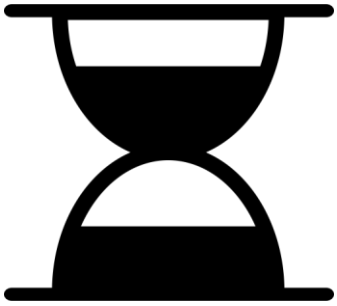


Unity Time Rewinder documentation (ver 3.1)

Unity Time Rewinder lets you rewind defined object states and move freely on time axis. It supports custom trackers with custom variable tracking that is easy to setup. System uses highly efficient circular buffer implementation to store and retrieve the values.

Table of contents

Downloading Time Rewinder and importing to custom Unity Project

You can either download the whole Unity example project from Github or download the prepared Unity package in Github releases. To use the Time Rewinder, only TimeRewinderImplemention folder is required, but i highly suggest checking the demoscene with prepared examples.

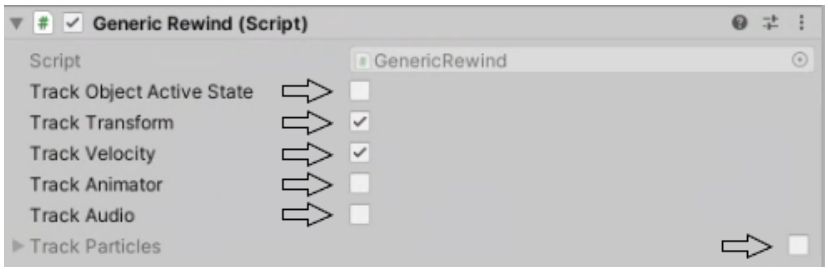
Start using Time Rewinder in your own project

To start using Time Rewinder, each scene must contain **RewindManager.cs** script, that is essential. In TimeRewinderImplementation/Scripts/RewindInputs folder, there are prepared two scripts. **RewindBySlider.cs** and **RewindByKeyPress.cs.** As names suggest, both scripts let you rewind the time differently. You can choose between these two prepared solutions that best suit your needs, or you can also write your own input rewind logic (more on that in the last chapter of this documentation).

Showcases of rewinding with these two solutions are prepared in 3 demoscenes (TimeRewindExample 1,2,3) that are located in ExampleScene folder.

RewindManager.cs contains property of how many seconds should be tracked, you can change this value directly from editor field.

For straight from the box use, import one of the prefabs that you choose from TimeRewinderImplementation/Prefabs into the scene (these prefabs already contain **RewindManager.cs** and respective input solution). Then on desired object that you want to rewind, you can use prepared script **GenericRewinds.cs**, that you attach to the object and then check what properties you want to track.

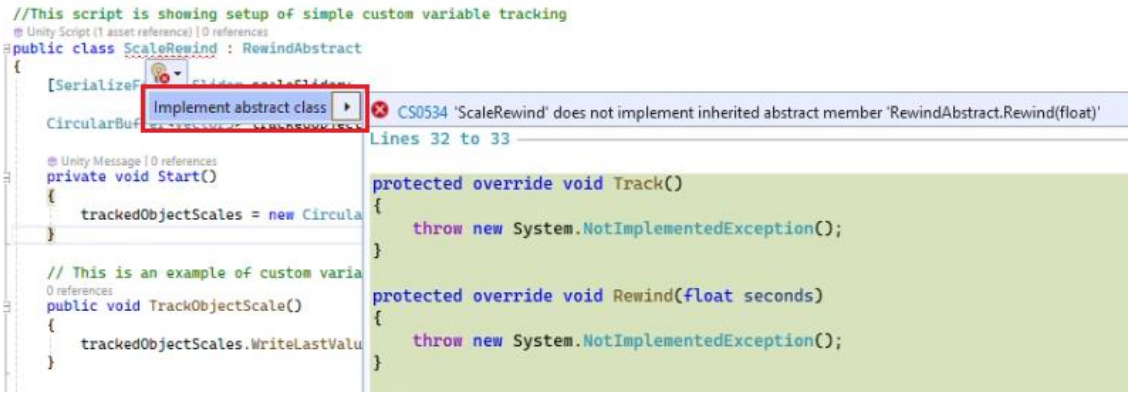


Currently straight from the box you can track object active states, Transforms (position, rotation, scale), ridigbody velocity, animator states, audiosource states (also audio in mixer groups) and particle effects. Adding custom variable is easy, and i will show you how to do it in next section. To start rewinding, play the scene. If you have choosen **RewindByKeyPress.cs**, you simply hold SpaceBar to rewind back. If you choosen **RewindBySlider.cs**, grab the slider handle and you can see, that you can move on time axis freely and see preview

