

LitMotion Overview



```
LMotion.Create(Vector3.zero, Vector3.one, 2)
    .BindToPosition(transform);
```

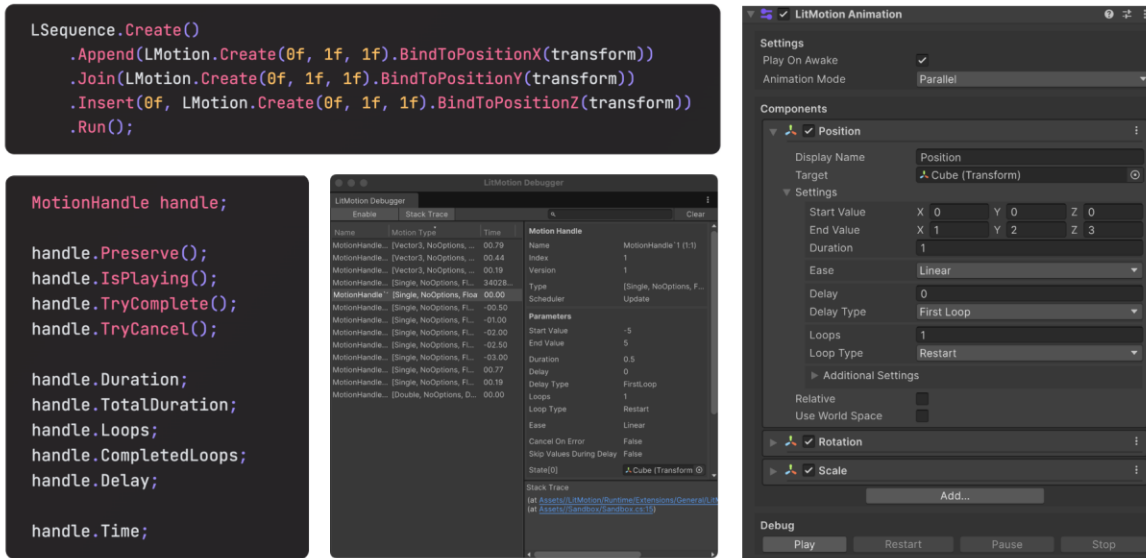
Features

- Animate anything easily and quickly
- Zero allocation
- DOTS-based high-performance implementation
- Completely open source
- Easing, loops and callbacks
- Conversion to Observable<T> with UniRx / R3
- Support async/await with UniTask
- Support Sequence and Visual Editor
- etc...

LitMotion is a high-performance tweening library for Unity. It encompasses a rich set of features to animate components like Transform, along with custom fields and properties, allowing for easy creation of animations.

The biggest feature of LitMotion is its excellent performance. Designed to take full advantage of Unity's latest technology, "DOTS", including the C# Job System and Burst Compiler, LitMotion runs 2-20 times (or more) faster than other tween libraries in various situations, such as creating and executing tweens. In addition, there are no allocations during tween creation.

LitMotion v2.0 is now available!



Additionally, v2 introduces Sequence for combining multiple motions and the LitMotion.Animation package, which allows you to create tween animations directly from the Inspector. With these additions, LitMotion is now as powerful, if not more, than DOTween Pro or PrimeTween in terms of features.

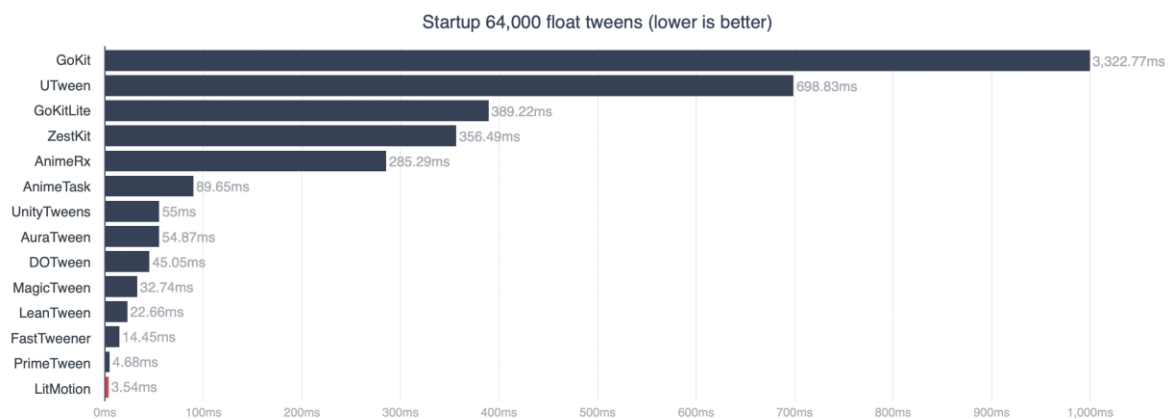
Features

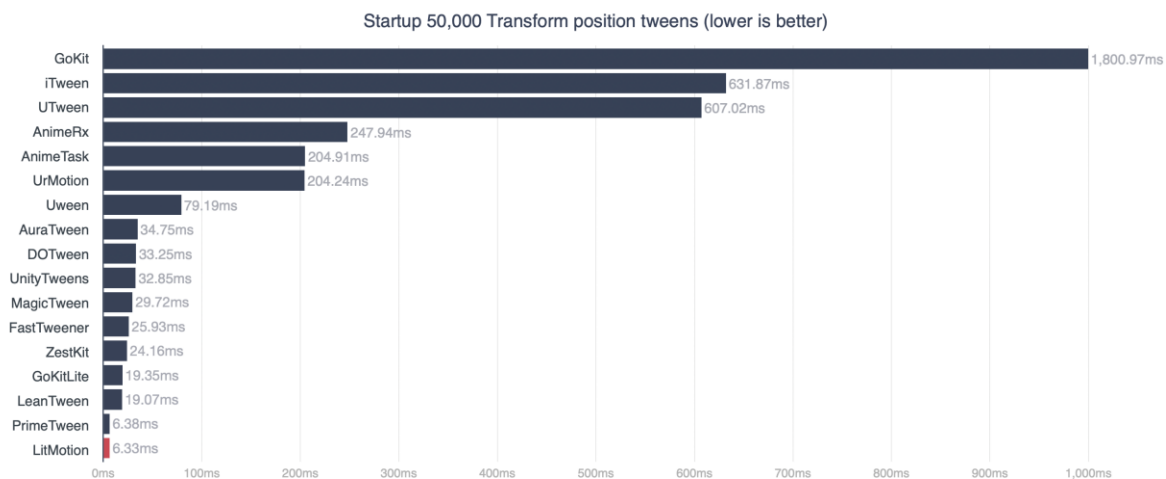
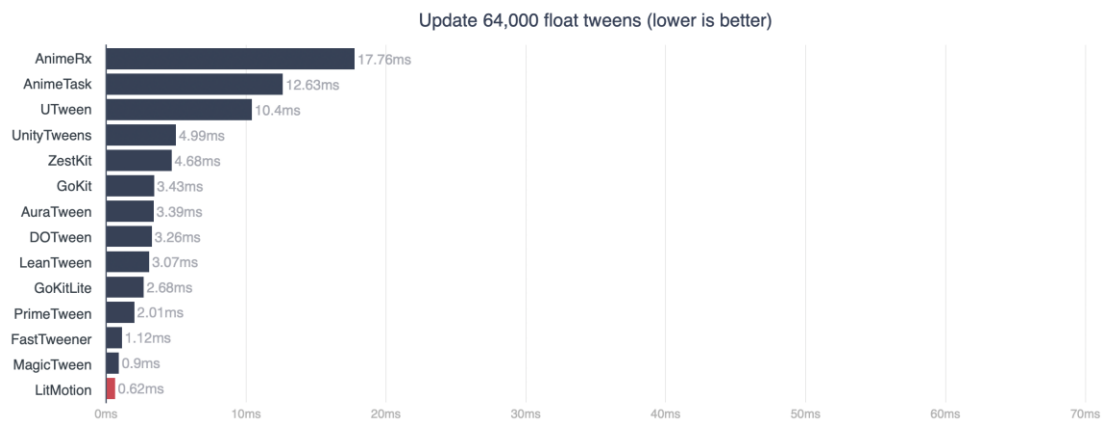
- Animate anything in one line of code.
- Achieves zero allocations with the struct-based design
- Extremely high-performance implementation optimized using DOTS (Data-Oriented Technology Stack)
- Works in both runtime and editor
- Supports complex settings like easing and looping
- Waits for completion using callbacks/coroutines
- Zero allocation text animation Supports TextMesh Pro / UI Toolkit
- Special motions like Punch, Shake, etc.

