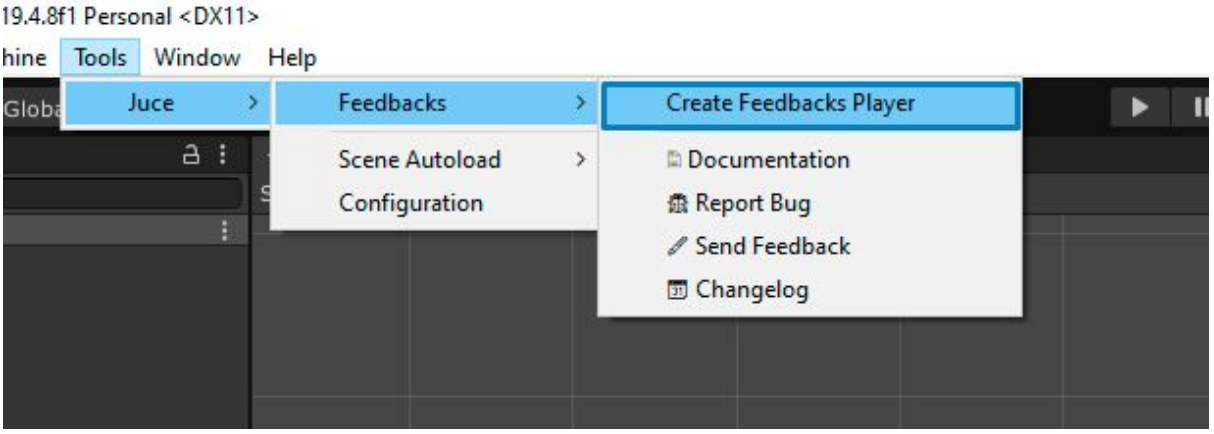
Juce-Feedbacks

If you want more detailed documentation, please visit:
https://github.com/Juce-Assets/Juce-Feedbacks/wiki/Usage-guide

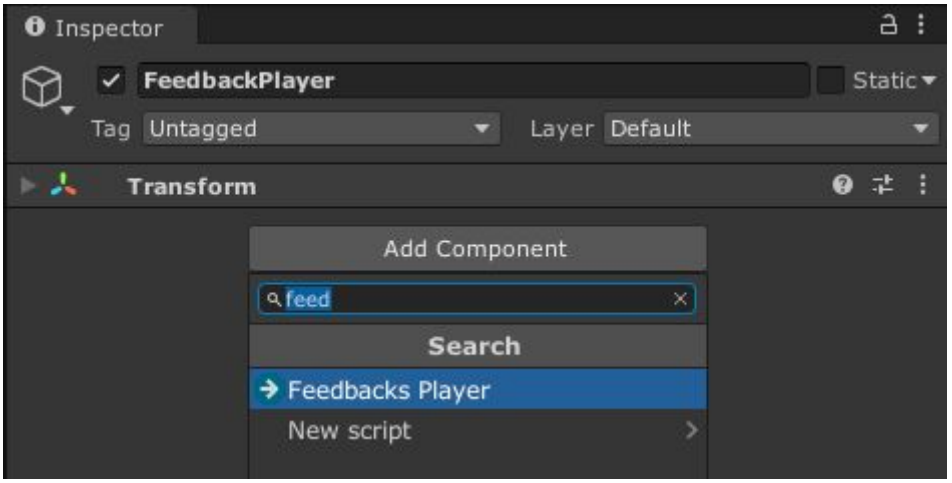
1. Getting started:

Setting up your feedbacks is extremely easy, fast, and fun. The inspectors have been designed to be simple yet efficient, and provide you with all the info you need at runtime. Simply add feedbacks, tweak them to your liking, test them in real-time, and enjoy all that good game feel!

To begin with, go to "Tools/Juce/Feedbacks/Create Feedback Player" to create your first FeedbacksPlayer!



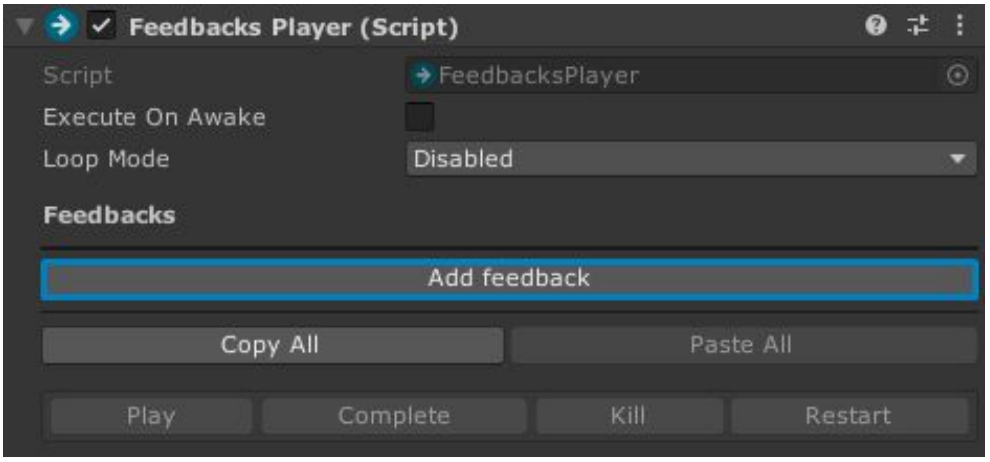
Or you can also create a new GameObject yourself, and add the FeedbacksPlayer component to it:

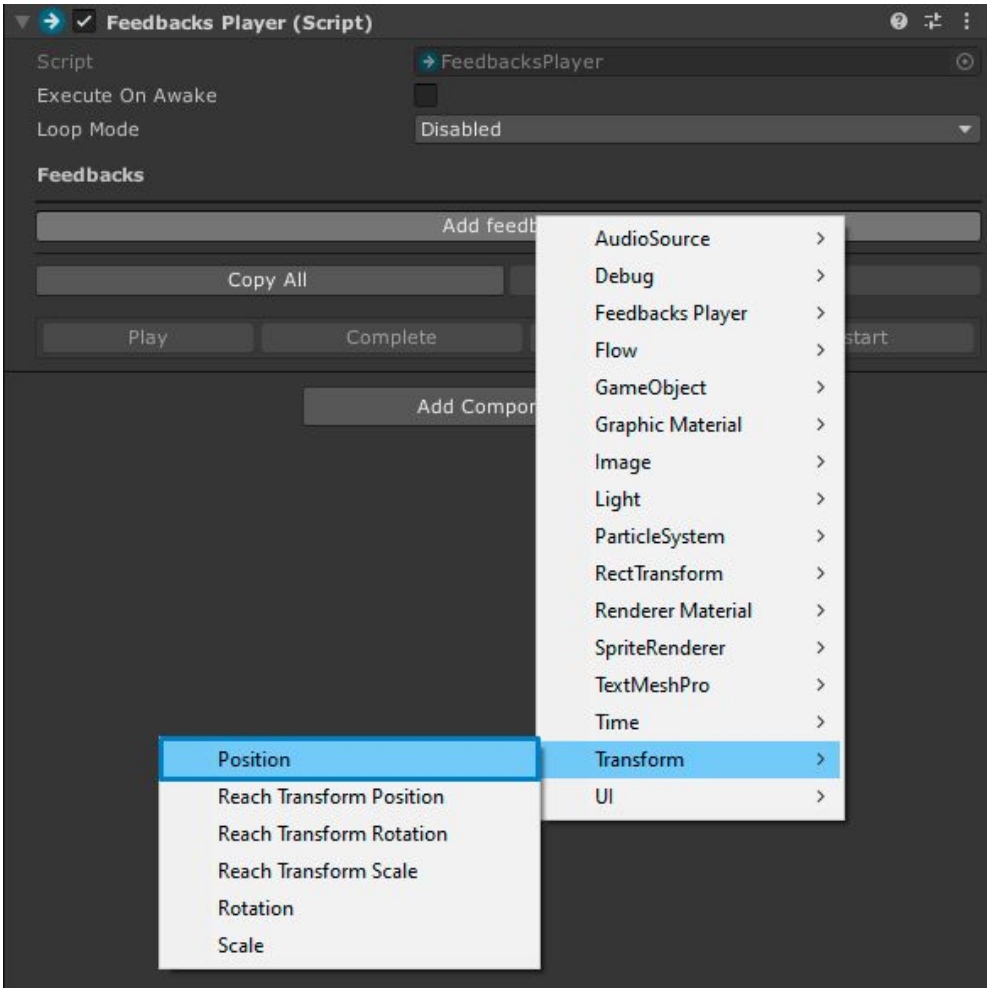


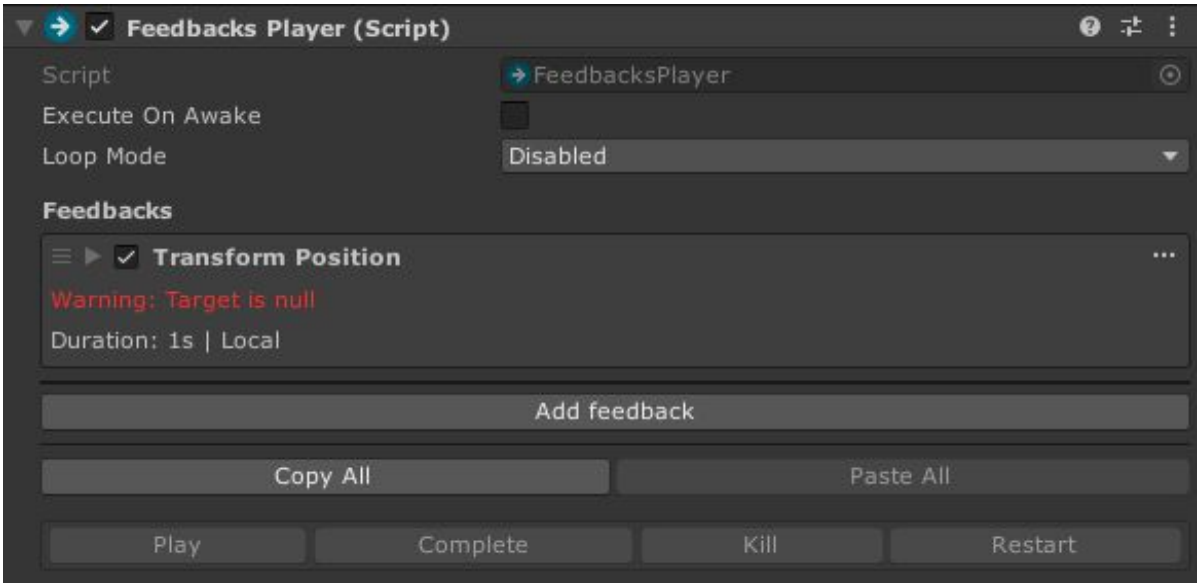
2. Feedbacks player:

Creating feedbacks:

Creating new feedbacks is very easy and intuitive. First, go to the FeedbacksPlayer component, press the "Add Feedback" button, and select the feedback that you want to add.







In the example, we have created a Transform Position feedback, which is used to move things around!

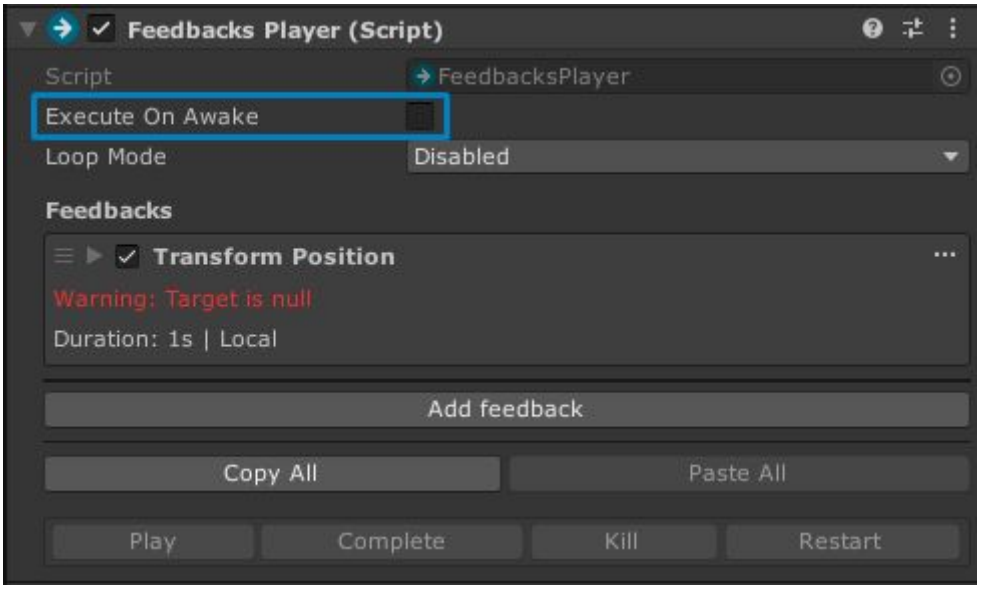
Feedback sections:

Mostly, all the feedbacks follow the same structure, which you can see here. Some of them may have more or less elements, depending on the needs of each feedback.

3. Controlling feedbacks

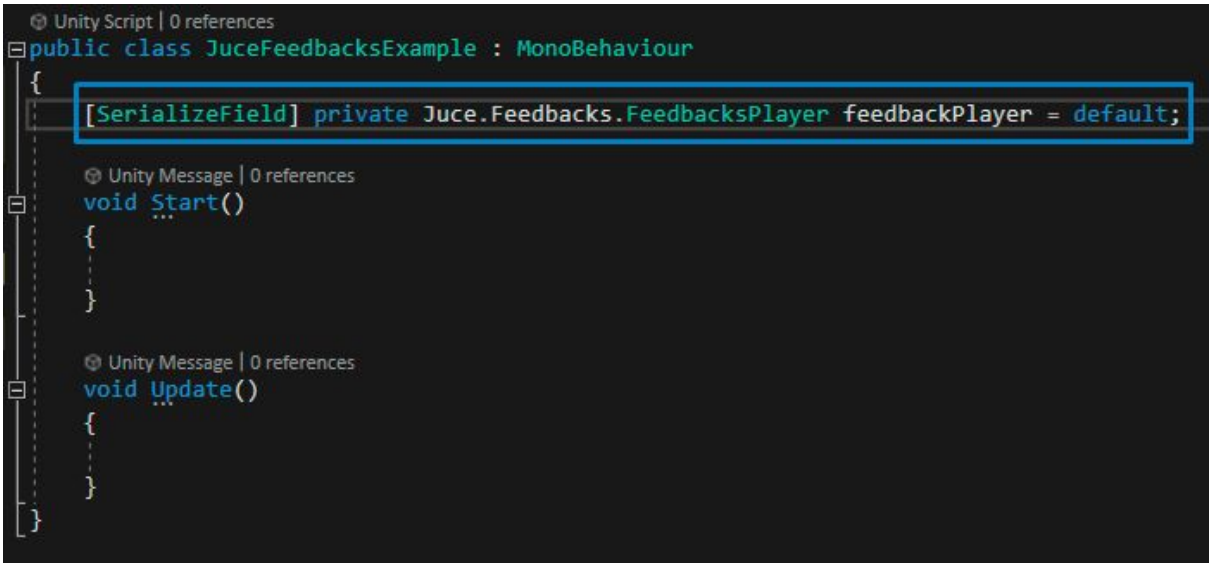
Playing:

You have two principal ways of playing a FeedbacksPlayer. The first of them is setting the component to Execute On Awake. If the setting is toggled, it will play at the moment the component is spawned.