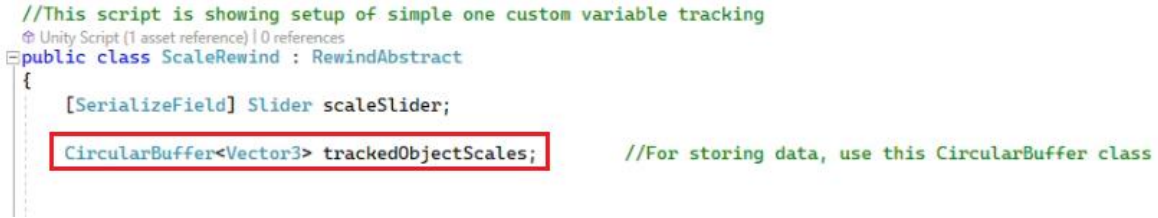
of historic snapshots. When you release the slider, active object state will be overwritten with preview values.

Adding custom tracker with custom variables tracking

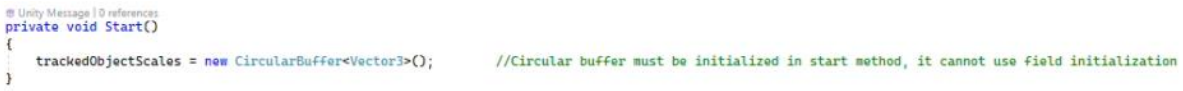
In TimeRewinderImplementation/Scripts/ImplementedObjects, you can find **ScaleRewinds.cs** and **TimerRewinds.cs** which are two examples of implemented custom trackers. The first thing when doing custom tracker is implementing **RewindAbstract.cs** abstract class, which contains all needed things for custom tracker. Simply inherit this class and let the editor implement all its methods.



To add custom variable tracking, add CircularBuffer (CircularBuffer is implemented in **CircularBuffer.cs**) with data structure you want to track. In our case, in **ScaleRewind.cs**, we want to track Vector3 values



Circular buffer must be initialized in Start() method, it cannot use field initialization because Time.fixedDeltaTime that is used inside the buffer doesnt exist yet.



Now you can implement what values will be tracked and restored. I suggest adding 2 separate methods for clarity (Tracking method and Restore method).

In these methods you simply tell what values will be tracked and then what will be restored on rewind for your custom object.

```
// This is an example of custom variable tracking
1 reference
public void TrackObjectScale()
{
    trackedObjectScales.WriteLastValue(transform.localScale);
}


// This is an example of custom variable restoring
1 reference
public void RestoreObjectScale(float seconds)
{
    transform.localScale = trackedObjectScales.ReadFromBuffer(seconds);

    //While we are at it, we can also additionaly restore slider value to match the object scale
    scaleSlider.value = transform.localScale.x;
}
```

Use **CircularBuffer.WriteLastValue()** and **CircularBuffer.ReadFromBuffer()** to write and restore from buffer positions. The parameter in the method **ReadFromBuffer(float seconds)** should be same as parameter in the **Rewind(float seconds)** method as shown below, so simply pass it to your method.

*Note: From version 3.0 of this plugin, there is additional method **ReadFromBuffer(float seconds, out bool wasLastAccessedIndexSame),** it exists for performance reasons*, as in certain cases, the values do not need to be set again if same values has been set the frame before. This method methods gives you info about that. Example is in RewindAbstract audio implementation. Use in caution!

