TpTweener

The TilePlus system optimized Tweener for Tiles, GameObjects, and custom uses.

Tweener Service

Intro

TpTweener is a TPT service that can be used for tweening TilePlus tile sprites, GameObjects; just about anything. The Tweener also has support for sequences. Tweens and Sequences can be Awaited or used from Coroutines.

When used with TilePlus tiles, tweens are used within TilePlus tile code, ZoneActions, or EventActions. The main restriction is that a TPT tile reference must be provided when creating Tweens or Sequences for TPT tiles.

When used with GameObjects you can use built-in methods to tween GameObject position, rotation, scale, or Color as well as a method to tween a GameObject position along a Bezier path.

To tween anything else, you create a small class containing start and end values, and some custom code that is called when the tween is evaluated. The custom code uses the 'small class' instance to:

- Reference whatever it is that's being tweened
- Update a current value for diagnostic use.

You can see how this works in the provided examples in the next few pages. It's pretty easy to extend the Tweener service.

Careful null-checking ensures that if a tile or GameObject is destroyed then the tween and/or its parent sequence is auto-killed. This is especially important when used with the TilePlus Layout system where tiles and prefabs are frequently added and destroyed (or restored to a pool).

Important

TpLibInit (`Runtime/Resources/Tp`) has a checkbox `Tweener Safe Mode`.

When checked, additional null checks are used in the Tweener. Advised to leave on.

- If this is not checked and a tweened object becomes null there can be an exception.
- Primary reason to uncheck this is for profiling.

Why Another Tweener?

But why create another Tweener in the first place? The Unity Asset store is full of free Tweeners on the Unity Asset Store and DemiGiant's DOTween is really great.

Here's why: none of them support natively tweening Tile sprites.

- You can do it in DOTween with Getters and Setters but DOTween can't handle tiles suddenly becoming null (if they're deleted) in the same way as it does for GameObjects.
- TilePlus Toolkit has a 'DOTween Adapter' which handles some of that, but it's inconvenient (and deprecated).
- DOTween is great but is way more complex than needed just for tiles.

TpTweener is non-generic and hard-coded, but can tween anything. It doesn't have a lot of frills such as the ability to pause tweens or start with a delay (easy to do with a sequence): low GC pressure and speed were the most important design considerations.

Unlike most other Tweeners, TweenerService Tweens and Sequences can be Awaited. Coroutine support is trivial, read about it here.

It's not supposed to be a substitute for DOTween but you can make it do almost anything that DOTWeen does with some custom code.

For Tiles

You use it by first obtaining a service handle from `protected static TpTweener TweenerService` property within any TPT tile code.

For example:

```
var tId = TweenerService.CreateTween(this,
                    Vector3.zero,
                    Color.red,
                    Matrix4x4.identity,
                    TpEasingFunction.Ease.Linear,
                    TpTweener.EaseTarget.Color,
                    1,
                    0,
                    Tpweener.LoopType.PingPong,
                    -1,
```

```
                    OnFinished);
```

This creates and starts a Color tween (EaseTarget.Color) with the final color being Color.red. The return value is a `long` which is used as an ID or 'handle.'

Although the Linear easing function is specified here, it's ALWAYS linear (Lerp) for Color tweens (the value is ignored for Color tweens).

The duration is one second. The -1 means that this tween repeats a PingPong tween until killed. When the tween is killed then the OnFinished callback is invoked. When you use `PingPong` for tweens, the duration value is used for each phase, hence, a two second duration for a tween running in ping-pong mode takes a total of 4 seconds to complete per loop.

The first four parameters are a reference to the tile itself (this), a Vector3 value, a Color value and a Matrix value. For the Vector3, Color, and Matrix parameters you just use the one you need for a particular tween.

Here, the Vector3 field is set to Vector3.zero and the Matrix field is set to Matrix4x4.identity because this is a Color tween.

The Vector3 field is used for tweens that use Vector3 values like Position, Rotation, or Scale.

The Matrix parameter is used for Matrix tweens which are a special tween variety.

Demos

TilePlus Extras has a TpTweener folder with a few example tiles and several scenes.

