Introduction

TMPEffects is a tool for Unity that allows you to easily apply many different kinds of effects to your text. It consists of two main components:

- TMPAnimator allows you to animate text over time
- TMPWriter allows you to show and hide text over time, as well as execute commands or raise events at any given index

Using both components in conjunction also allows you to apply special animations to text that is in the process of being shown or hidden.

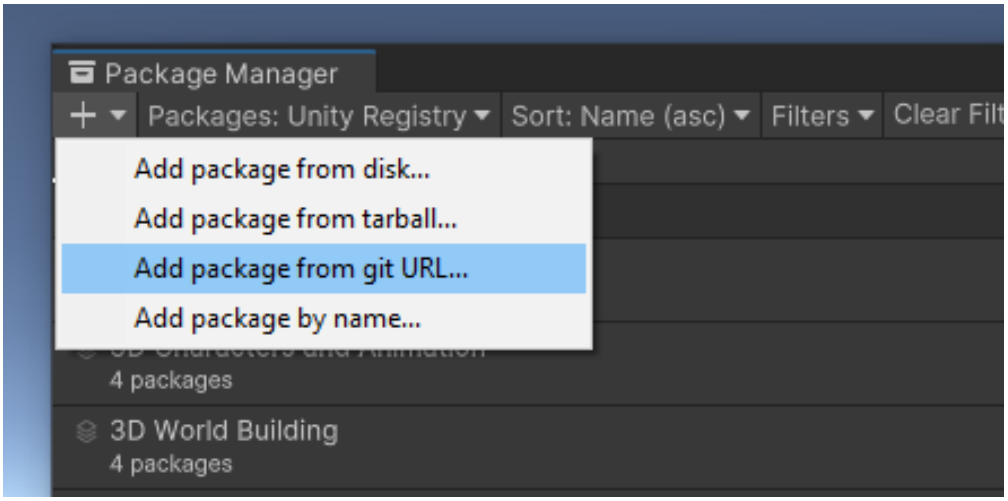
Installation

TMPEffects is available on the OpenUPM registry.

Alternatively, you can install TMPEffects through the Unity Package Manager, using the git url:

```
https://github.com/Luca3317/TMPEffects.git
```

For that, open the Package Manager window in Unity and click on the '+' icon in the top left.



Then, paste the above git url and hit enter. Done!

Importing Resources

Once installed, when you first access a TMPEffects component, you will be prompted to import the required resources.
These are the asset files for the built-in animations, commands and databases, as well as a settings file (which you can access through Unity's `Preferences`). They will be imported into `Assets/TMPEffects`.

⚠ Ideally, you should reload the current scene once you imported these resources, otherwise some asset references might not be correctly updated until the next domain reload.

After that, you should be good to go 😃
Optionally, there are a few basic samples you can import.

TMPAnimator

TMPAnimator is one of the two main components of TMPEffects, along with TMPWriter. Primarily, it allows you to

Getting started with TMPAnimator

After adding TMPEffects to your project, add a TMPAnimator component to a GameObject with a TMP_Text component (either TextMeshPro - Text or TextMeshPro - Text (UI)).

Applying your first animation

Write some placeholder text in the TextMeshPro's component textbox. Analogous to TextMeshPro's built-in rich text tags (e.g. <color>, <s>), you can add animations to your text by simply adding animation tags. Try adding `<wave>` before your placeholder text, and then hitting the `Toggle Preview` button in the TMPAnimator's inspector. In the scene and game view, you should now see that your text is being animated. It should look something like this:

⚠ If the `<wave>` tag is still visible in the scene / game view, the tag is not being correctly processed. Make sure to use the default database by toggling `Use default database` in the TMPAnimator inspector's `Animations` foldout.

You can close the animation using `</wave>`. Only text between the opening and closing tag is animated.

Modifying the animation

Optionally, you can pass various parameter to animation tags. For example, the `<wave>` tag supports `amplitude` and `uniformity`, among others. You could modify the tag like so: `<wave amplitude=10 uniformity=0.5>`, which should result in something like this:

Try to play around with these two values to get something you like! Theres a lot more customization you can apply for almost all animation tags; a complete overview of all tags and their respective parameters can be found in Built-in animations.