Now define what you really want to track in **Track()** method override. You can tell it to track your own variables also with combination of already implemented tracking solutions (eg. Tracking position,rotation,velocity,animator…). This is the place where you should add your implemented tracking method.

```
//In this method define what will be tracked. In our case we want to track already implemented audio tracking,particle tracking + new custom added variable scale tracking
2 references
protected override void Track()
{
    TrackParticles();
    TrackAudio();
    TrackObjectScale();
}
```

Similarly to **Track()** method, you must also fill **Rewind()** method override, where is defined what will be restored on rewind. This is the place where you should add you own implemented restore method.

```
//In this method define, what will be restored on time rewinding. In our case we want to restore Particles, Audio and custom scale tracking
3 references
protected override void Rewind(float seconds)
{
    RestoreParticles(seconds);
    RestoreAudio(seconds);
    RestoreObjectScale(seconds);
}
```

This is all that is required, now you can add your custom tracking script as a component to the object you want to track and start rewinding.

General mindset

Time rewind system is implemented with **RewindManager.cs,** that acts as main controller of all rewinds and scripts that implement **RewindAbstract.cs**. Each tracked object that you want to track must contain script component, that inherits from **RewindAbstract.cs** class.

Tracking and rewinding is implemented with help of FixedUpdate(). I was experimenting with tracking intensity that you could set thru variable, but at the end i decided to stick with certainty that FixedUpdate() provides. Maybe in the future if requested i might look into it again, but for now all implementation is hardcoded with FixedUpdate() so it is stable.

Try to keep in mind, that if you want to add track and rewind functionality to the object that is regularly being changed in Update() method, it might result in slight synchronization problems, due to FixedUpdate() usually not catching with Update(). In rewind related stuff try to use FixedUpdate() instead of Update() whenever possible to completely avoid these issues. One simple way how to fix these problems if you need to use Update() method is to use **IsBeingRewinded** property from **RewindManager.cs**, and temporarily disabling Update() method while this property is true. Example of this simple solution is shown in **ParticleTimer.cs** (script is located in ExampleScenes/ScriptsForScene), where otherwise Update() and FixedUpdate() would fight against each other (although the problem in the script could be also solved just by using FixedUpdate()).

The reason why Time Rewinding is not implemented with the Update() method by default is, that it would add total unpredictability when rewinding by certain amount seconds. Also in my opinion, it is usually unecessary luxury to track values more often than in FixedUpdate() and it would also result in less overall performance.

Regarding the time settings in Unity. The FixedTimestep can only be modified prior to application launch, as this impacts the physics precision and rewinding smoothness.

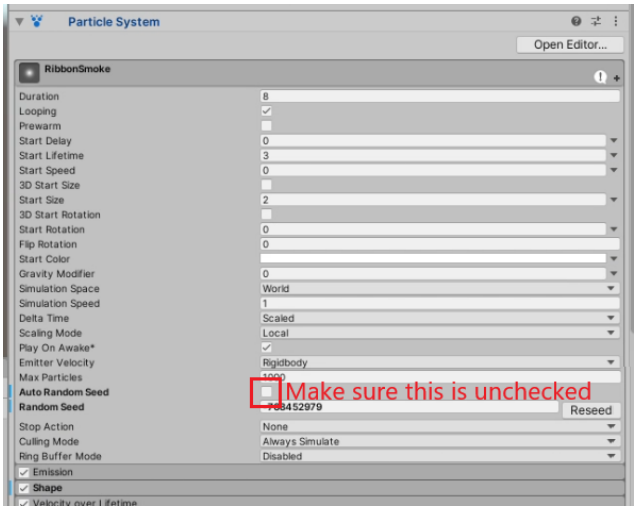
Note: From version 2.1+ of this library, the Time.timeScale can be altered at any point, even during runtime. However, the Rewind system overall expects the timeScale to be 1. Therefore, if you change the timeScale, the amount of seconds you can rewind will also change (eg: if you initially set to track 12 seconds and you have timeScale=2, you will be able to only rewind 6 realtime seconds). You can also classically pause or resume the whole game by setting the Time.timeScale to 0 or to 1 respectively.

Note 2: When entering the rewind phase, the timeScale needs to be greater than 0. If it is not, even rewind process will be frozen, due to FixedUpdate() function being completely stopped.