The other way (and the most common) is doing it through script.

You just need to get a reference of the FeedbacksPlayer, and then call Play();



```
Unity Script | 0 references
public class JuceFeedbacksExample : MonoBehaviour
{
    [SerializeField] private Juce.Feedbacks.FeedbacksPlayer feedbackPlayer = default;

    Unity Message | 0 references
    void Start()
    {
        feedbackPlayer.Play();
    }

    Unity Message | 0 references
    void Update()
    {

    }
}
```

```
51 references
public override void GetFeedbackInfo(ref List<string> infoList)
{
    InfoUtils.GetTimingInfo(ref infoList, delay, duration);
    InfoUtils.GetStartEndVector3PropertyInfo(ref infoList, value);
    InfoUtils.GetCoordinatesSpaceInfo(ref infoList, coordinatesSpace);
}
```

Knowing when finished

To know if a FeedbacksPlayer has finished playing, you have two ways:

- Using a System.Action, that you pass at the Play method

```
Unity Script | 0 references
public class JuceFeedbacksExample : MonoBehaviour
{
    [SerializeField] private Juce.Feedbacks.FeedbacksPlayer feedbackPlayer = default;

    Unity Message | 0 references
    void Start()
    {
        feedbackPlayer.Play(() =>
        {
            // This feedback has finished playing
        });
    }

    Unity Message | 0 references
    void Update()
    {

    }
}
```

- Or using Task, so you can await.

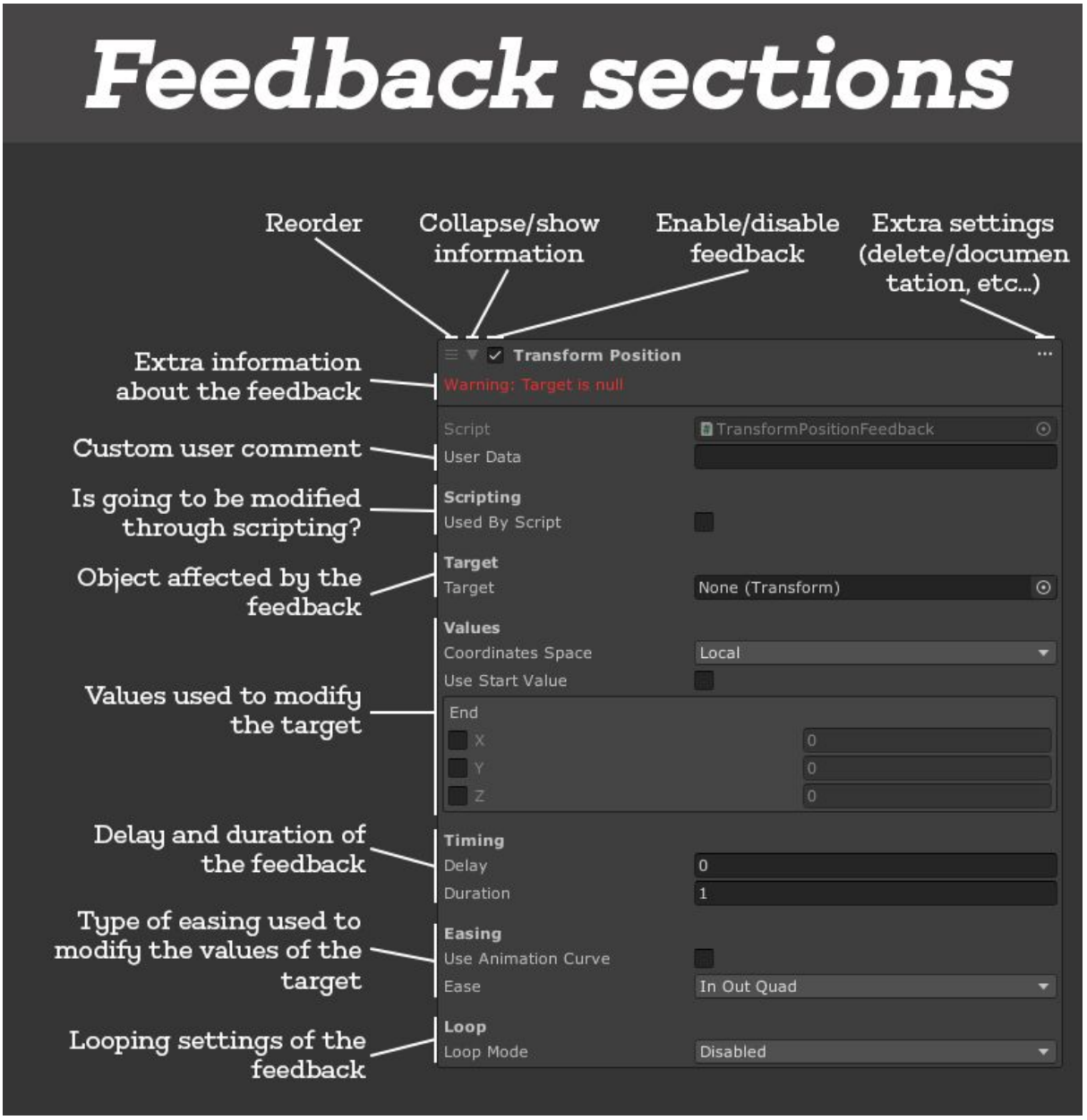
```
1 reference
private async Task ThisIsAFunction()
{
    // Do some things

    await feedbackPlayer.Play();

    // Do some other things
}
```

