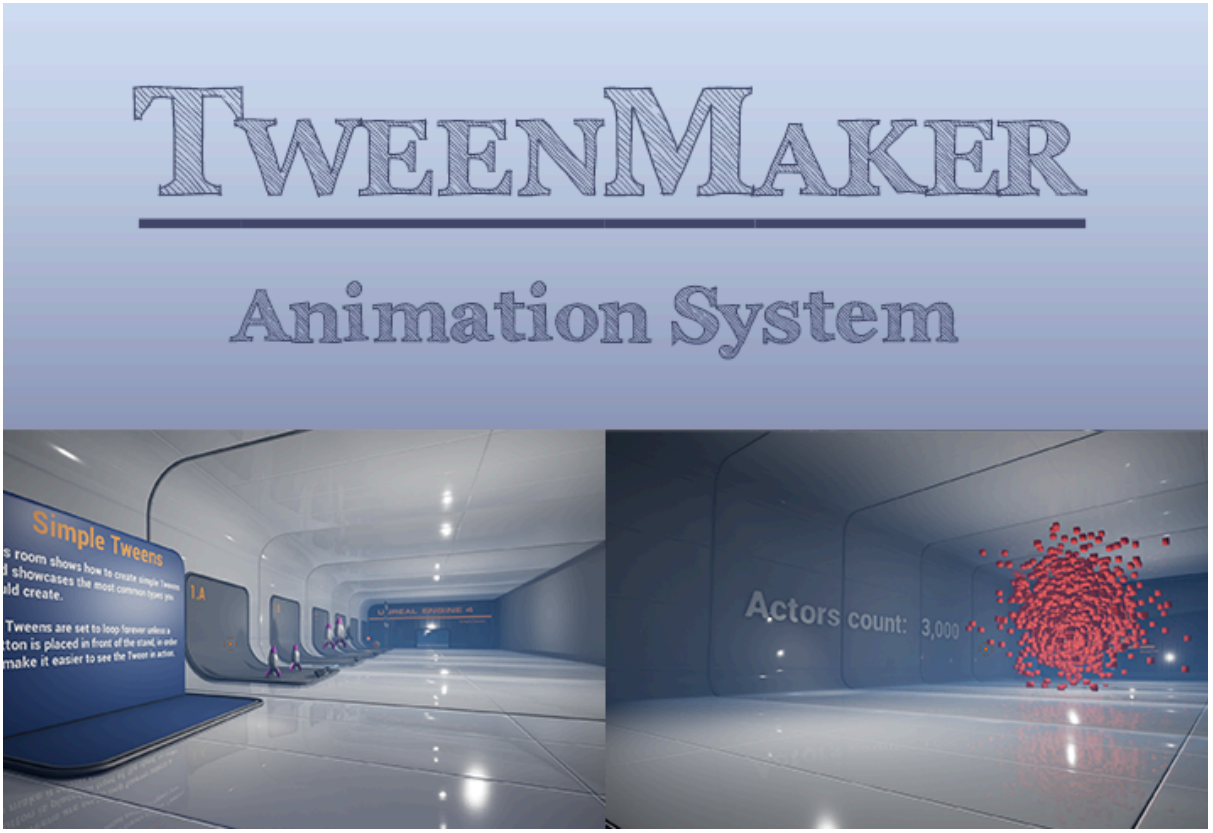
TweenMaker

Easy to use animation system for Blueprints and C++



Version 1.7.4

Available in the [Unreal Engine Marketplace]

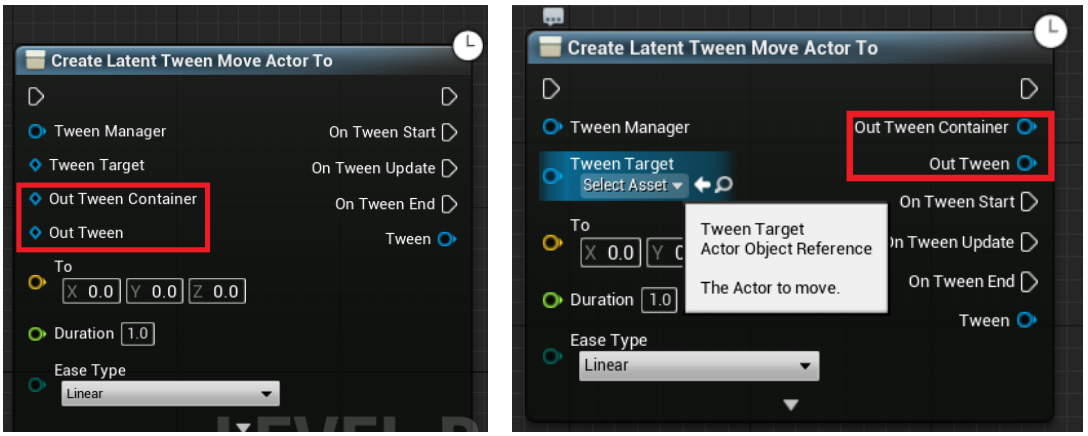
Changelog

Version 1.7.4 - 17-05-2023

[**ADD**] Added support for version UE 5.2 (current version up to UE 5.4)

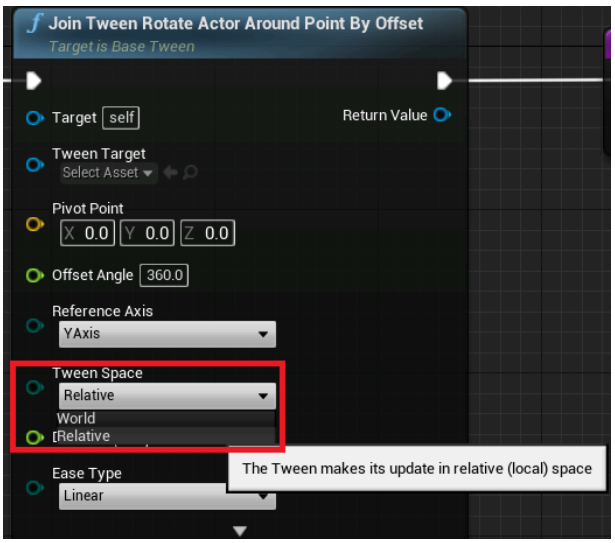
Version 1.7.3 - 05-09-2022

[**FIX**] General code cleanup. **Latent tweens are affected by this change** and might require updating. In particular, if you were using the parameters shown in the images below in latent Tweens, you will need to re-adjust and use the pin in the ‘output’ section instead.

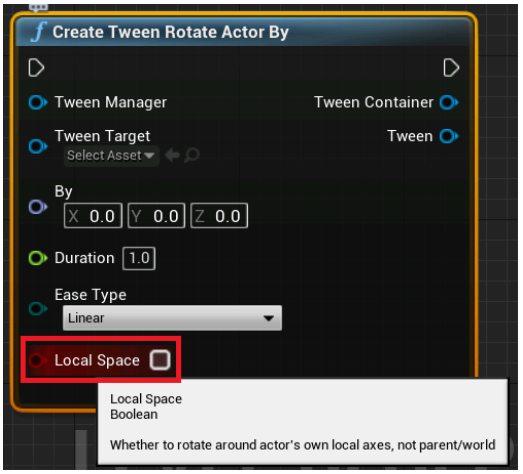


Before the update (left), after the update (right)

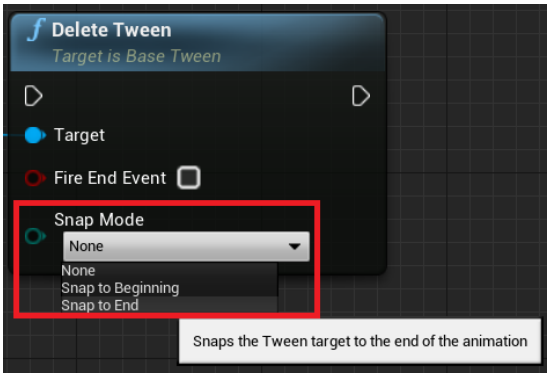
[**ADD**] Added the "tween space" (world or relative) for the "Rotate Around Point" type of Tweens, which allows to choose in which coordinate system it should make the location update:



[**ADD**] Added a 'local space' parameter to the “RotateBy” Tweens that allows to rotate the object around the Actor's own local axes, not parent/world (credits to Steven Streeting for the addition):



[**ADD**] Added the possibility to choose to snap at the beginning/end of theTween when deleting it manually:



Version 1.7.2 - 09-04-2022

[**ADD**] Added support for version UE 5.0.

[**FIX**] Fixed problem that might affect Tweens created during BeginPlay of the level in packaged games

Version 1.7.1 - 15-09-2021

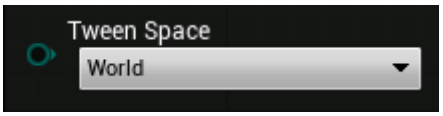
[**FIX**] Removed warnings when packaging the plugin while the Python Editor Script plugin is enabled.
[**FIX**] Fixed possible garbage collection issues.

Version 1.7 - 01-09-2021

[**ADD**] Added support for version UE 4.27.