Particles System Rewinds

The only option that i have found to rewind particle systems is to use Unity Particle.Simulate() method, which comes with huge performance hit if the particle system is running for long time especially in loop. Because of that i added limiter option for particle system tracking, that will reset the tracking after limit is hit. This will save a ton of performance in long particle systems. Although limiting particle system means that the rewind for long particle effects will not be so smooth everytime, it can be usually set up well enough to not see obvious jump transition. Short particle systems, dont need limiter and will rewind smoothly.

Important thing you must make sure you do, is to uncheck Auto Random Seed in ParticleSystem settings on every Particle System you want to rewind.



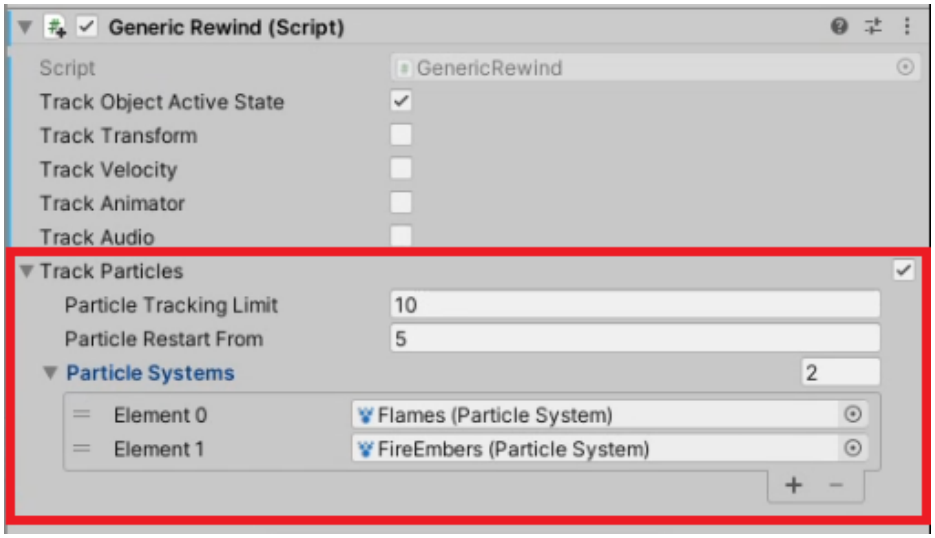
So the Particle.Simulate() is simulated with correct seed. You can set the Random Seed to whatever you like, just make sure you dont change it during runtime.

Before you start rewinding particles, you need to set up **Particle Settings** in editor.

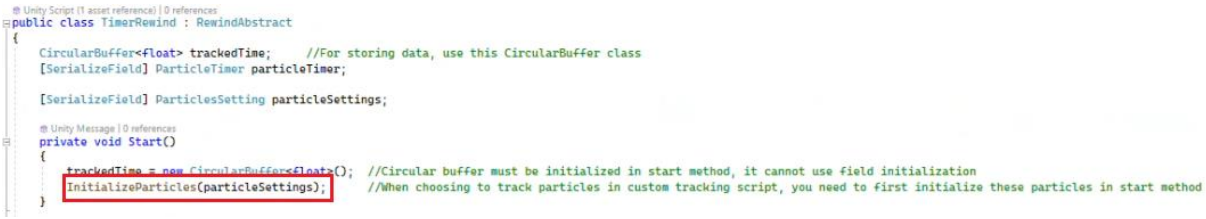
> **Particle Restart From** shown on picture below is variable containing the time the Particle System will reset to after the Particle Tracking Limit is hit. Try different values in this variable, to reach the smoothest possible transitions.
>
> If your particle system is short enough (eg. 5-10 seconds)**,** set **Particle Tracking Limit** to arbitraty number above your Particle System duration.

If your Particle System is long, experiment with different values that will give you acceptable performance.

Particle Systems List should contain the systems you will want to rewind.



If you also want to setup your own custom tracker with Particle Tracking (similarly shown in **TimerRewind.cs** example), you must call **InitializeParticles()** in Start method of your own custom tracker. Initialization of particles is shown on image below.

