Trigger Me Timbers

Overview Guide

Last updated: 23 Feb 19

Online Documentation

docs.productionsofdust.com

Table of Contents

A word (or number of words) from the author

Ahoy!

I wanted to take a moment to thank you for your purchase. I started Dust Productions LLC with the intention of creating elegant interactive experiences (which is just marketing BS for wanting to make awesome software). I've been making games and software for over 10 years now, and I've learned a thing or two along the way. I hope to share that knowledge with you.

If you are coming in with little to no programming experience, I think you will find care was taken to make this powerful package as simple to use as possible. If you are coming in as a seasoned programming veteran of the industry with years of experience, I think you will also find care was taken to make extending the classes and Components simple if you need to create and customize more. There's a sea of possibilities out there, I truly hope this is the map to the treasure you've been looking for.

YAARRG!

Dust Productions

Frequently Asked Questions

Where do I go to get help?

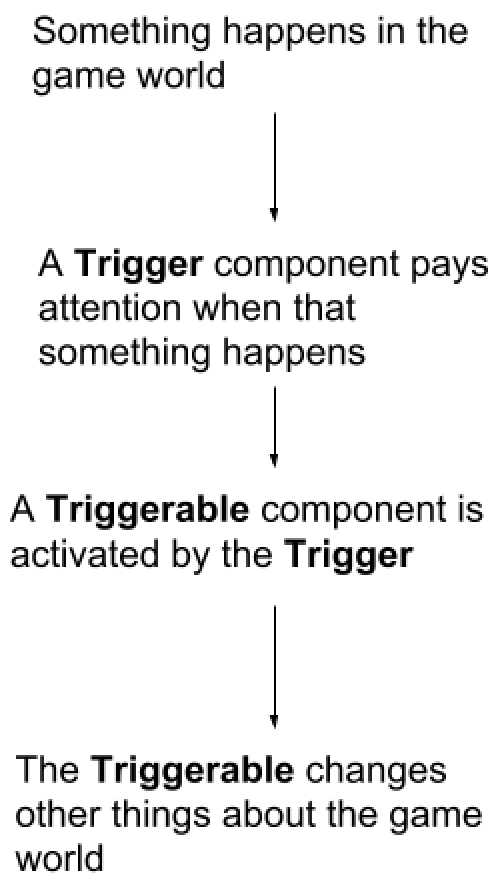
The first thing to do is check out this FAQ and guide or the online documentation for answers.

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.TriggerMeTimbers/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 Trigger Me Timbers, exactly?

Trigger Me Timbers seeks to accomplish one thing: abstract out the idea of what causes something to happen, and what happens as a result of the cause. In other words, Trigger Me Timbers breaks apart cause and effect and standardizes how they communicate with one another. Here's a nice graphic that summarizes the concept:


Something happens in the game world
A **Trigger** component pays attention when that something happens
A **Triggerable** component is activated by the **Trigger**
The **Triggerable** changes other things about the game world

What's even better is that you can have many **Triggers** activate a **Triggerable**, and many **Triggerables** be activated by a **Trigger**. The result of this is the ability to have something like a button press, do something like turn a UI GameObject on. Maybe that same button press also fades the transparency, plays a sound, and moves the GameObject slightly. The result of which is you have a nice polished UI panel that fades in and out of existence making a soft swishing sound as it comes and goes. Instead of creating one script called "TurnOnUiObject" that does those things, and isn't reusable, you have the

TriggerOnButtonClick, **TriggerableSetGameObjectsActive**, **TriggerableSetTransformValues**, **TriggerableAudioSourcePlayback**, and **TriggerableSetFloat** with a **CanvasGroupAlphaSetter**. These Components can be used as building blocks to get the behaviour we are looking for, and they can be used over and over again.

Of course, that was just one example of what you could create. There are plenty of unique Components that cause a behaviour to happen (i.e. a **Trigger**) and create the effect (i.e. a **Triggerable**). And if what this package provides isn't enough, you can create your own code that hooks into the system to leverage what the powerful Components have to offer. We'll go into more details later, but it's so easy, a picaroon could do it.

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 Trigger Me Timbers

Triggers

Triggers are the "Cause" part of the Cause and Effect relationship in Trigger Me Timbers. A **Trigger** can be anything from a button click, to a scene loading, to something custom in your game.

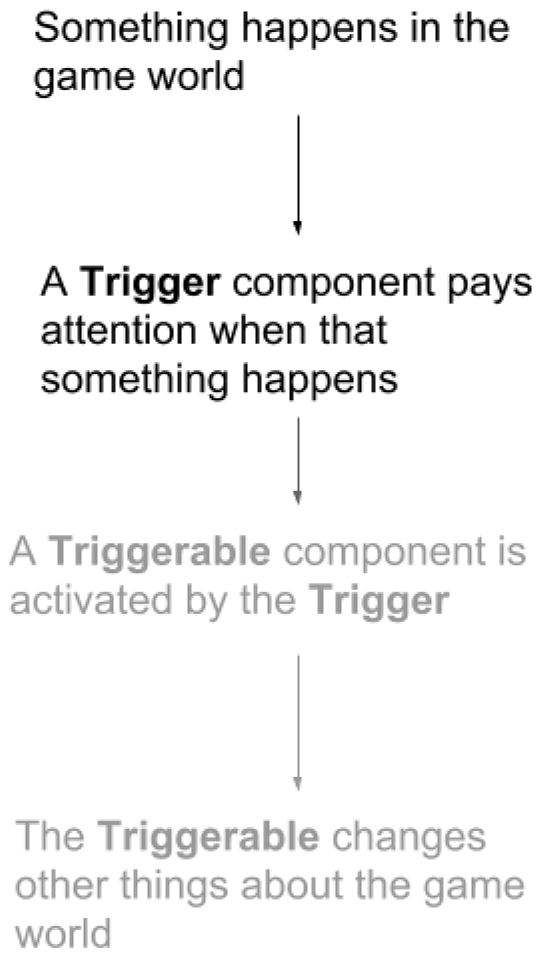
List of Triggers

For a full list of all the current Triggers, please visit the online documentation. Keep in mind that some Triggers might inherit from other Triggers.

Using Triggers In the Editor

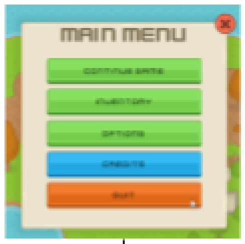
Summary

Remember this diagram?

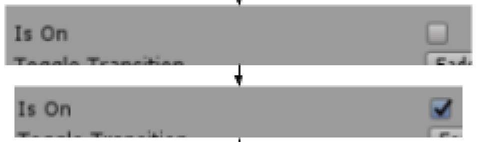

Something happens in the game world
A **Trigger** component pays attention when that something happens
A **Triggerable** component is activated by the **Trigger**
The **Triggerable** changes other things about the game world

We are concerned with the top half in this diagram, not so much the bottom half (we'll get to that later). Here is the high level overview for what this looks like with an example, the **TriggerOnToggled** Component:

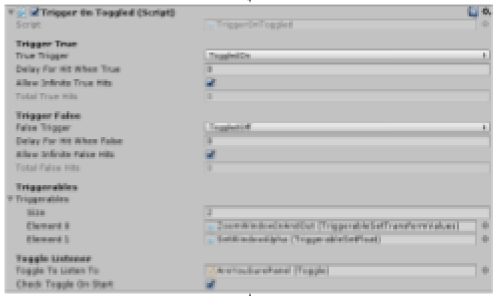
The user clicks the “Quit” Button in the Main Menu



This causes a **Toggle’s isOn** to change to true via some other code.



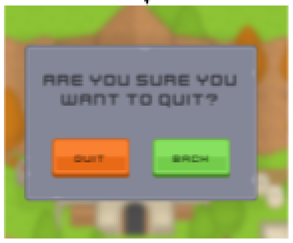
The **TriggerOnToggled** Component tells **Triggerables** the **Toggle** was **ToggledOn**.