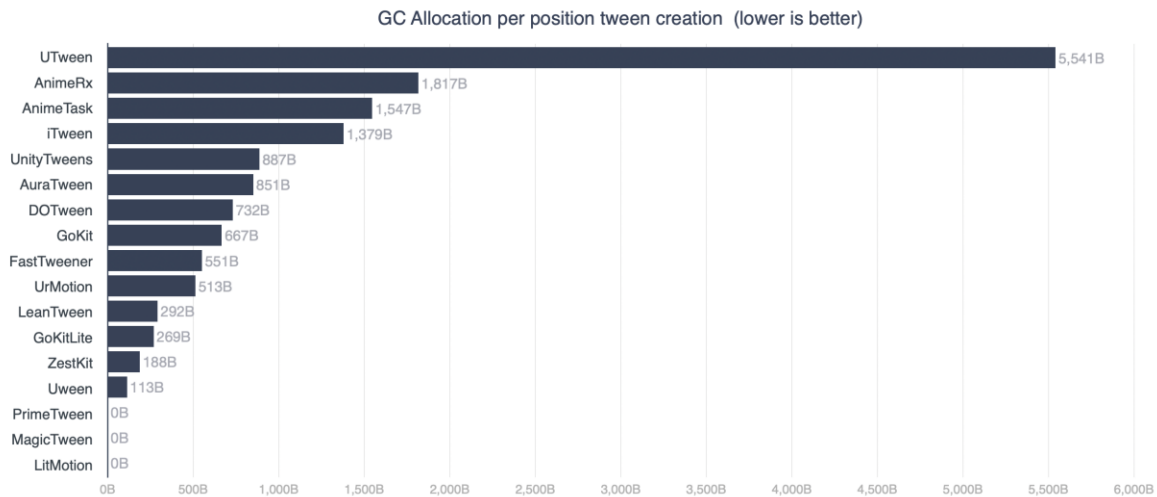
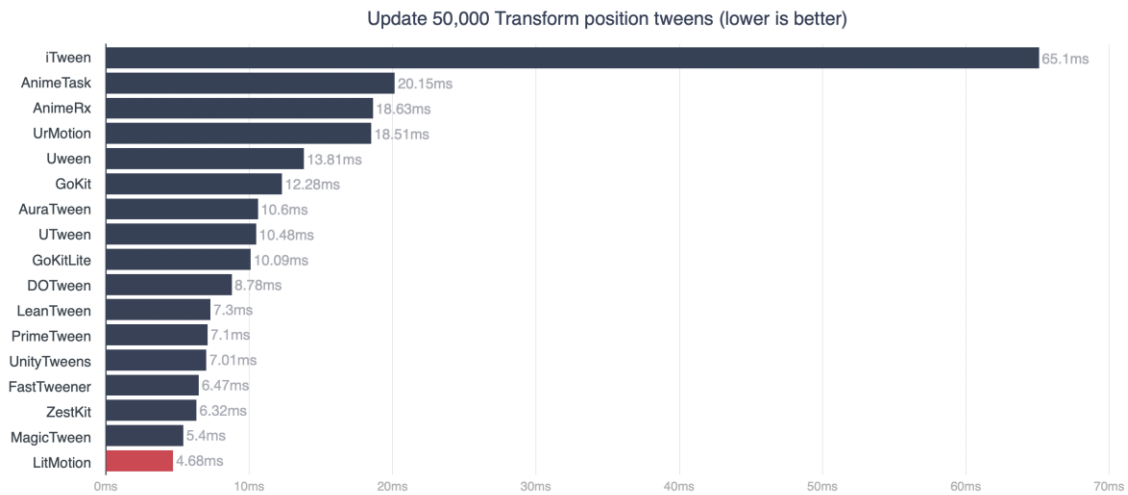
- Conversion to Observable using [UniRx](#) / [R3](#)
- async/await support using [UniTask](#)
- Type extension with IMotionOptions and IMotionAdapter
- Integration with the Inspector via SerializableMotionSettings<T, TOptions>
- Debugging API and LitMotion Debugger window
- Combine animations using LSequence
- Create complex animations directly from the Inspector with the [LitMotion.Animation](#) package

Performance

LitMotion operates faster than any Unity tweening library in both motion creation and execution. Below are the results of benchmarks.







The project and source code used for the benchmarks can be reviewed in [this repository](#).

Comparison with DOTween/Magic Tween

Among other Unity tweening libraries like DOTween and the previously mentioned Magic Tween, LitMotion possesses distinct features in comparison.

- Excellent Performance
 - LitMotion operates about 5 times faster than DOTween.
 - It is about 1.5 times faster than the faster Magic Tween.

- Additionally, there are no allocations during motion creation.
- Curated Features
 - While LitMotion provides sufficient features, it offers fewer functionalities compared to Magic Tween or DOTween. This approach aligns with the library's concept of being "Simple" (although room for extension is available).
- Simple and Flexible API
 - With a natural feel using method chaining, LitMotion enables smooth writing from motion creation to binding.
 - Unlike Magic Tween or DOTween, LitMotion does not include extension methods for components. While I acknowledge the advantages of extension methods, they can sometimes lead to confusion in terms of readability. LitMotion prioritizes API simplicity and unifies the entry point into the LMotion class.

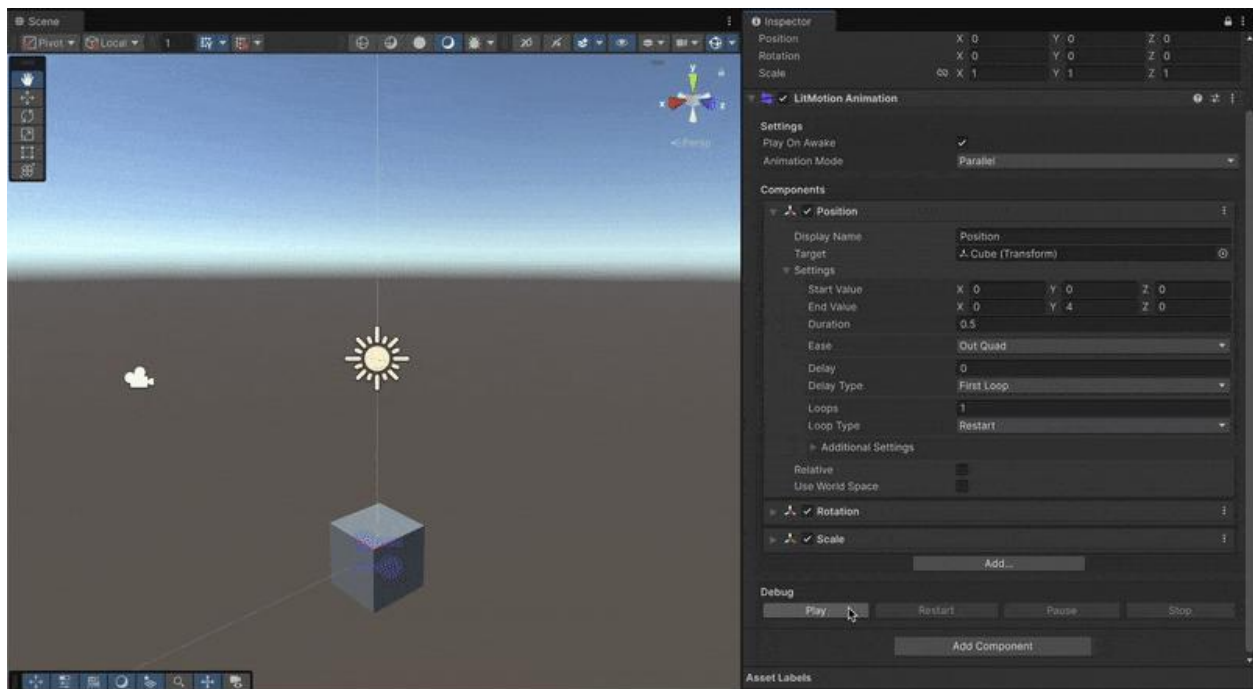
What's new in v2

LitMotion v2 introduces significant enhancements to its functionality. In this section, we will introduce the features added in v2.

For migration from v1, refer to [Migrate from LitMotion v1](#).

LitMotion.Animation Package

The LitMotion.Animation package has been added, which provides functionality for creating animations directly from the Inspector. This package is separate from the main LitMotion package and, when integrated into your project, it enables the use of the LitMotion Animation component, which allows you to create animations through a visual editor in the Unity Inspector.



For more details, see the [LitMotion.Animation](#) section.

Sequence

The Sequence feature has been added to compose multiple motions. You can create a builder with `LSequence.Create()` and add motions to it.

```
LSequence.Create()
    .Append(LMotion.Create(-5f, 5f, 0.5f).BindToPositionX(target))
    .Append(LMotion.Create(0f, 5f, 0.5f).BindToPositionY(target))
    .Append(LMotion.Create(-2f, 2f, 1f).BindToPositionZ(target))
    .Run();
```

For more details, refer to the [Sequence](#) section.