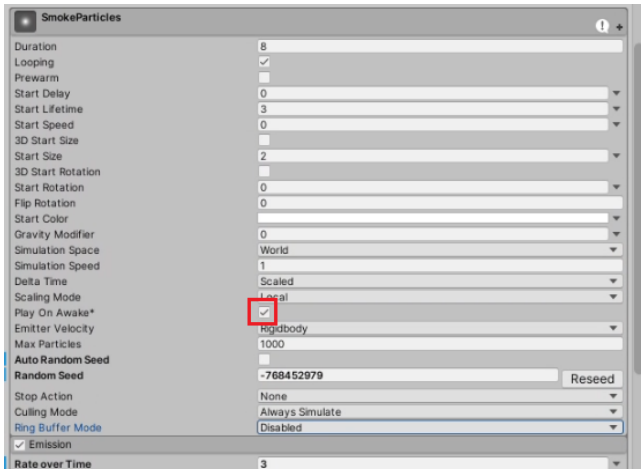


One limitation of rewinding particles is, that it doesnt track and rewind changes that you make on particle system modules at runtime (eg. changing emission rate from code). Particle rewinds rely solely on ParticleSystem.Simulate() method. I considered adding tracking to some important modules, but there are simply too many variables to track and it would be bloated.

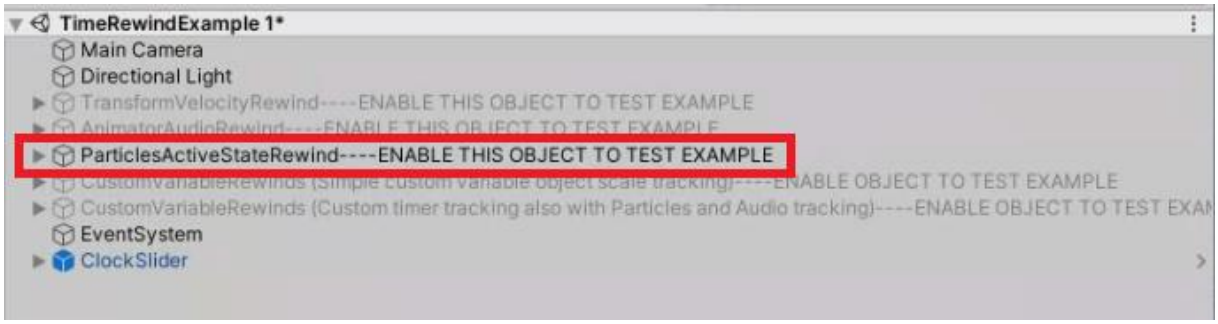
Even if particle system modules were tracked, rewinding would not really work accurately. For example, it would not rewind correctly when in Particle System you would suddenly change emission to 0. When doing that emission change in non-rewind mode, you would see that particles just dont dissapear all of sudden. The already existing particles will still be alive and they would dissapear only when their time runs out or are killed by something else.

On the other hand when rewinding, particles would suddenly dissapear, because ParticleSystem.Simulate() works with current attributes on particle system and cannot be simulated with attributes changing over time.

That is why it is best to have particle systems prepared before hand and just launch them at runtime. The best method with least performance impact is to set them to to Play on Awake and then enabling/disabling its parent GameObject in runtime when you want to turn them on/off.