[**ADD**] As requested, added the possibility to **choose the space (i.e. world or relative)** in which certain types of Tweens are executed (by default it's **World**).
The following Tweens will now have a new parameter that allows to choose which space to use (it applies for both Actors and SceneComponents):

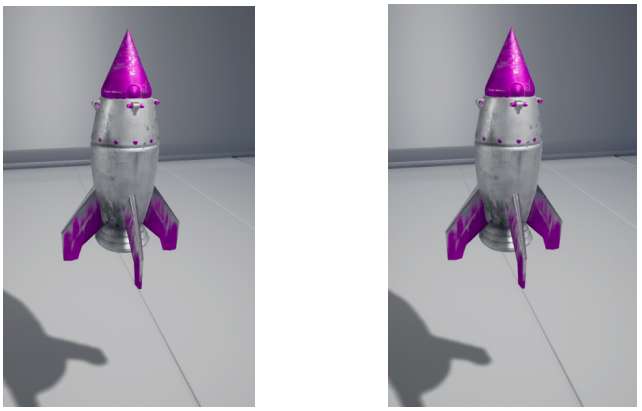
- MoveTo;
- RotateTo;
- ScaleTo.



The new option (available for the previously mentioned Tweens) is displayed in the advanced section

[**ADD**] As requested, added the possibility to **choose the rotation mode (i.e. shortest path or full path)** in which the RotationTo Tweens are executed (by default it's **Shortest path**).

The difference between them is the following: suppose you want to rotate an Actor by 270° on the X axis, starting from a 0° angle. The shortest path will move toward -90° (as seen on the left image below), while the full path will take the long way around and actually reach 270°.



Shortest path on the left; full path on the right.

The new option (available for the RotateTo Tweens) is displayed in the advanced section

NOTE for C++ users: if you're using the plugin with C++ and you might need to manually add the new parameters in order to compile.

Version 1.6.2 - 8-12-2020

[**FIX**] Minor fixes and code cleaning.
[**ADD**] Added support for version 4.26.

Version 1.6.1 - 15-05-2020

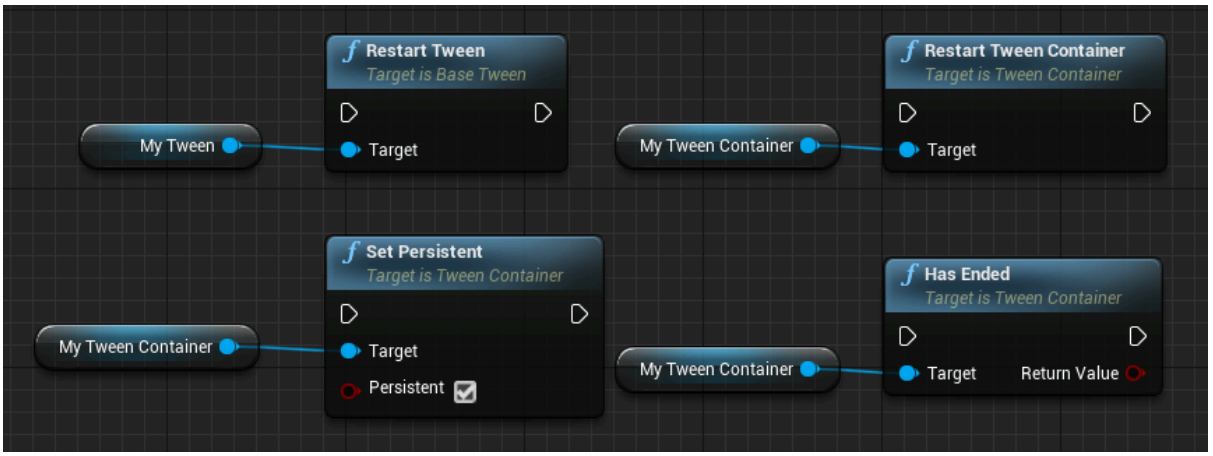
[**FIX**] Minor fixes due to compilation/packaging issues on certain platforms.

Version 1.6 - 10-05-2020

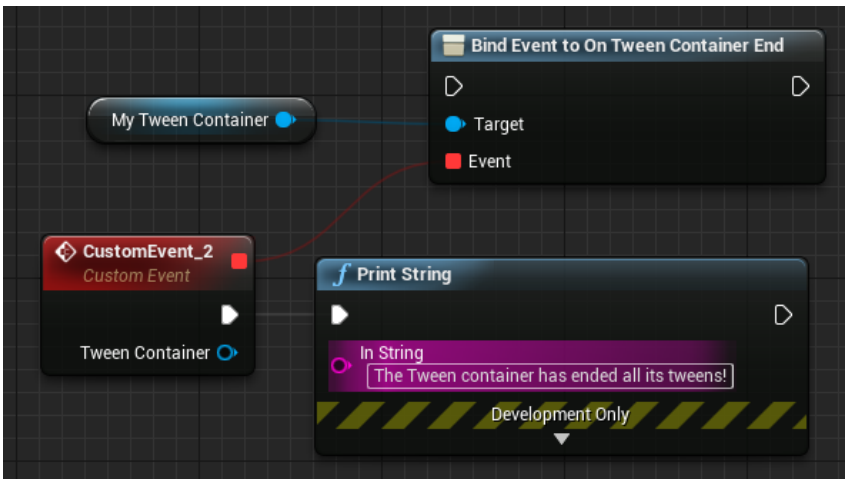
[**ADD**] As requested, added the possibility to **restart a Tween or a whole TweenContainer**. Along with this, added the possibility to make a TweenContainer **persistent**. This means that when the container ends, it will not be deleted by the TweenMaker system.

This is meant for those who need to use a Tween sequence that might end and restart multiple times and where performance is very important. Instead of recreating the sequence each time, you can set it as persistent and call the new RestartTweenContainer function to restart the whole sequence.

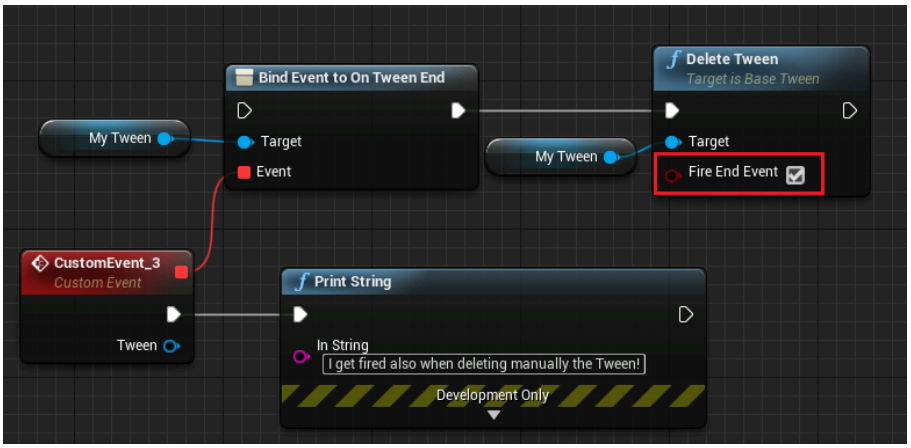
The functions that were added can be seen in the next picture.



[**ADD**] As requested, added the **OnTweenContainerEnd** delegate that is fired when a TweenContainer ends the whole sequence (i.e. all its Tweens).

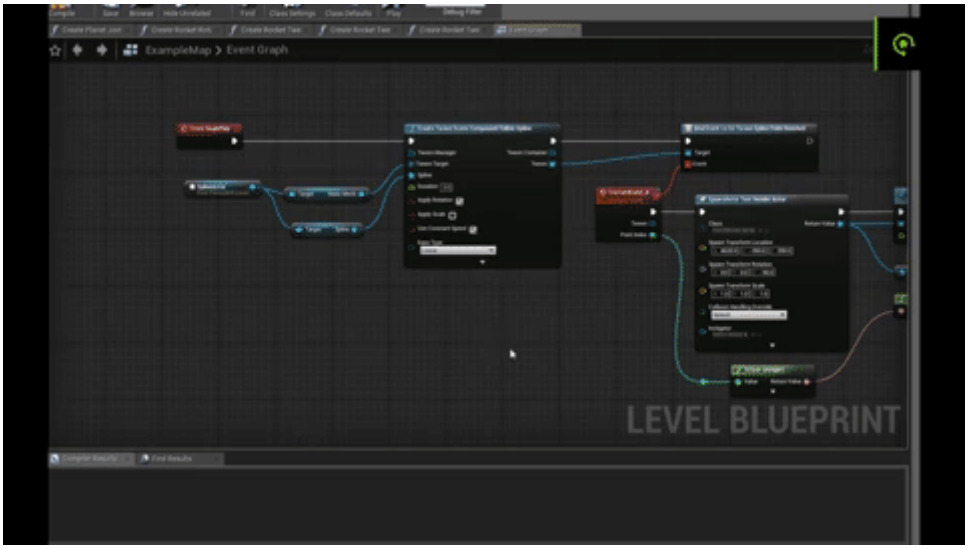


[**ADD**] As requested, added a way to fire the OnTweenEnd delegate before manually deleting a Tween. This can be useful in case you need to perform the same ending operation when you end a Tween manually as the one it would do when ending on its own.