MotionHandle

The following properties and methods have been added to `MotionHandle`:

```
MotionHandle handle;

// methods
handle.Preserve();
handle.IsPlaying();
handle.TryComplete();
handle.TryCancel();

// readonly properties
```

```
handle.Duration;  
handle.TotalDuration;  
handle.Loops;  
handle.CompletedLoops;  
handle.Delay;  
  
// properties  
handle.Time;
```

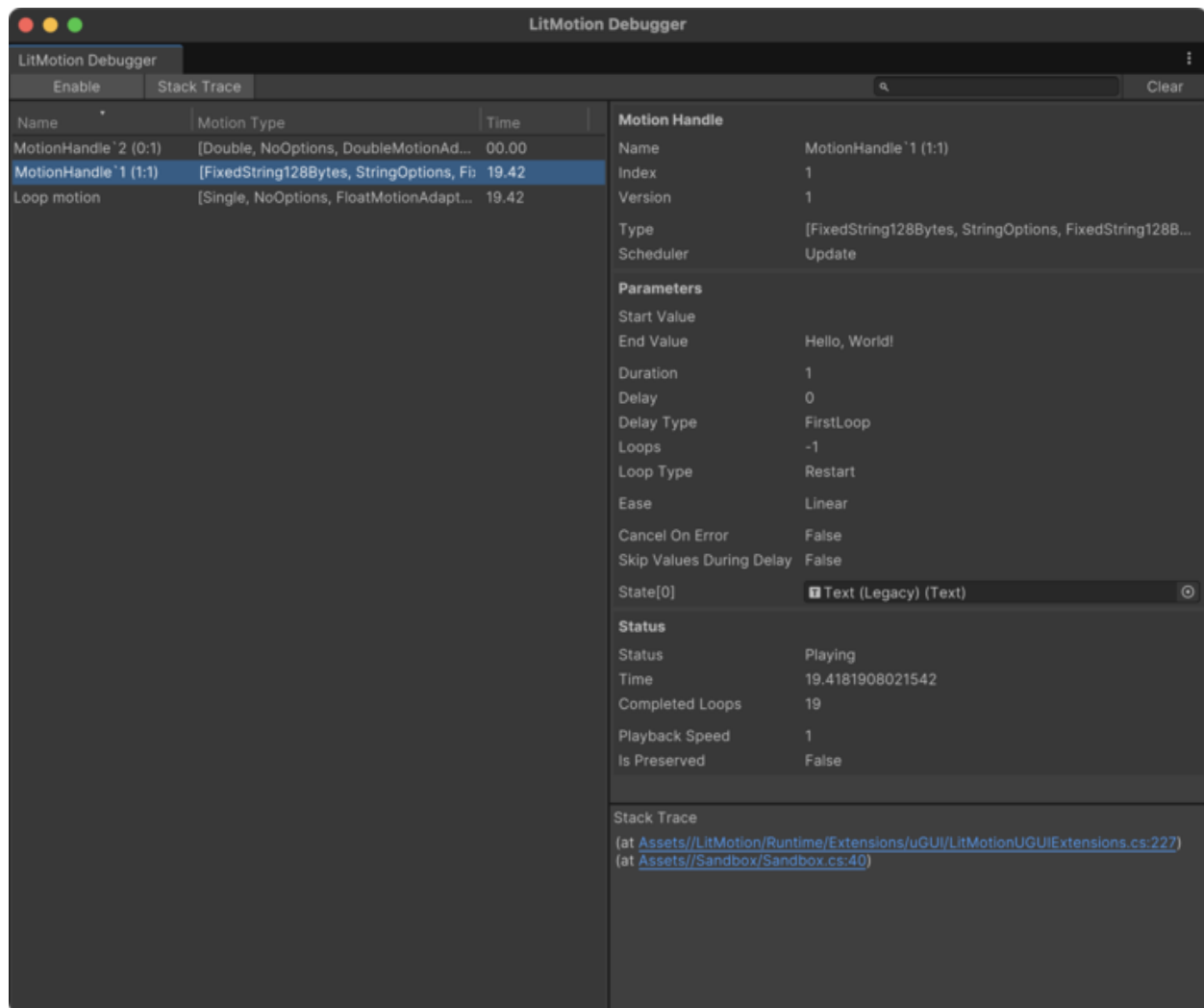
WithOnLoopComplete

A new callback has been added that is triggered upon the completion of each loop. You can add this callback using `WithOnLoopComplete()`.

```
LMotion.Create(...)  
    .WithOnLoopComplete(completedLoops => { })  
    .Bind(x => { })
```

LitMotion Debugger

In place of the MotionTracker window from v1, we have added the more powerful LitMotion Debugger. Additionally, new debugging APIs have been introduced.



For more details, refer to the [LitMotion Debugger](#) section.

Quick Start

Using LitMotion allows you to easily animate values such as Transform or Material. To create a motion, use `LMotion.Create()`.

```
using UnityEngine;
```

```
using LitMotion;
```

```
using LitMotion.Extensions;
```

```
public class Example : MonoBehaviour
```

```
{
```

```

[SerializeField] Transform target;

void Start()
{
    LMotion.Create(Vector3.zero, Vector3.one, 2f) // Animates values from (0f, 0f, 0f) to (1f,
1f, 1f) over 2 seconds

    .BindToPosition(target); // Binds to target.position

    LMotion.Create(0f, 10f, 2f) // Animates values from 0f to 10f over 2 seconds

    .BindToUnityLogger(); // Binds to Debug.unityLogger, displaying values in the Console
on updates

    var value = 0f;

    LMotion.Create(0f, 10f, 2f) // Animates values from 0f to 10f over 2 seconds

    .Bind(x => value = x); // Can bind to any variable, field, or property
}
}

```

The entry point for LitMotion is `LMotion.Create()`. You can construct motions and animate values by binding them to targets.

Package Structure

Namespace	Description
LitMotion	Contains essential functionalities for creating and driving motions.
LitMotion.Adapters	Includes adapters for Unity-specific types like primitive types and <code>Vector3</code> .
LitMotion.Editor	Contains functionalities to operate motions within the editor.

Namespace	Description
LitMotion.Extensions	Contains extension methods to bind motions to Unity components. These functionalities are separated into a different asmdef file.
LitMotion.Animation	A package that provides animation functionality working with LitMotion. The LitMotion Animation component, which allows you to create complex animations directly from the Inspector, becomes available. This is provided as a separate package.

Supported Types

The built-in supported types in LitMotion are as follows:

- int
- long
- float
- double
- Vector2
- Vector3
- Vector4
- Quaternion
- Color
- Rect

If you want to animate types other than these, you can create a [Custom Adapter](#).

FAQ

Is there a feature equivalent to `DelayedCall()`?

There is no method in LitMotion equivalent to `DelayedCall()`. This is by design.

In the modern era of `async/await`, delayed calls using callbacks should be avoided whenever possible. This is because exceptions are not propagated externally, making error handling more difficult. Instead, please use `async` methods.

However, if you are migrating from another tweening library and it is difficult to replace, you can use the following code as an alternative:

```
// The value doesn't matter, so any value will work
```

```
LMotion.Create(0f, 1f, delay)
```

```
.WithOnComplete(action)
```

```
.RunWithoutBinding();
```

Can I add a callback to Sequence?

In DOTween, the Sequence has AppendCallback(), but LitMotion does not implement this method. This is for the same reason that DelayedCall() is not implemented.

LitMotion's Sequence is designed for "combining multiple motions," and it intentionally omits other features. This is to avoid code complexity. (For more details, refer to the [Design Philosophy](#) section.)

Basic Concepts

The LitMotion API consists of several components that are crucial to understand when using LitMotion.

MotionBuilder

A structure used to construct motions. LMotion.Create() returns a MotionBuilder instance.

```
var builder = LMotion.Create(0f, 10f, 3f);
```

MotionBuilder provides methods to configure motions, allowing method chaining.

```
LMotion.Create(0f, 10f, 3f)
```

```
.WithEase(Ease.OutQuad)
```

```
.WithDelay(2f)
```

```
.WithLoops(4, LoopType.Yoyo);
```

LitMotion motions are typically bound to some value. Binding triggers motion creation and playback.

```
var value = 0f;
```

```
LMotion.Create(0f, 10f, 3f)
```

```
.WithEase(Ease.OutQuad)
```

```
.Bind(x => value = x);
```

For more details, refer to [Binding](#) and [Motion Configuration](#).

MotionHandle

MotionHandle controls created motions. Methods like Bind() from MotionBuilder return this handle.

```
var handle = LMotion.Create(0f, 10f, 3f).Bind(x => value = x);
```

You can manage motion presence, completion, or cancellation through MotionHandle.

```
var handle = LMotion.Create(0f, 10f, 3f).Bind(x => value = x);
```

```
if (handle.IsActive())  
{  
    handle.Complete();  
    handle.Cancel();  
}
```

For further details, refer to [Motion Control](#).

MotionScheduler

Motion update timing is determined by MotionScheduler. Set the Scheduler using WithScheduler().

```
LMotion.Create(0f, 10f, 2f)  
    .WithScheduler(MotionScheduler.FixedUpdate)  
    .Bind(() => Debug.Log(x));
```

Refer to [Motion Configuration](#) for more information.

MotionAdapter

The interpolation between two values is described by structures implementing IMotionAdapter<T, TOptions>. Built-in adapters are defined within the LitMotion.Adapters namespace.

To add specific options to a motion, define a structure implementing IMotionOptions.

Refer to [Custom Adapter](#) for creating custom adapters.

Binding

In LitMotion, it is required to bind the values to the target fields or properties when creating a motion. The values of the motion are updated within the PlayerLoop specified by the Scheduler, and the latest values are reflected in the bound fields/properties.

```
var value = 0f;
```

```
LMotion.Create(0f, 10f, 2f)
```

```
    .Bind(x => value = x); // Pass Action<T> to update the value
```

Avoiding Allocations

Lambda expressions passed to Bind() cause allocations due to capturing external variables (known as closures).