Extra control:

Once it's playing, you can:

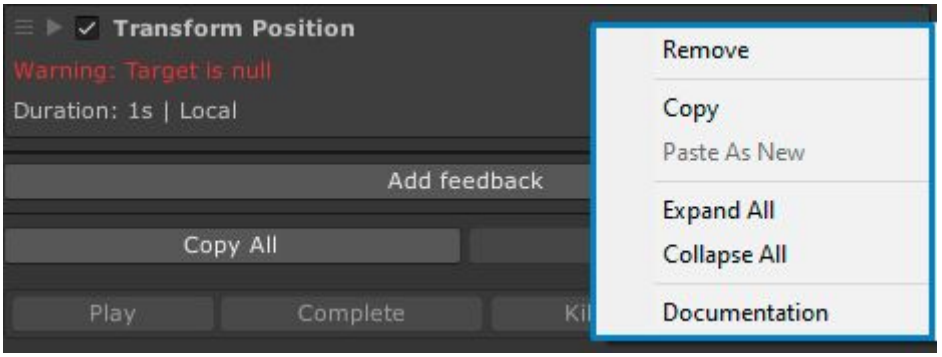
- Kill the feedback, so it instantly stops playing.
- Complete the feedback, so it instantly reaches its final state.

```
Unity Script | 0 references
public class JuceFeedbacksExample : MonoBehaviour
{
    [SerializeField] private Juce.Feedbacks.FeedbacksPlayer feedbackPlayer = default;

    Unity Message | 0 references
    void Start()
    {
        feedbackPlayer.Play();

        feedbackPlayer.Kill();

        feedbackPlayer.Complete();
    }

    Unity Message | 0 references
    void Update()
    {

    }
}
```