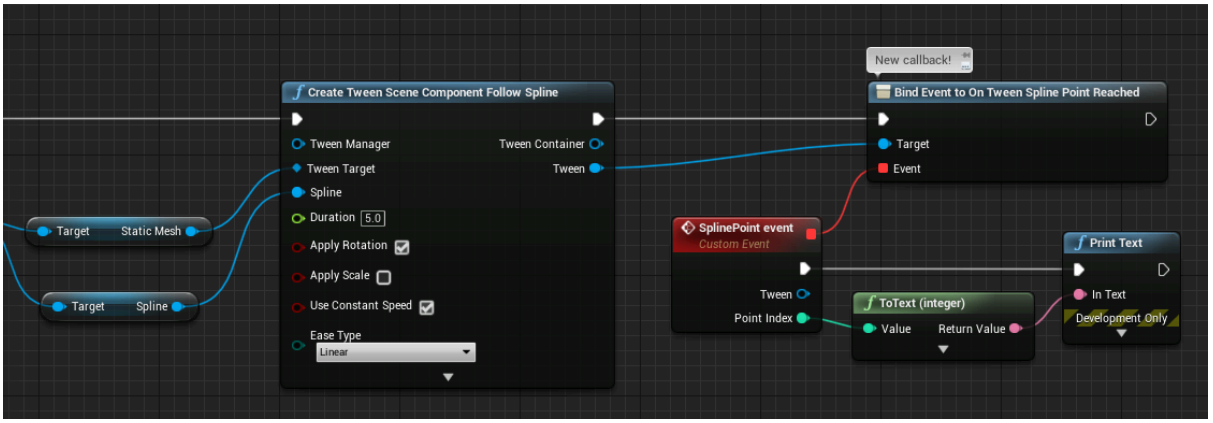
Version 1.5

[**ADD**] As requested, added a **callback for spline Tweens** that gets called whenever the Tween reaches a spline point (notice the text at the center of the gif, which gets updated whenever a new point is reached):

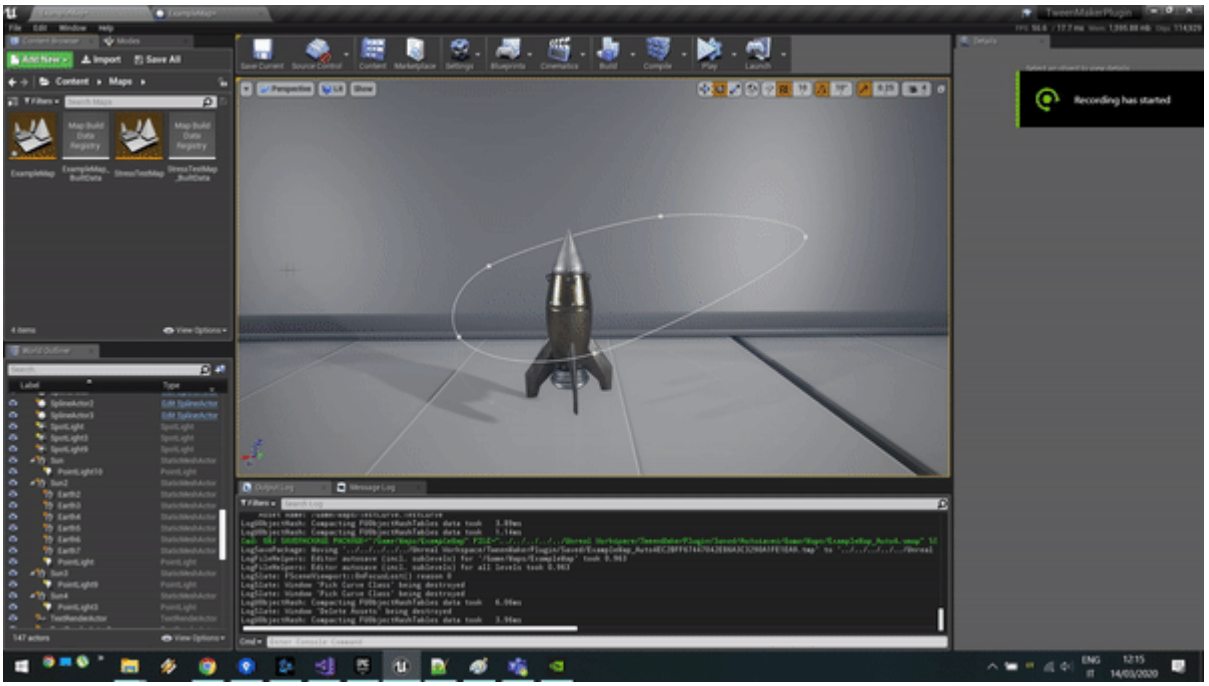


The callback receives as parameters the point that was reached and the Tween itself.

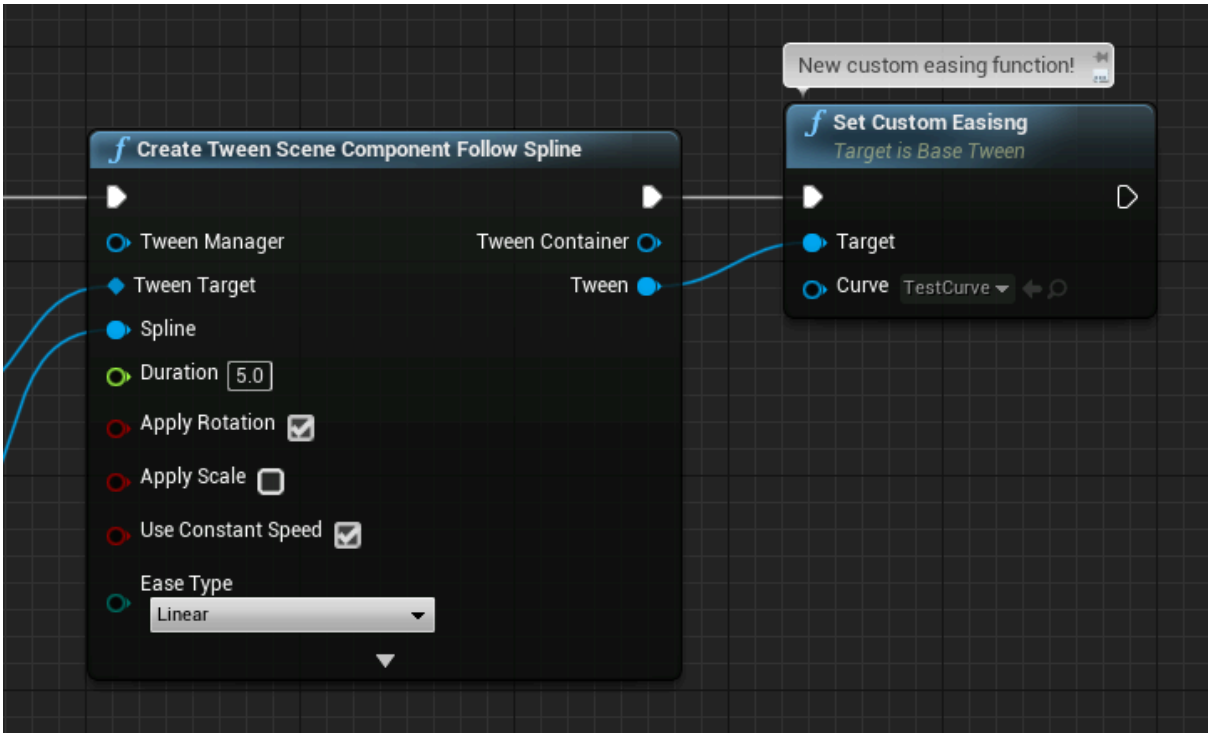
To use the callback simply add an event for it, like so:



[**ADD**] As requested, added a the possibility to implement **custom easing functions** by using the **CurveFloat** asset:

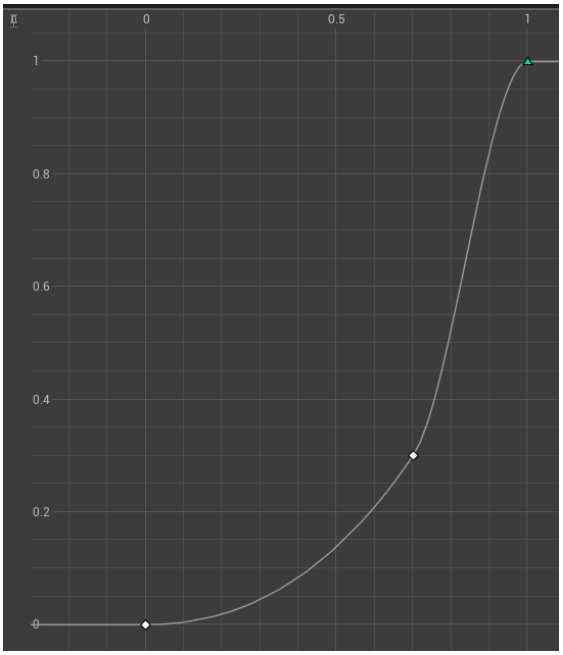


To use it, simply call the new function after creating a Tween (of any type!) and pass the curve asset:

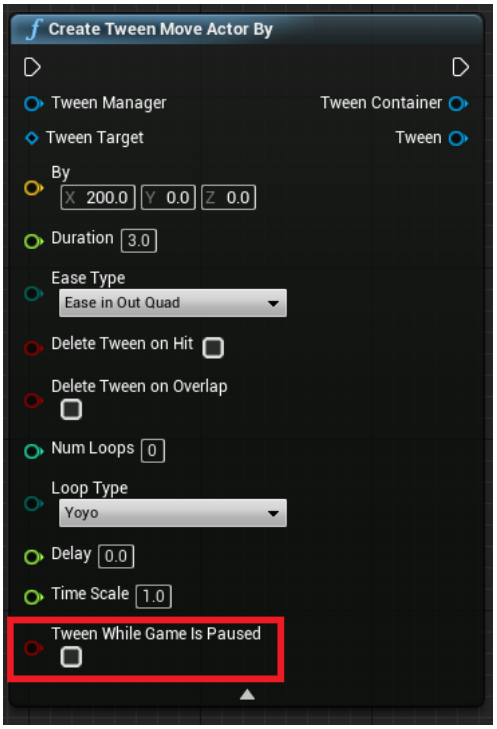


NOTE: the CurveFloat is expected to be in the range [0, 1] both for the X axis and for the Y axis, otherwise the Tween might behave in unexpected ways.