Additionally, by passing the state as an argument to Bind(), you can avoid allocations caused by closures.

```
class FooClass
```

```
{  
    public float Value { get; set; }  
}
```

```
var target = new FooClass();
```

```
LMotion.Create(0f, 10f, 2f)
```

```
    .Bind(target, (x, target) => target.Value = x); // Pass the target object as the first argument
```

Extension Methods

The LitMotion.Extensions namespace provides extension methods to simplify value binding.

```
using UnityEngine;
```

```
using LitMotion;
```

```
using LitMotion.Extensions; // Include this namespace
```

```

public class Example : MonoBehaviour
{
    [SerializeField] Transform target;

    void Start()
    {
        LMotion.Create(Vector3.zero, Vector3.one, 2f)
            .BindToPosition(target); // Bind value to target.position

        LMotion.Create(1f, 3f, 2f)
            .BindToLocalScaleX(target); // Bind value to target.localScale.x
    }
}

```

Playing Without Binding

To play motions without value binding, use `RunWithoutBinding()` to create a motion without binding.

```

LMotion.Create(0f, 0f, 2f)
    .WithOnComplete(() => Debug.Log("Complete!"))
    .RunWithoutBinding();

```

If you require further assistance or have more questions, feel free to ask!

Motion Control

Methods for creating motions, such as `Bind()` and `RunWithoutBinding()`, all return a `MotionHandle` struct.

```
var handle = LMotion.Create(0f, 10f, 2f).RunWithoutBinding();
```

You can control the motion through this struct.

Completing/Canceling Motion

To complete a running motion, call `Complete()`.

```
var handle = LMotion.Create(0f, 10f, 2f).RunWithoutBinding();
```

```
handle.Complete();
```

To cancel a running motion, call `Cancel()`.

```
var handle = LMotion.Create(0f, 10f, 2f).RunWithoutBinding();
```

```
handle.Cancel();
```

Motion Existence Check

The above methods/properties will throw an exception if the motion has already finished or if the `MotionHandle` is uninitialized. To check if the motion the `MotionHandle` points to exists, use `IsActive()`.

```
var handle = LMotion.Create(0f, 10f, 2f).RunWithoutBinding();
```

```
if (handle.IsActive()) handle.Complete();
```

If you want to call `Complete()` or `Cancel()` only if the motion exists, you can simplify the code using `TryComplete()` / `TryCancel()`.

```
handle.TryComplete();
```

```
handle.TryCancel();
```

Reusing Motion

By default, completed motions are automatically discarded, so `MotionHandle` cannot be reused.

If you want to reuse the same motion, call `Preserve()`. This prevents the motion from being discarded after completion.

```
// Call Preserve()
```

```
handle.Preserve();
```

```
// It can be reused after completion
```



```
handle.Complete();
```

```
handle.Time = 0;
```

However, motions that have called `Preserve()` will continue to run until `Cancel()` is explicitly called. After use, you should either call `Cancel()` or link the lifetime to a `GameObject` using `AddTo()`.

Adjusting Playback Speed

You can modify the playback speed of a motion using the `MotionHandle.PlaybackSpeed` property. This allows you to slow down, reverse, or pause the motion.

```
var handle = LMotion.Create(0f, 10f, 2f).RunWithoutBinding();
```

```
handle.PlaybackSpeed = 2f;
```

Manually Controlling Motion

You can manually control a motion using the `Time` property.

```
// Manually set the elapsed time of the motion
```

```
handle.Time = 0.5;
```

However, motions that are completed by manually adjusting `Time` are also automatically discarded by default. If you want to manually control a motion, it's recommended to call `Preserve()`.

Getting Motion Information

You can get motion data from the properties of the `MotionHandle`.

```
// Duration per loop
```

```
var duration = handle.Duration;
```

```
// Total duration of the motion
```

```
var totalDuration = handle.TotalDuration;
```

```
// Delay time
```

```
var delay = handle.Delay;
```

```
// Number of loops
```

```
var loops = handle.Loops;
```

```
// Number of completed loops
```

```
var completedLoops = handle.CompletedLoops;
```

To check if a motion is currently playing, use `IsPlaying()`. This is similar to `IsActive()`, but unlike `IsActive()`, which always returns true until the motion is discarded, `IsPlaying()` will return false once the motion completes (if `Preserve()` was called).

```
if (handle.IsPlaying())
```

```
{
```

```
    DoSomething();
```

```
}
```

Linking Cancellation to GameObject

You can use `AddTo()` to automatically cancel the motion when the `GameObject` is destroyed.

```
LMotion.Create(0f, 10f, 2f)
```

```
    .Bind(() => Debug.Log(x))
```

```
    .AddTo(this.gameObject);
```

Managing Multiple MotionHandle

To manage multiple `MotionHandle` instances together, the `CompositeMotionHandle` class is provided. You can bind `MotionHandle` instances to this class using `AddTo()`.

```
var handles = new CompositeMotionHandle();
```

```
LMotion.Create(0f, 10f, 2f)
```

```
    .Bind(() => Debug.Log(x))
```

```
    .AddTo(handles);
```

Motion Configuration

You can add settings such as easing, repeat count, callbacks, etc., when creating a motion. To add settings, use the With- methods.

When writing, you can apply multiple settings simultaneously using method chaining.

```
var value = 0f;
```

```
LMotion.Create(0f, 10f, 2f)
```

```
    .WithEase(Ease.OutQuad)
```

```
    .WithComplete(() => Debug.Log("Complete!"))
```

```
    .Bind(x => value = x);
```

List of Methods

WithEase

Specifies the easing function to apply to the motion.

You can also specify AnimationCurve for this. (Ease.CustomAnimationCurve is automatically set when using WithEase(AnimationCurve). Do not specify this option with WithEase(Ease).)

WithDelay

Delay the start of a motion by a specified number of seconds. You can adjust the behavior by specifying DelayType and SkipValuesDuringDelay.

- DelayType

Specifies the behavior of delay during looping.

DelayType	Behavior
DelayType.FirstLoop	Default setting. Applies delay only in the first loop.
DelayType.EveryLoop	Applies delay in each loop.

- SkipValuesDuringDelay

Specifies whether to skip the processing of Bind during the delay time. It is set to true by default.

WithLoops

Sets the number of times the motion will repeat. It's set to 1 by default. Setting it to -1 creates a motion that repeats infinitely until stopped.

Additionally, you can set the behavior during repetition by specifying `LoopType` as the second argument.

LoopType	Behavior
<code>LoopType.Restart</code>	Default behavior. Resets to the start value at the end of each loop.
<code>LoopType.Flip</code>	Animates the value back and forth between start and end values.
<code>LoopType.Increment</code>	Value increases with each loop.
<code>LoopType.Yoyo</code>	Animates between the start and end values in a back-and-forth (yo-yo) motion.

WithOnComplete

Specifies a callback at the end of the playback.

WithOnCancel

Specifies a callback for when the motion is canceled. Here's the English translation for the provided text:

WithOnLoopComplete

Specifies a callback to be called at the end of each loop. This is also called when the motion completes, with the order being `OnLoopComplete` → `OnComplete`.

WithScheduler

Specifies the Scheduler used for motion playback.

Scheduler	Behavior
<code>MotionScheduler.Initialization</code>	Updates at the Initialization timing.
<code>MotionScheduler.InitializationIgnoreTimeScale</code>	Updates at the Initialization timing, ignores the influence of <code>Time.timeScale</code> .
<code>MotionScheduler.InitializationRealtime</code>	Updates at the Initialization timing, ignores the influence of

Scheduler

Behavior

MotionScheduler.EarlyUpdate

Time.timeScale, and calculates time using Time.realtimeSinceStartup.

Updates at the EarlyUpdate timing.

MotionScheduler.EarlyUpdateIgnoreTimeScale

Updates at the EarlyUpdate timing, ignores the influence of Time.timeScale.

MotionScheduler.EarlyUpdateRealtime

Updates at the EarlyUpdate timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.

MotionScheduler.FixedUpdate

Updates at the FixedUpdate timing.

MotionScheduler.PreUpdate

Updates at the PreUpdate timing.

MotionScheduler.PreUpdateIgnoreTimeScale

Updates at the PreUpdate timing, ignores the influence of Time.timeScale.

MotionScheduler.PreUpdateRealtime

Updates at the PreUpdate timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.

MotionScheduler.Update

Updates at the Update timing.

MotionScheduler.UpdateIgnoreTimeScale

Updates at the Update timing, ignores the influence of Time.timeScale.

MotionScheduler.UpdateRealtime

Updates at the Update timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.

MotionScheduler.PreLateUpdate

Updates at the PreLateUpdate timing.

Scheduler	Behavior
MotionScheduler.PreLateUpdateIgnoreTimeScale	Updates at the PreLateUpdate timing, ignores the influence of Time.timeScale.
MotionScheduler.PreLateUpdateRealtime	Updates at the PreLateUpdate timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.
MotionScheduler.PostLateUpdate	Updates at the PostLateUpdate timing.
MotionScheduler.PostLateUpdateIgnoreTimeScale	Updates at the PostLateUpdate timing, ignores the influence of Time.timeScale.
MotionScheduler.PostLateUpdateRealtime	Updates at the PostLateUpdate timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.
MotionScheduler.TimeUpdate	Updates at the TimeUpdate timing.
MotionScheduler.TimeUpdateIgnoreTimeScale	Updates at the TimeUpdate timing, ignores the influence of Time.timeScale.
MotionScheduler.TimeUpdateRealtime	Updates at the TimeUpdate timing, ignores the influence of Time.timeScale, and calculates time using Time.realtimeSinceStartup.
MotionScheduler.Manual	Updates will be performed using the ManualMotionDispatcher. For more details, please refer to ManualMotionDispatcher .

Scheduler

Behavior

EditorMotionScheduler.Update (LitMotion.Editor)

Updates at the EditorApplication.update timing. This Scheduler is limited to the editor.

