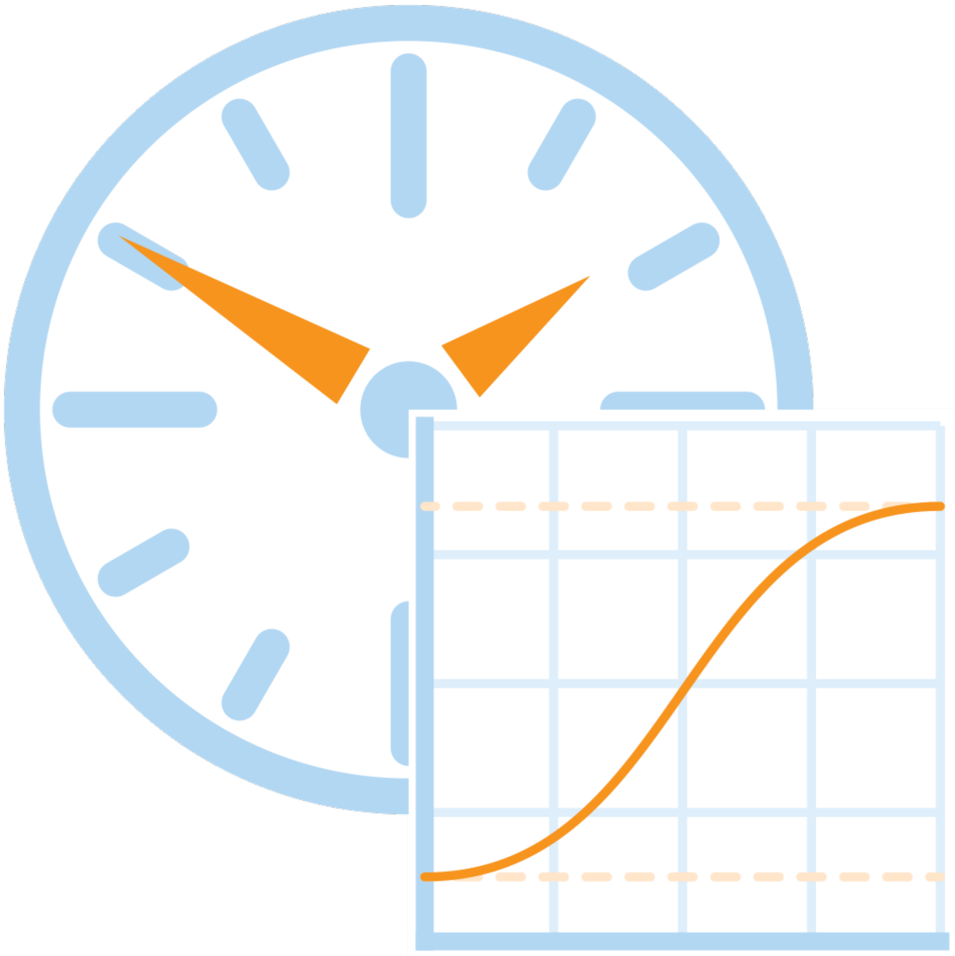
Time and Terp

User Manual 1.0



Contents

Overview / Quick Start

Time and Terp is a collection of lightweight, interconnected C# utility classes that provide fire-and-forget, simple timing and scaled interpolation (easing) functionality for your Unity projects. The package contains five core elements:

- The **GameTime** class provides an easy way to alter the timescale Unity operates on while keeping the physics timestep adjusted appropriately. It provides time deltas for both the scaled and real time. GameTime also provides the Update() calls from Unity that the rest of the system needs to run properly.
- The **Timer** class allows you to perform simple scheduling of tasks and actions on a one-shot or recurring basis. For example, performing an action a set time after a button has been pressed, or performing the same action every few seconds.
- The **Interpolator** class provides interpolation between two values over a set time, similar to the familiar Mathf.lerp() method. Unlike lerp(), Interpolator provides an assortment of scaling functions beyond linear scaling for interpolation, and operates without further management after it has been created.
- The **Pool** class is used internally to handle allocations for the Timers and Interpolators, providing a recycled object pool to reduce garbage overhead.
- An **Example** scene/script to help illustrate usage.

Basic usage is quite simple and follows these steps:

1. Include the FIL.Utilities namespace in the script you plan to use T&T for. This is done with the following line of code, typically at the top of the file:

   ```
   using FIL.Utilities;
   ```

2. Call Timer.Create() or Interpolator.Create(), passing a simple lambda expression, like so:

   ```
   // DoSomething() is some arbitrary method we've declared elsewhere-- it's what is
   going to get called when the Timer fires.
   Timer.Create(timeToWait, (Timer timer) => { DoSomething(); });

   // variableToInterp is a variable we're using somewhere else-- a position, an
   alpha, whatever. The assignment code is called each time the Interpolator
   updates.
   Interpolator.Create(startValue, endValue, duration, (Interpolator interp) => {
   variableToInterp = interp.Value; }, null);
   ```

And that's it! The necessary supporting script objects will automatically create themselves and update any Timers and Interpolators you're currently running, and the Timers and Interpolators you create will clean themselves up when they're done. You don't have to worry about it further. Naturally, this is the simplest use of the tools—to see more advanced and interesting usage, refer to the appropriate sections of this document.