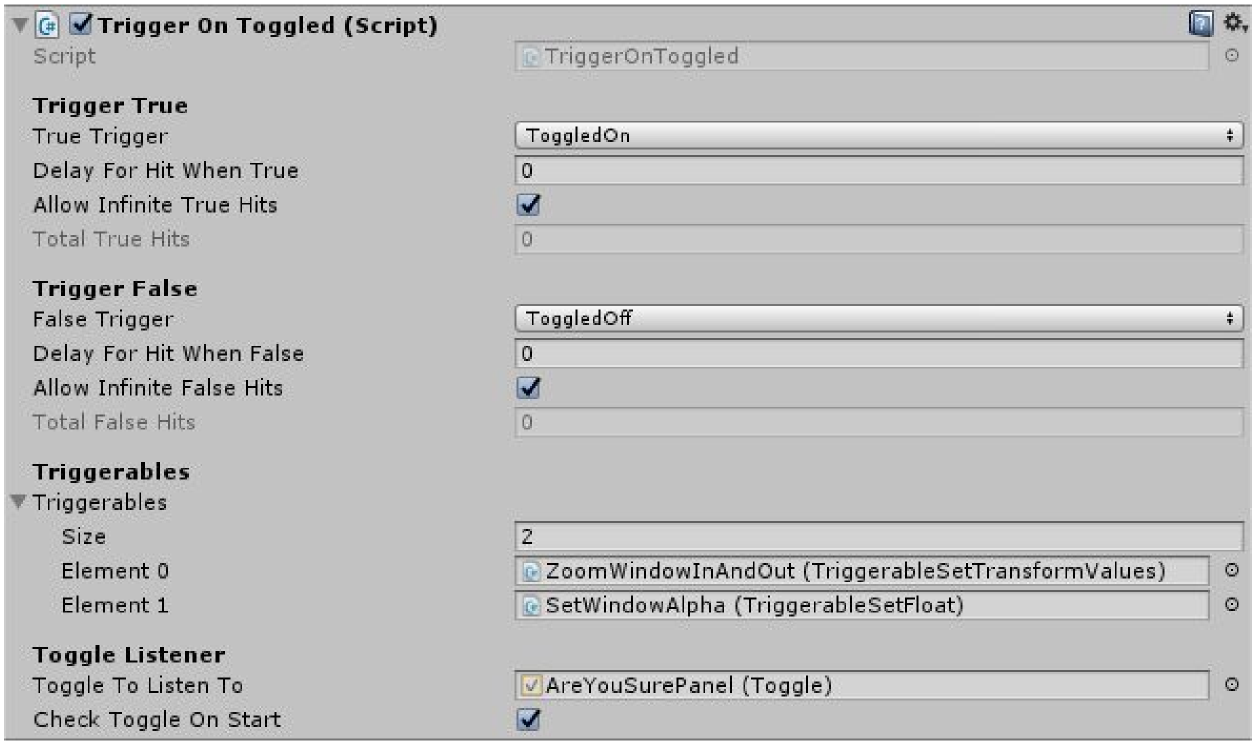
```
Trigger On Toggled (Script)
Script                          TriggerOnToggled
Trigger True
True Trigger                    ToggledOn
Delay For Hit When True         0
Allow Infinite True Hits        ✓
Total True Hits                 0
Trigger False
False Trigger                   ToggledOff
Delay For Hit When False        0
Allow Infinite False Hits       ✓
Total False Hits                0
Triggerables
▼ Triggerables
    Size                        2
    Element 0                   ZoomWindowInAndOut (TriggerableSetTransformValues)
    Element 1                   SetWindowAlpha (TriggerableSetFloat)
Toggle Listener
Toggle To Listen To             AreYouSurePanel (Toggle)
Check Toggle On Start           ✓
```

TriggerableSetFloat with **CanvasGroupAlphaSetter** fades the UI panel in.

TriggerableSetTransformValues with **TransformValueSetter** moves the UI panel into position



You can check out the **Triggerables** section for discussion on how those work, but for now we will focus on how the **TriggerOnToggled** Component is set up. Here’s a larger view of that Component:



Trigger On Toggled (Script)

Script	TriggerOnToggled
Trigger True	
True Trigger	ToggledOn
Delay For Hit When True	0
Allow Infinite True Hits	✓
Total True Hits	0
Trigger False	
False Trigger	ToggledOff
Delay For Hit When False	0
Allow Infinite False Hits	✓
Total False Hits	0
Triggerables	
Triggerables	
Size	2
Element 0	ZoomWindowInAndOut (TriggerableSetTransformValues)
Element 1	SetWindowAlpha (TriggerableSetFloat)
Toggle Listener	
Toggle To Listen To	AreYouSurePanel (Toggle)
Check Toggle On Start	✓

Going from top to bottom, you will see the sections titled **Trigger True**, **Trigger False**, **Triggerables**, and **Toggle Listener.** We will go into detail on these, starting from the bottom section.

Bottom Section

The bottom section is where you will find configuration specific to that **Trigger**, if any is needed**.** In this case the **Toggle Listener** section has a reference to the **Toggle** we want to pay attention to, as well as the option to check it when the game starts.

Triggerables

The next section up is where you can link any **Triggerables** for this **Trigger** to activate. We have the two **Triggerables** mentioned previously. Again, we will cover those in a later part of this document. If you add this to a GameObject that already has **Triggerables** on it, they will be automatically added here, otherwise you can drag and drop from the hierarchy to add them.

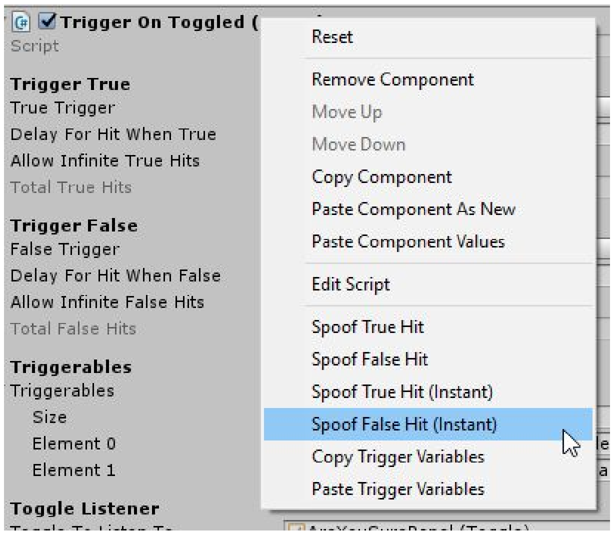
Trigger True / Trigger False

Above that we have the sections for **Trigger True** and **Trigger False**. These are the most important sections, and determine when to tell the **Triggerables** to do things. The field named **True Trigger** and **False Trigger** have dropdowns that list all the things the **Trigger** Component can listen for. In this case the options are **ToggledOn**, **ToggledOff**, and **OnAnyToggle** (not pictured). Each **Trigger** Component will have its own set of options in this dropdown. There are also options to delay hitting the **Triggerables** until some time has passed, and whether or not there should be a maximum number of hits allowed.

All of the **Triggers** are set up set up the same way, so figuring out how to use it once, means you will know how to use it for everything else. If you prefer to see more clear example, you should check out the **TriggerableSetTransformValues** scene. If you'd like to see some more practical examples, you should check out the **MainMenu** or **PhoneScreen** scene

Extra Features

Debugging / Testing

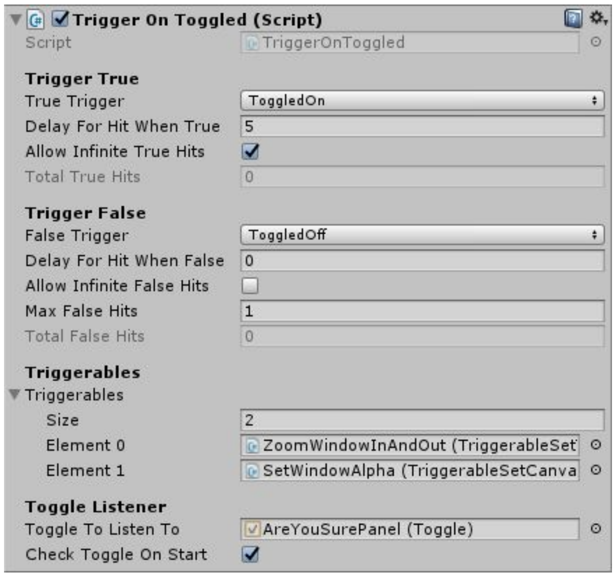


There's a few more things about **Triggers** I should mention. There are some convenience features that you can find in the context menu (accessed by right clicking on the title of the Component).