WithCancelOnError

Cancels the motion when an uncaught exception occurs within the motion's Bind function. It is set to false by default.

WithImmediateBind

Executes the Bind operation at the time of scheduling the motion. By default, it is set to true.

WithDebugName

Sets a name for debugging purposes. For more details, refer to the [LitMotion Debugger](#) section.

WithRoundingMode (int)

Sets the rounding mode for decimal values. This option is only applicable to int-type motions.

RoundingMode

Behavior

RoundingMode.ToEven

Default setting. Rounds the value to the nearest integer and, in case of a tie, rounds to the nearest even number.

RoundingMode.AwayFromZero

Typical rounding behavior. Rounds the value to the nearest integer and away from zero in case of a tie.

RoundingMode.ToZero

Rounds the value towards zero.

RoundingMode.ToPositiveInfinity

Rounds the value towards positive infinity.

RoundingMode.ToNegativeInfinity

Rounds the value towards negative infinity.

WithScrambleMode (FixedString-)

You can fill the yet-to-be-displayed characters with random characters. This option is applicable only to string motions.

ScrambleMode	Description
ScrambleMode.None	Default setting. Nothing is displayed in the yet-to-be-displayed parts.
ScrambleMode.Uppercase	Fills spaces with random uppercase alphabet characters.
ScrambleMode.Lowercase	Fills spaces with random lowercase alphabet characters.
ScrambleMode.Numerals	Fills spaces with random numeral characters.
ScrambleMode.All	Fills spaces with random uppercase/lowercase alphabet or numeral characters.
(ScrambleMode.Custom)	Fills spaces with random numeral characters from the specified string. This option cannot be explicitly specified and is set when passing a string argument to WithScrambleMode.

WithRichText (FixedString-)

Enables RichText support, allowing character advancement in text containing RichText tags. This option is applicable only to string motions.

WithFrequency (Punch, Shake)

Sets the frequency (number of oscillations until the end) for Punch and Shake vibrations. The default value is set to 10.

WithDampingRatio (Punch, Shake)

Sets the damping ratio for Punch and Shake vibrations. When this value is 1, it fully dampens, and when it's 0, there is no damping at all. The default value is set to 1.

WithRandomSeed (FixedString-, Shake)

Allows you to specify a random seed used during motion playback. This controls the random behavior of ScrambleChars or vibrations.

MotionSettings

By using MotionSettings<T, TOptions>, you can store motion configuration settings for reuse.

MotionSettings<T, TOptions> can be created either using object initializers or via the MotionBuilder. The type arguments should be the type of the value to animate and the

options type (such as NoOptions, IntegerOptions, StringOptions, PunchOptions, ShakeOptions, etc.).

// Created using an object initializer

```
var settings = new MotionSettings<float, NoOptions>
```

```
{
```

```
    StartValue = 0f,
```

```
    EndValue = 10f,
```

```
    Duration = 2f,
```

```
    Ease = Ease.OutQuad
```

```
};
```

// Created using MotionBuilder

```
var settings = LMotion.Create(0f, 10f, 2f)
```

```
    .WithEase(Ease.OutQuad)
```

```
    .ToMotionSettings();
```

The created MotionSettings<T, TOptions> can be passed as an argument to LMotion.Create().

```
LMotion.Create(settings)
```

```
    .Bind(x => { });
```

Since MotionSettings<T, TOptions> is a record type, you can also overwrite part of the settings using the with expression.

```
var newSettings = settings with
```

```
{
```

```
    StartValue = 5f
```

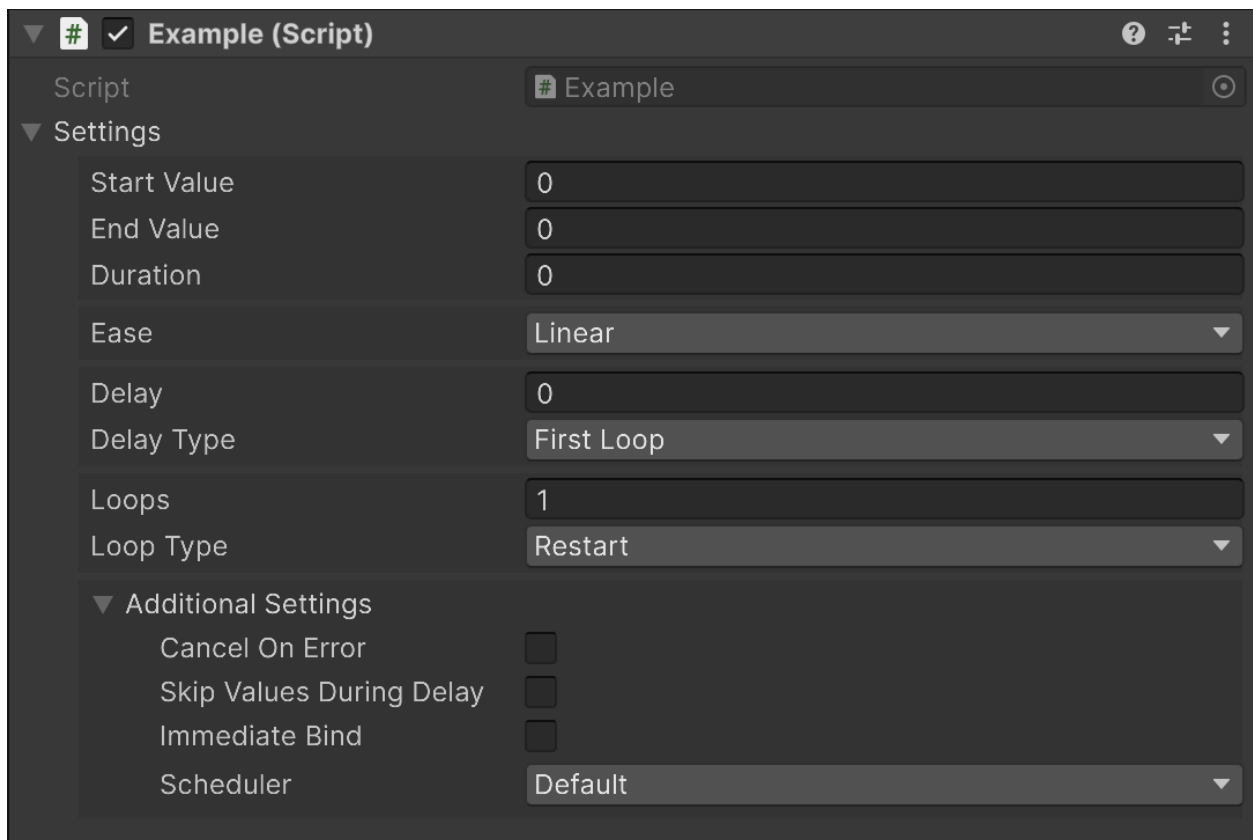
```
};
```

SerializableMotionSettings

Instead of using `MotionSettings<T, TOptions>`, you can use `SerializableMotionSettings<float, NoOptions>` to make the values editable in the Inspector.

```
public class Example : MonoBehaviour
{
    [SerializeField] SerializableMotionSettings<float, NoOptions> settings;

    void Start()
    {
        LMotion.Create(settings)
            .BindToPositionX(transform);
    }
}
```



Sequence

Using a Sequence, you can combine multiple motions. This is useful when controlling complex animations.

To create a Sequence, you build it with `LSequence.Create()`, add motions, and then call `Run()`.

```
LSequence.Create()  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionX(transform))  
    .Join(LMotion.Create(0f, 1f, 1f).BindToPositionY(transform))  
    .Insert(0f, LMotion.Create(0f, 1f, 1f).BindToPositionZ(transform))  
    .Run();
```

The return value of `Run()` is a `MotionHandle`, so it can be treated the same as any other motion.

Warning

You cannot add motions that are already playing or those with infinite loops to a sequence. (An exception will occur.)

Append

`Append()` adds motions sequentially after the previous motion in the sequence.

// The x, y, z motions will be played in order

```
LSequence.Create()  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionX(transform))  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionY(transform))  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionZ(transform))  
    .Run();
```

You can also add a delay between motions using `AppendInterval()`.

```
LSequence.Create()  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionX(transform))  
    .AppendInterval(0.5f)  
    .Append(LMotion.Create(0f, 1f, 1f).BindToPositionY(transform))
```

```
.AppendInterval(0.5f)
.Append(LMotion.Create(0f, 1f, 1f).BindToPositionZ(transform))
.Run();
```

Join

Join() adds a motion at the start of the last added motion in the sequence, making them play simultaneously.

// The x, y, and z motions will play simultaneously

```
LSequence.Create()
.Join(LMotion.Create(0f, 1f, 1f).BindToPositionX(transform))
.Join(LMotion.Create(0f, 1f, 1f).BindToPositionY(transform))
.Join(LMotion.Create(0f, 1f, 1f).BindToPositionZ(transform))
.Run();
```

Insert

Insert() inserts a motion at a specified position in the sequence.