Time and Terp Classes

This section of the manual provides in-depth usage information on the classes that make up the Time and Terp package. Here you'll find a discussion of the various member properties and methods, as well as some advanced usage examples. Those who wish to get the most out of the package, this section is for you. We will be assuming a moderate level of C# knowledge from this point out.

GameTime Class

The GameTime class is a Monobehaviour that provides two major functions: First and foremost, it hooks into Unity's regular Update cycle and provides the Timer and Interpolator pools with a regular update loop. Second, it maintains a secondary time delta based on realtime seconds independent of Unity's Time.deltaTime. This second function is important because Unity's time delta can be scaled—that is, made to run faster or slower—by manipulating Time.timeScale (for more on how this extra delta is useful, see RealTimeDelta). As a convenience feature, GameTime provides a facility to alter Time.timeScale and automatically adjust the Physics timestep to be consistent with the altered scale.

The GameTime class is set up with the Singleton pattern, so there is only ever one instance at a time. It can be manually added to the object of your choice in your scene (for instance, if you like to keep all your Singletons on a globals object), but this step is not necessary. If the GameTime instance does not exist the first time it is called, GameTime will create an appropriate GameObject and attach itself. Interpolators and Timers both call the instance when they are created, making this process (essentially) totally automatic.

GameTime Properties

Instance

(static GameTime) Returns the GameTime instance, creates a new one if it doesn't yet exist.

All of the GameTime public properties are static, so you will almost never have to call this property yourself. However, if you want to force the GameTime instance to bootstrap itself at a particular time, this would be the easiest way to do so. Otherwise, this is really only useful to the package internals.

GameTimeDelta

(static float) Returns the scaled time delta in seconds.

This, at present, simply returns the same value as Time.deltaTime. It's included here in wrapped form for consistency, and to foster the idea that this delta represents time flow within the game itself and encourage good practices.

RealTimeDelta
(static float) Returns the real-world time delta in seconds.

This will always return the time since the last frame in actual real seconds, unaffected by the timescale value. This allows you to decouple certain aspects of your code which should always operate normally from Time.deltatime. For instance, using RealTimeDelta for UI functions will keep the UI responsive and functional, even if you have modified the speed the rest of the game runs at. This approach also prevents certain kinds of unwanted behavior—for example, if you were to slow down the timescale and then speed it back up after two seconds, if you base this behavior on Time.deltaTime, it will not behave as expected. RealTimeDelta is provided to prevent this kind of problem by providing a secondary delta that always operates at a 1:1 scale.

Timers and Interpolators can be created using either realtime or scaled time behavior for maximum flexibility.

EnableInterpolators
(static bool) Enables or disables the Update() calls to the Interpolator pool.

Used to “turn off” all Interpolators. This property is also exposed to the Inspector if you assign the script to your own object in the scene. Note that this will NOT ‘clean up’ existing Interpolators—they will simply pause. Intended use is to turn off the Update() calls if you only need one class or the other for some reason.

EnableTimers
(static bool) Enables or disables the Update() calls to the Timer pool.

Used to “turn off” all Timers. This property is also exposed to the Inspector if you assign the script to your own object in the scene. Note that this will NOT ‘clean up’ existing Timers—they will simply pause. Intended use is to turn off the Update() calls if you only need one class or the other for some reason.

TimeScale
(static float) Returns or sets the time scale to operate at. “Normal” scale is 1.0.

A convenience property to change the scale Unity’s time operates at. Unlike setting Time.timeScale, using this will also adjust the physics timestep to be consistent with the new value, keeping the simulation in sync with the rest of the game. In short, setting this to 0.5 slows the game to half-speed and makes sure the Physics simulation also updates at a sensible rate for that speed.

Interpolator Class

Interpolators are used to smoothly transition a float value from a starting value to an ending value over a period of time using a function to control how the value moves from start to finish. This is conceptually very similar to the way Mathf.lerp() is most often used. Interpolation is often referred to as *tweening* or *easing*, though strictly speaking those terms refer to other, related techniques. In addition to linear interpolation, values can be interpolated using scaling functions to further control how the value is moved from start to finish. Common easing functions are provided, and it is relatively simple to provide your own custom function (See InterpolatorScaleDelegate for more information).

Interpolator Properties

Progress

(float) Returns the current progress of the Interpolator.