Stacking animations

First, close the `<wave>` if you haven't already. After the closing tag, add another tag, `<palette>`, as well as some text after this tag that will be animated by it. Your text should look like this now: "<wave> *Text* </wave> <palette> *More text* </palette>", although you will notice the last closing tag is completely optional in this case. The animated text should look like this:

If you now remove the closing `</wave>` tag, the second text will be animated by both tags (assuming the `Animations override` toggle in the `Animator settings` foldout is set to false, which it will be by default). It should look something like this:

In this manner, you can stack a (theoretically) limitless amount of animations. Of course, there's no guarantee all combinations will mesh well together 😉

If you switch the `Animations override` toggle in the TMPAnimator's inspector to true, the second text will be animated as before. This toggle only defines the default behavior of animation tags; you can decide whether a tag should override the previous tags individually by adding the `override` parameter to a tag, like so: `<palette override=true>`. All animation tags support this parameter.

Late animations / second pass

Another parameter supported by all animation tags is `late`, and is used like so: `<wave late>`. If set, the animation will be applied in a second pass within the TMPAnimator, meaning it will be applied after all animations that do not have this parameter.

You will not need this parameter in the large majority of cases. It is useful primarily for when you need the mesh data of the character to consider the changes made by the other animations. For example, the flashlight effect shown below needs the `late` parameter to work correctly, as it operates on the vertex positions of the characters. If it was applied before the wave animation, the flashlight would use the incorrect, initial vertex positions.

More information about how animations work (and how to create your own) can be found in Creating Animations.

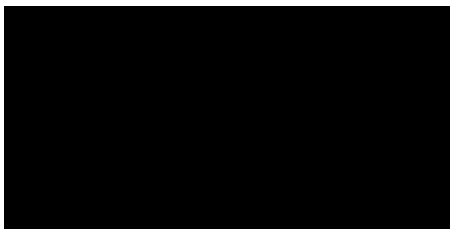
Built-in animations

This section gives you a complete overview of all built-in basic animations and their parameters (for show / hide / scene animations see the respective sections). Basic animations are those type of animation seen in the previous section, which animate a piece of text continuously over time.