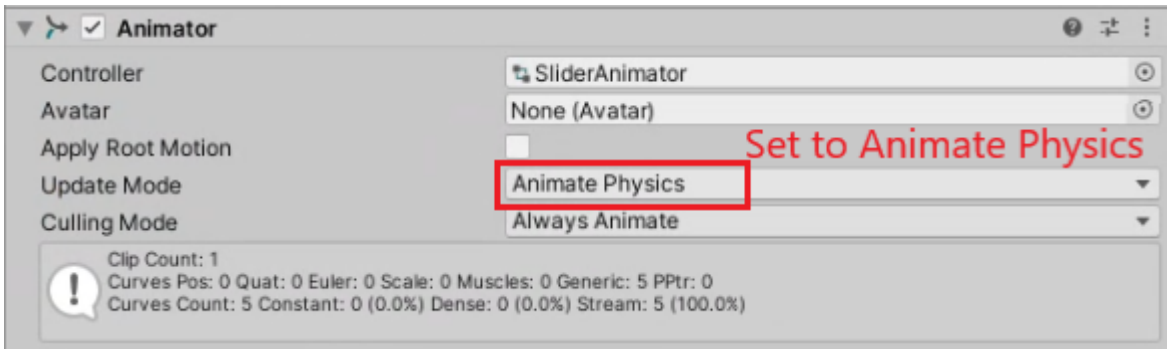


You can combine it with tracking the active state of Particle system parent by tracking its object active state. Special example with enabling and disabling particle system is prepared in this showcase in first demoscene.



Animator rewinds

When rewinding animators, i recommend changing its Update mode to **Animate Physics** (FixedUpdate). On next image is what i mean by that.



By default, animator is set to Normal settings (it uses regular Update() loop) and it is usually not a big deal. You might also require animator to be tied to Update() method for various reasons. But for other things, that require absolute synchronization and precise tracking of animator states, it could lead to some problems due to rewind system working primarily with FixedUpdate().

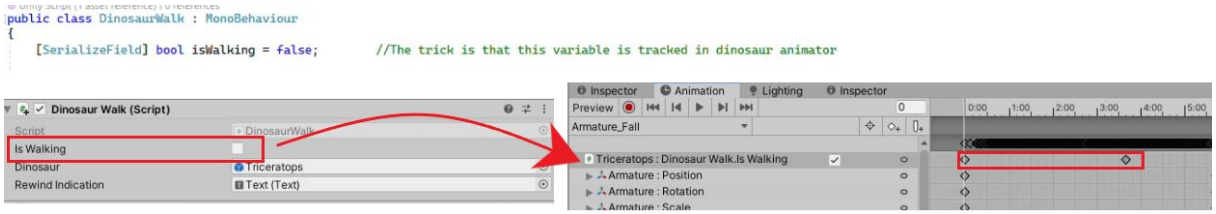
If you dont need Animator to run specifically in Update() method, change it to **Animate Physics** settings as shown above, otherwise test what works for you and if it is not doing problems leaving it in Normal setting is probably fine.

Note: From version of 3.0 of this plugin, Animator transitions are also supported. Avoid transition offset though, as this is still not supported yet.

Another topic is **Animation events**. Which can be really usefull especially when used in combination with rewind system. Sometimes rewinding stuff like for example Coroutines can be a real issue due to how Coroutines really work and that the state of the Coroutine cannot really be tracked. One solution that can replace Coroutines in some cases is appropriate use of Animation events in Animator. Be carefull though**,** Animation event will not get triggered during the rewind, because timescale of the animator is set to 0 and Animator is effectively paused. But, when you restore normal game flow, and there is Animator event infront, it will surely get triggered.

Animation events are of course not solution for everything, for example lets say you wanted to track state of some variable that you are changing in Animation event. Because Animation event will not get triggered when you rewind back, for some time, the state of that variable would not be correct. For these cases

it is better to track the state of the variable directly in the animation. The example of this method is shown below (demoscene with walking dinosaur).



With this method, you will make sure the variable has correct value all the time, even on rewinds.

Note: From 3.0 version of the plugin, all animator parameters (Triggers, Bools, ...) are also tracked.

Audio Rewinds

Audio rewinds dont have any known caviats, additionaly the active state of the audio source is also tracked (enabled/disabled). So you could set the Audio source to play on awake and then when you want to play the audio, you can simply enable the Audio Source on the object, so it is not staying active all the time.

Note: From 3.0 version of the plugin, the pitch and volume is also automatically tracked.

Rewinding spawned objects *(From version 3.0)*

Sometimes you will probably want to track and rewind certain objects that you spawn in the middle of the game. Because the **RewindManager.cs** registers all objects at the game start, these spawned objects would normally not be tracked.

One way to deal with this problem is to use pooled objects that are already spawned at the start of the game and reuse these when you need to. If you cant however do that, there is also option to add these object to tracking loop manually.