You can simulate hitting a trigger by choosing **Spoof True Hit**, **Spoof False Hit, Spoof True Hit (Instant),** or **Spoof False Hit (Instant)**. The "**(Instant)**" versions will bypass any restrictions or delays, while the ones without will respect things like the max hits and delays. These options are useful for debugging so you don't have to get to the part in the game where you are trying to hit the **Trigger**. It can also be useful as a debug when you aren't seeing the behaviour you expect so you can narrow down whether it's the thing hitting the **Trigger**, the **Triggerables** themselves, or something outside of that system causing your unexpected behaviour.

Copy / Paste

The next really handy feature is the ability to **Copy Trigger Variables** and **Paste Trigger Variables**. This is similar to **Copy Component** and **Paste Component Values**, except it can be used across different **Trigger** types. Here's an example:

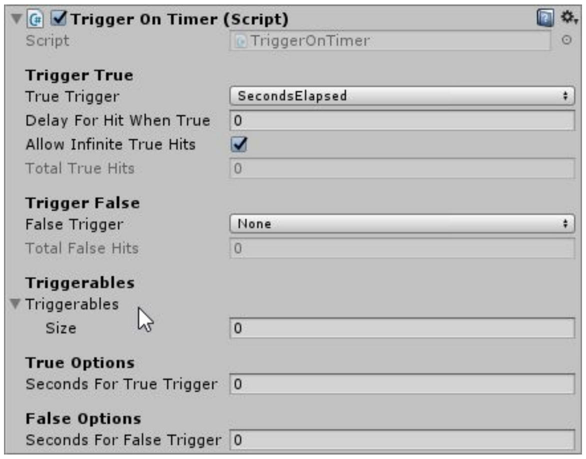

Trigger On Toggled (Script)
Script: TriggerOnToggled

Trigger True
True Trigger: ToggledOn
Delay For Hit When True: 5
Allow Infinite True Hits: ✓
Total True Hits: 0

Trigger False
False Trigger: ToggledOff
Delay For Hit When False: 0
Allow Infinite False Hits: ☐
Max False Hits: 1
Total False Hits: 0

Triggerables
Triggerables
Size: 2
Element 0: ZoomWindowInAndOut (TriggerableSet
Element 1: SetWindowAlpha (TriggerableSetCanva

Toggle Listener
Toggle To Listen To: AreYouSurePanel (Toggle)
Check Toggle On Start: ✓


Trigger On Timer (Script)
Script: TriggerOnTimer

Trigger True
True Trigger: SecondsElapsed
Delay For Hit When True: 0
Allow Infinite True Hits: ✓
Total True Hits: 0

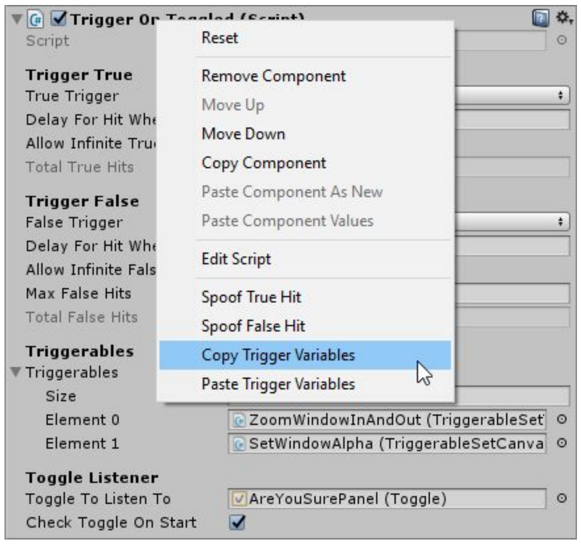
Trigger False
False Trigger: None
Total False Hits: 0

Triggerables
Triggerables
Size: 0

True Options
Seconds For True Trigger: 0

False Options
Seconds For False Trigger: 0

Let's say we have an existing **TriggerOnToggled,** and a Component we just added called **TriggerOnTimer**. **TriggerOnToggled** might have a bunch of things we want to also use, like the **Triggerables**, the delays, and the maximum number of hits for different things. **TriggerOnToggled** also has some things we don't want on our new Component.



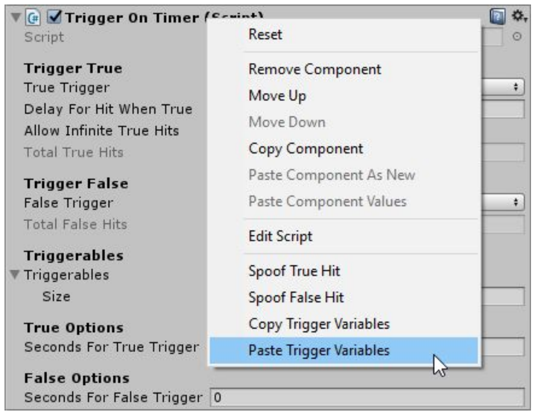
Trigger On Toggled (Script)
Script

Trigger True
True Trigger
Delay For Hit Whe
Allow Infinite Tru
Total True Hits

Trigger False
False Trigger
Delay For Hit Whe
Allow Infinite Fals
Max False Hits
Total False Hits

Triggerables
Triggerables
Size
Element 0: ZoomWindowInAndOut (TriggerableSet
Element 1: SetWindowAlpha (TriggerableSetCanva

Toggle Listener
Toggle To Listen To: AreYouSurePanel (Toggle)
Check Toggle On Start: ✓



Trigger On Timer (Script)
Script

Trigger True
True Trigger
Delay For Hit When True
Allow Infinite True Hits
Total True Hits

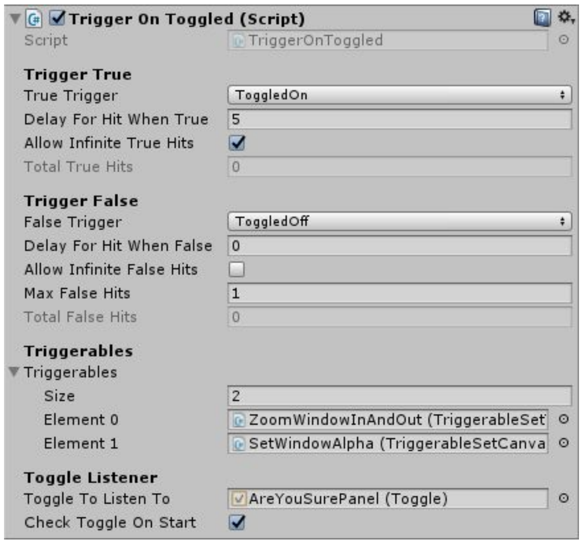
Trigger False
False Trigger
Total False Hits

Triggerables
Triggerables
Size

True Options
Seconds For True Trigger

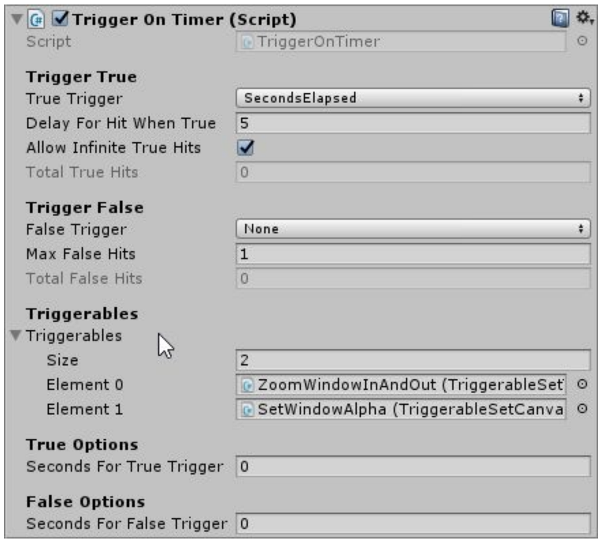
False Options
Seconds For False Trigger: 0

In this instance, we can do our copy and paste, and it is smart enough to not bring over things that are unique to the **TriggerOnToggle** Component. So what we end up with looks something like this:



Trigger On Toggled (Script)