- TweenerFlexTile has fields for all possible types of tweens (except DELAY) and the fields which are shown change with what's being affected
  - I.E., if the tween target is Color then a color picker is available, but for, say, position, a Vector3 field is shown.
- TweenSpecTile uses a TpTweenSpec asset to run a tween from that asset.
- TweenSpecSequenceTile uses all the entries from a TpTweenSpec asset to create and run a sequence. You can also check a toggle for interactive use. This allows you to tweak the TweenSpec asset while the Editor is playing: each time the sequence ends it re-loads from the asset rather than internally repeating the cached sequence.

These three tiles are part of the normal distribution in the Plugins folder but for many uses, in a real app, you'd run a tween as the result of a Message being sent to a tile or because of a Zone entry or exit.

However, the stock 'Tweening' tiles are great for your experimentation with this feature; especially TweenSpecSequenceTile.

To use tweens in the code for Event or Zone actions, check out the TpTweenerSubObject. SubObjects are scriptable objects which are attached as references to TpTileZoneActions or TpTileEventActions.

TpTweenerSubObject can be used by a TileAction to easily tween the tile's sprite when an event is handled (EventAction) or when a Zone is entered or exited (ZoneAction.)

This approach is used in the TileFabDemos/LayoutSystem demo.

Tweens and Sequences

Tweens

You can tween Position, Rotation, Scale, and Color of Tile sprites.

For color tweens you can also specify 'Constant A', which means that the alpha value doesn't change from the value found when the tween begins.

You can also tween the Tile's Matrix, or the Matrix with Color, or the Matrix with Color-Constant-A. These let you create a single tween that can change Position, Rotation, Scale and/or Color/Color-Constant-A, with optional fine-grained control over which parts of the transform change. For example, one could specify a Matrix tween where only position and scale are changed.

- Matrix tweens also allow specifying different Ease functions for Position, Rotation, and Scale.
- MatrixColor and ColorConstantA use Lerp for the Color tween.
- Color Tweens always Lerp

It's incredibly flexible, albeit a bit more work to set up.

Don't get confused: if you use one of the Matrix varieties, including those with Color, it's one tween. When the tween updates it can affect Postion, and/or Rotation, and/or Scale, and/or Color in one operation. It's not three or four tweens in parallel.

To that end, the TweenSpecSequenceTile has an interactive mode. In this mode, the sequence is forced to not loop. When the sequence completes, it restarts with a fresh set of value fom the Tween Spec asset. Hence, if you change values of this asset while the editor is Playing, such changes will be seen when the sequence restarts. As is true for project assets, the changes will remain after you exit play mode. It's a great way to play with sequences and Matrix-style tweens.

The tweener supports one-shot, looping, and ping-pong tweens. For looping tweens there's a loop count and as usual if the loop count is -1 the loop continues forever until cancelled or the tile is deleted.

Delay tweens can also be created, but they're only for sequences.

Sequences

A sequence can be created and tweens can be created with automatic adding to a sequence. Sequences do what you'd expect: run all the tweens in a list of tweens. Delay tweens can be added to a sequence. They don't tween anything, just time out. Since a callback can be issued each time the sequence changes to the next tween, a sequence of just Delay tweens can be used to create a simple sequencer.

Callbacks

Individual tweens have callbacks for each Update, on completion, and when a looping tween loops. Tweens are auto-deleted when their parent tile becomes null. Since a callback is issued to a tile and the tile is null in this case, the tween-ending callback is not issued.

Note that callbacks for each update can get extremely processor-intensive depending on how many tweens are running and what is done during execution of the callback. For example, if you have 100 tweens running then you'd get 100 callbacks every frame. Hence, these are recommended ONLY during development but there's no explicit restriction on this.

Note that while each callback provides a reference to the specific tween that caused the callback, most of the values in the instance are read-only properties so

1. Do not hold a reference outside the local scope. These are pooled at at the end of the tween: it is returned to the pool and reset.
2. You can't change anything except a "Diagnostics" property.
3. You can't copy a tween and inject it back into the system.

Sequences have callbacks when complete and when the next tween in the sequence begins.

