

# **Unity Time Rewinder documentation** (ver 1.1)

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 .....	1
Start using Time Rewinder in your own project.....	2
Adding custom tracker with custom variables tracking .....	3
General mindset.....	5
Particles System Rewinds.....	6
Animator rewinds.....	9
Audio Rewinds.....	10
Rewinding time thru code (Custom rewind inputs) .....	10

## **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 TimeRewinderImplementation folder is required, but i highly suggest checking the demoscene with prepared examples.

## Start using Time Rewinder in your own project

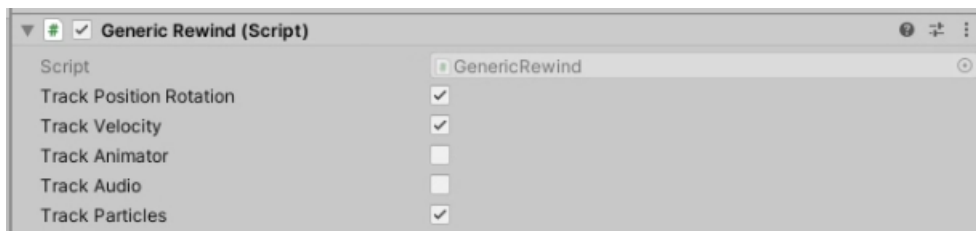
To start using Time Rewinder, each scene must contain **RewindManager.cs** script, that is essential. In [TimeRewinderImplementation/Scripts/RewindInputs](#) folder, there are prepared two scripts. **RewindBySlider.cs** and **RewindByKeyPress.cs**. As names suggest, both scripts let you rewind the time 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 demoscenes [TimeRewindExample 1](#) and [TimeRewindExample 2](#) that are located in [ExampleScene](#) folder.

**RewindManager.cs** contains variable of how many seconds should be tracked, you can change this value, although provided Slider is setup with animations to 12 seconds by default, so you would have to also update the slider settings. In case you want to use **RewindByKeyPress.cs**, no additional setting similar to Slider is required and you can simply change the value in the script as you like.

```
/// <summary>
/// Variable defining how much into the past should be tracked, after set limit is hit, old values will be overwritten in circular buffer
/// </summary>
public readonly float howManySecondsToTrack = 12;
```

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 just mentioned). 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.

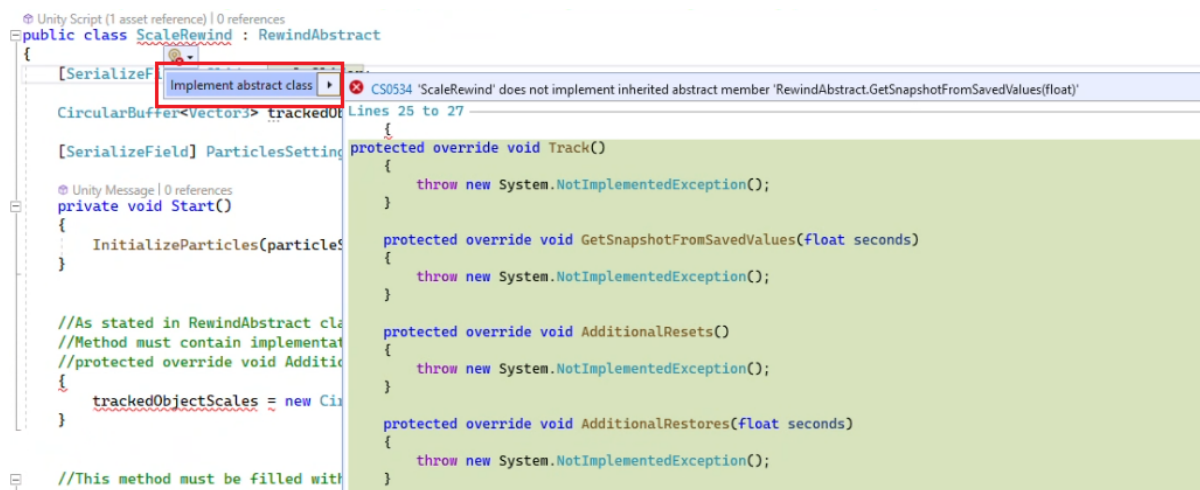


Currently straight from the box you can track object positions, rotations, rigidbody velocity, animator states, audiosource states (also audio in mixer groups) and particle effects. Adding custom variable is easy, and i will show you how to do it in next section. To start rewinding, play the scene. If you 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

```
//This script is showing setup of simple one custom variable tracking
//Unity Script (1 asset reference) | 0 references
public class ScaleRewind : RewindAbstract
{
    [SerializeField] Slider scaleSlider;

    CircularBuffer<Vector3> trackedObjectScales; //For storing data, use this CircularBuffer class
}
```