An example of a correct range for a CurceFloat is the following:



[**ADD**] As requested, added a way to keep Tweening while the game is paused. Now **each Tween has a new option called "Tween while game is paused"**, which can be set when the Tween is created. A Tween marked that way will keep animating even if the game is paused, which is particularly useful for animating UI elements such as menus, an inventory, and so on.

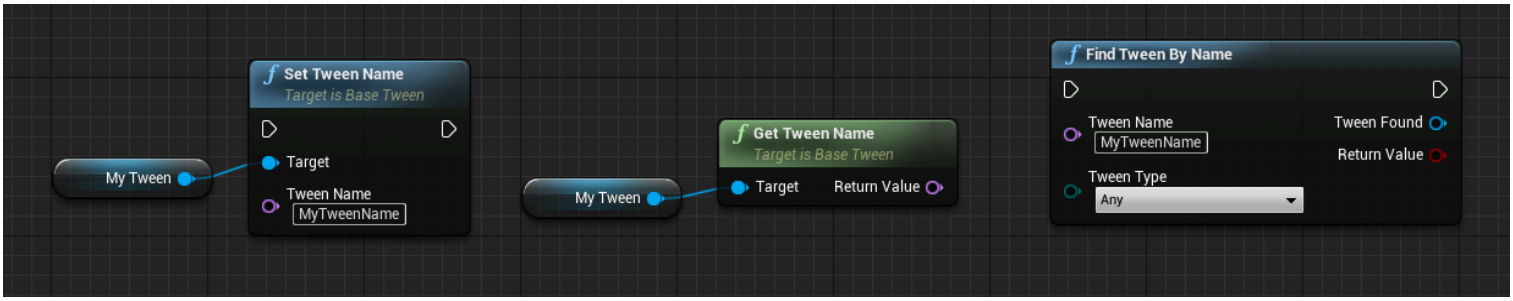


The new option (available for each Tween) is highlighted in red

Version 1.4

[**ADD**] As requested, added a simple but efficient “tag” system. You can now set a name for a Tween, which be used to differentiate it among other Tweens (imagine sharing an “OnTweenUpdate” event with multiple Tweens. By adding a name to each one you could do something different with each one by looking at their names). You can also find a Tween by its name among all other Tweens.

The following functions were added:



NOTE: you can use the *FindTweenByName* function to find a particular Tween, but remember to make sure that each Tween has a unique name, otherwise only the first Tween found will be retrieved.
Also, if you know the type of the Tween you’re looking for, it’s a lot more efficient to set it as the TweenType instead of using “Any”.

Version 1.3

[**ADD**] Added a optional parameter that enables using a constant speed in "*FollowSpline***"** type of Tweens. The parameter is called **"UseConstantSpeed"** and it is set by default.

NOTE for C++ users: if you're using the plugin with C++ and you have created "*FollowSpline"* type of Tweens, you will need to add the parameter in order to compile.

Version 1.2

Starting from this version only the **example project** for the 4.21 onward will be updated. Previous versions of the example project are no longer supported.

[**ADD**] Added support for version 4.22.

[**ADD**] As requested, added a new version for the "*RotateAroundPoint*" type of Tweens: ***RotateAroundPointByOffset***. This Tween does the same as *RotateAroundPoint*, but it automatically computes the radius and the axis of the rotation so that the tween target starts rotating exactly where it stands when the Tween starts. Much easier to use!

[**MOD**] Modified the **example project** for the 4.21 and 4.22 with the new *RotateAroundPointByOffset* Tween. In particular, examples **1.D**, **2.B**, **2.D** now use the new Tween.

Version 1.1

[**ADD**] Added support for version 4.21.
[**FIX**] Fixed a rare bug that occurred when using the plugin from C++.

Table Of Contents

Changelog **2**
- Version 1.7.4 - 17-05-2023 2
- Version 1.7.3 - 05-09-2022 2
- Version 1.7.2 - 09-04-2022 3
- Version 1.7.1 - 15-09-2021 3
- Version 1.7 - 01-09-2021 3
- Version 1.6.2 - 8-12-2020 4
- Version 1.6.1 - 15-05-2020 4
- Version 1.6 - 10-05-2020 5
- Version 1.5 6
- Version 1.4 9
- Version 1.3 10
- Version 1.2 10
- Version 1.1 10

Table Of Contents **12**
Introduction **12**
Content **13**
Installing TweenMaker **13**
Using TweenMaker **14**
- Tween Containers 16

Creating Tweens inside Blueprints **18**
- Standard and Latent Tweens 19
- Tween Utils/Info 21

Creating Tweens in C++ **21**
Conclusion / Getting Help **25**

Introduction

This is the manual for TweenMaker plugin. It will cover its installation and usage, both from Blueprints and C++. This document should be used along with the example project, which you can download at these links:

Example project:

- For version 4.21
- For version 4.22

Packaged example (built with UE 4.19 for Windows 64 bit, doesn't require the plugin).

The term "Tween" is short for "in-betweening", the process of generating intermediate frames between two images to give the appearance that the first image evolves smoothly into the second image. Tweening is a key process in all types of animation, including computer animation.

Content

This plugin provides several types of Tweens that can be applied to Actors, SceneComponents, Materials and UMG widgets. The following types are supported:

- **Actors** and **SceneComponents**:
    - Move;
    - Rotate;
    - Scale;
    - RotateAroundPoint;
    - FollowSpline.
- **Materials:**
    - MaterialScalar;
    - MaterialVector.
- **UMG Widgets:**
    - Move;
    - Scale;
    - AngleTo;
    - ShearTo;
    - OpacityTo.

TweenMaker also offers the possibility to create custom Tweens, in order to to be able

to Tween any type of object. The example project provides a few examples regarding custom Tweens.

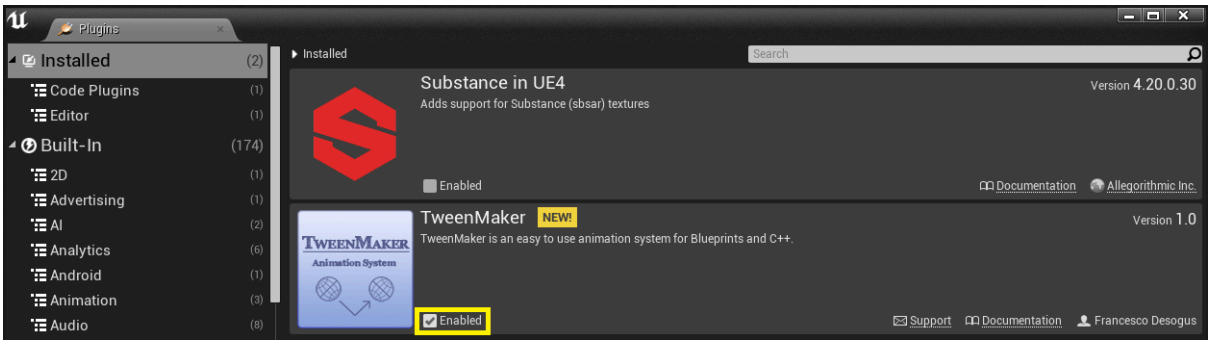
There are **41 easing functions** implemented, which lets Tweens behave differently depending on what the situation requires.

Installing TweenMaker

In this section, I'll quickly show you how to install TweenMaker using the Epic Games Launcher. Once you've purchased TweenMaker, please ensure that you restart the launcher. This will ensure that your Vault refreshes, and it should show TweenMaker now. As for the Vault, this is located at the bottom of your Library tab (UNREAL ENGINE > Library > Scroll Down). Once you're in here, search for "TweenMaker", and press the "Install to Engine" button. This will bring up a popup asking you which engine version you want to install it to.

Use the dropdown to select whatever version your project is using, and press the "Install" button. TweenMaker will now begin installing to the Unreal Engine and will be ready for use shortly!

