and the CanvasGroup will still be locked.

Utilities

The utility classes are mostly static libraries used to help with specific tasks. There are a few exceptions, however. We will first talk about the MonoBehaviours, then we will discuss the other classes.

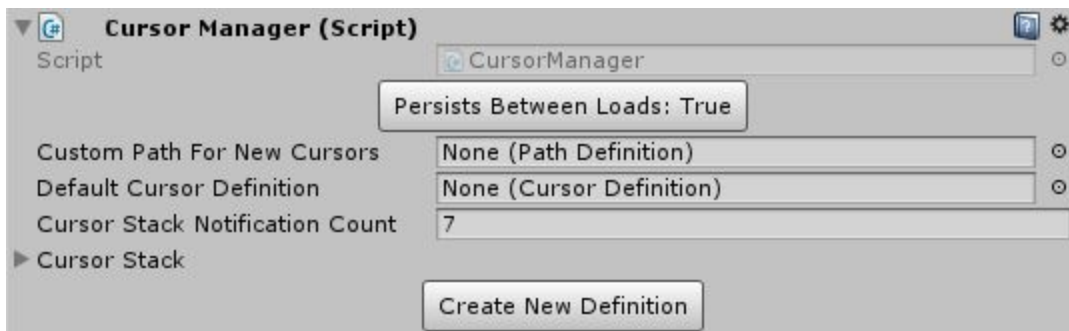
MonoBehaviours

Comment



This is used to leave comments on GameObjects (similar to how you'd leave comments in code). This information is stripped from the build.

CursorManager



CursorManager is a **Singleton** that allows you to change the mouse cursor from just about anywhere. The three main ways you will interact with **CursorManager** are calling **AddCurrentCursor()**, **RemoveCurrentCursor()**, and **SetCursorToDefault()**. Trigger Me Timbers has some nice Components to do this for you, but if you wanted to hook into it yourself, here's how you'd do it.

Cursors are kept as a stack. The cursor on the top of the stack is the cursor that will be displayed in Unity, so it would follow that in order to set the cursor, you would call **CursorManager.Instance.AddCurrentCursor()**.

```
public void AddCurrentCursor(CursorDefinition cursorDefinition, bool clearStackBeforeAdd = false)
```







You will be required to pass in a **CursorDefinition**, which is a ScriptableObject provided by Core we covered above. You can also optionally clear the current stack before pushing this to the top. (**Note:** By default, **CursorManager** will warn you if your stack gets to 7 or larger. This number can be changed, but if you are only pushing to the stack, and never popping from it, you are bound to have memory problems.)

```
public void RemoveCurrentCursor()
```

Because this is set up as a stack, once you want to switch the cursor back to what it was, all you have to do is call **CursorManager.Instance.RemoveCurrentCursor()**.

```
public void SetCursorToDefault(bool clearStackBeforeSetting = false)
```

If you don't want to go back to the previous cursor, and instead want to just go back to the default, you can use **CursorManager.Instance.SetCursorToDefault()** instead. By default, this does **NOT** clear the stack, you have to pass in **true** if you would like to do a full reset. Here's a little diagram for how you might practically use this:

Player behaviour	Code you call	Cursor stack contents
Player hasn't done anything		Active Cursor 
Player hovers over a draggable object in your game	<code>CursorManager.AddCurrentCursor(OpenHand)</code>	Active Cursor 
Player clicks to drag object	<code>CursorManager.AddCurrentCursor(ClosedHand)</code>	Active Cursor 
Player lets go of object	<code>CursorManager.RemoveCurrentCursor()</code>	Active Cursor 
Player stops hovering over draggable object	<code>CursorManager.RemoveCurrentCursor()</code>	Active Cursor 

DeveloperInformationMonoBehaviour

Holds information on how to contact us at DustProductions, LLC.

DontDestroyOnLoad

Makes it so an object persists between scene loads.