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

```
//Unity Message | 0 references
private void Start()
{
    trackedObjectScales = new CircularBuffer<Vector3>(); //Circular buffer must be initialized in start method, it cannot use field initialization
}
```

Now you can implement what values will be tracked and restored. I advise you to add 2 own 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. Use **CircularBuffer.WriteLastValue()** and **CircularBuffer.ReadFromBuffer()** to write and restore from buffer positions

```
// 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 additionally restore slider value to match the object scale
    scaleSlider.value = transform.localScale.x;
}
```

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. Just be careful not to override any method from RewindAbstract in your tracker implementation, rewinding would then not work correctly.

In previous tool version, you also had to implement AdditionalReset() and AdditionalRestore(), but workflow is now simplified and these methods are no longer needed.

## General mindset

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

Being component of specified object also means, that tracking cannot be done, when object you want to track is disabled. So while rewinding time, do not disable game objects you want to track. Use other techniques as for example moving the object from visible area or disable its mesh renderer. This rewind mindset applies to all kinds of tracking, except for Particles tracking, where you can disable parent game object of specified particle systems. Particle system tracking is implemented slightly differently from other types of tracking, due to internal Unity reasons. Particle system caviats will be described in next chapter.

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** bool 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 unnecessary luxury to track values more often than in FixedUpdate() and it would also result in less overall performance.

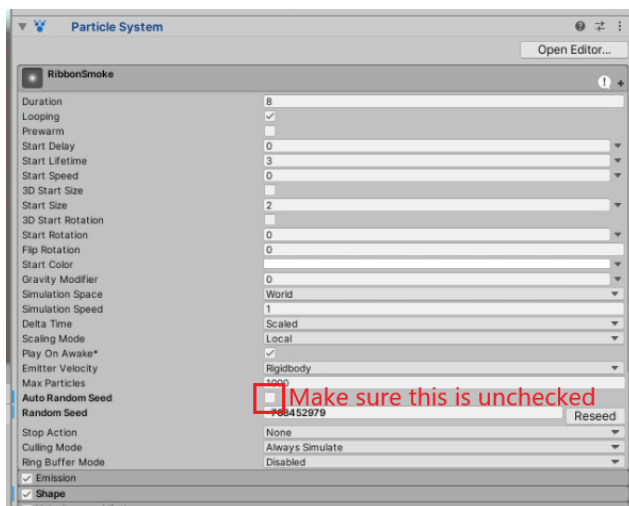
Note that you can change Fixed Timestep or Time.timeScale in Unity Time-settings preferences, but i advise you being carefull with that and have valid reasons before you start changing these values.

**Be warned though:** You cannot change these settings in runtime (Fixed Timestep or Time.timeScale). Changing it would lead to inconsistent values in circular buffer that differ in time, and because rewind system takes seconds as parameter for rewind, it would do a lot of problems.

## 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.

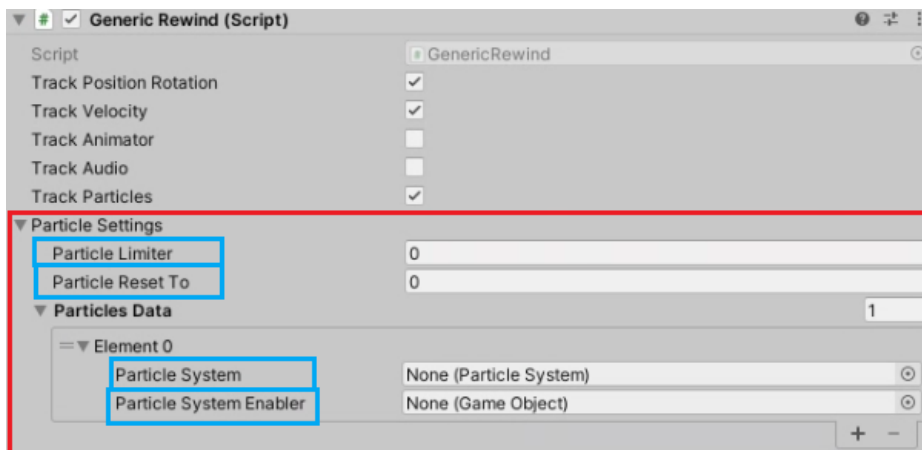
Before you start rewinding particles, you need to set up **Particle Settings**. When using **GenericRewind.cs** you only need to set it up in Unity Editor.