Once you've opened the project you want to use TweenMaker in, head to Edit > Plugins, and find TweenMaker under the Installed > Code Plugins tab. Press the enabled checkbox for TweenMaker, and restart your editor (as prompted by a little yellow popup in the bottom). You are now ready to start using TweenMaker in this project!



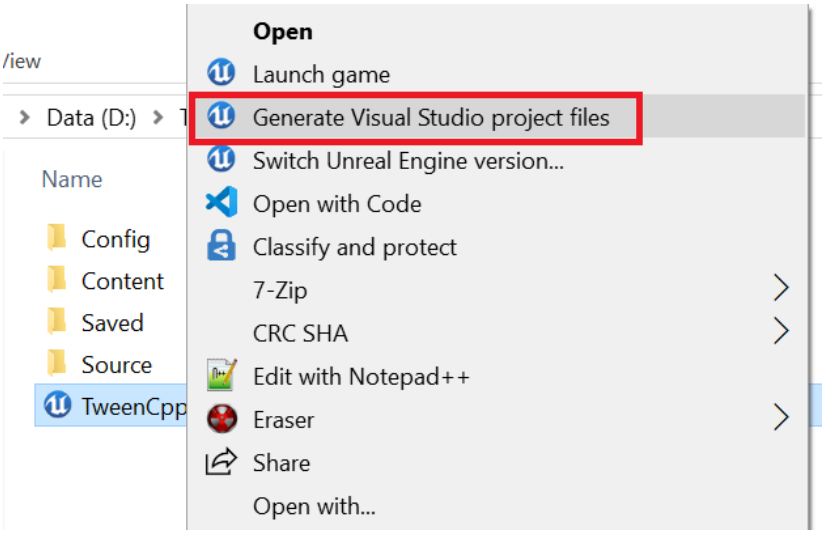
Installing TweenMaker for C++

If you're using TweenMaker in C++, you have to follow the steps of the previous paragraph, plus you have to add "TweenMaker" in the Build.cs file, like in the picture below.

```
using UnrealBuildTool;

public class TestTweenMaker : ModuleRules
{
    public TestTweenMaker(ReadOnlyTargetRules Target) : base(Target)
    {
        PCHUsage = PCHUsageMode.UseExplicitOrSharedPCHs;

        PublicDependencyModuleNames.AddRange(new string[] { "Core", "CoreUObject", "Engine", "InputCore", "TweenMaker" });


        PrivateDependencyModuleNames.AddRange(new string[] {  });
    }
}
```

Add “TweenMaker” to the dependencies if you’re using it in C++

After adding the “TweenMaker” to the list of modules, don’t forget to generate the Visual Studio project files again, as shown in the picture below. After that, you can access the TweenMaker classes as you would any other class.



Generate the Visual Studio files again after adding “TweenMaker” to the list of modules

Using TweenMaker

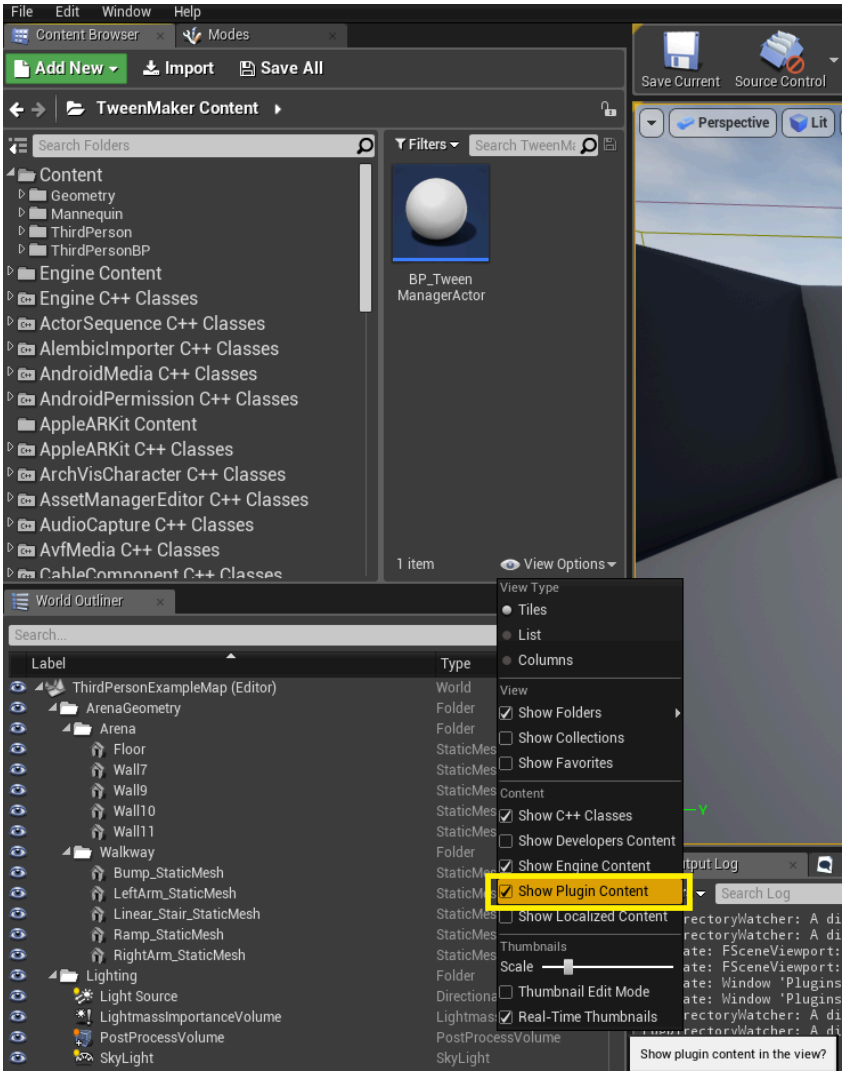
Now that you’ve installed and enabled TweenMaker for your project, it’s time to learn how to use it.

All Tweens are managed by a *TweenManagerComponent*. You can either add one to any Actor that needs to create Tweens, or you can place a *TweenManagerActor* anywhere in your level and it will allow you to create Tweens anywhere without having to worry about passing a reference to a TweenManagerComponent.

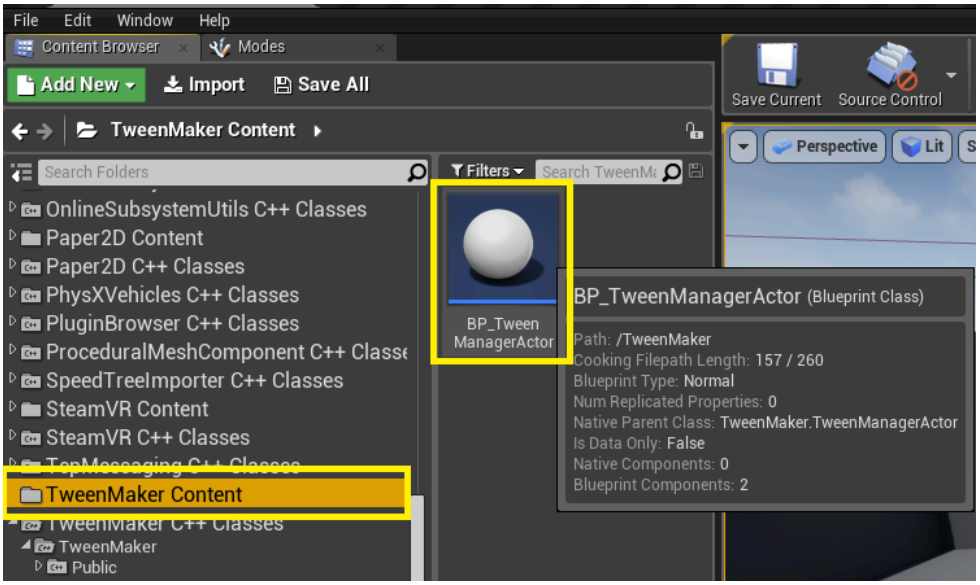
The latter is the easiest way to create Tweens and it’s the method I suggest using. Still, it might be useful to create TweenManagerComponents that are independent from each other from time to time, so there’s an option.

In the following section I'll consider the case in which a TweenManagerActor is being used. In order to place one, follow these steps:

1. Make sure that "Show Plugin content" is enabled in the Content Browser, like in the picture below.



2. Scroll down until you find the "TweenMaker Content" folder. Inside you'll find the TweenManagerActor Blueprint. Place it anywhere in your level.



Only **one** TweenManagerActor should placed in a level at any given time. If there are more than one, a warning will be displayed at runtime.

Tween Containers

Before explaining how to create Tweens it's important to understand the structure behind all Tweens and what can be done with them. If don't care much about this, head to the next section instead.

Each Tween belongs to a TweenContainer object, which is in charge of executing the Tween and it manages its lifecycle.

The TweenContainer can be seen as a holder for a sequence of Tweens. Tweens can be *appended* to a TweenContainer in order to be executed one after the other. Tweens can also be *joined* together in order to play them in parallel. At any time a Tween can be appended or joined to another, even in the middle of the sequence. If two or more tweens are joined together, before moving to the next Tween in the sequence they will all need to be completed.