This is a value in the range of 0-1 which reflects what fraction of the total duration has elapsed. This value is **not** scaled by the current scaling function and will always progress in a linear fashion with time.

Start

(float) The start value to interpolate from.

End

(float) The end value to interpolate to.

Value

(float) The current interpolated value according to the current progress and scaling function.

This is the property of primary interest, as it's the end result of the Interpolator's work. Assign this to whatever you're trying to interpolate in your code.

Realtime

(bool) Returns whether this Interpolator is operating in realtime (true) or scaled time (false).

This allows Interpolators to operate independently of scaled game time when desirable. For instance, you might apply a slow motion effect by changing GameTime.timescale to a low number, and then gradually speed back up to normal over some number of seconds. In order for this to work properly, you would want to set your Interpolator to operate in realtime (otherwise the speed-up effect would occur much slower than intended).

Tag

(Object) Used to attach additional information to the Interpolator.

This is very open-ended, and there isn't a lot to say about it. This gives you an opportunity to attach some extra data of your own choice to the Interpolator that you can then retrieve during the step and completed callbacks.

Interpolator Methods

Create

```
static Interpolator Create(float start, float end, float length,
        Action<Interpolator> step, Action<Interpolator> completed)

static Interpolator Create(float start, float end, float length, bool realtime,
        InterpolatorScaleDelegate scale, InterpolatorEaseDirection direction,
        Action<Interpolator> step, Action<Interpolator> completed)
```

Initializes, starts, and returns an Interpolator instance for use.

Unsurprisingly—as this function is at the core of the Interpolator usage—a proper discussion of the Create() method is going to be one of the more extensive sections of this manual. Calls to Create() are going to form upwards of 90% of your use of the system in most cases, and there is quite a lot you can do here. So let's dive in. The first item to note is that there are two overloads for this method—the first simply defaults scale to InterpolatorScales.Linear and direction to InterpolatorEaseDirection.InOut.

Parameters

`float start` - Defines the initial value that the Interpolator is going to interpolate from.

`float end` - Defines the final value to interpolate toward.

`float length` - Defines the duration of the interpolation, in seconds.

`bool realtime` - Sets whether this Interpolator runs in real time, or scaled time (see Realtime).

`InterpolatorScaleDelegate scale` - Chooses a scaling function to use for this Interpolator's output. This is how you add easing to the interpolated values. The built-in scales are stored in the InterpolatorScales class as static methods. See InterpolatorScaleDelegate for details.

`InterpolatorEaseDirection direction` - Sets a direction for the scaling function to be applied in. See InterpolatorEaseDirection.

`Action<Interpolator> step` - The method to run each time the Interpolator updates. This should be in the form of an Action<Interpolator> (the signature is `void MethodName(Interpolator)`), and can be either a normally-declared method or—more commonly—a lambda expression. Lambda expressions are convenient for any one-off Interpolator use, whereas a declared method approach is more desirable in cases where you'll be using the same step function in many places. Most of the time, this will be passed something very simple, such as `(Interpolator i) => { alpha = i.Value; }` to assign the current value of the Interpolator to the variable being interpolated.

`Action<Interpolator> completed` - The method to run when the Interpolator completes. This is an Action<Interpolator> just like step, so much of the same information applies here. This parameter is typically used for performing cleanup tasks and other processing that is dependent on the Interpolator completing. For example, the DemoUI example script uses this to re-enable the UI button after both Interpolators are finished.

Stop

```
void Stop()
```

Stops the Interpolator completely. In order to use this effectively, you will need to store the reference returned by Interpolator.Create() in some fashion.

Update

```
static void Update()
```

Triggers all Interpolators to update themselves based on the time deltas from GameTime. Call this yourself at your own peril—it is neither recommended nor supported.

InterpolatorEaseDirection

Enumerations of this type control the *direction* of the applied easing function. For functions that support it, different InterpolatorEaseDirection values will yield different results in the output:

In: The Interpolator will "ease in" to the motion or other change—that is, the function will be applied to the start of the Interpolator's range and will come to a sudden stop at the end value.

Out: The Interpolator will "ease out" of the change—the changes will begin abruptly and the function's effects will be seen at the end of the Interpolator's range.

InOut: The Interpolator will both "ease in" and "ease out" of the motion or change, with the function being applied near both the start and end values.

While we call this "easing" and often times it does resemble smoothly accelerating in and out of changes, the precise effects are dependent on the InterpolatorScaleDelegate performing the easing.

InterpolatorScaleDelegate

The InterpolatorScaleDelegates are in some ways the heart of the Interpolator class, as they provide the functions that modify the intermediate progress values into the final scaled output you see in Interpolator.Value. Time And Temp comes with many of the most common easing functions already defined for your use, stored in the InterpolatorScales class:

Linear

This function produces no scaling, allowing the value to change at a constant rate with the progress of the Interpolator. Functionally, this works the same way lerp() functions do and produces the same result.