**Particle Reset To** shown on picture below is variable containing the time the Particle System will reset to after the Particle Limiter 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 Limiter** to arbitrary number above your Particle System duration. If your Particle System is long, experiment with different values that will give you acceptable performance.

**Particle System enabler** also shown below is here to track particle active states, because Particle Systems are designed to be enabled or disabled during tracking. Particle System enabler should be GameObject of single Particle System that you want to track, or in case of many many Particle Systems and Sub-Particle Systems, it should be their Parent GameObject (also note that every sub Particle System must be included as one element of Particles Data).

**Particle System** field should contain the Particle System 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.

```
using UnityEngine;
using System.Collections;

public class TimerRewind : RewindAbstract
{
    [SerializeField] ParticleTimer particleTimer;
    [SerializeField] ParticlesSetting particleSettings;

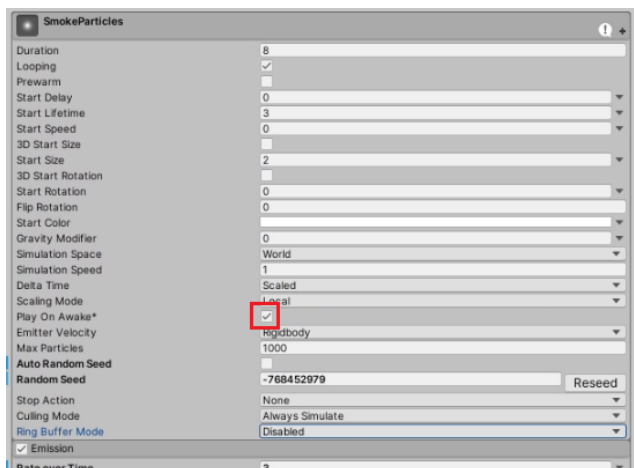
    private void Start()
    {
        trackedTime = new CircularBuffer<float>(); //Circular buffer must be initialized in start method, it cannot use field initialization
        InitializeParticles(particleSettings); //When choosing to track particles in custom tracking script, you need to first initialize these particles in start method
    }
}
```

One limitation of rewinding particles is, that it doesn't 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.

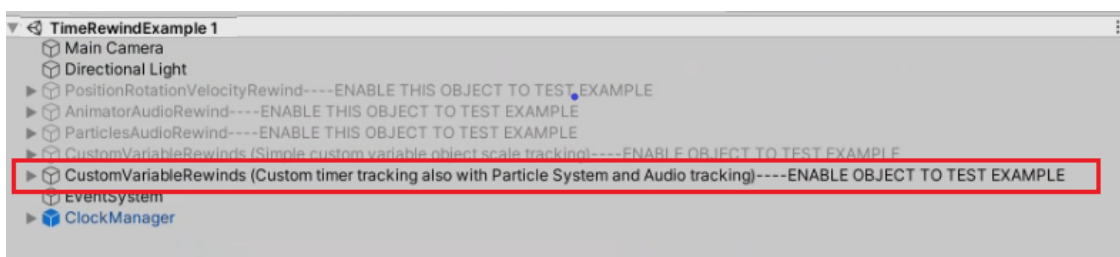
Even if particle system modules were tracked, rewinding would not really work accurately. For example, it would not rewind correctly when in Particle System you would suddenly change emission to 0. When doing that emission change in non-rewind mode, you would see that particles just don't disappear all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

On the other hand when rewinding, particles would suddenly disappear, 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 `Play On Awake` and then enabling/disabling its parent `GameObject` in runtime when you want to turn them on/off. That is the reason why **Particle System enabler** mentioned previously is important here.



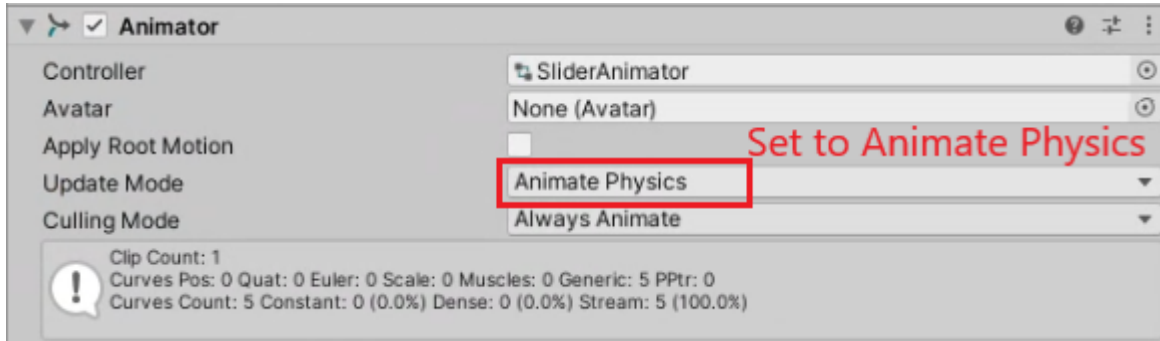
Special example with enabling and disabling particle system is prepared in last showcase in first demoscene.