Script	TriggerOnToggled
Trigger True	
True Trigger	ToggledOn
Delay For Hit When True	5
Allow Infinite True Hits	✓
Total True Hits	0
Trigger False	
False Trigger	ToggledOff
Delay For Hit When False	0
Allow Infinite False Hits	
Max False Hits	1
Total False Hits	0
Triggerables	
Triggerables	
Size	2
Element 0	ZoomWindowInAndOut (TriggerableSet
Element 1	SetWindowAlpha (TriggerableSetCanva
Toggle Listener	
Toggle To Listen To	AreYouSurePanel (Toggle)
Check Toggle On Start	✓



Trigger On Timer (Script)

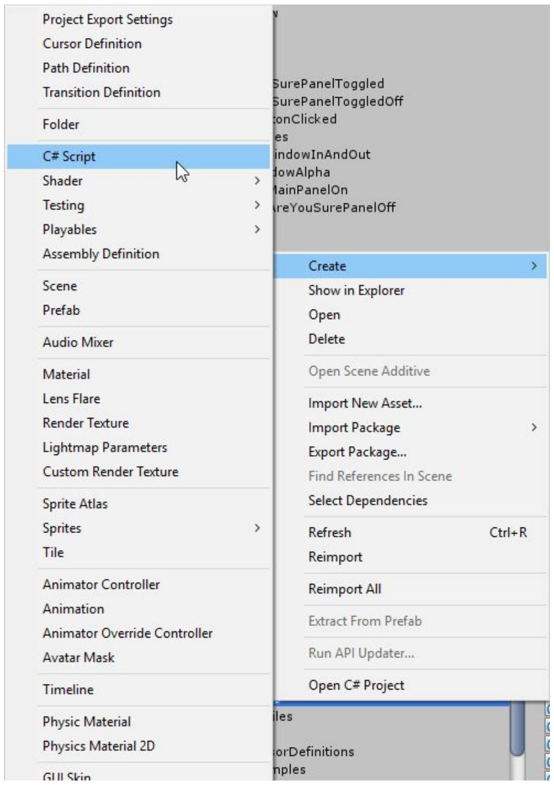
Script	TriggerOnTimer
Trigger True	
True Trigger	SecondsElapsed
Delay For Hit When True	5
Allow Infinite True Hits	✓
Total True Hits	0
Trigger False	
False Trigger	None
Max False Hits	1
Total False Hits	0
Triggerables	
Triggerables	
Size	2
Element 0	ZoomWindowInAndOut (TriggerableSet
Element 1	SetWindowAlpha (TriggerableSetCanva
True Options	
Seconds For True Trigger	0
False Options	
Seconds For False Trigger	0

Notice the **True Trigger** and **False Trigger** haven't changed, nor has anything in the bottom section, because those are unique to the Trigger, but everything else copied over.

If you are curious about making your own **Triggers**, you can go to the next section, otherwise it might be a good idea to go on to the **Triggerables** section in this document.

Using Triggers In Code

So you want to make your own **Triggers**, eh? No problem, here's how you do it. First thing you have to do is make sure **DustProductions.TriggerMeTimbers** exists in your Unity project. Then you make a new script however you prefer, I like to right-click on the folder and do this:



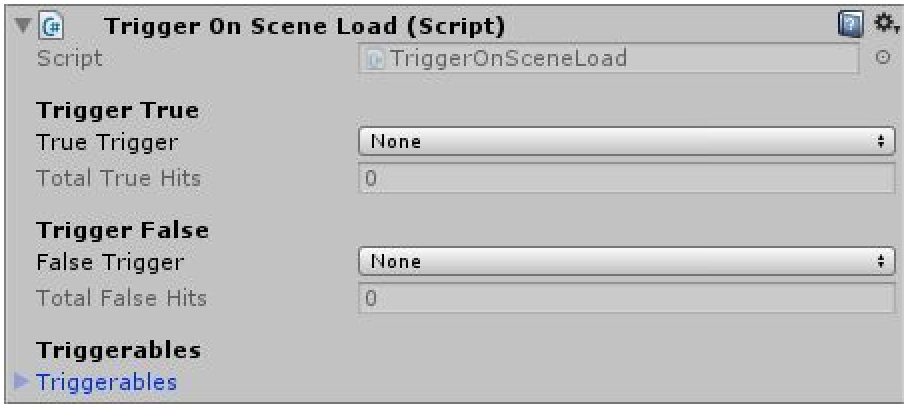
Then after you've named your file (I'm naming it **TriggerOnSceneLoad**), open it up in your IDE. You should be greeted with something like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerOnSceneLoad : MonoBehaviour
{

    // Use this for initialization
    void Start ()
    {

    }

    // Update is called once per frame
    void Update ()
    {

    }
}
```

Now before we start to write our code, we have to do a few things. The first is to make sure we add “using **DustProductions.TriggerMeTimbers**” to the top of the file. Then we can inherit from **Trigger** instead of **MonoBehaviour**. I also like to delete the things Unity added for me and start with a blank slate. If you’ve made it this far, your compiler should hopefully start yelling at you that you haven’t implemented the abstract class. That’s the next step. Here’s what it looks like now.

```
using DustProductions.TriggerMeTimbers;
using UnityEngine;

public class TriggerOnSceneLoad : Trigger
{
    protected override string[] GetTriggerDropdownOptions()
    {
        throw new System.NotImplementedException();
    }
}
```

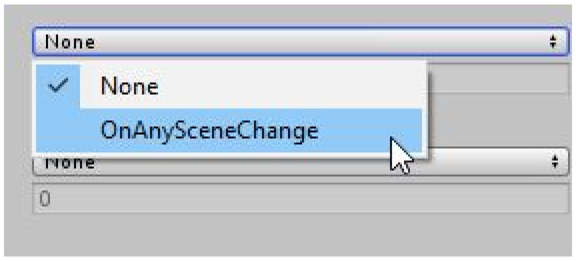
If we were to try and add the Component now, this is what we would see:


Trigger On Scene Load (Script)
Script TriggerOnSceneLoad
Trigger True
True Trigger None
Total True Hits 0
Trigger False
False Trigger None
Total False Hits 0
Triggerables
Triggerables

We would also see that **NotImplementedException** in the Unity console. Let’s fix that up. **GetTriggerDropdownOptions()** is called whenever the Unity inspector needs to show the **True Trigger** and **False Trigger** dropdowns. So what we want to do is make sure we set those so our **Trigger** actually has functionality. Without doing so, the dropdowns will always say **None**. For this example, we will do the following:

```
using DustProductions.TriggerMeTimbers;
using UnityEngine;

public class TriggerOnSceneLoad : Trigger
{
    // Required by Trigger
    protected override string[] GetTriggerDropdownOptions()
    {
       return new string[] { "OnAnySceneChange" };
    }

}
```

This tells the dropdown options to also contain the string “**OnAnySceneChange**”. Now our dropdown should look like this if we check it out in the editor:



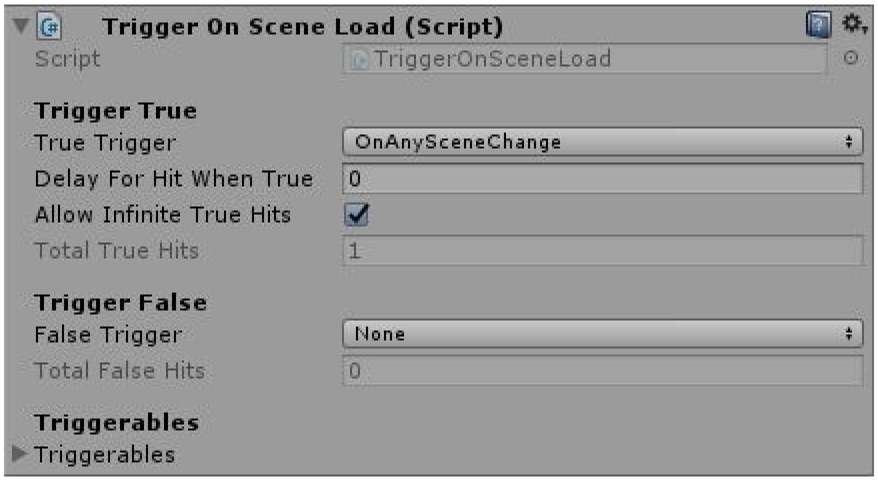
Nice! Now we have to make it so our **Trigger** can pay attention to scene changes.