Sequences intercept the individual tween callbacks for the sequence's internal use but don't relay them to tiles.

Create a tween

There are four ways to create and run a tween.

- Use the Create methods
  - CreateTween: create any sort of tween.
  - CreateDelayTween: Don't tween anything, just wait. Sequences only.

    - CreateTweenFromSpec: Use the TpTweenSpec asset to create tweens.
        - TpTweenSpec.Tween are the individual specs in a list of specs in the asset.
    - CreateGoTween: support for Rotation, Position, Scale, and Color tweens for GameObjects.
        - This method supports custom tweens.
        - See the next few pages for more information.

Create methods return a long integer which is the ID of the tween.

Tile Tweening

For tiles, tweens are relative: for example, if you tween position, the value provided as an endpoint is the CHANGE that you want and NOT the absolute position.

For example: Creating a tween that changes Position with an EndValue of Vector3(2.2,3.5,0) doesn't move the Tile's sprite to (2.2,3.5,0). It *offsets* the position by 2.2 units in the X direction and 3.5 units in the Y direction from the tile's position in the tilemap: it's relative to the tile.

So if the tile is at (0,0,0) for this example, the tile's sprite would move to 2.2,3.5,0. If the tile were at (10,15,0) then the tile's sprite moves to (10+2.2, 15+3.5, 0+0) = (12.2,18.5,0).

And that makes a lot of sense in this context: the tile itself doesn't move while you're tweening its color or transform. You're actually affecting the *Tilemap* and how it displays the tile's sprite. Nothing in the tile's data changes at all.

Similarly, if affecting rotation, the change is relative to the existing rotation of the tile's sprite, and if affecting scale, the change is relative to the existing scale of the tile's sprite.

Color tweens are a little different: you're tweening between the color when the tween is launched and an absolute end color specified in the CreateTween method call.

GameObject Tweening

GameObject tweens are normally absolute: for example, if you tween position you provide the initial position and the end position. If you don't specify an initial position, the Tweener uses the current position as the initial position and adjusts the end position, turning the tween into a 'relative' tween.

GameObject tweens have different Create methods but the workflow is similar.

- GameObject tweens can be created from a project asset which defines the tween parameters.
- GameObject tweens can be in sequences.
- Adding GameObject tweens to sequences with Tile tweens or vice versa will cause the tween to be rejected.

See the next section for more information about GameObject tweening.

CreateTweenFromSpec

In a practical application, CreateTweenFromSpec is the most useful. Most likely you'll have several tweens that you'll use repeatedly for many tiles. Note that this doesn't support GameObject tweens.

Rather than have individual tiles have all the fields for specifying tweens (as in TweenTileExample) you use a Tween specification from a TpTweenSpec asset; this can be seen in TweenTileExample2.

You can also use specs from this asset when creating sequences.

Ignoring CreateDelayTween, the Create methods launch the tween immediately. If you need a delay, create a two-element sequence with a DELAY as the first tween in the sequence.

Operations on Tweens

Most of these use the id or 'handle' returned from the `Create` methods.

- A running tween can be cancelled with KillTween.
- Get a reference to a running tween with GetTween. However, since these are pooled items, do not hold this reference outside a local scope or you WILL get memory leaks.
- Get an unpooled copy of a running tween with CopyTweener. Note that this is NOT pooled but is a COPY of the running tween at that point in time.
- Get the ToString of a running tween without affecting the instance with GetTweenInfo.
- Tweens are auto-deleted if their parent tile or the parent tile's Tilemap become null.

Create a sequence.

- Use CreateSequence, add tweens using a Create method, and use PlaySequence to run the sequence.
  - Note that you use one of the input parameters to CreateSequence to indicate if this is a sequence of Tile tweens or of GameObject tweens: they can't be mixed in the same sequence.
- Use CreateSequenceFromSpec and use PlaySequence to run the sequence.

CreateSequenceFromSpec uses the tween specifications in a TpTweenSpec asset and makes a sequence out of the tweens.

- An array of indices into the List of Tweens in a TpTweenSpec asset can be used if you want only specific tweens from the list to be used.