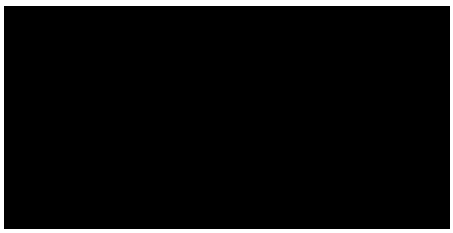
▸ WAVE PARAMETERS

▸ FADE PARAMETERS

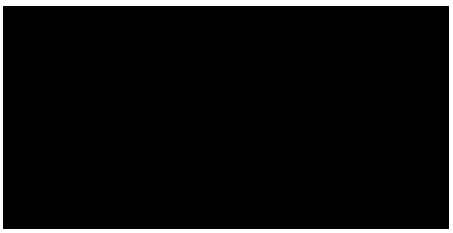
▸ PIVOT PARAMETERS



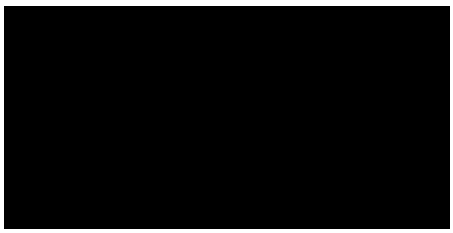
▸ FUNKY PARAMETERS



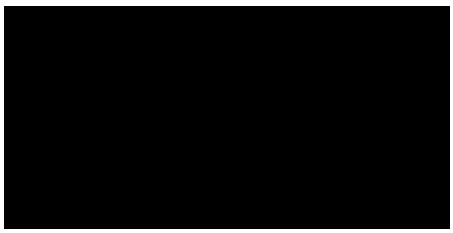
▸ FADE PARAMETERS



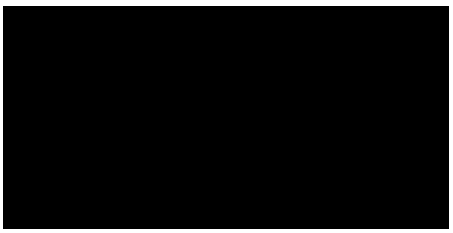
▸ SHAKE PARAMETERS



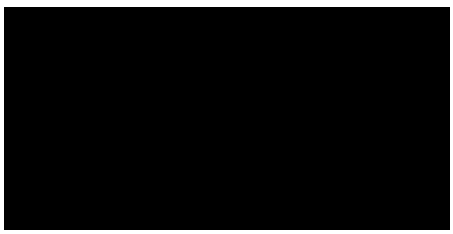
▸ GROW PARAMETERS



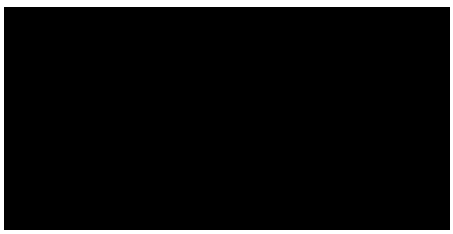
▸ FADE PARAMETERS

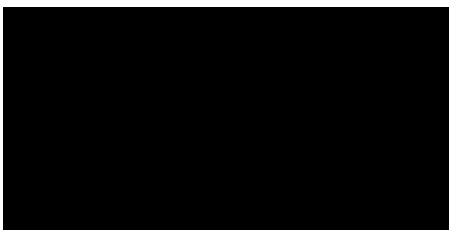


▸ SPREAD PARAMETERS



▸ PIVOTC PARAMETERS





`<swing>` and `<jump>` are based on previous animations; they use the same code with different default values. `<swing>` is based on `<pivot>`, `<jump>` is based on `<wave>`. The parameters are therefore identical with the ones they are based on.

TMPAnimator overview

This section gives an overview of the actual TMPAnimator component, both for the inspector and scripting. The full API documentation can be found here.

Preview

To toggle the editor preview of animations, press the `TogglePreview` at the top of the TMPAnimator inspector. Next to it, the button labeled `Reset time` resets the time tracked by the TMPAnimator component, and therefore all animations.

Updating the animations

In the inspector or through the `SetUpdateFrom` method, you can set how the animations are updated.

If `UpdateFrom` is set to either `Update`, `LateUpdate` or `FixedUpdate`, the animations are automatically updated in the respective Unity callback. In order to play animations in play mode, you will have to call `StartAnimating` or set the `Play On Start` to true in either the inspector or some other script's `Awake` function. You can then stop animating again by simply calling `StopAnimating`.

Alternatively, if you want more fine-tuned control over when and how often animations are updated, for example if you want to limit the updates per second to at most 300, you can set the TMPAnimator's `UpdateFrom` to `Script`. This causes the animations to no longer be updated automatically; instead you may call `UpdateAnimations(float deltaTime)` manually whenever you like.

Note that if `UpdateFrom` is set to `Script`, you should not call `StartAnimating` or `StopAnimating`, since this will have no effect besides logging a warning to Unity's console. Vice versa, if `UpdateFrom` is set to be automatically updated, you should not call `UpdateAnimations(float deltaTime)`; it again does nothing but log a warning.

The state of `UpdateFrom` has no effect on the editor preview.

Animation databases

The TMPAnimator inspector has a foldout labeled `Animations`. There, you may choose the animation database that is used to process animation tags from the TextMeshPro component's text. If you toggle `Use default database` to true, the default animation database is automatically selected for you. The default database is defined in the TMPEffects preferences file. You can also set the database used by the TMPAnimator component through script, using the `SetDatabase(TMPAnimationDatabase db)` method.

Below the database, there are three other fields: `SceneAnimations`, `SceneShowAnimations` and `SceneHideAnimations`. These are simply dictionaries that allow you to map tag names to SceneAnimations. Tags defined here are also parsed by the TMPAnimator.

For more about databases, see Databases. For more about SceneAnimations, see SceneAnimations.

Animator settings

TMPAnimator has various settings that modify the way it animates its text. Each of these is settable through both the inspector and through script.