## Animator rewinds

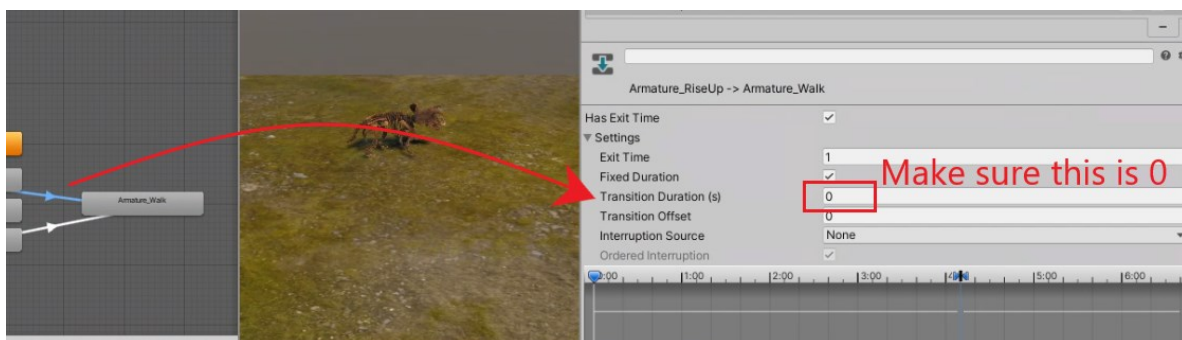
When rewinding animators, i recommend changing its Update mode to **Animate Physics** (FixedUpdate). On next image is what i mean by that.



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.

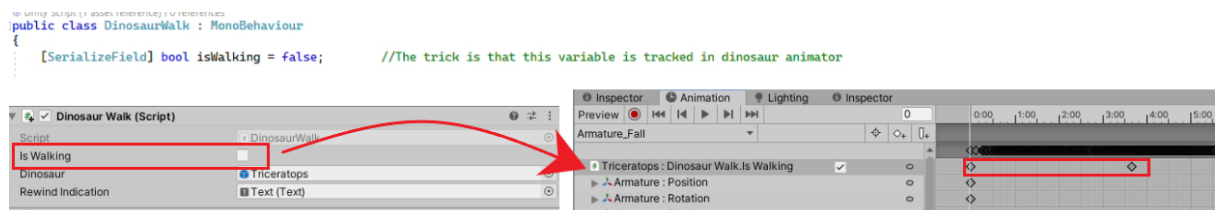
One known problem of Animator rewinds is **Transition duration**. When transition is set to value that is greater than 0, on edge cases it will not rewind the animator correctly and animation will get stuck. Unfortunately i havent found a solution for this yet, so **Transition Durations** must be disabled for now.



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 careful 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 in front, it will surely get triggered.

Animation events are of course not solution for everything, for example let's 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.

## Audio Rewinds

Audio rewinds don't have any known caveats, additionally 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.

## Rewinding time thru code (Custom rewind inputs)

**RewindBySlider.cs** and **RewindByKeyPress.cs** are prepared for you, you can use it, customize it and visually enhance it in your game. 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.

0 references  
`public void RewindTimeBySeconds(float seconds)`

Method for rewind without preview

2 references  
`public void StartRewindTimeBySeconds(float seconds)`

2 references  
`public void SetTimeSecondsInRewind(float seconds)`

2 references  
`public void StopRewindTimeBySeconds()`

Methods for rewind  
with preview

First way to rewind the time is with **RewindTimeBySeconds()** that is used for instant rewind, where you don't 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.

To update the preview when rewinding, use **SetTimeSecondsInRewind()** method to update the time of the preview you want to see.

After you are satisfied with currently shown preview and you want to return time to the shown preview, you must call **StopRewindTimeBySeconds()** method, which will stop the rewind and all stats shown in preview will be set to current objects, so game can continue from this point.