```
using DustProductions.TriggerMeTimbers;
using UnityEngine;
using UnityEngine.SceneManagement;

public class TriggerOnSceneLoad : Trigger
{
    // Required by Trigger
    protected override string[] GetTriggerDropdownOptions()
    {
       return new string[] { "OnAnySceneChange" };
    }

    // Used to fire callback for implementation
    private void Awake()
    {
       SceneManager.activeSceneChanged += OnSceneChanged;
    }
    private void OnDestroy()
    {
       SceneManager.activeSceneChanged -= OnSceneChanged;
    }

    // This will be called whenever the scene changes in play mode
    private void OnSceneChanged(Scene current, Scene next)
    {
       // Tell Trigger to fire here
    }
}
```

Great, now we just need to tell the **Trigger** to fire when **OnSceneChanged()** is called. This is simple enough.

```
    // This will be called whenever the scene changes in play mode
    private void OnSceneChanged(Scene current, Scene next)
    {
       CheckTriggerString("OnAnySceneChange");
    }
```

This is a method in the **Trigger** class that checks if any of your dropdowns match that string and fires a hit when they match. Now make sure one of your dropdowns is set to the **OnAnySceneChange** option, press the play button, and notice **Total True Hits** was incremented!



Trigger On Scene Load (Script)

Script	TriggerOnSceneLoad
Trigger True	
True Trigger	OnAnySceneChange
Delay For Hit When True	0
Allow Infinite True Hits	✓
Total True Hits	1
Trigger False	
False Trigger	None
Total False Hits	0
Triggerables	
Triggerables	

You might also notice that the console is yelling at you because there are no **Triggerables** assigned to this script. That is to be expected. Just drag and drop any old **Triggerable** if you haven't already, and you should be good to go.

There might be a time where you might not want to automatically **CheckTriggerString()**, and need to do some additional checks before hitting the trigger. In this case, instead of using **CheckTriggerString()**, you would do your checks manually, and call **HitTrigger(bool triggerValue)** Here's a quick example of what that looks like:

```
// This will be called whenever the scene changes in play mode
private void OnSceneChanged(Scene current, Scene next)
{
   if (current.name.Equals("MainMenu") &&
       TrueTrigger == "OnCurrentIsMainMenu")
   {
         HitTrigger(true);
   }
   if (current.name.Equals("MainMenu") &&
       FalseTrigger == "OnCurrentIsMainMenu")
   {
         HitTrigger(false);
   }
}
```

Note that **HitTrigger()** doesn't automatically mean the linked **Triggerables** will be hit. It will still respect any delays or maximum number of hits set in the inspector. But that's it, you have now made it so some code you wrote can hook into everything Trigger Me Timbers has to offer.

! It's probably not a good idea to leave all those strings laying around your code, turn them into consts so there's no way they can be mistyped! And null check your variables! We run a tight ship around here!

Triggerables

A **Triggerable** Component is the "Effect" part of the Cause and Effect relationship in Trigger Me Timbers. A **Triggerable** changes the game world, and can do things like activating GameObjects, change the mouse cursor, moving Transforms, and anything else you might want it to do.

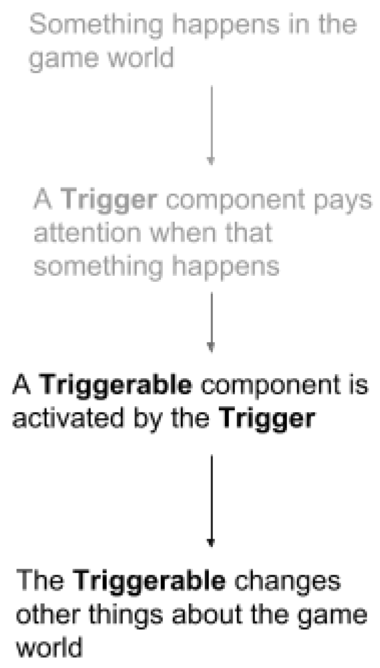
List of Triggerables

For a full list of all the current Triggerables, please visit the online documentation. Keep in mind that some Triggerables might inherit from other Triggerables.

Using Triggerables in the Editor

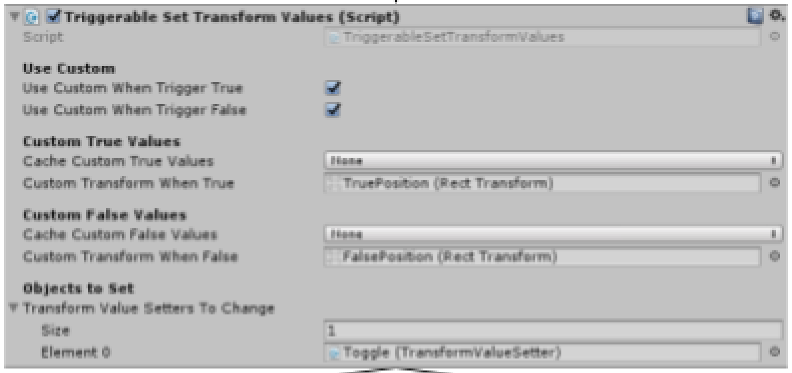
Summary

Remember this diagram?


Something happens in the game world
A **Trigger** component pays attention when that something happens
A **Triggerable** component is activated by the **Trigger**
The **Triggerable** changes other things about the game world

We are now concerned with only the bottom half of the diagram. Here's how we move the Toggle around in the included **TriggerableSetTransformValues** scene. You can follow along in the Unity editor, if you like. Here's a summary of what's happening:

Clicking the Toggle tells the **TriggerOnToggled** Component to send either **true** or **false** to **TriggerableSetTransformValues.**


Triggerable Set Transform Values (Script)
Script: TriggerableSetTransformValues

Use Custom
Use Custom When Trigger True: ✓
Use Custom When Trigger False: ✓

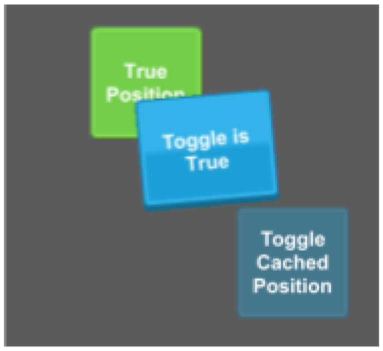
Custom True Values
Cache Custom True Values: None
Custom Transform When True: TruePosition (Rect Transform)

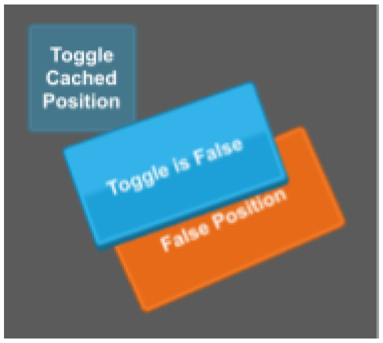
Custom False Values
Cache Custom False Values: None
Custom Transform When False: FalsePosition (Rect Transform)

Objects to Set
Transform Value Setters To Change
Size: 1
Element 0: Toggle (TransformValueSetter)

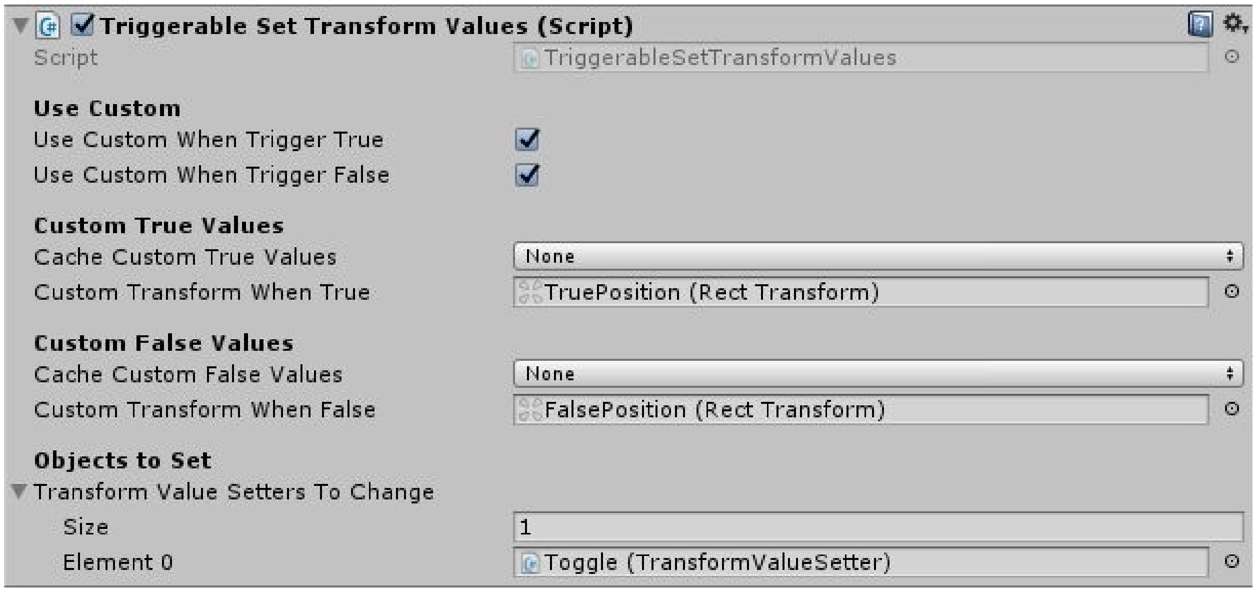
When **true** is hit, the Toggle moves to **Custom Transform When True's** RectTransform values.

When **false** is hit, the Toggle moves to **Custom Transform When False's** RectTransform values.





So let's break down how the **TriggerableSetTransformValues** Component is setup, and what each of the options do. Here's a bigger image of the Component.



Triggerable Set Transform Values (Script)

Script	TriggerableSetTransformValues
Use Custom	
Use Custom When Trigger True	✓
Use Custom When Trigger False	✓
Custom True Values	
Cache Custom True Values	None
Custom Transform When True	TruePosition (Rect Transform)
Custom False Values	
Cache Custom False Values	None
Custom Transform When False	FalsePosition (Rect Transform)
Objects to Set	
Transform Value Setters To Change	
Size	1
Element 0	Toggle (TransformValueSetter)

From top to bottom, we have the sections **Use Custom, Custom True Values, Custom False Values,** and **Objects to Set.** Now, let's go into more detail on these, starting from the bottom.

Objects to Set

These are the objects in the game world that we want to manipulate with the **Triggerable**. This one is manipulating **TransformValueSetter** Component, which is a nice utility Component to move Transforms over time. We are able to assign more than one **TransformValueSetter** by increasing the size of the array if we wanted to set multiple at the same time.

Custom True Values / Custom False Values

Here's where most of the configuration is going to be. The custom values will vary depending on the particular **Triggerable** you are using. This Component has custom values for Transforms. These Transform values are what we will set the Toggle's Transform values to. On many **Triggerables**, you will find the option to **Cache Custom True / False Values**. We are not doing any caching on this object, so that means whatever the Transform values are when the **Triggerable** gets hit, that's where we will move the Toggle. There are also options to cache **OnAwake**, **OnStart**, and **BeforeFirstModification**. The first two options mean that whenever Awake() or Start() are called on the Component, the Transform values will be saved off. This means you could move around the **TruePosition** and **FalsePosition** Transforms while the game is running, and the Toggle would still go to wherever they were when Awake() or Start() were called. **BeforeFirstModification** will cache the values before we ever modify anything with this Component. Similarly, other Components are able to cache colors, alpha values, Toggle isOn values, etc. Basically whatever the **Triggerable** is modifying, it will be saved off before that Component touches it.

Use Custom

	True in Inspector	**False in Inspector**
Use Custom When Trigger True	True passed in: Custom Value Used	True passed in: Cached Value Used
	False passed in: N/A	False passed in: N/A
Use Custom When Trigger False	True passed in: N/A	True passed in: N/A
	False passed in: Custom Value Used	False passed in: Cached Value Used