- Animations override:
  The default override behavior for all animation tags. If true, each tag overrides any of its category (basic / show / hide) that came before it, and only that one is applied. Otherwise, animations are stacked by default. Each tag can manually define its override behavior by using the `override` (shorthand: `or`) parameter.

- Default show / hide string:
  Allows you to define a default show / hide animation that is used for the entirety of the text, if no other show / hide animation tag effects it. Set this like you would add any tag to your text, e.g. `<+fade dur=0.65 anc=a:bottom>>`, `<-spread crv=easeinoutsine>>`.

- Exclusions:
  For each of the animationg categories (basic / show / hide), you can define a set of characters that is excluded from all animations. For example, if you don't want numbers to be animated, you could set `Excluded Characters` to "1234567890". In addition to this, there is an `Exclude Punctuation` toggle for each of the categories.

- Scale animations:
  Defines whether animations should be scaled to the font size property of the TMP_Text component. If true, animations will look identical regardless of font size.

- Use scaled time:
  Defines whether animations should use scaled time or not.

Adding & removing tags through script

The TMPAnimator class exposes four different TagCollections: `BasicTags`, which contains all parsed basic animation tags, `ShowTags`, which contains all parsed show animation tags and `HideTags`, which contains all parsed hide animation tags. Additionally, `Tags` is the union of the other three collections.

For each of the `TagCollections`, you may freely add and remove tags at any point.

Show / hide animations

In addition to basic animations, which are applied continuously, there are also show animations and hide animations, which will be applied only when the effected text is in the process of being shown / hidden.

Applying show / hide animations

Generally speaking, both show and hide animations require you to add a TMPWriter component to the same GameObject as your TMPAnimator.
(You could also write a custom script to show and hide the text in the manner required for show / hide animations to take effect, using TMPAnimator's or TMPWriter's `Show/Hide` methods.)

Show and hide animations are applied in much the same way as basic animations are; in your TMP_Text component, simply add the supported show / hide animation tags like you would regular TextMeshPro tags. Show animation tags are prefixed with a '+', for example `<+fade>`. Hide animation tags are prefixed with a '-', for example `<-move>`. For both, the corresponding closing tag must also include the prefix, after the slash: `</+fade>`, `</-move>`.

So, for example, the string "<+fade><-move>My placeholder text" would animate the text like this:

If you want to try out recreating this animation, you will have to add a TMPWriter component your TMPAnimator GameObject, and set it up like described in TMPWriter.

Setting parameters and stacking show / hide animations works completely analogous to basic animations.

Built-in animations

This section gives you a complete overview of all built-in show and hide animations and their parameters (for basic / scene animations see the respective sections). Each of the animations listed here has both a show and a hide version.

▸ FADE PARAMETERS

▸ PIVOT PARAMETERS

▸ GROW PARAMETERS

▸ SPREAD PARAMETERS

▸ MOVE PARAMETERS

▸ SHAKE PARAMETERS

▸ CHAR PARAMETERS

Scene animations

In addition to `TMPAnimation`, `TMPShowAnimation` and `TMPHideAnimation`, which the animations we've seen so far derive from and which are stored on disk, there is also a `SceneAnimation` equivalent for each. These are alternative versions of the base classes for the respective category which derive from Unity's MonoBehaviour. The primary purpose of them is to allow you to easily reference Scene objects.

Adding scene animations

When you have a GameObject with a `SceneAnimation` component on it, you can simply drag it into the corresponding dictionary in TMPAnimator's `Animations` foldout, the same way you would assign any scene reference. Once you did that, enter a fitting name for the animation in the field next to where you dragged the `SceneAnimation`. That's it! You can now use the animation in your text through a tag like any of the built-in animations.

Applying scene animations

Scene animations are applied in the exact same way as their respective counterpart: Scene animation tags can be directly inserted into the text, where scene show animations are prefixed with a '+', scene hide animations with a '-', and basic scene animation tags are not prefixed.

Creating scene animations

For information about how to create scene animations, see Creating animations.

Creating animations

This section walks you through creating your own animations, specifically basic animations, or animations that derive from `TMPAnimation`. For what small differences there are, see Creating show / hide animation and Creating scene animations.