All type of Tweens provide (optional) callbacks, which are the following:

- **OnTweenStart**: executed when the Tween starts (if the Tween loops, it gets executed at the beginning of each cycle);
- **OnTweenUpdate**: called at each Tick until the Tween is done;
- **OnTweenEnd**: called when the Tween ends (if the Tween loops, it gets executed at the end of each cycle).

Moreover, Tweens that move in any way the target of the animation also provide the following callbacks:

- **OnActor/SceneComponentHit**: called when the target hits something;

- **OnActor/SceneComponentBeginOverlap**: called when the target overlaps something.

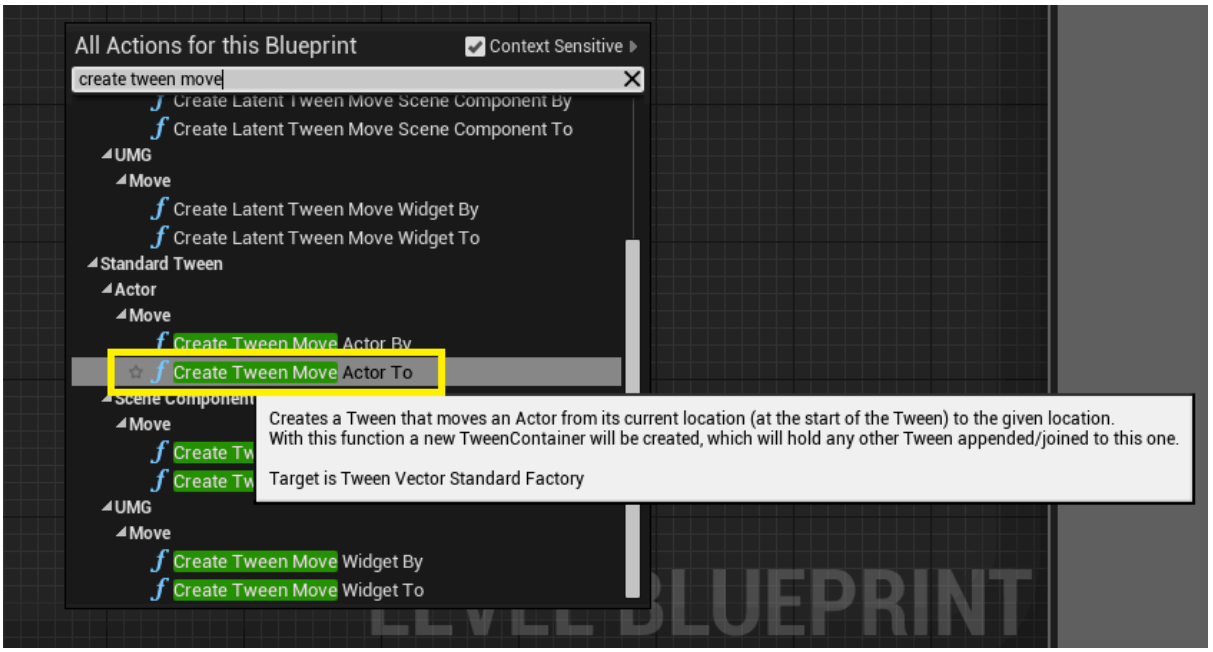
Regarding their lifecycle, Tweens are kept alive as long as their TweenContainer is alive (or they're terminated manually by the user). A TweenContainer is kept alive as long as its Tweens are still being executed. So for example, if a Tween is the first of a sequence of 100 Tweens, it will be kept alive until all 100 Tweens are done. This is because at any time a Tween or a TweenContainer can be set to loop, so all Tweens must be alive for that to happen. When a Tween or a TweenContainer is terminated, any reference to it will be set to null. So at any time it's important to check if a reference is valid before using it (unless you're absolutely that the Tween is being executed).

That said, since this plugin was designed to exploit the best features of both Blueprints and C++, the usage will be explained in different sections. Let's start by explaining how to create Tweens in Blueprints and then move on to C++.

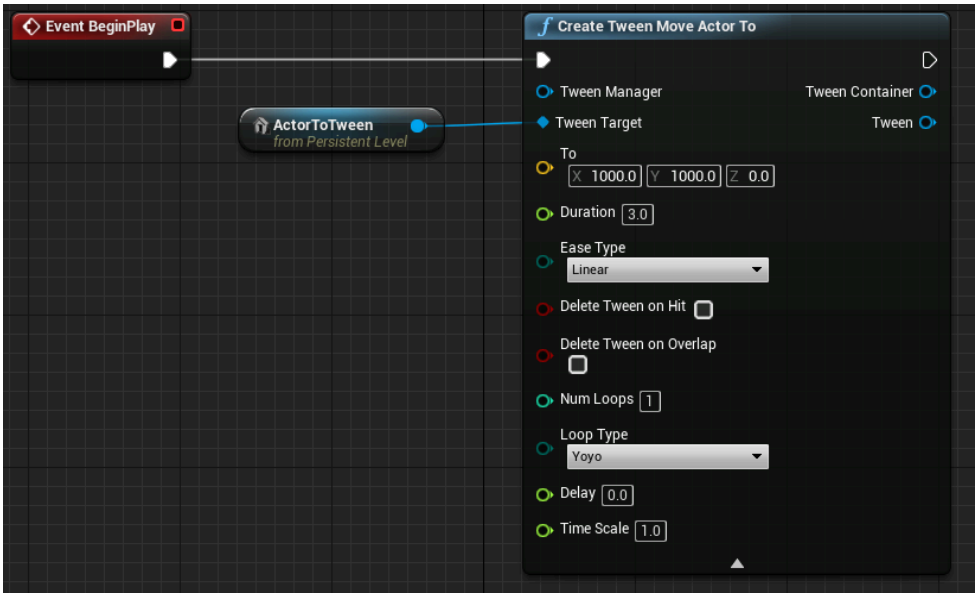
Creating Tweens inside Blueprints

To create a Tween, simply get a reference to the object you'd like to Tween and type "Create Tween" followed by the type of Tween you want (Move, Rotate, Scale, etc.). Each function comes with a description of what it does.

For example, let's create a "MoveTo" Tween, as in the picture.



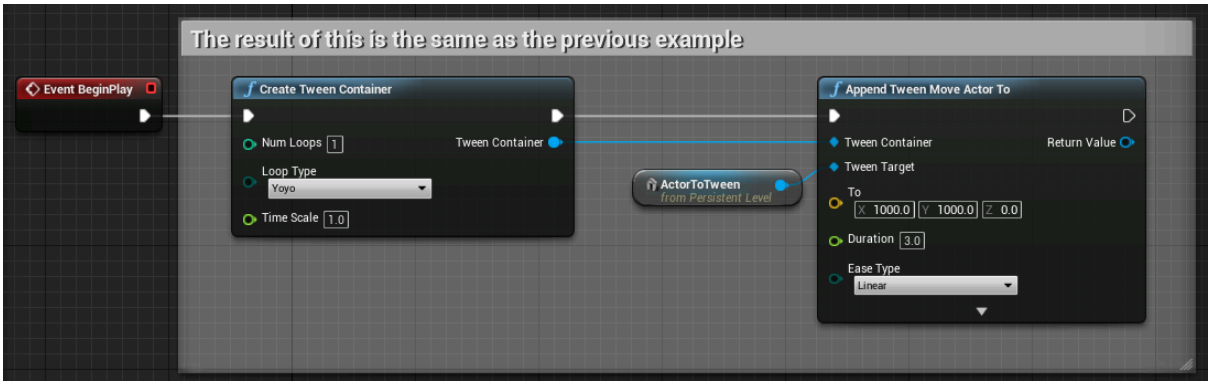
Now get a reference to an Actor you'd like to animate and plug it into the "Tween Target" input pin. Insert the values you'd like in the other pins (mouse over each input pin to read a description of what it does).



It's worth noting that the "Tween Manager" input pin is left unplugged. If a TweenManagerActor is being used to manage the Tweens, no pin should be attached to it. If your Actor is using an independent TweenManagerComponent, then you should plug that to the "Tween Manager" input pin.

The outputs of this function are the TweenContainer that holds the newly created Tween, and the Tween itself.

This is the easiest way to create a Tween, but you could also create an empty TweenContainer and then append Tweens to it, like in the image below.