Quadratic/Cubic/Quartic/Quintic/Exponential

These very similar functions each use an exponential approach to easing, resulting in the output value accelerating into and out of the end and start values, respectively. As the functions progress from Quadratic to Exponential, the acceleration effect becomes increasingly pronounced.

Circular
An easing function based on a circular curve. This produces a motion very similar to the above exponentials, with a sharp rate increase on the end opposite the easing direction (or the middle in the case of InOut).

Spring
This function overshoots the target slightly (or winds up for the motion, depending on ease direction), producing an effect not unlike a spring.

Extending the Interpolators with Custom Scales
Adding your own scaling to an Interpolator is both simple and potentially quite difficult. Programmatically, it's quite simple, as all you need to do is define your own function in the form specified by the InterpolatorScaleDelegate definition, which has the following signature:

```
float InterpolatorScaleDelegate(float progress, InterpolatorEaseDirection direction)
```

progress is a value in the range of 0-1 giving the fractional distance along the Interpolator's range which the Interpolator has currently reached. direction gives the requested ease direction for those scaling functions that support scaling direction.

Once you've created your scaling function, simply pass it to Interpolator.Create() in the scale parameter. The difficult part in creating custom scales is determining exactly how to return a continuous range of values in the fashion you wish based on the progress of the Interpolator. Those details are outside the scope of this document or support for this Asset.

Timer Class

The Timer class is for simple scheduling of events and tasks that need to happen at a set point in the future, or on a rigidly-repeating basis. The most common usage is for a one-shot, fire-and-forget scheduling of an event when you know it needs to happen a certain number of seconds from the creation of the timer.

Timer Properties

TickLength

(float) This is the length of time the Timer will run for before triggering its tick action, ie: the Timer duration.

Repeats

(bool) Whether the Timer expires when it finishes or repeats indefinitely.

Timers that repeat will continue running and triggering their action at intervals equal to Timer.TickLength until Timer.Stop() is called. For this reason, it is recommended to store the reference returned by Timer.Create() when creating a repeating Timer, so that you may stop the Timer when necessary. It's also possible to test conditions to stop the Timer from within the tick action, but it's recommended you do this with particular care.

Realtime

(bool) Returns whether the Timer is operating in real or scaled time.

It can be desirable to operate a given Timer in either in-game time (which can be scaled) or real time, depending on circumstances. For instance—you might want to limit a round of gameplay to five real-world minutes, while allowing the speed of gameplay to run in slow or fast motion for various effects. A Timer that is not set to realtime would vary its duration according to the timescale, resulting in a round length much longer or shorter than desired, while a realtime Timer would keep correct time for the entire period.

Tag

(Object) Used to attach additional information to the Timer.

This is very open-ended, and there isn't a lot to say about it. This gives you an opportunity to attach some extra data of your own choice to the Timer that you can then retrieve during the step and completed callbacks.

Timer Methods

Create

```
static Timer Create(float tickLength, Action<Timer> tick, bool realtime = false, bool repeats = false)
```

Initializes, starts, and returns an Interpolator instance for use.

Like Interpolator.Create(), calls to this method form roughly 90% of the interaction most people are going to have with the Timer system, and it functions in much the same way.

Parameters

`float tickLength` - The duration (in seconds) the Timer waits until performing the tick action.

`Action<Timer> tick` - The method to run each time the Timer updates. This should be in the form of an Action<Timer> (the signature is `void MethodName(Timer)`), and can be either a normally-declared method or—more commonly—a lambda expression. Lambda expressions are convenient for one-off Timer use, whereas a declared method approach is more desirable in cases where you'll be using the same step function in many places. The DemoUI example script's StartSpheres() method contains an example of both methods in action.

`bool realtime` - Sets whether this Timer runs in real time, or scaled time (see Realtime). This is an optional parameter and defaults to false.

`bool repeats` - Sets whether this Timer repeats, or completes and then expires. This is an optional parameter and defaults to false.

Stop

```
void Stop()
```

Stops the Timer completely. In order to use this effectively, you will need to store the reference returned by Timer.Create() in some fashion.

Update

```
static void Update()
```

Triggers all Timers to update themselves based on the time deltas from GameTime. Call this yourself at your own peril—it is neither recommended nor supported.

Pool Class

The Pool class is used internally to store and recycle Timers and Interpolators as they are created and stopped or completed. Pools are generic collections and may therefore store any type of class—but as they are intended primarily for internal use, usage is outside the scope of this document and they are not officially supported *for use outside of Time and Temp*. You are, of course, welcome to use Pools anyway if you feel they are useful to you—the code should be sufficiently commented.