Creating the class

First, create a new C# script in the Unity editor. Add the following using statement at the top of your class: `using TMPEffects.TMPAnimations;`.
Then, make the created class derive from `TMPAnimation`. In order to be able to create the animation object in the Unity editor and add it to your database, make sure to decorate the class with the CreateAssetMenu attribute.

Methods

You will have errors due to `TMPAnimation's` abstract members not being implemented. Auto-implement them using your IDE, or add them manually. When you are done, your class should look something like this:

```
using UnityEngine;
using TMPEffects.TMPAnimations;

[CreateAssetMenu(fileName="new YourFirstAnimation",
menuName="Your/Path/YourFirstAnimation")]
public class YourFirstAnimation : TMPAnimation
{
    public override void Animate(CharData cData, IAnimationContext context)
    {
        throw new System.NotImplementedException();
    }

    public override bool ValidateParameters(IDictionary<string, string> parameters)
    {
        throw new System.NotImplementedException();
    }

    public override object GetNewCustomData()
    {
        throw new System.NotImplementedException();
    }

    public override void SetParameters(object customData, IDictionary<string,
string> parameters)
    {
```

```
        throw new System.NotImplementedException();
    }
}
```

Let's go over each method individually.

Animate(CharData cData, IAnimationContext context) : The primary method of your animation. This method will be called each animation update, once for each animated character. We'll go into more detail about this method in the next section.

ValidateParameters(IDictionary<string, string>) : This method is called once during tag processing. It allows you to specify whether a given tag for this animation has valid parameters. ParameterUtility will come in handy here. Return true if the parameters are valid, return false if not. If false, the tag will not be processed.

GetNewCustomData() : Allows you to create a piece of custom data that will be passed into `Animate` as part of the `IAnimationContext`. Used for storing parameters, keeping other consistent values (for example, create an RNG once and store it here instead of creating it every `Animate` call), and anything else you need. In here, you should also set the default values for the parameters defined in the inspector.

SetParameters(object customData, IDictionary<string, string>) : This method is called once, right after tag processing is done. The passed in `customData` object is the object you created and returned in `GetNewCustomData`. It allows you to store the tag parameters in your object and access them in `Animate`.

Full example

The code below is the full implementation of the built-in `wave` animation.
If the code seems somewhat daunting don't worry; you will have to have looked at AnimationUtility, ParameterUtility, and Animating a character to fully get what's going on here 🙂

```
using System.Collections.Generic;
using UnityEngine;
using TMPEffects.CharacterData;
using static TMPEffects.ParameterUtility;
using static TMPEffects.TMPAnimations.AnimationUtility;
using TMPEffects.Extensions;

namespace TMPEffects.TMPAnimations.Animations
{
    [CreateAssetMenu(fileName = "new WaveAnimation", menuName
= "TMPEffects/Animations/Wave")]
    public class WaveAnimation : TMPAnimation
    {
```

```
        [Tooltip("The wave that defines the behavior of this animation. No prefix.\nFor more
information about Wave, see the section on it in the documentation.")]
        [SerializeField] Wave wave = new Wave(AnimationCurveUtility.EaseInOutSine(),
AnimationCurveUtility.EaseInOutSine(), 0.5f, 0.5f, 1f, 1f, 0.2f);
        [Tooltip("The way the offset for the wave is calculated.\nFor more information
about Wave, see the section on it in the documentation.\nAliases: waveoffset, woffset,
waveoff, woff")]
        [SerializeField] WaveOffsetType waveOffsetType = WaveOffsetType.XPos;

        // Animate the character
        public override void Animate(CharData cData, IAnimationContext context)
        {
            // Cast your custom data object to the type
            Data data = (Data)context.customData;

            // Evaluate the wave data structure at the current time, with the characters
offset (see AnimationUtility section for info on this)
            float eval = data.wave.Evaluate(context.animatorContext.PassedTime,
GetWaveOffset(cData, context, data.waveOffsetType)).Item1;

            // Set the new position of the character
            cData.SetPosition(cData.info.initialPosition + Vector3.up * eval);
        }

        // Validate the tag's parameters
        public override bool ValidateParameters(IDictionary<string, string> parameters)
        {
            // If there is no parameters, return true (wave does not have any
required parameters)
            if (parameters == null) return true;

            // If there is a parameter "waveoffset" (or one of its aliases)
            // but it has the wrong type, return false
            if (HasNonWaveOffsetParameter(parameters, "waveoffset", WaveOffsetAliases))
return false;

            // If the wave parameters could not be validated, return false
            // Note: "WaveParameters" does not refer to anything specific to
"WaveAnimation" here.
            // WaveParameters is a predefined parameter bundle in ParameterUtility. See the
section on it for more info.
            if (!ValidateWaveParameters(parameters)) return false;

            // else return true
            return true;
        }
```

```
        // Create the custom data object, set the default values for the parameters, and
return it
        public override object GetNewCustomData()
        {
            return new Data() { wave = this.wave, waveOffsetType = this.waveOffsetType };
        }

        // Set the parameters defined in the tag
        public override void SetParameters(object customData, IDictionary<string,
string> parameters)
        {
            // If there is no parameters, return early
            if (parameters == null) return;

            // Cast your custom data object to the type
            Data data = (Data)customData;

            // If has the waveoffset parameter set it in your custom data object
            if (TryGetWaveOffsetParameter(out var wot, parameters, "waveoffset",
WaveOffsetAliases)) data.waveOffsetType = wot;

            // Set the wave in your custom data object
            // As with ValidateWaveParameters, "Wave" refers to the parameter bundle
            // defined in ParameterUtility, not "WaveAnimation".
            data.wave = CreateWave(this.wave, GetWaveParameters(parameters));
        }

        // The class used to store the parameter values
        private class Data
        {
            public Wave wave;
            public WaveOffsetType waveOffsetType;
        }
    }
}
```