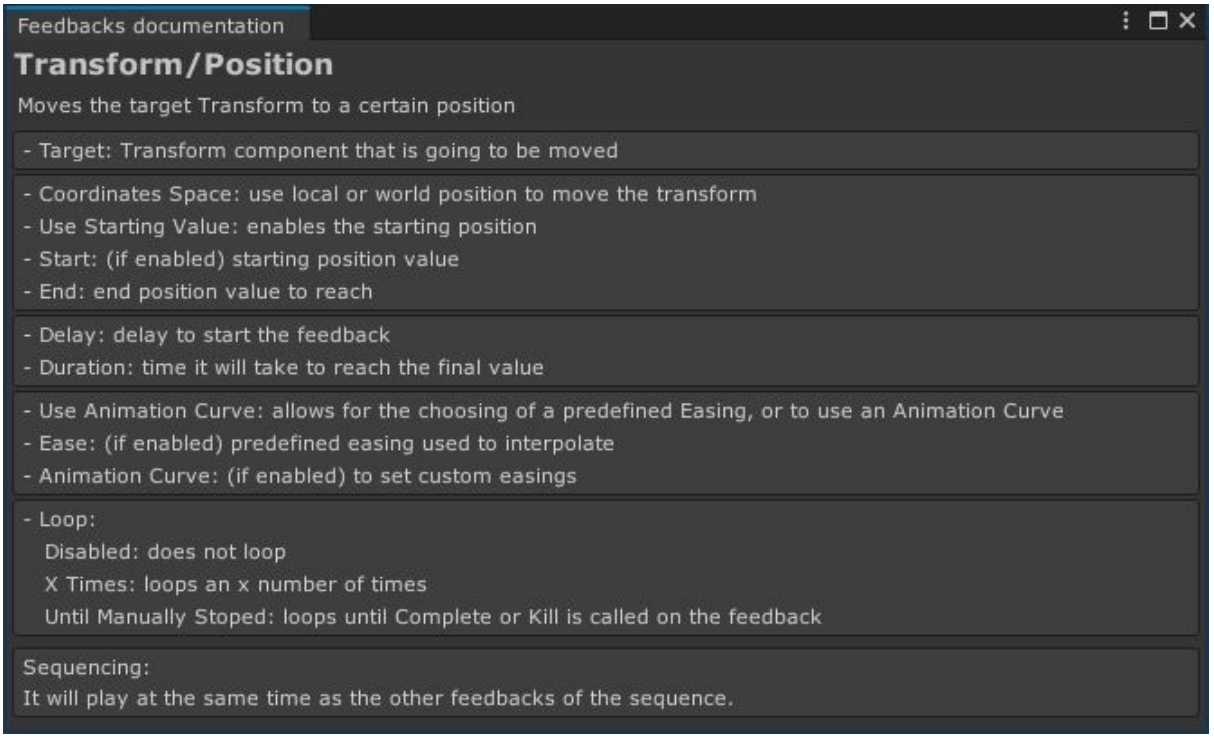


Extra functions:

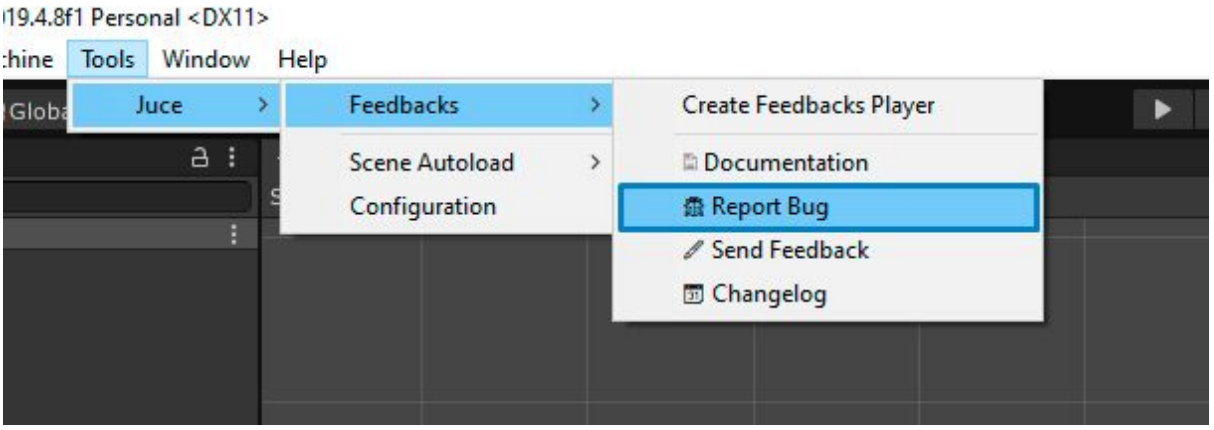
If you press the three dotted button to the right of any feedback, you can see extra functionality:



From here, you can remove a feedback, see its documentation and collapse/hide all feedbacks. Each one of the feedbacks has some documentation inside the editor, so you can check it without having to leave Unity while you learn how to use the feedbacks.



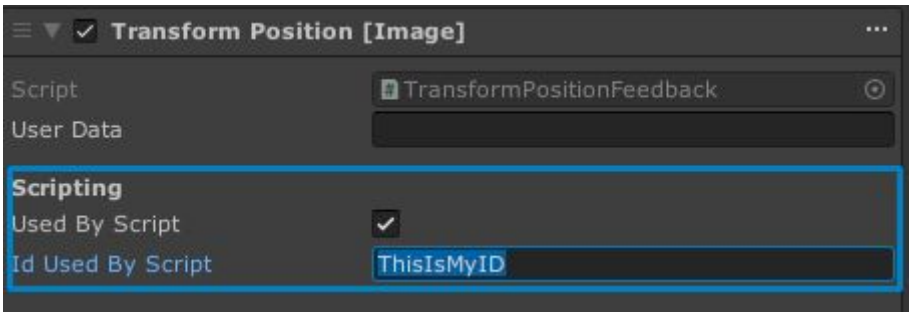
If you see anything that's wrong with the documentation, or you think that it can be improved, please don't hesitate on sending a bug/feedback.


19.4.8f1 Personal <DX11>
hine
Tools
Window
Help
Juce
Feedbacks
Scene Autoload
Configuration
Create Feedbacks Player
Documentation
Report Bug
Send Feedback
Changelog

4. Modifying values through scripting

Sometimes, you may want to set some values dynamically through scripting. Juce-Feedbacks embraces that and allows you to modify any value of a feedback on code.

To get a feedback to be able to modify it, first you have to enable the 'Used by script' field, so the FeedbacksPlayer knows you will want to retrieve this feedback at some point, and it does not have to iterate over all the feedbacks to find the one you want. Next, you have to set an ID that you will use on the code to retrieve this feedback. This ID can be whatever you want, but it has to be unique on that FeedbacksPlayer.



Next, from code you just have to get the reference of the FeedbacksPlayer, and then call GetFeedback, with the type of the feedback and the ID that we set on the editor. If you don't know the type of the feedback that you want to get, just go to the editor and it will show under the feedback name like this: Script SomeFeedback.