    - Don't do this unless really needed: if you change the TpTweenSpec asset later you have to update the array.
- If the array is null (the default) then the entire list of Tweens is used.

CreateSequence and CreateSequenceFromSpec return a long integer which is the ID of the sequence.

If you're not using CreateSequenceFromSpec:

- Create a sequence using CreateSequence.
- Use CreateTween or CreateDelayTween and supply the ID of the sequence in the method call.
    - When the sequence ID is supplied, the created tween is NOT immediately run, but its data structure (a TpTween instance) is added to the sequence.
- Use PlaySequence to start the sequence.

Operations on Sequences

Just like Tweens, the id/handle retrurned from the Create methods for sequences is used as an identifier for the following:

- Delete an unused sequence with DeletePendingSequence.
- Kill a running sequence with (wait for it...) KillRunningSequence.
- Sequences are auto-deleted if their parent tile or the parent tile's Tilemap become null.

Reset

Reset the tweener with ResetTweener and the Sequencer with ResetSequencer. Both release all active tweens and those which are included in the sequences' lists of tweens.

Observe

- Use the Tween Monitor (menu item). This opens an Editor window which displays all of the information about running tweens and sequences.
    - If you have many tweens running this can slow down your game.
- Use the Services Inspector (menu item). This shows all running services (including this one).

Tweener and GC

When you tween the tile sprite the Tilemap will execute one or more callbacks, with allocations for each one. For example, the Tween demo called "StressTest" runs about 400 tweens and generates about 60 Kb of garbage every Update. Of course this is an extreme example but it's important to bear in mind.

- This is entirely a Tilemap phenomenon.

Tweening GameObjects

Intro

TpTweener can also be used to tween GameObjects' Position, Rotation, Scale, or Color; or to tween a GameObject's position along a Bezier path.

Unlike tweens for TPT tiles, there's no equivalent 'Matrix' tween that can tween all of these at once.

CreateGoTween is the most basic entry point for GameObject tweening:

```
public long CreateGoTween(GameObject?            gameObject,
                          Action<TpTween,float>? gameObjectAction,
                          TpEasingFunction.Ease    ease,
                          float                    duration,
                          long                     sequenceId         = 0L,
                          LoopType                 loopType           = LoopType.None,
                          int                      numLoops           = -1,
                          Action<TpTween>?         onFinished         = null,
                          Action<TpTween, float>?  onRewindOrPingPong = null,
                          Action<TpTween>?         onLoopEnded        = null,
                          Action<TpTween, float>?  onUpdate           = null )
```

This is very flexible: provide a target GameObject and an Action to modify same. The Action gets the Tween instance and a 0 - 1F progress value. The remaining parameters are the same as for TPT tile tweens: ease function, duration, etc.

One can also create a sequence and add GameObject tweens to the sequence just like with TPT tile tweens. However, one cannot mix GameObject and TPT tile tweens in the same sequence: you'll get a 0L return value, indicating an error.

There are also two higher-level methods with preset Actions: GameObjectTrs and GameObjectColor

GameObjectTrs allows transform-related GameObject tweens for the most common use: supply an enum value indicating which component (position, rotation, scale) to affect.

The `startValue` parameter is nullable. If this is null when the method is invoked then the GameObject's transform.position, transform.rotation, or transform.scale is used for `startValue` and this obtained value is added to the `endValue`: that is, the tween is 'relative'.

For example, if:

- `startValue` == null
- `endValue` = Vector3(5,6,7)
- the GameObject's transform.position was (3,4,5)

Then internally:

- `startValue` becomes (3,4,5) (the tranform.position from the GameObject)
- `endValue` becomes (5,6,7) + (3,4,5).
- This can produce strange results when tweening rotation or scale.
- Another way to have greater control is to always supply the startValue but make the tween 'relative' by modifying endValue yourself before calling the method.