The **Use Custom** section has two options. The chart above summarizes what happens when different options are set, and different values are passed in, but if that isn't clear, here's a description of what you are looking at. **Use Custom When Trigger True** means that when this **Triggerable** gets the **true** value passed in, it will use the information defined in **Custom True Values** to move the Toggle. In the pictured Component above, it would move to **TruePosition's** RectTransform's Values. If **Use Custom When Trigger True** was set to **false**, it could move to the Toggle's original position. This is further reinforced by the fact that the custom values disappear from the inspector when these are set to **false**. This concept is important, so we are going to use another example.

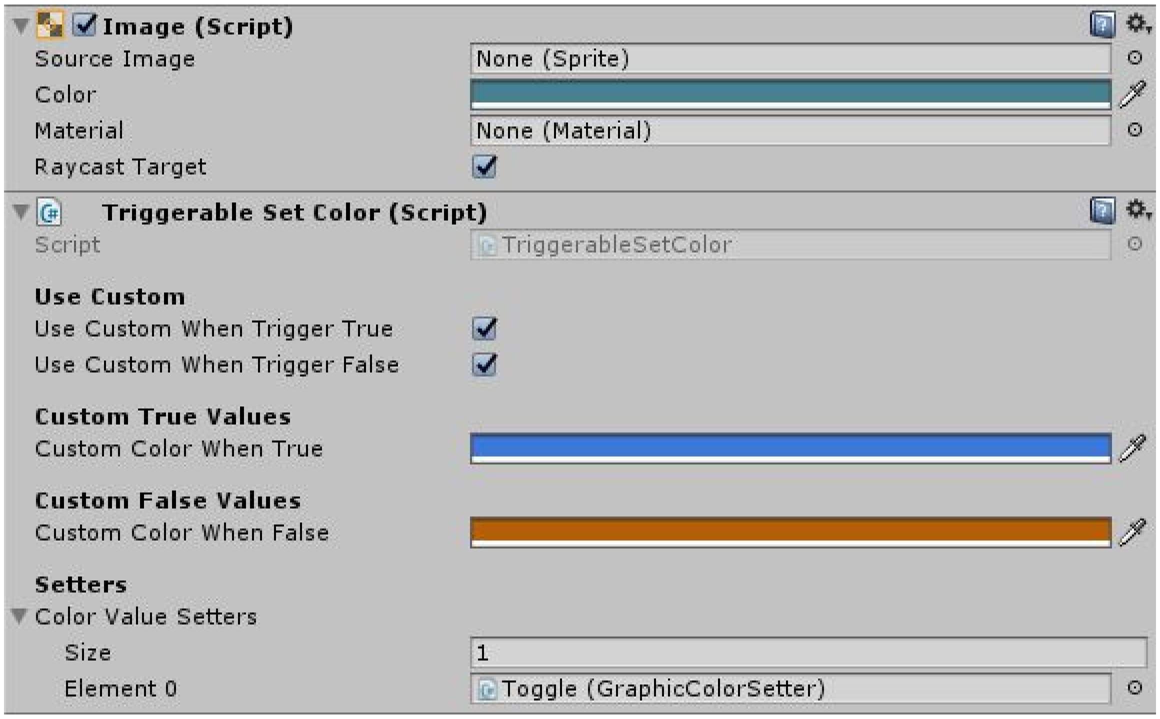


Image (Script)
Source Image: None (Sprite)
Color
Material: None (Material)
Raycast Target: ✓

Triggerable Set Color (Script)
Script: TriggerableSetColor

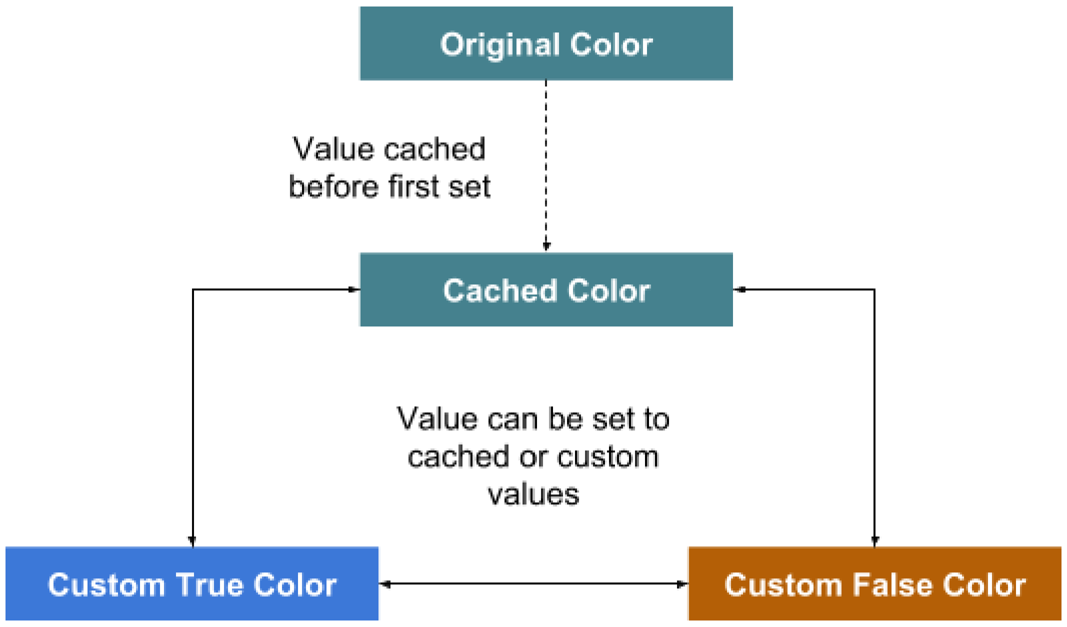
Use Custom
Use Custom When Trigger True: ✓
Use Custom When Trigger False: ✓

Custom True Values
Custom Color When True

Custom False Values
Custom Color When False

Setters
Color Value Setters
Size: 1
Element 0: Toggle (GraphicColorSetter)

Let's imagine a Toggle with Components **TriggerOnToggled,** and **TriggerableSetColor** with a **GraphicColorSetter.** Let's imagine our Toggle starts off as **teal**, and we have set the **Custom Color When True** to be **blue** and the **Custom Color When False** to be **orange**. Here is a flowchart of how the colors would be set.


Original Color
Value cached before first set
Cached Color
Value can be set to cached or custom values
Custom True Color
Custom False Color

And here's the chart of what options would get what colors.

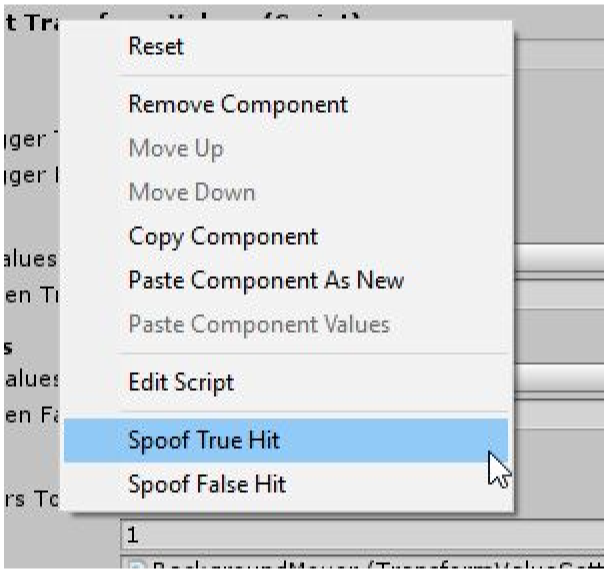
	True in Inspector	**False in Inspector**
Use Custom When Trigger True	True passed in: Custom Value Used	True passed in: Cached Value Used
	False passed in: N/A	False passed in: N/A
Use Custom When Trigger False	True passed in: N/A	True passed in: N/A
	False passed in: Custom Value Used	False passed in: Cached Value Used

! If you run into an error that says you are trying to use the cached value before the value has been cached, then you should try to set the value to be cached sooner. So if it's **BeforeFirstModification**, try setting it to **OnStart**. If it's **OnStart**, then try setting it to **OnAwake**. If that doesn't work, then you might have to delay whatever **Trigger** is attempting to set the value. You can check the **Trigger** in debug mode (covered below).

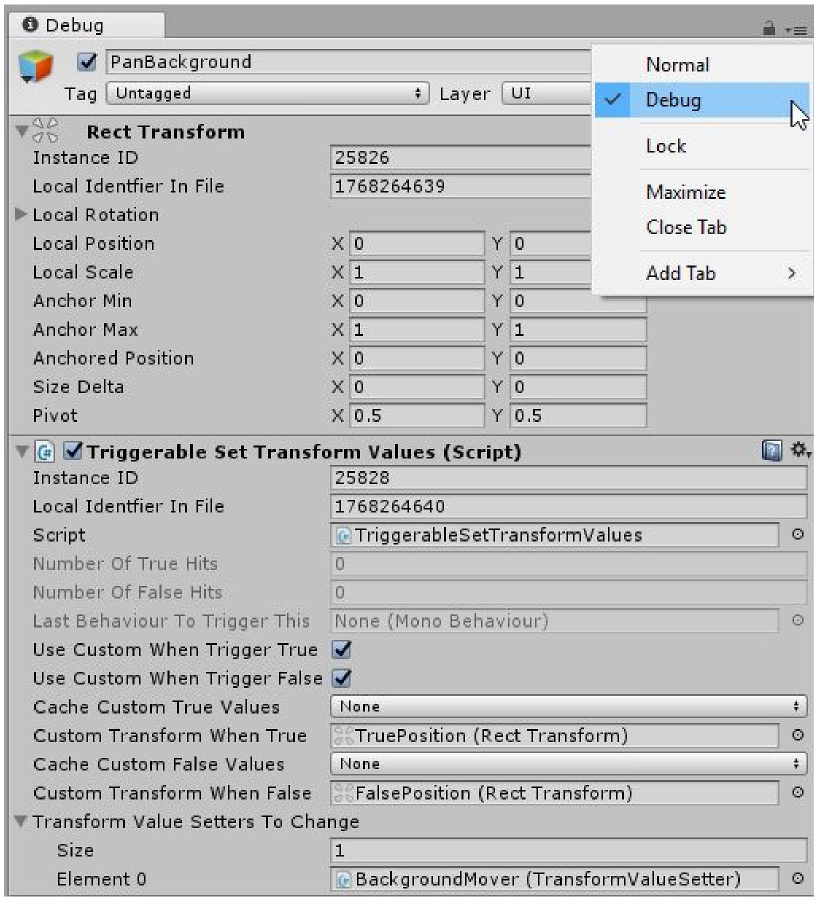
Extra Features

Debugging / Testing

Like the **Trigger** Components, **Triggerable** Components also have a convenient context menu for easily testing and debugging behaviour.



You only have to right click on the title of the Component, and you can spoof true and false hits, without needing to trigger the Component in game. Some other nice debug features are available in the inspector's built in debug mode. In case you have forgotten how to enable that, it's by clicking on the hamburger dropdown (☰) icon in the top right corner of your inspector and click **Debug**.



PanBackground
Tag Untagged Layer UI

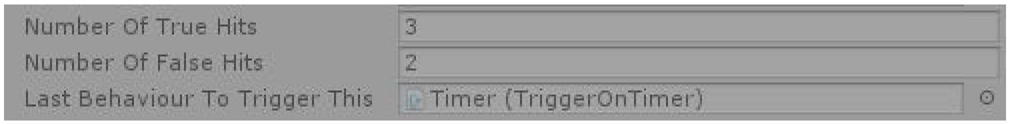
Rect Transform

Instance ID	25826	
Local Identfier In File	1768264639	
Local Rotation		
Local Position	X 0	Y 0
Local Scale	X 1	Y 1
Anchor Min	X 0	Y 0
Anchor Max	X 1	Y 1
Anchored Position	X 0	Y 0
Size Delta	X 0	Y 0
Pivot	X 0.5	Y 0.5

Triggerable Set Transform Values (Script)

Instance ID	25828
Local Identfier In File	1768264640
Script	TriggerableSetTransformValues
Number Of True Hits	0
Number Of False Hits	0
Last Behaviour To Trigger This	None (Mono Behaviour)
Use Custom When Trigger True	✓
Use Custom When Trigger False	✓
Cache Custom True Values	None
Custom Transform When True	TruePosition (Rect Transform)
Cache Custom False Values	None
Custom Transform When False	FalsePosition (Rect Transform)
Transform Value Setters To Change	
Size	1
Element 0	BackgroundMover (TransformValueSetter)

Doing so will give you a dropdown like this, and change how your inspector looks. You can see here, we have our same **TriggerableSetTransformValues** Component, but with a few fields that you couldn't see before. The most interesting ones are the **Number Of True Hits, Number Of False Hits,** and **Last Behaviour To Trigger This**. If we were to be in the game and need to know what was going on, we could reference it in debug mode.



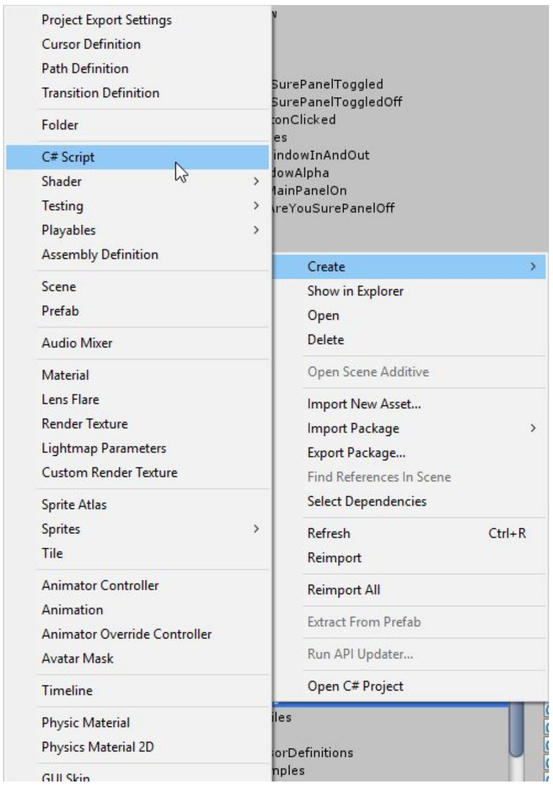
Number Of True Hits	3
Number Of False Hits	2
Last Behaviour To Trigger This	Timer (TriggerOnTimer)

This is the same Component after the game has been running for a bit. We can clearly see the number of times the object has been triggered, as well as the reference to the **TriggerOnTimer** Component. If you like, you can also click on this reference to be brought to the object in the hierarchy.

I think that covers the basics of how **Triggerables** work. Next up, we will discuss how to make your own.

Using Triggerables in Code

Trigger Me Timbers can handle many things, but it may be the case that you need to manipulate something in your game world that isn't built in to Unity. Here's how you'd make your own **Triggerable**. Easy enough, let's start from the beginning. First thing you have to do is make sure **DustProductions.TriggerMeTimbers** exists in your Unity project. Then you make a new script however you prefer, I like to right-click on the folder and do this:



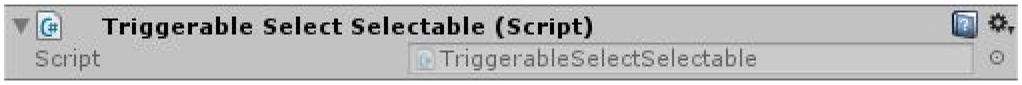
Then after you've named your file (I'm naming it **TriggerableSelectSelectable**), open it up in your IDE. You should be greeted with something like this:

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerableSelectSelectable : MonoBehaviour
{

    // Use this for initialization
    void Start ()
    {

    }

    // Update is called once per frame
    void Update ()
    {

    }
}
```