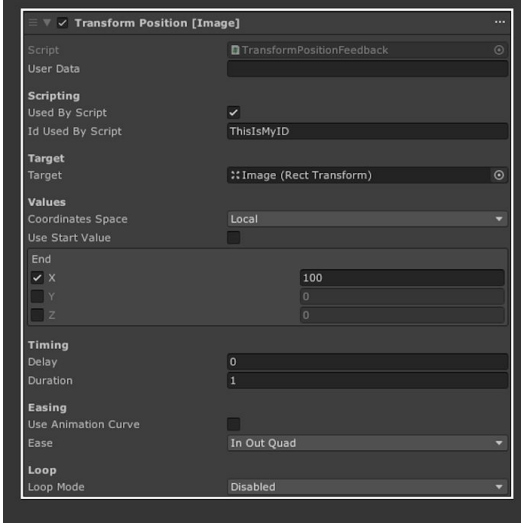
Keep in mind that if you don't use the proper type matching the ID that's on the editor, the GetFeedback function will return null!

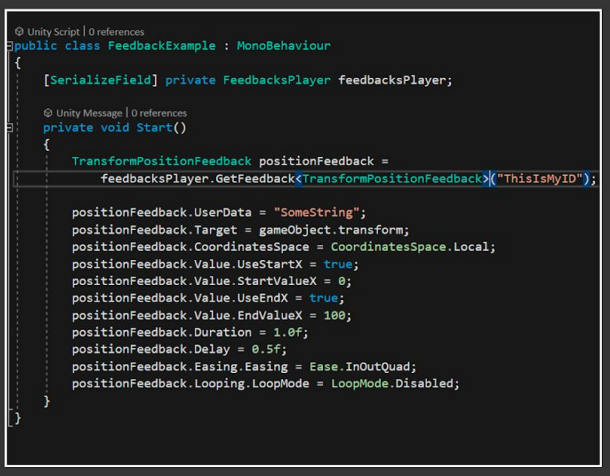
```
Unity Script | 0 references
public class JuceFeedbacksExample : MonoBehaviour
{
    [SerializeField] private Juce.Feedbacks.FeedbacksPlayer feedbackPlayer = default;

    Unity Message | 0 references
    private void Start()
    {
        TransformPositionFeedback feedback = feedbackPlayer.GetFeedback<TransformPositionFeedback>("ThisIsMyID");

        feedback.Value.StartValueX = 10.0f;
    }
}
```

From there, you have a direct reference to the feedback, and you can start chaning any of its values.

Keep in mind that you need the change the values before playing the FeedbacksPlayer.

Complete control (on editor and on code)





```
Unity Script | 0 references
public class FeedbackExample : MonoBehaviour
{
    [SerializeField] private FeedbacksPlayer feedbacksPlayer;

    Unity Message | 0 references
    private void Start()
    {
        TransformPositionFeedback positionFeedback =
            feedbacksPlayer.GetFeedback<TransformPositionFeedback>("ThisIsMyID");

        positionFeedback.UserData = "SomeString";
        positionFeedback.Target = gameObject.transform;
        positionFeedback.CoordinatesSpace = CoordinatesSpace.Local;
        positionFeedback.Value.UseStartX = true;
        positionFeedback.Value.StartValueX = 0;
        positionFeedback.Value.UseEndX = true;
        positionFeedback.Value.EndValueX = 100;
        positionFeedback.Duration = 1.0f;
        positionFeedback.Delay = 0.5f;
        positionFeedback.Easing.Easing = Ease.InOutQuad;
        positionFeedback.Looping.LoopMode = LoopMode.Disabled;
    }
}
```

4. Creating new feedbacks

Creating new feedbacks is very easy if you know what you are doing. In this section, you will learn all the necessary steps!

Setup:

To start, you need to create a new class that inherits from Feedback. For the feedback to be shown by the FeedbacksPlayer, you will need to add the FeedbackIdentifier attribute, where the first value is the name of the feedback, and the second one is the category.

```
using System;
using System.Collections.Generic;
using Juce.Feedbacks;
using Juce.Tween;

[FeedbackIdentifier("FeedbackName", "FeedbackCategory/")]
Unity Script | 0 references
public class NewFeedbackExample : Feedback
{
    45 references
    public override bool GetFeedbackErrors(out string errors)
    {
        errors = string.Empty;

        return false;
    }

    45 references
    public override string GetFeedbackTargetInfo()
    {
        return string.Empty;
    }

    47 references
    public override void GetFeedbackInfo(ref List<string> infoList)
    {

    }

    50 references
    public override ExecuteResult OnExecute(FlowContext context, SequenceTween sequenceTween)
    {
        return null;
    }
}
```

Next you will need to override the following functions:

- GetFeedbackErrors: allows the feedback to show error information on the editor, about something that is not properly set up. This error will be shown then as red text error under the feedback. Return true if there is an error and set the message error on the out string errors.

```
45 references
public override bool GetFeedbackErrors(out string errors)
{
    if (target == null)
    {
        errors = ErrorUtils.TargetNullErrorMessage;
        return true;
    }

    errors = string.Empty;
    return false;
}
```