If `startValue` isn't null then the tween is absolute:

- If the current position, rotation, or scale doesn't match the value of `startValue` then the current position, rotation, or scale becomes `startValue`
  - For example, in the case of position, the GameObject's position will instantly change when the tween starts.
- `endValue` isn't changed.

```
public long GameObjectTrs(GoEaseTrsTarget trsEaseTarget,
                          GameObject              go,
                          Vector3?                startValue,
                          Vector3                 endValue,
                          float                   duration,
                          TpEasingFunction.Ease   ease,
                          long                    sequenceId         = 0L,
                          LoopType                loopType           = LoopType.None,
                          int                     numLoops           = -1,
                          Action<TpTween>?        onFinished         = null,
                          Action<TpTween, float>? onRewindOrPingPong = null,
                          Action<TpTween>?        onLoopEnded        = null,
                          Action<TpTween, float>? onUpdate           = null)
```

GameObjectColor is for color tweening. The `startValue` parameter is evaluated in the same way: use the value if provided but if null use the current color value.

This method accomodates SpriteRenderers and Renderers. It will use the first one it finds in the GameObject or its children.

```
public long GameObjectColor(GameObject             go,
                            Color?                 startValue,
```

```
                                Color                 endValue,
                                float                   duration,
                                TpEasingFunction.Ease   ease,
                                long                    sequenceId         = 0L,
                                LoopType                loopType           = LoopType.None,
                                int                     numLoops           = -1,
                                Action<TpTween>?        onFinished         = null,
                                Action<TpTween, float>? onRewindOrPingPong = null,
                                Action<TpTween>?        onLoopEnded        = null,
                                Action<TpTween, float>? onUpdate           = null)
```

The Tween instance variables for GameObject-related tweens have public property accessors rather than internal so you can modify them just after creation (as seen in the code for these two methods). Modifying these during the execution of a tween is unsupported and may cause exceptions or other issues.

Like all other Tween instances, GameObject tween instances are pooled. If you hold a reference to a tween instance it will become invalid at some point and may interfere with the pooling operation.

Creating GameObject Tweens from an asset

Similar to TPT tile tweens, you can use an asset `TpGoTweenSpec` to create GameObject tweens or Sequences of same.

There are corresponding methods:

- CreatGoTweenFromSpec: use an individual GameObject tween spec from the list of these in the asset.
- CreateSequenceFromGoSpec: use all or some of the asset's GameObject tween specs to create a sequence.

NOTE: since the 'spec' is a project asset you can't modify its fields without affecting everything that uses this 'spec'. If you want to modify fields in a spec prior to sending it to the Create method you need to clone it first and then make changes to the cloned spec's fields. Then send the cloned instance to the Create method.

NOTE: you can add Delay tweens to the 'spec'. However, as usual Delay tweens can *only* be used in sequences.

Other uses

One can create a tween using CreateGoTween with an Action that modifies other values besides the transform or color.

For example, when using GameObjectTrs to create a Position, Rotation, or Scale GameObject tween, the Action supplied to CreateGoTween actually performs the calculations and transform changes. Instead, you can use the progress value passed to the Action to change, say, the intensity of a Light.

Note that if you just want a periodic timer, use TpLib.InvokeRepeatingUntil instead. It's way more efficient.

An example custom tween is shown below. This is built-in to the Tweener.

There's an example of a custom Tween affecting the character size of a TextMesh. Find it in the Demos folder, and see the next page for a deeper dive.

Bezier Curve Position Tweening

```
public long GameObjectPositionBezierTween(GameObject
go,
                                          Vector3[] points,
                                          float
duration,
                                          long                    sequenceId             =
0L,
                                          LoopType                loopType               =
LoopType.None
                                          int                     numLoops               = -
1,
                                          Action<TpTween>?        onFinished             =
null,
                                          Action<TpTween, float>? onRewindOrPingPong     =
null,
                                          Action<TpTween>?        onLoopEnded            =
null,
                                          Action<TpTween, float>? onUpdate               =
null)
```

This is similar to GameObjectTrs. Provide a set of points and the GameObject's local position will tween between these points.

- Inclusion in sequences is allowed.
- points[0] is the start position.
  - If not the current position of the GameObject then the GameObject will immediately move there.
- points[N] is the end position.
- The size of the points array must be between 2 and 16.
- LoopType.Loop means that the GameObject will jump back from points[N] to points[0] before repeating.
- LoopType.PingPong means that the GameObject will tween back to points[0] from points[N] when repeating.