// Insert the x, y, and z motions at specified positions

```
LSequence.Create()
.Insert(0.1f, LMotion.Create(0f, 1f, 1f).BindToPositionX(transform))
.Insert(0.2f, LMotion.Create(0f, 1f, 1f).BindToPositionY(transform))
.Insert(0.3f, LMotion.Create(0f, 1f, 1f).BindToPositionZ(transform))
.Run();
```

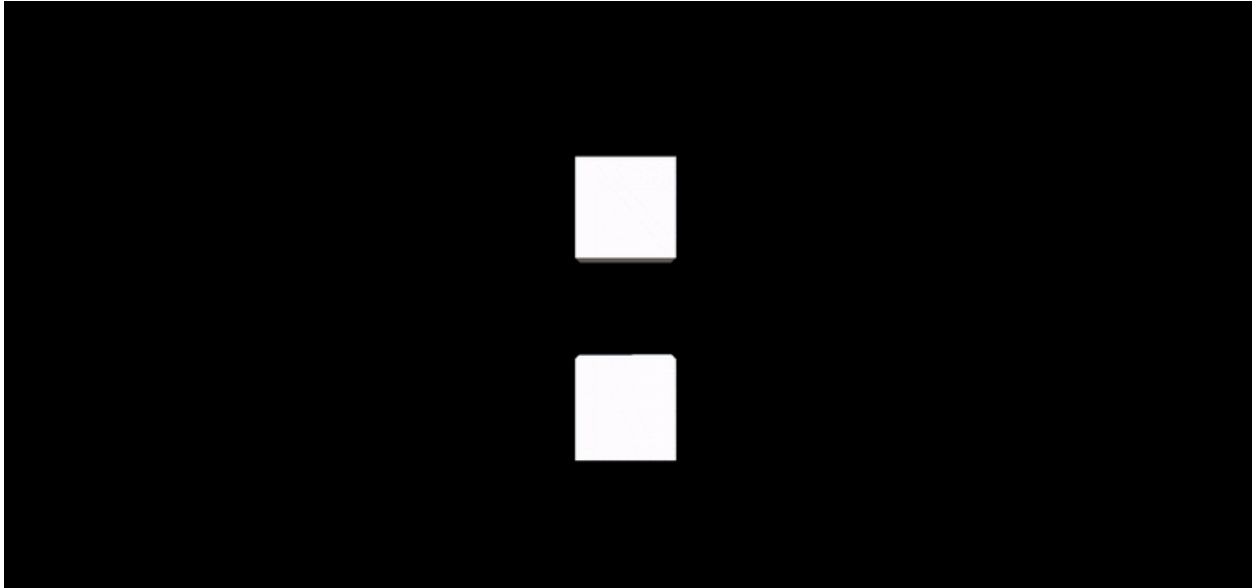
Vibration Motion with Punch/Shake

You can create a vibration motion using LMotion.Punch.Create() or LMotion.Shake.Create().

```
LMotion.Punch.Create(0f, 5f, 2f)
.BindToPositionX(target1);
```

```
LMotion.Shake.Create(0f, 5f, 2f)
```

```
.BindToPositionX(target2);
```



When creating a vibration motion, specify the initial value (startValue) as the first argument and the strength of the vibration (strength) as the second argument. The vibration will fluctuate within the range of $\text{startValue} \pm \text{strength}$. Please note that the value range for vibration motion differs from the usual motion.

The difference between Punch and Shake lies in the behavior of the vibration. With Punch, the vibration is regular, while Shake exhibits random movement.

These motions offer specific settings:

```
LMotion.Punch.Create(0f, 5f, 2f)
```

```
.WithFrequency(20)
```

```
.WithDampingRatio(0f)
```

```
.BindToPositionX(target1);
```

```
LMotion.Shake.Create(0f, 5f, 2f)
```

```
.WithFrequency(20)
```

```
.WithDampingRatio(0f)
```

```
.WithRandomSeed(123)
```

```
.BindToPositionX(target2);
```

For further details, please refer to the [Motion Configuration](#) section.

Text Animation

You can create a motion to animate strings using `LMotion.String.Create**Bytes()`.

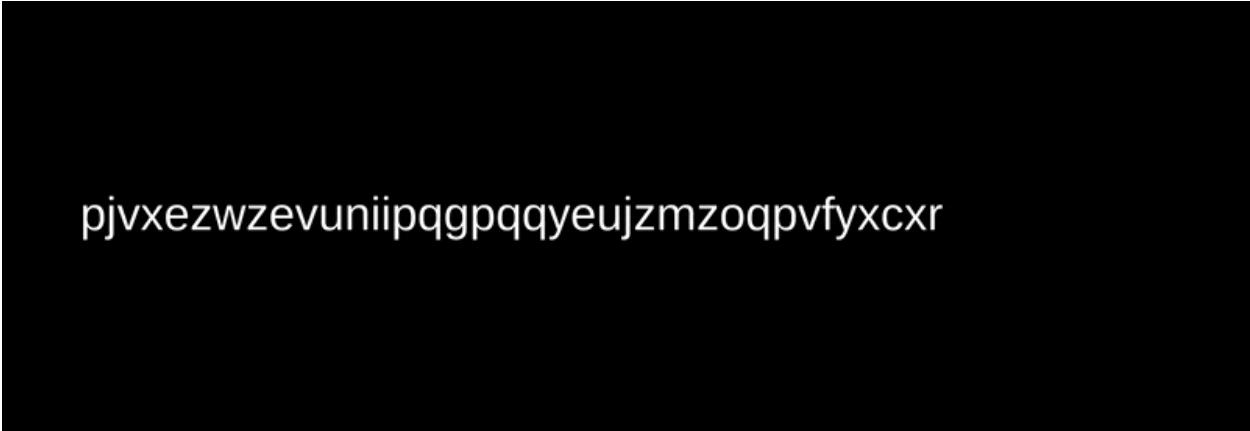
```
TMP_Text text;
```

```
LMotion.String.Create128Bytes("", "<color=red>Zero</color> Allocation <i>Text</i> Tween!  
<b>Foooooo!!</b>", 5f)
```

```
.WithRichText()
```

```
.WithScrambleChars(ScrambleMode.Lowercase)
```

```
.BindToText(text);
```



pjvxewzewuniipqgppqyeujzmzoqpvfyxcxr

Binding Numeric Motion to Text

It's also possible to bind numeric motion to text. When using `TMP_Text` as the target, you can perform binding with zero allocation using `BindToText()`.

```
TMP_Text text;
```

```
LMotion.Create(0, 999, 2f)
```

```
.BindToText(text);
```

Moreover, it's possible to set formatting by passing a format string. Below is a sample motion for displaying a float number with comma separation up to two decimal places.

```
TMP_Text text;
```

```
LMotion.Create(0f, 100000f, 2f)
```

```
.BindToText(text, "{0:N2}");
```



1,000.00

However, motions including this formatting use `string.Format()` internally, leading to GC allocations. To avoid this, you'll need to introduce `ZString` into your project. For more details, refer to the section on [ZString](#).

TextMesh Pro Character Animation

In addition to the ability to animate text and values in TextMesh Pro, LitMotion provides the ability to animate specified characters.



LitMotion

`TMP_Text text;`

```

for (int i = 0; i < text.textInfo.characterCount; i++)
{
    LMotion.Create(Color.white, Color.red, 1f)
        .WithDelay(i * 0.1f)
        .WithEase(Ease.OutQuad)
        .BindToTMPCharColor(text, i);

    LMotion.Punch.Create(Vector3.zero, Vector3.up * 30f, 1f)
        .WithDelay(i * 0.1f)
        .WithEase(Ease.OutQuad)
        .BindToTMPCharPosition(text, i);
}

```

While manipulating characters with LitMotion, the character information is maintained even if the characters are rewritten during motion playback. However, after playback, it returns to the initial values through mesh updates (such as rewriting the text or calling `ForceMeshUpdate()`).

Await Motion in Coroutine

It's possible to await the completion of a motion within a coroutine using `ToYieldInstruction()` of `MotionHandle`.

```

IEnumerator CoroutineExample()
{
    yield return LMotion.Create(0f, 10f, 2f).Bind(x => Debug.Log(x))
        .ToYieldInstruction();
}

```

However, coroutines have functional limitations such as the inability to return values and await in parallel due to language constraints.

As an alternative, it's recommended to use `async/await` with `UniTask` for waiting on motions. For information on integrating with `UniTask`, please refer to [UniTask](#) Integration.

Await Motion in async/await

MotionHandle implements the GetAwaiter() method, allowing you to directly await it to wait for completion.

```
await handle;
```

If you want to pass a CancellationToken, you can use ToValueTask() / ToAwaitable() or UniTask.

ValueTask

You can convert a motion to a ValueTask using MotionHandle.ToValueTask(). This allows you to use async/await to wait for the completion of the motion.

```
async ValueTask ExampleAsync(CancellationToken cancellationToken)
{
    await LMotion.Create(0f, 10f, 1f)
        .RunWithoutBinding()
        .ToValueTask(cancellationToken);
}
```

However, using ValueTask in Unity has some performance concerns. For the best performance, it is recommended to use UniTask. For integration with UniTask, refer to the [UniTask](#) guide.

Awaitable

Starting from Unity 2023.1, Unity provides the [Awaitable](#) class to enable efficient async/await handling within Unity.