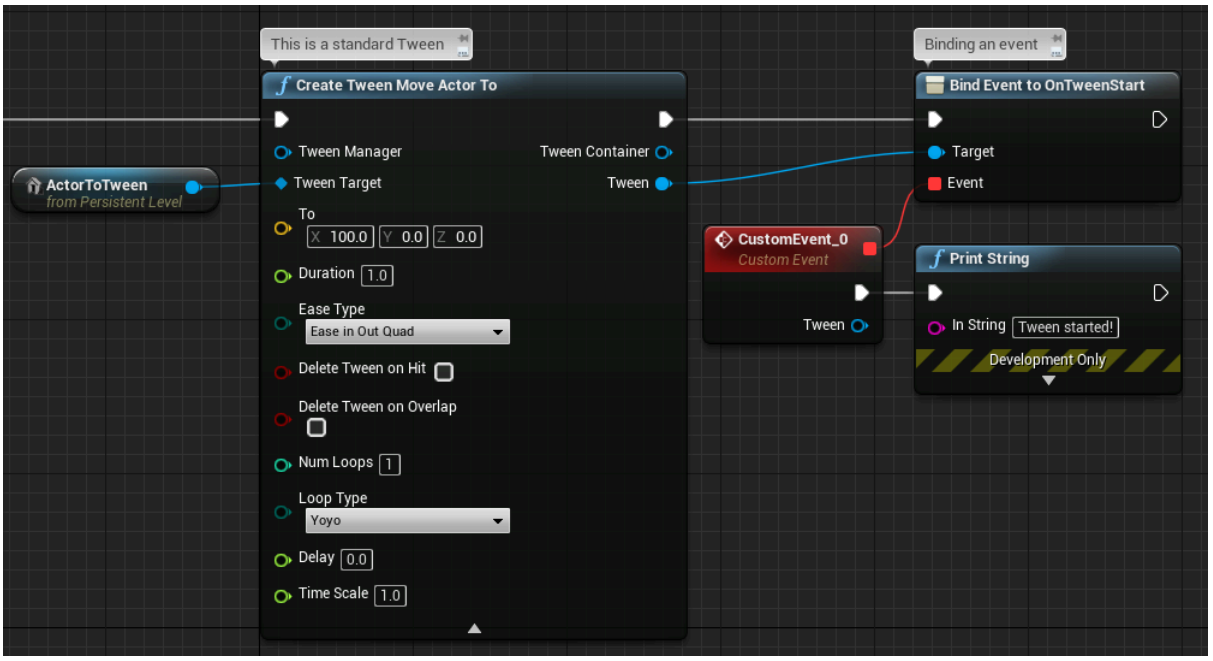


The advantage of this is that you can set the loop count of the TweenContainer (and so of the whole sequence of Tweens that it manages) and its timescale, while using the "Create…" functions you have to manually call "SetLoop" from the TweenContainer.

Standard and Latent Tweens

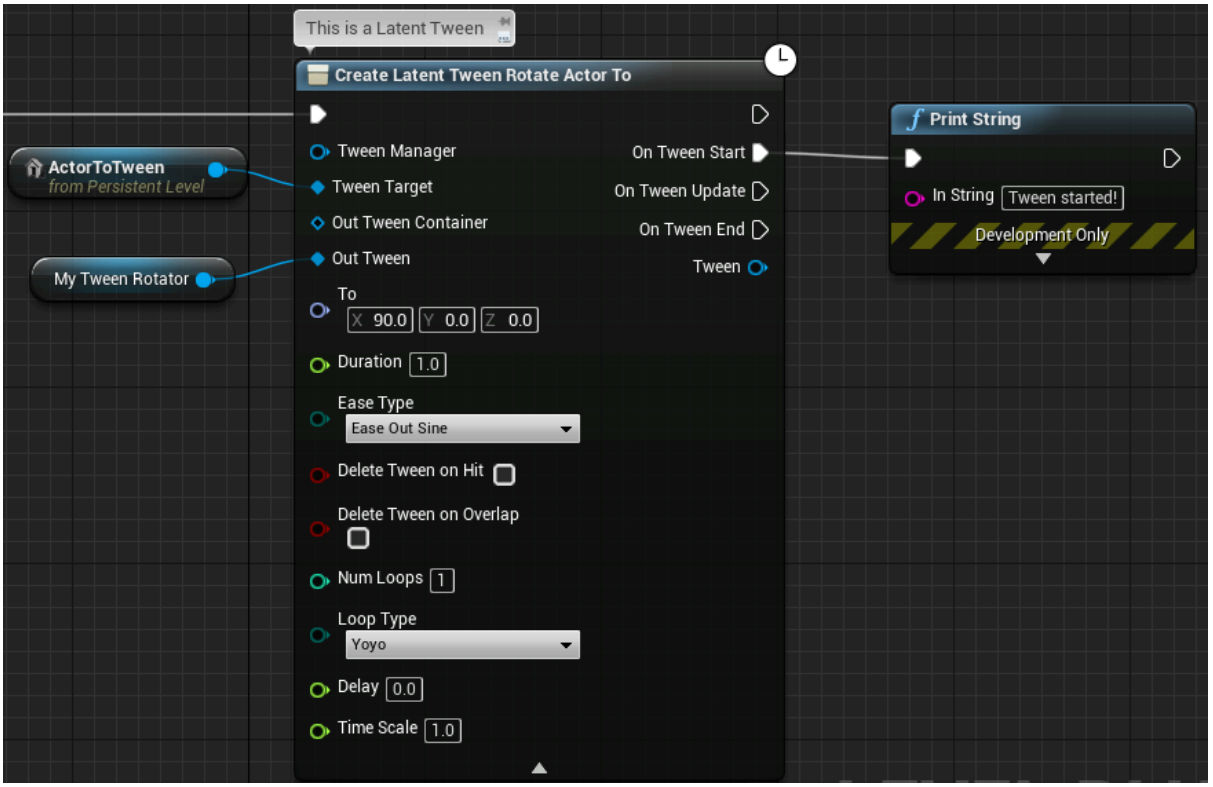
Inside Blueprints, Tweens can be of 2 main types: latent and standard. The difference between these 2 types is mainly due to the way they handle callbacks.

Standard Tweens are created as in the picture below, which shows as an example a Tween move. Standard Tweens always output the newly created Tween, to which can be bound an event or a function.



Standard Tween with a callback

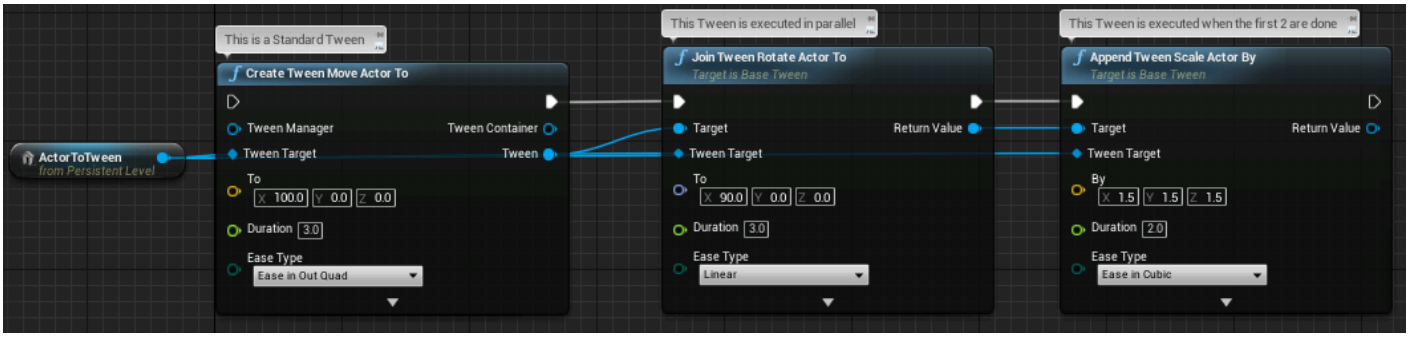
Latent Tweens provide an easy way to associate behaviours to the callbacks (as can be seen in the picture below), but are less flexible. The output pin will only be valid between the OnTweenStart and OnTweenEnd. Because of this, if a reference to the Tween is needed, a variable of the appropriate type must be passed as input to the function, as shown in the picture. The same must be done if a reference to the owning TweenContainer is needed.



Latent Tween with a callback (note the “Latent” word in the function’s name)

Moreover, in Unreal latent functions in general can’t be used inside functions. Because of this, Latent Tweens can only be created in event graphs.

Because of these limitations, I would suggest using Latent Tweens when you need to create simple autonomous Tweens. If you need something more complex, like joined Tweens or a sequence (or both), I would suggest using standard Tweens, since they’re easier to “chain together”, as shown in the picture below, in which I sequence of 2 Tweens is created, with another one joined to the first Tween of the sequence.