Custom Tweening

The Tween class' properties are mostly `{get; internal set;}` so you can't make changes. However, there are some deliberately-exposed properties that you can use for custom tweening.

```
///
<summary>

/// The GameObject reference for GameObject
tweens
///
</summary>

public GameObject? GameObject { get; set;
}



///
<summary>

/// The Action for GameObject
tweening
///
</summary>

public Action<TpTween,float>? GameObjectAction { get; set;
}



///
<summary>

/// Custom user data, example: Used for Bezier curve
evaluation.
///
</summary>

public object? UserData { get; set;
```

```
}


///
<summary>

/// Copy of original user data for
rewinds
///
</summary>

public object? OriginalUserData { get; set;
}


///
<summary>

/// The Renderer for Color
Tweens
///
</summary>

public Renderer? Renderer { get; set;
}


///
<summary>

/// The SpriteRenderer for Color
tweens
///
</summary>

public SpriteRenderer? SpriteRenderer { get; set;
}


///
```

```
<summary>

/// TRUE for GameObject tweens using the
SpriteRenderer
///
</summary>

public bool IsSpriteRenderer { get; set;
}
```

These exposed properties let you create custom tweens that don't use the built-in actions. Care must be taken **NEVER EVER** change any of these once a tween is running.

The field and property values of whatever you use for `UserData` are modified during the custom tween's execution. The `UserData` class' ToString method is used to show information in the Services Inspector and the Tween Monitor windows. You can see how this custom data is used below.

As mentioned earlier, tweens created by one of the Create tween methods launch immediately. However, immediately means the next Update. So a new tween can change these public properties right after creation.

Here's an example of a Component that can be attached to a legacy TextMesh component to tween the character size. Obviously this same approach can be used to tween anything that takes a float value. In essence, this approach is similar to using DOTween's 'Getters' and 'Setters' for specialized tweens.

Follow along: additional comments in CAPS

```
[RequireComponent(typeof(TextMesh))]
public class TextFontSizeTweenExample : MonoBehaviour
{
   private class ActionData
   {
      public float StartValue { get; set; } = 0;   //THE START VALUE
      public float EndValue   { get; set; } = 1;   //THE END VALUE

      public float CurrentValue { get; set; }      //CURRENT VALUE

      public TextMesh? TextMesh { get; set; } = null;   //TEXTMESH
```

REFERENCE

```
      /// <inheritdoc />
      public override string ToString()
      {
          return $"Char Size: Current: {CurrentValue}";
      }
   }



    /// <summary>
    /// Min
    /// </summary>
    [Tooltip("Minimum Character size")]
    public float m_MinimumCharSize = 1f;

    /// <summary>
    /// Max
    /// </summary>
    [Tooltip("Maximum Character size")]
    public float m_MaximumCharSize = 1f;

    /// <summary>
    /// duration
    /// </summary>
    [Tooltip("tween duration")]
    public float m_Duration = 2f;

    void Start()
    {
        var textMesh = GetComponent<TextMesh>();
        if (!textMesh)
            return;

        //GET THE TWEENER SERVICE HANDLE
        var tweener          = TpServiceManager.GetServiceOfType<TpTweener>();
        if (!tweener)
            return;
```

```
        //CREATE AND INIT AN INSTANCE OF OUR CUSTOM DATA
        var ld = new ActionData
                 {
                     StartValue   = m_MinimumCharSize,
                     EndValue     = m_MaximumCharSize,
                     TextMesh     = textMesh,
                     CurrentValue = textMesh.characterSize
                 };

        //Create a custom GameObject tween with our own custom GameObjectAction.
        var id = tweener.CreateGoTween(gameObject,
                                       GameObjectAction,
                                       TpEasingFunction.Ease.CustomEase, //this enum value
causes GameObjectAction to be invoked.
                                       m_Duration,
                                       0,
                                       TpTweener.LoopType.PingPong,
                                       -1);



        var tween = tweener.GetTween(id); //note that this would be incorrect if the tween was
part of a sequence.
        //If we have a sequence ID use GetTweenFromSequence.
        //If it's not known then generally use:
        //   tween = sequenceId != 0 ? GetTweenFromSequence(sequenceId, id) : GetTween(id);

        if (tween == null)
            return;

        //ADD CUSTOM DATA REFERENCES TO THE TWEEN
        //Add our data to the tween, it's sent to the Action below.
        tween.UserData                = ld;
        tween.OriginalUserData        = ld;
        return;

        //This action does exactly the same thing as built in actions
        //but we use the User data items.
        //The Tweener handles looping, etc.
```

```
        //One can hook into the OnRewindOrPingPong to customize this to some extent.
        void GameObjectAction(TpTween t, float progress)
        {
            //unboxing
            if(t.UserData is not ActionData d || d.TextMesh == null)
                return;
            //get start and end values for Lerp
            var sValue = t.Forward ? d.StartValue : d.EndValue;
            var eValue = t.Forward ? d.EndValue : d.StartValue;

            //compute a new value
            var newValue = Mathf.Lerp(sValue, eValue, progress);

            d.CurrentValue = newValue;
            d.TextMesh.characterSize  = newValue;
            //THE TARGET FOR THIS TWEENED VALUE CAN BE ANYTHING THAT TAKES A FLOAT, JUST
CHANGE THE ACTIONDATA CLASS.
        }
    }


}
```