If you're using Unity 2023.1 or later, LitMotion provides the ToAwaitable() extension method to convert a MotionHandle to an Awaitable. This allows you to use async/await to wait for motion completion.

```
async Awaitable ExampleAsync(CancellationToken cancellationToken)
{
    await LMotion.Create(0f, 10f, 1f)
        .RunWithoutBinding()
```

```
.ToAwaitable(cancellationToken);  
}
```

Changing Behavior on Cancellation

By specifying `CancelBehavior` in the arguments for `ToValueTask()` / `ToAwaitable()`, you can change the behavior when the async method is canceled. Additionally, setting `cancelAwaitOnMotionCanceled` to true allows the async method to be canceled when the `MotionHandle` is canceled.

```
await LMotion.Create(0f, 10f, 1f)  
    .RunWithoutBinding()  
    .ToAwaitable(CancelBehavior.Complete, true, cancellationToken);
```

Convert to IDisposable

`MotionHandle` can be converted to `IDisposable`.

```
var disposable = handle.ToDisposable();
```

By default, calling `Dispose()` on this `IDisposable` will cancel the motion. This behavior can be changed by specifying a `DisposeBehavior` argument in `ToDisposable()`.

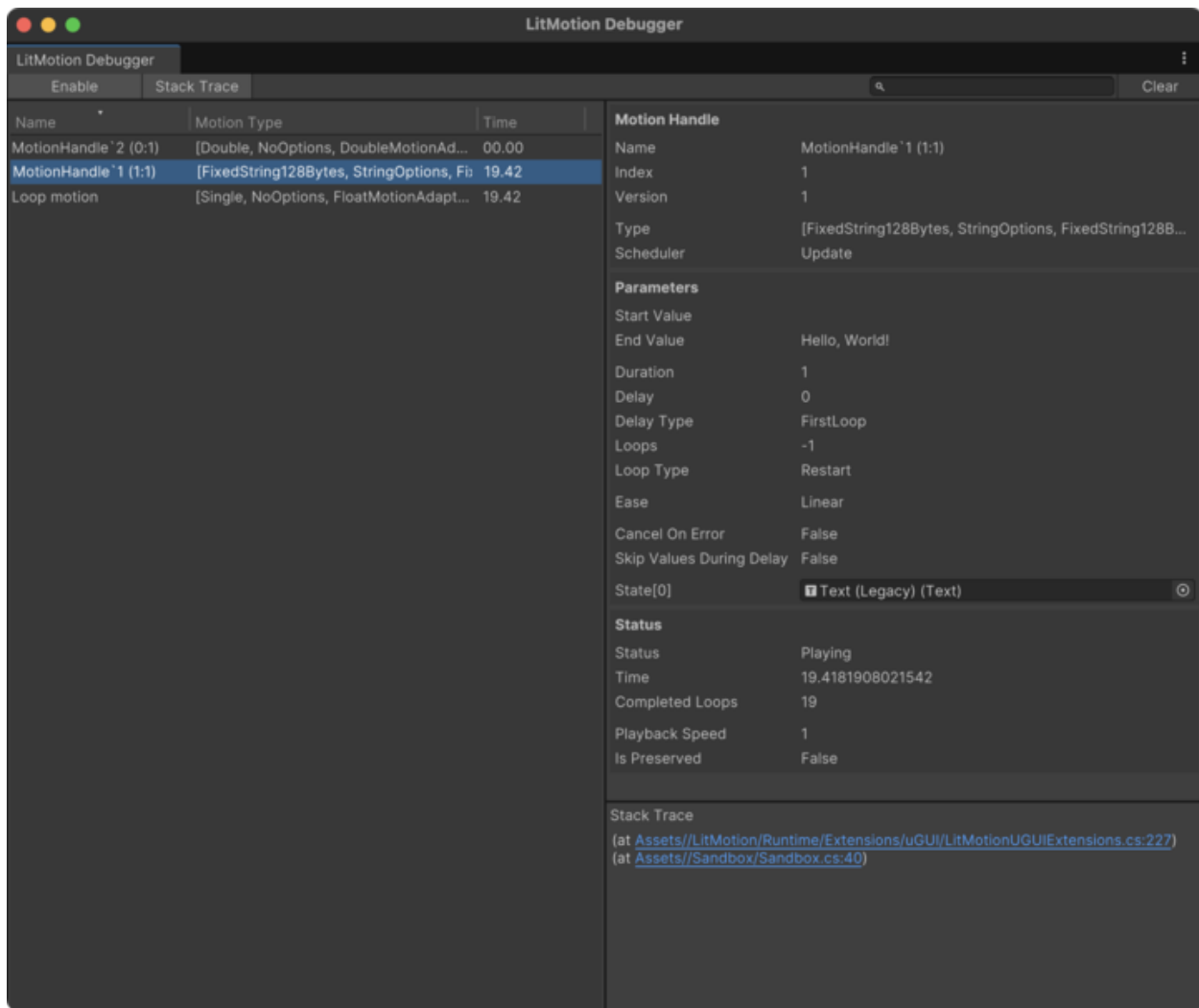
```
var disposable = handle.ToDisposable(DisposeBehavior.Complete);
```

```
// handle.Complete() will be called
```

```
disposable.Dispose();
```

LitMotion Debugger

By using the LitMotion Debugger window, you can view information about the motions you've created.



Open the window from Window > LitMotion Debugger, and click [Enable] to activate the debugger. You can also click [Stack Trace] to enable and view the stack trace when the motion is created.

Warning

Enabling the debugger can significantly impact performance. It is recommended to keep it disabled and only enable it during debugging sessions.

Debug Name

In the debugger, you can search for a specific motion by its name. The default debug name follows this format:

```
$"MotionHandle`{StorageId}({Index};{Version})"
```

This debug name can be changed by passing a custom name using `WithDebugName()` when creating the motion.

```
var handle = LMotion.Create(0f, 1f, 1f)

    .WithDebugName("name")

    .Bind(x => { });
```

You can also get the debug name using `GetDebugName()`.

```
var name = handle.GetDebugName();
```

This debug name is only preserved in Debug builds and will be removed in Release builds. To use this feature in a Release build, add `LITMOTION_DEBUG` to the Scripting Define Symbols.

Play Motion in Unity Editor

LitMotion supports playback in the Unity Editor. When creating motions in Edit Mode, they are automatically scheduled in the Editor. In this case, the motions are driven on `EditorApplication.update`.

```
using UnityEngine;

using LitMotion;
```

```
if (Application.isPlaying)
{
    // In Play Mode (Runtime), it operates as usual on Update

    LMotion.Create(0f, 10f, 2f)

        .Bind(x => Debug.Log(x));
}
else
{
    // In Edit Mode, it operates on EditorApplication.update

    LMotion.Create(0f, 10f, 2f)
```

```
.Bind(x => Debug.Log(x));  
}
```

You can explicitly specify EditorMotionScheduler.Update to the scheduler as well.

```
using LitMotion;
```

```
using LitMotion.Editor;
```

```
LMotion.Create(0f, 10f, 2f)
```

```
.WithScheduler(EditorMotionScheduler.Update)
```

```
.Bind(x => Debug.Log(x));
```

Exception Handling

You can use MotionDispatcher.RegisterUnhandledExceptionHandler() to set up handling for unhandled exceptions that occur within Bind or WithOnComplete. By default, it is configured to display exceptions in the Console using UnityEngine.Debug.LogException(ex).

```
using UnityEngine;
```

```
using LitMotion;
```

```
// Retrieve the currently set exception handling (Action<Exception>)
```

```
var handler = MotionDispatcher.GetUnhandledExceptionHandler();
```

```
// Change to display a warning using LogWarning instead of LogException
```

```
MotionDispatcher.RegisterUnhandledExceptionHandler(ex => Debug.LogWarning(ex));
```

ManualMotionDispatcher

By using the ManualMotionDispatcher, you can manually update multiple motions.

```
var dispatcher = new ManualMotionDispatcher();
```

You can set the motion update process to be handled by the created dispatcher by specifying dispatcher.Scheduler in the Scheduler.

```
// Set the Scheduler to the created ManualMotionDispatcher's Scheduler
```

```
var handle = LMotion.Create(value, endValue, 2f)
```

```
.WithScheduler(dispatcher.Scheduler)
```

```
.BindToUnityLogger();
```

You can perform updates using `dispatcher.Update(double deltaTime)`.

```
dispatcher.Update(0.1);
```

ManualMotionDispatcher.Default

If you need a globally available `ManualMotionDispatcher`, you can use `ManualMotionDispatcher.Default`.

```
ManualMotionDispatcher.Default.Update(0.1);
```

`MotionScheduler.Manual` is equivalent to `ManualMotionDispatcher.Default.Scheduler`.

Warning

When using `ManualMotionDispatcher.Default`, setting `Domain Reload` to off may cause unexpected behavior, as the motion state is not reset. To avoid this, explicitly initialize it at startup by calling `Reset()`.

```
void Awake()
```

```
{
```

```
    ManualMotionDispatcher.Default.Reset();
```

```
}
```

Avoid Dynamic Memory Allocation

You can expand the capacity of the internal array that holds motions beforehand by calling `MotionDispatcher.EnsureStorageCapacity()`. By ensuring the maximum anticipated capacity, such as during the app's startup, you can mitigate runtime dynamic memory allocation.

```
MotionDispatcher.EnsureStorageCapacity<float, NoOptions, FloatMotionAdapter>(500);
```

```
MotionDispatcher.EnsureStorageCapacity<Vector3, NoOptions,  
Vector3MotionAdapter>(1000);
```

Since storage differs for each combination of value and options, you need to call `EnsureStorageCapacity()` for each respective type.

Custom Binding Extension Method

Although it is possible to animate properties not provided in `LitMotion.Extensions` using `Bind()`, it can be convenient to define extension methods for frequently used properties. Here, we will demonstrate how to add a custom binding extension method to `MotionBuilder`.