Now before we start to write our code, we have to do a few things. The first is to make sure we add “using **DustProductions.TriggerMeTimbers**” to the top of the file. Then we can inherit from **Triggerable** instead of **MonoBehaviour**. I also like to delete the things Unity added for me and start with a blank slate. If you’ve made it this far, your compiler should hopefully start yelling at you that you haven’t implemented the abstract class. That’s the next step. Here’s what it looks like now.

```
using DustProductions.TriggerMeTimbers;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class TriggerableSelectSelectable : Triggerable
{
    protected override void TriggerHit(bool triggerValue)
    {
       throw new System.NotImplementedException();
    }
}
```

If we were to try to add the Component now, this is what we would see:


Triggerable Select Selectable (Script)
Script TriggerableSelectSelectable

Not very impressive, so let’s make it actually do stuff. The first thing I need for my new Component is a reference to the Selectable I want to select. In order to use the Selectable class, I will also need to use UnityEngine.UI. Here’s what we have now:

```
using DustProductions.TriggerMeTimbers;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class TriggerableSelectSelectable : Triggerable
{
    [SerializeField]
    private Selectable SelectableToSelect;

    protected override void TriggerHit(bool triggerValue)
    {
       throw new System.NotImplementedException();
    }
}
```

Now we just need to select our Selectable whenever a **Trigger** calls **TiggerHit()** on our object.

```
using DustProductions.TriggerMeTimbers;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class TriggerableSelectSelectable : Triggerable
{
    [SerializeField]
    private Selectable SelectableToSelect;

    protected override void TriggerHit(bool triggerValue)
    {
       if (SelectableToSelect == null)
       {
             Debug.LogError("Can't select Selectable because it is null on:
" + name, this);
             return;
       }
       SelectableToSelect.Select();
    }
}
```

I also added a null check just in case we forgot to assign the **SelectableToSelect** in the inspector. You'll also notice that I'm not using the **triggerValue** for anything. So the behaviour we'd see from this Component at this point is that any time a **Trigger** called **TriggerHit()**, the Selectable would be selected. We might want to change that so it is only selected when we pass in **true.** All that takes is a quick check, and we are good to go. Here's what our final code looks like:

```
using DustProductions.Core;
using DustProductions.TriggerMeTimbers;
using UnityEngine;
using UnityEngine.UI;

public class TriggerableSelectSelectable : Triggerable
{
    [Header("Object to Set")]
    [Tooltip("Select this Selectable when true is passed in")]
    [SerializeField]
    private Selectable SelectableToSelect;

    private void Reset()
    {
       this.TryGetComponent(ref SelectableToSelect);
    }

    protected override void TriggerHit(bool triggerValue)
    {
       if (SelectableToSelect == null)
       {
             Debug.LogError("Can't select Selectable because it is null on:
" + name, this);
             return;
       }
       if (triggerValue)
       {
             SelectableToSelect.Select();
       }
    }
}
```

I also added some nice to have features like the header, tooltip text, and automatically grabbing the Selectable if it exists when the Component is added using the nice utility method provided by **DustProductions.Core** called **TryGetComponent()**.

Now you are ready to test your Component by starting the game, finding a Selectable, and using the nice context menu mentioned up in the **Extra Features** section to make sure it works.

I think that's all you need to get started with making your own **Triggerables**. Of course this one was pretty simple, yours might be more complicated, but the key things to remember are that you have to inherit from **Triggerable**, keep things generic and simple, and once you have created a **Triggerable**, you can trigger it using any of the **Triggers** in the game, allowing you ultimate flexibility in how to use it.

! Technically, you just need to implement the **ITriggerable** interface, meaning you can make just about anything into a **Triggerable**, you don't even need to inherit from the **Triggerable** class!

Best Practices

Use a consistent style when building objects in the editor

I like to define one entry point one exit point for a GameObject or group of GameObjects. I find myself using Toggles as entry and exit points just because they can be interactable for UI elements. They are also nice to use with ToggleGroups, which can only allow one Toggle to be active at a time for the group. Regardless of what you choose to do, try to do it everywhere, so you never have to dig around and figure out what is triggering what.

If you are planning on having a variable set over time for a Triggerable, use a ValueSetter instead

Trigger Me Timbers comes with **Triggerables** for setting floats, Colors, and the values of Transforms. So if you need to change any of these types of objects, you don't need to create a **Triggerable**, what you want instead is to create a **ValueSetter**. Let's say you wanted to set the speed of one of your characters, and that speed is a float value. You might create a Component called **CharacterSpeedValueSetter** that would inherit from **ValueSetter<float>**. Then you can drop your **CharacterSpeedValueSetter** onto a **TriggerableSetFloat**, and be hooked into everything Trigger Me Timbers has to offer. No need to write your own code for transitioning the values, no need to write your own logic for handling the **Trigger**, just need to write the code that says how to get and set the value you want to change.