- GetFeedbackTargetInfo: gets information about the target of the feedback. Normally this is the name of the GameObject, but it can be whatever you want. This text is going to be shown beside the name of the feedback.

```
45 references
public override string GetFeedbackTargetInfo()
{
    return target != null ? target.gameObject.name : string.Empty;
}
```

- GetFeedbackInfo: allows for showing important preview information about the feedback, that is going to be shown on the editor when the feedback is collapsed. For every property that you want to show, you need to add a new entry on the ref List infoList, so it's properly formatted.

```
51 references
public override void GetFeedbackInfo(ref List<string> infoList)
{
    InfoUtils.GetTimingInfo(ref infoList, delay, duration);
    InfoUtils.GetStartEndVector3PropertyInfo(ref infoList, value);
    InfoUtils.GetCoordinatesSpaceInfo(ref infoList, coordinatesSpace);
}
```

- OnExecute: function that is actually called when the feedback starts. We will see more information about this in the following sections.

Variables:

Next, you will need to define which variables do you need to use. The Feedbacks library normally follows the image pattern, but you can do what you please.

Also, you can see that we are using a custom type on the image named StartEndVector3Property. These are custom-made properties that have some extra

functionality on the editor, and help keep the inspector clean and compact. Those are the list of custom properties:

- EasingProperty
- GraphicMaterialColorProperty
- GraphicMaterialFloatProperty
- LoopProperty
- RendererMaterialColorProperty
- RendererMaterialFloatProperty
- StartEndColorNoAlphaProperty
- StartEndColorProperty
- StartEndFloatProperty
- StartEndTransformVector3Property
- StartEndUnitFloatProperty
- StartEndVector2Property
- StartEndVector3Property

There are examples of each of them on the code.

```
[FeedbackIdentifier("Position", "Transform/")]
Unity Script | 1 reference
public class TransformPositionFeedback : Feedback
{
    [Header(FeedbackSectionsUtils.TargetSection)]
    [SerializeField] private Transform target = default;

    [Header(FeedbackSectionsUtils.ValuesSection)]
    [SerializeField] private CoordinatesSpace coordinatesSpace = default;
    [SerializeField] private StartEndVector3Property value = default;

    [Header(FeedbackSectionsUtils.TimingSection)]
    [SerializeField] [Min(0)] private float delay = default;
    [SerializeField] [Min(0)] private float duration = 1.0f;

    [Header(FeedbackSectionsUtils.EasingSection)]
    [SerializeField] private EasingProperty easing = default;

    [Header(FeedbackSectionsUtils.LoopSection)]
    [SerializeField] private LoopProperty looping = default;
```

Scripting:

To make sure your new feedback can be modified through scripting, you need to create public acces to your variables, so they can be modified from the outside.

```
0 references
public Transform Target { get => target; set => target = value; }
0 references
public CoordinatesSpace CoordinatesSpace { get => coordinatesSpace; set => coordinatesSpace = value; }
0 references
public StartEndVector3Property Value => value;
0 references
public float Delay { get => delay; set => delay = Mathf.Max(0, value); }
0 references
public float Duration { get => duration; set => duration = Mathf.Max(0, value); }
0 references
public EasingProperty Easing => easing;
0 references
public LoopProperty Looping => looping;
```

Functionality:

To see how to create functionality, we will follow the TransformPositionFeedback as an example.

Internally, the feedbacks work as a tweening sequence that stack one after the other, so the only thing that we need to do is add our functionality to the sequence stack.

First, we append the delay at the beginning, if there is any.

```
50 references
public override ExecuteResult OnExecute(FlowContext context, SequenceTween sequenceTween)
{
    if (target == null)
    {
        return null;
    }

    Tween.Tween delayTween = null;

    if (delay > 0)
    {
        delayTween = new WaitTimeTween(delay);
        sequenceTween.Append(delayTween);
    }
```

Next, we append the necessary tweens to create the expected functionality.

```
if (value.UseStartValue)
{
    SequenceTween startSequence = new SequenceTween();

    switch (coordinatesSpace)
    {
        case CoordinatesSpace.Local:
            {
                if (value.UseStartX)
                {
                    startSequence.Join(target.TweenLocalPositionX(value.StartValueX, 0.0f));
                }

                if (value.UseStartY)
                {
                    startSequence.Join(target.TweenLocalPositionY(value.StartValueY, 0.0f));
                }

                if (value.UseStartZ)
                {
                    startSequence.Join(target.TweenLocalPositionZ(value.StartValueZ, 0.0f));
                }
            }
            break;
```

Finally, we set the easing and the looping properties.

The ExecuteResult class that is returned needs to receive two different types of sequences, so it shows the progress properly on the editor. First, it needs the delay sequence, and then, the sequence that has all the functionality of the tween.

```
Tween.Tween progressTween = endSequence;

sequenceTween.Append(endSequence);

EasingUtils.SetEasing(sequenceTween, easing);
LoopUtils.SetLoop(sequenceTween, looping);

ExecuteResult result = new ExecuteResult();
result.DelayTween = delayTween;
result.ProgresTween = progressTween;

return result;
```

If you don't have delay nor functionality, you can leave them to null.