As an example, let's define an extension method to bind a motion to the `Value` property of a `Foo` class like the one below:

```
public class Foo
{
    public float Value { get; set; }
}
```

Define an extension method tailored to the type of the target property (in this case, `float`) as follows:

```
public static class FooMotionExtensions
{
    public static MotionHandle BindToFooValue<TOptions, TAdapter>(this
MotionBuilder<float, TOptions, TAdapter> builder, Foo target)
        where TOptions : unmanaged, IMotionOptions
        where TAdapter : unmanaged, IMotionAdapter<float, TOptions>
    {
        return builder.Bind(target, (x, target) =>
        {
            target.Value = x;
        });
    }
}
```

By defining it as a generic method, you can bind values of float type regardless of the types of TOptions and TAdapter. Additionally, when the target is a class, using Bind(TState, Action<T, TState>) avoids the use of closures, allowing you to create motions with zero allocation.

The code to animate a value using this extension method is below.

```
var foo = new Foo();  
  
LMotion.Create(0f, 10f, 2f)  
    .BindToFooValue(foo);
```

Custom Adapter

In LitMotion, IMotionAdapter<T, TOptions> and IMotionOptions interfaces are provided for extension purposes.

Implementing an Adapter

Implementing a structure that conforms to IMotionAdapter<T, TOptions> allows you to animate custom types. Below is an example of implementing Vector3MotionAdapter to add support for Vector3:

```
using Unity.Jobs;  
  
using Unity.Mathematics;  
  
using UnityEngine;  
  
using LitMotion;  
  
  
// The assembly attribute is necessary to support Burst for Jobs  
  
// Add the RegisterGenericJobType attribute to register MotionUpdateJob<T, TOptions,  
TAdapter>  
  
[assembly: RegisterGenericJobType(typeof(MotionUpdateJob<Vector3, NoOptions,  
Vector3MotionAdapter>))]  
  
  
// Create a structure implementing IMotionAdapter with type arguments specifying the  
target value type and additional options (if required, else use NoOptions)  
  
public readonly struct Vector3MotionAdapter : IMotionAdapter<Vector3, NoOptions>
```



```

{
    // Implement the interpolation logic within the Evaluate method

    public Vector3 Evaluate(ref Vector3 startValue, ref Vector3 endValue, ref NoOptions
options, in MotionEvaluationContext context)
    {
        return Vector3.LerpUnclamped(startValue, endValue, context.Progress);
    }
}

```

Adapters cannot hold state, so refrain from defining unnecessary fields within them.

LitMotion utilizes generic Jobs internally. Therefore, when using custom types, you need to add the RegisterGenericJobType attribute to the assembly to ensure Burst recognizes the types.

If you want motions to have special states, implement an unmanaged structure conforming to IMotionOptions. Here is a partial implementation example for IntegerOptions used in motions involving int or long types:

```

public struct IntegerOptions : IMotionOptions, IEquatable<IntegerOptions>
{
    public RoundingMode RoundingMode;

    // Implement Equals, GetHashCode, etc.

    ...
}

```

Specify NoOptions if options are not necessary.

Creating Motions Using Custom Adapters

When creating a MotionBuilder with LMotion.Create, you can use a custom adapter by passing the adapter type as a type argument.

```

LMotion.Create<Vector3, NoOptions, Vector3MotionAdapter>(from, to, duration)
    .BindToPosition(transform);

```

UniTask

Introducing [UniTask](#) into your project adds extension methods that allow motion waiting to be compatible with async/await.

If you have installed UniTask via the Package Manager, the following functionality will be automatically added. However, if you have imported UniTask via a unitypackage or similar method, you'll need to add LITMOTION_SUPPORT_UNITASK to Settings > Player > Scripting Define Symbols.

Waiting for Motion

You can convert a MotionHandle to a UniTask using the ToUniTask() method.

```
var cts = new CancellationTokenSource();  
  
await LMotion.Create(0f, 10f, 2f).Bind(x => Debug.Log(x))  
  
    .ToUniTask(cts.Token);
```

The options that can be specified in the argument are the same as for ToValueTask() / ToAwaitable().

ZString

By integrating [ZString](#) into your project, you can achieve zero-allocation string formatting.

If you install ZString via the Package Manager, the internal processing will automatically be replaced with ZString. If you use a unitypackage or similar method to install, you'll need to add LITMOTION_ZSTRING_SUPPORT to Project Settings > Player > Scripting Define Symbols.

Zero-Allocation for BindToText

When passing a string format argument to BindToText(), LitMotion internally uses string.Format(). Since it takes an object as an argument, it incurs allocation due to boxing.

```
TMP_Text text;  
  
LMotion.Create(0, 100, 2f)  
  
    .BindToText(text, "{0:0}"); // Causes GC allocation per frame
```

The internal implementation of BindToText() is partially as follows:

```
// Part of the code for BindToText()  
  
builder.BindWithState(text, format, (x, target, format) =>
```

```
{
    ...

    target.text = string.Format(format, x); // Allocates due to boxing here
});
```

Introducing ZString replaces this with processing using ZString.Format() within LitMotion. This reduction eliminates unnecessary boxing allocations.

```
builder.BindWithState(text, format, (x, target, format) =>
{
    ...

    target.text = ZString.Format(format, x); // Allows zero-allocation formatting
});
```

Furthermore, if the target is TMP_Text, instead of using ZString.Format(), LitMotion uses SetTextFormat(), an extension method of ZString. This method internally utilizes TMP_Text.SetText(), enabling entirely zero-allocation formatting processing.

```
builder.BindWithState(text, format, (x, target, format) =>
{
    ...

    target.SetTextFormat(format, x);
});
```

Migrate from DOTween

This page provides a simple guide for migrating from DOTween to LitMotion.

Position Tweens

```
var endValue = new Vector3(1f, 2f, 3f);

var duration = 1f;

// DOTween
transform.DOMove(endValue, duration);
```

```
// LitMotion
```

```
LMotion.Create(transform.position, endValue, duration)
```

```
.BindToPosition(transform);
```

Value Tweens

```
// DOTween
```

```
var value = 0f;
```

```
DOTween.To(() => value, x => value = x, endValue, duration);
```

```
// LitMotion
```

```
LMotion.Create(value, endValue, duration)
```

```
.Bind(x => value = x);
```

From

```
// DOTween
```

```
transform.DOMoveX(endValue, duration).From(startValue);
```

```
// LitMotion
```

```
LMotion.Create(startValue, endValue, duration)
```

```
.BindToPositionX(transform);
```

Punch / Shake

```
// DOTween
```

```
transform.DOPunchPosition(...);
```

```
transform.DOShakePosition(...);
```

```
// LitMotion
```

```
LMotion.Punch.Create(...)
```

```
.BindToPosition(transform);
```

```
LMotion.Shake.Create(...)
```

```
.BindToPosition(transform);
```

Tween Settings

```
// DOTween
```

```
tween.SetLoops(2, LoopType.Yoyo)
```

```
.SetEase(Ease.OutQuad);
```

```
// LitMotion
```

```
builder.WithLoops(2, LoopType.Yoyo)
```

```
.WithEase(Ease.OutQuad);
```

Tween Control

```
// DOTween
```

```
tween.Pause();
```

```
tween.Complete();
```

```
tween.Kill();
```

```
// LitMotion
```

```
handle.PlaybackSpeed = 0f;
```

```
handle.Complete();
```

```
handle.Cancel();
```

Sequence

```
// DOTween
```

```
DOTween.Sequence()
```

```
.Append(...)
```

```
.Join(...)
```

```
.Insert(...);
```

```
// LitMotion
```

```
LSequence.Create()
```

```
.Append(...)
```

```
.Join(...)
```

```
.Insert(...)
```

```
.Run();
```

Adding callbacks to Sequences is not supported in LitMotion. This is intentional, as LitMotion encourages the use of async methods for complex animations that require callbacks. For more details, refer to the [Design Philosophy](#).

Update Timing Changes

```
// DOTween
```

```
tween.SetUpdate(UpdateType.Fixed);
```

```
// LitMotion
```

```
builder.WithScheduler(MotionScheduler.FixedUpdate);
```

Linking to GameObject

```
// DOTween
```

```
tween.SetLink(gameObject);
```

```
// LitMotion
```

```
handle.AddTo(gameObject);
```

Coroutines, async/await

```
// DOTween
```

```
yield return tween.WaitForCompletion();
```

```
await tween.AsyncWaitForCompletion();
```

```
// LitMotion
```

```
yield return handle.ToYieldInstruction();
```

```
await handle;
```

Safe Mode

To display exceptions within a tween as warnings, similar to DOTween's Safe Mode, you can configure LitMotion as follows:

```
// Log caught exceptions as warnings in the console
```

```
MotionDispatcher.RegisterUnhandledExceptionHandler(ex => Debug.LogWarning(ex));
```

Unsupported Features

DelayedCall

There is no direct equivalent of DOTween.DelayedCall() in LitMotion. For more details, refer to the [FAQ](#).

SetSpeedBased

LitMotion does not support a SetSpeedBased() equivalent. You should calculate the required duration based on the distance between the start and end points.

DoPath

There is no equivalent to transform.DoPath() in LitMotion. However, you can achieve similar functionality by combining LitMotion with Unity's [Splines](#) package.