Singleton

Ensures only one version of the script can exist at a time. Can be useful for something like a manager class. Be careful though, with great power comes great responsibility. It's very easy to set up a situation where overdependence on these leads to bad organization and race conditions. Consider yourself warned. To turn a script into a **Singleton**, take your new class (in this example, it's called "NewManagerClass"):

```
using UnityEngine;

// Instead of this...
public class NewManagerClass : MonoBehaviour
{
    public void ManageAllTheThings()
    {
        // Code to manage things
    }
}
```

And change what it inherits from to **Singleton**, and pass in the Type of **Singleton** it should be.

```
using UnityEngine;
using DustProductions.Core;

// ...do this
public class NewManagerClass : Singleton<NewManagerClass>
{
    public void ManageAllTheThings()
    {
        // Code to manage things
    }
}
```

Then you can access the **Singleton** from any class like the following example:

```
NewManagerClass.Instance.ManageAllTheThings();
```

Libraries

Libraries contain many useful methods for reducing the overall code that needs to be written.

CoroutineEssentials

Contains an easy way to track coroutines, cancel them, and see if they have finished execution. Use by saving the Coroutine to a variable, then assigning the variable with **mySavedRoutine = this.StartCoroutine<>()**.

CurveInterpolator

Takes in AnimationCurves, and returns a value on that curve. Works a bit like Mathf.Interpolate().

DiskUtilities

Contains methods related to reading and writing on the disk. These have only been lightly tested on anything other than Windows, so your mileage may vary if you are using a different OS. I recommend checking out the online documentation for a full list of available functionality.

ExtensionMethods

Contains some useful methods for development, sort of a catch-all. I recommend checking out the online documentation for a full list of available functionality.

Interpolate

This is a library filled with methods to get mathematical interpolations between points. Can also handle curves and splines. I recommend checking out the online documentation for a full list of available functionality.

OpenInFileBrowser

Used to easily open files in the default file browser for Windows and macOS.

ReflectionUtilities

Helpful methods when dealing with C# Reflection, especially in Unity.

UIUtilities

Some handy things when dealing with Unity UI, RectTransforms, and Canvases.

Other Stuff

These don't fit into any other grouping, so they just exist here.

Cacheable

An abstract class that can be used to easily cache values and set to the cached values.

DeveloperInformation

A class that has developer information hardcoded. **DeveloperInformationMonoBehaviour** pulls from these values.

IEditorComponentAddedHandler

An interface that sets up the ability to do something when a Component is added in the editor. The basic premise is the object that's going to do something implements this interface, and the Component you are adding calls **this.EDITORONLY_AnnounceComponentAddedInEditor()**. Usually you'd call this in the Reset() method provided by Unity. Don't forget to wrap your call in `#if UNITY_EDITOR`, otherwise your project won't build.

Locker

Locker is an easy way to simplify if multiple objects want control of something. A good example that uses **Locker** is **CanvasGroupInteractableSetter**. Many objects might want to control the value of the CanvasGroup's Interactable value. The Locker is used in the backend to see what objects have requested what values for that. The main way to use this is with **Lock()**, **Unlock()**, and **IsLocked**.

MultiValueDictionary

An easy way to have a Dictionary that has one key for multiple values. Internally these multiple values are just a List of whatever Type you use to create the Dictionary.

ScriptableObjectFactory

Used by Core to create ScriptableObjects of any type. You can also hook into this with **ScriptableObjectFactory.EDITORONLY_TryCreateNewScriptableObjectAtPath<>()**. This will throw a warning if the asset you are trying to create already exists.

SerializableCursorDefinitionStack

See **SerializableStack**.

SerializableStack

A way to view a “Stack” in the Unity inspector. The unfortunate thing is the inspector does not play nice with generic types, so you have to create a class like **SerializableCursorStack** that inherits from this in order for it to work properly. Example:

```
[Serializable]  
public class SerializableCursorDefinitionStack :  
    SerializableStack<CursorDefinition>
```

Internally, the “Stack” is just a List, which means it’s not optimized the same way a regular Stack might be. Chances are, you’ll never notice the difference.

TransformValues

This is a data holding struct that’s used to cache off all the important values of a Transform. We do this because Transforms must be accompanied by a GameObject in Unity, and we don’t want to go through all of that upkeep just to hold some values. The data held by this is quite large for a struct, so it is recommended to always pass this around by reference using the “ref” keyword.