Other examples

See the TilePlusExtras/SimpleDemos/TpTweener/GameObjectExamples project folder.

Future Expansions

Additional built-in tweens for various Unity 'things' will be added over time. But they are really easy to create on your own, as seen here.

Coroutines and Awaitables

Coroutines

TilePlus generally uses Awaitables rather than coroutines. However, it's easy to make a coroutine wait for a Tween or Sequence to complete:

```
var tId = TweenerService.CreateTween(.....);


yield return new WaitUntil(() => !TweenerService.IsRunningTween(tId));
```

Here we just poll TweenerService to see if the tween we created is still running. The Coroutine waits until IsRunningTween returns false.

However, if all you want is to know when the tween or sequence completes, just use one of the 'Finished' callbacks and save the coroutine overhead.

Important: You'll note below that Awaitable tweens and sequences can't have infinite loop counts and will fail if the loop count value passed to the create method is < 0. A similar situation exists for using Coroutines: if the loop count is infinite then the coroutine won't ever terminate. However, you need to handle this yourself or wrap the create method so that that circumstance is avoided.

Awaitable

Tweens and Sequences can be wrapped into `Awaitable` methods. The main restriction is that one cannot have Awaitable tweens or sequences with infinite loop counts (i.e., < 0).

Examples of how to use this feature are part of the TweenerFlex and TweenSpecSequence tiles.

Here's a bit of the code for the TweenerFlex tile:

```
private async Awaitable WaitForCompletion(long tId)
{
    if (m_LoopCount < 0)
    {
        //NOTE: The TweenAsAwaitable call below will fail if the loop count is < 0
        Debug.Log("infinite loop count tweens should not be awaited!");
```

```
        return;
    }
    var at = TweenerService.TweenAsAwaitable(tId);
    if (at == null)
        return;
    //The elapsed time calculation is obviously not required.
    var now = Time.time;
    var s   = TweenerService.GetTweenInfo(tId);

    await at;
    Debug.Log($"Awaitable tween complete, ET: {Time.time-now}. Id = {tId}\n{s}");
}
```

A simpler version (untested) could be:

```
private async Awaitable WaitForCompletion(long tId)
{
    var at = TweenerService.TweenAsAwaitable(tId);
    if (at == null)
        return;
    await at;
}
```

Since it's almost impossible for the TweenerService to be null, one could do this:

```
private async Awaitable WaitForCompletion(long tId)
{
    await TweenerService.TweenAsAwaitable(tId);
}
```

Or inline that type of statement in any async method that needs it.

It's pretty simple to use Awaitable tweens. Awaitable sequences are very similar, but you'll note that the Awaitable method is cancelled if the Sequence can't launch for some reason.

TpLibTasks has this helper method for Awaitables:

`public static async Awaitable WhenAll(List<Awaitable> awaitables)`

Using this method one can run a number of tweens and Await for them ALL to complete.