Example how to do that is prepared in **TimeRewindExample 3**, additionaly the process is shown on next picture.

```
RewindAbstract someOjectToSpawn = Instantiate(_projectilePrefab, _spawnPoint.position, Quaternion.identity);
RewindManager.Instance.AddObjectForTracking(someOjectToSpawn, RewindManager.OutOfBoundsBehaviour.DisableDestroy);
```

```
void RewindManager.AddObjectForTracking(RewindAbstract objectToRewind, RewindManager.OutOfBoundsBehaviour outOfBoundsBehaviour)
Lets you add object to track pool even when game is already running.
```

After instantiation, you simply call **AddObjectForTracking(...)** method, where you pass required parameters and choose out OutOfBoundsBehaviour.

The OutOfBounds behaviour is the behaviour of the object when you rewind before the object was attached to the tracking loop (typically before object creation). You have two options that are described on next picture.

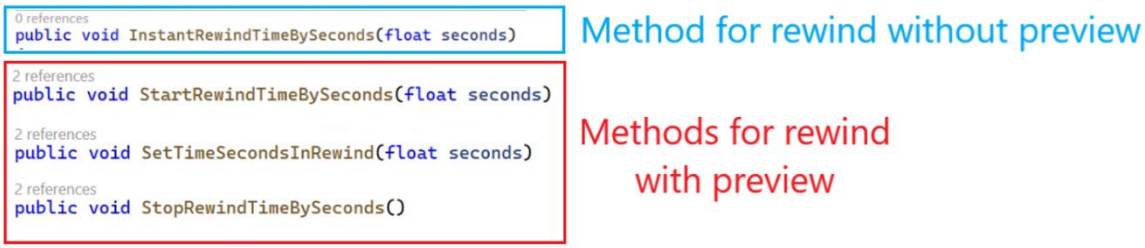
```
public enum OutOfBoundsBehaviour
{
    /// <summary>
    /// Only disables the object when out of bounds
    /// </summary>
    Disable,
    /// <summary>
    /// Disables the object when out of bounds. Then, if the game resumes in out of bound state, the object is destroyed
    /// </summary>
    DisableDestroy
}
```

You will usually want to use DisableDestroy behaviour as it destroys the object automatically if it wasnt attached to the tracker at the time when you resume the game. If you are using pooling technique with your spawned objects, you typically want to use just Disable option, to reuse the objects later in the game.

Rewinding time thru code (Custom rewind inputs)

RewindBySlider.cs and **RewindByKeyPress.cs** are prepared for you, you can use it straight away or customize it for your needs. However, if you want to rewind time by different input logic, you can write it yourself. The only thing that is needed, is that you call appropriate methods from **RewindManager.cs** that are also documented. I also suggest looking into **RewindByKeyPress.cs**, which contains very easy to understand and simple implementation. **RewindManager.cs** provides two ways how to rewind time.

There are 4 main methods in **RewindManager.cs** that are important for you.



First way to rewind the time is with **InstantRewindTimeBySeconds()** that is used for one time instant rewind, where you dont need rewind previews and you know you just want to rewind time by specified amount of seconds.

Second way of rewinding is with rewind previews (this way of doing rewinds is also shown in demo scene examples). Start rewinding time by calling **StartRewindTimeBySeconds()** method. After calling this method, all objects will be instantly rewinded by specified seconds and tracked attributes will be freezed (this effectively pauses the game for tracked attributes and shows you the preview).

To update the preview, call **SetTimeSecondsInRewind()** method to update the time of the preview.

After you are satisfied with currently shown preview and you want to resume the game, you **must** call **StopRewindTimeBySeconds()** method, which will stop the rewind and all rewinded objects will again start behaving as usually from this point.

Shader rewinds (moving shaders)

It is also possible to rewind custom moving shaders. This rewinding requires a little bit of setup, and the result can be seen in **TimeRewindExample 1** in the ShaderRewinds example**.** The main idea here is to track certain variable/variables that your shader uses to create the moving parts. In the example scene, simple stripe-flowing shader is showcased that uses *time variable* to create the movement. I recommend looking into the **ShaderRewind.cs** where the setup is shown. You can use the similar technique to pass variables to your shader.