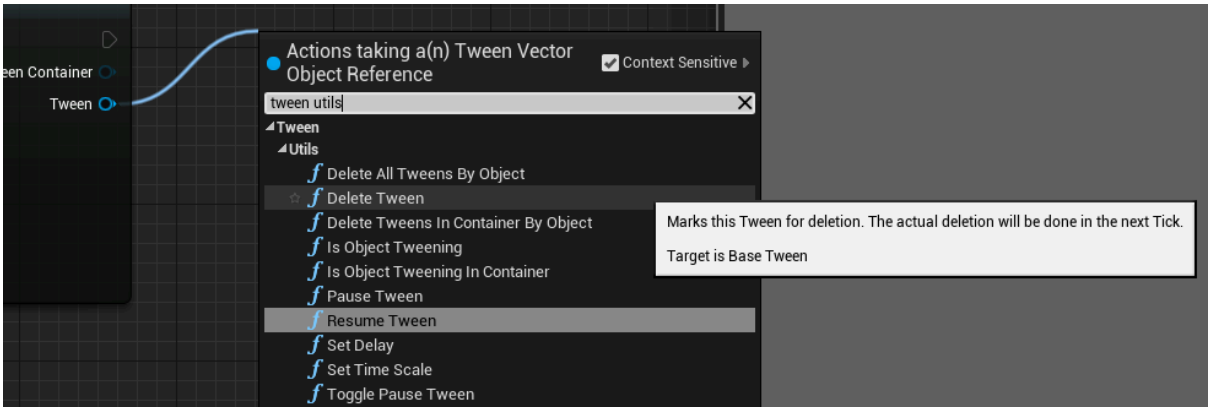


Appending and joining Tweens together

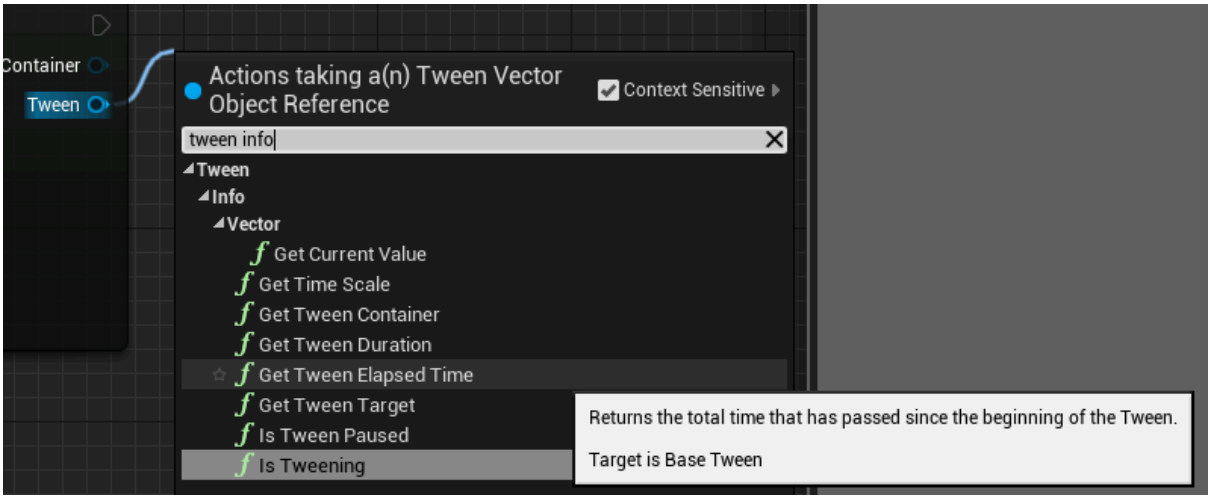
Tween Utils/Info

All Tween objects extend a BaseTween class, which provides some utility functions to help you manage Tweens. These functions can be accessed by typing “tween utils”, like in the picture below. Among the utility methods there are those that

pause/resume/delete a Tween and those that are more generic and help you find particular Tweens among all others.



Also, from a Tween object you can access some information about it, like it's current value, the elapsed time and the total time. All functions that retrieve information are under the "Info" category in the function structure, as you can see in the picture below.



For more examples and usages check out the example project.

Creating Tweens in C++

In this section we'll be discussing how to create Tweens in C++. Before starting, I will say that as a rule all methods that are marked with a "*BP_*" prefix should **only** be used in Blueprints. All other (public) methods can be safely used in C++.

The list of classes and files can be found here.

Unlike in Blueprints, there are no Latent Tweens in C++, and there are no "Create…" functions. In order to create a Tween you have to first create the UTweenContainer

object that will hold it:

```
UTweenContainer* tweenContainer = UTweenManagerComponent::CreateTweenContainerStatic();
```

Note that I'm using a static method. This way the Tween will exploit the ATweenManagerActor that should be placed in the scene (if it's not, a warning will be displayed at runtime and the Tween won't be created). If you need to create a UTweenManagerComponent, you should call the non static version of that method (it has the same name, minus the "Static" word in the end).

Just like the Blueprint version, when creating a TweenContainer you can specify the number and type of loops and its timescale, if you like (or you can do it in a second time by calling the respective "SetLoop" and "SetTimescale" methods).

After creating the TweenContainer you can start appending/joining Tweens to it. Appending a Tween to the TweenContainer means adding it as the last Tween in the sequence, while joining a Tween to the TweenContainer means adding it to the last Tween in the sequence to be played in parallel.

For example, let's create a sequence that will move an Actor while scaling it, and then rotates it by 90°. The code is the following:

```
#include "MyActor.h"
#include "TweenContainer.h"
#include "TweenManagerComponent.h"
#include "TweenVector.h"
#include "TweenRotator.h"

AMyActor::AMyActor()
{
    PrimaryActorTick.bCanEverTick = true;
}

void AMyActor::BeginPlay()
{
    Super::BeginPlay();

    UTweenContainer* container = UTweenManagerComponent::CreateTweenContainerStatic();
    UTweenVector* move = container->AppendTweenMoveActorBy(this, FVector(100, 0, 0), 3.0f);
    UTweenRotator* tweenRotate = move >AppendTweenRotateActorBy(this, FRotator(90, 0, 0), 2.0f);
    UTweenVector* tweenScale = move >JoinTweenScaleActorBy(this, FVector(0.5f, 0.5f, 0.5f), 1.0f);
}
```

Note that each Tween uses a particular Tween type, which represent the value that is modified by the interpolation. For example, a “Move” and a “Scale” Tween modifies a FVector, so they’re *UTweenVector*. The “Rotate” Tweens uses a FRotator, so it’s a *UTweenRotator*.
There are also *UTweenLinearColor*, *UTweenVector2D* and *UTweenFloat*.

If you want to add callbacks, you can bind functions to the delegates that start with “OnTween” and end with “Delegate” (if it doesn’t end with the word “Delegate” it’s a Blueprint delegate and should be left untouched). For example, let’s add some callbacks to the “tweenRotate” of the previous example:

```
{
   // … Previous code

    tweenRotate->OnTweenStartDelegate.BindLambda([] (UTweenRotator* pTweenRotator)
    {
        UE_LOG(LogTemp, Warning, TEXT("This Tween is starting! It's going to last for %f seconds"),
pTweenRotator->GetTweenDuration());
    });

    tweenRotate->OnTweenUpdateDelegate.BindLambda([] (UTweenRotator* pTweenRotator)
    {
        UE_LOG(LogTemp, Warning, TEXT("Current value: %s; time passed: %f"),
*pTweenRotator->GetCurrentValue().ToString(), pTweenRotator->GetTweenElapsedTime());
    });

    tweenRotate->OnTweenActorHitDelegate.BindUObject(this, &AMyActor::OnTweenHit);
}

void AMyActor::OnTweenHit(UTweenRotator* pTween, AActor* pThisActor, AActor* pOtherActor,
FVector pNormalImpulse, const FHitResult& pHitResult)
{
    UE_LOG(LogTemp, Warning, TEXT("I hit this actor: %s. I'll destroy myself in a second!"),
*pOtherActor->GetName());
    SetLifeSpan(1.0f);
    pTween->DeleteTween();    // Let’s remove the Tween
}
```

Here I'm using C++ lambdas to easily bind behaviours to the Tween's delegates (the reason why the C++ delegates are different from the Blueprint ones is to allow you to take advantage of the C++ lambdas, which do not work in Blueprint delegates). In particular, I'm printing to the log when the Tween starts and I keep printing at each update the current value of the Tween and the elapsed time. I also bind an event when the Actor hits something, but this time I'm using a function instead of a lambda. The function is defined as a UFUNCTION(), with the following signature in the header:

```
UFUNCTION()
void OnTweenHit(class UTweenRotator* pTween, AActor* pThisActor, AActor* pOtherActor, FVector
pNormalImpulse, const FHitResult& pHitResult);
```

Before concluding, you should know that some utility functions that allow you to delete particular Tweens by object or finding out if a UObject is Tweening (and in that case also getting the reference to the given Tween) are in the ATweenManagerActor class as static functions. For example, the snippet below will delete all Tweens of the Actor in the previous example when the Actor is hit:

```
void AMyActor::OnTweenHit(UTweenRotator* pTween, AActor* pThisActor, AActor* pOtherActor,
FVector pNormalImpulse, const FHitResult& pHitResult)
{
    // Let's delete any kind of Tween currently active for this Actor
    ATweenManagerActor::DeleteAllTweensByObject(this, ETweenGenericType::Any);
}
```

The following snippet will check if there is a "Move" Tween currently active for this Actor, and if so it prints out the elapsed time:

```
void AMyActor::OnTweenHit(UTweenRotator* pTween, AActor* pThisActor, AActor* pOtherActor,
FVector pNormalImpulse, const FHitResult& pHitResult)
{
    UBaseTween *tweenFound = nullptr;
    if(ATweenManagerActor::IsObjectTweening(this, ETweenGenericType::Move, tweenFound))
    {
        UE_LOG(LogTemp, Warning, TEXT("Object is moving! Elapsed time: %f seconds"),
tweenFound->GetTweenElapsedTime());
    }
}
```

That will be all for C++, since these are the main things that you need to know in order to start creating Tweens in C++.

Conclusion / Getting Help

This concludes the TweenMaker documentation. Please feel free to reach out to me at “francesco.desogus9@gmail.com” if you have any further questions, suggestions or features you’d like to see.