Adding the animation to a database

To actually use the animation in your text, you will have to follow these steps:

1. Create an animation object: Right click in your project view and create it (it will be in the path you specified in the CreateAssetMenu attribute).
2. Add that object to the database you want to use and give it a name
3. Use that database in the TMPAnimator component

Done! You can now use your custom animation like any of the built-in ones.

Creating show / hide animations

Creating show and hide animations works 99% the same as creating basic animations. The only differences are:

- Instead of deriving from `TMPAnimation`, you must derive from `TMPShowAnimation` or `TMPHideAnimation` respectively.
- ⚠ You HAVE to call `context.FinishAnimation(cData)` at some point in the animation; This will notify the animator that this show animation is finished, and the character may transition from the `Showing` state to the `Shown` state. Because of this, all built-in show and hide animations have a `duration` parameter, and `context.FinishAnimation(cData);` is called when that duration is exceeded. See the example below for a simple way to do that.

```
public void Animate(CharData cData, IAnimationContext context)
{
    ReadOnlyAnimatorContext ac = context.animatorContext;
    Data d = context.customData as Data;

    // Check if the difference between the time that passed since the animator started
    // playing and the time the character entered the SHOWING state exceeds the duration
    if (ac.PassedTime - ac.StateTime(cData) >= d.duration)
    {
        context.FinishAnimation(cData);
        return;
    }

    // Actual animation logic here...
}
```

Creating scene animations

Creating a scene animation, scene show animation or scene hide animation is almost the exact same as creating a basic animation, show animation or hide animation; the only difference is that you will have to derive from `TMPSceneAnimation`, `TMPSceneShowAnimation` or `TMPSceneHideAnimation` respectively, and that you don't add the CreateAssetMenu attribute. Since it is not a ScriptableObject, you of course don't add it to a database either; instead, you add it as a component to a GameObject in your scene, and add that GameObject to your TMPAnimator as described here: Adding scene animation.

⚠ The `context.FinishAnimation(cData)` call is required for `TMPSceneShowAnimation` and `TMPSceneHideAnimation` as well.

Animating a character

This section guides you through how you animate characters in TMPEffects.

As shown in Creating animations, the `Animate` method takes two parameters:

- CharData
- IAnimationContext

For general information about these types, see the respective documentation.

Applying transformations to a character

Moving the character

To move the character, simply use the `SetPosition(Vector3 position)` or `AddPositionDelta(Vector3 delta)` method on `CharData`.
Most of the time, you will want to do this using the original position of the character and an offset.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Move the character up 125 units over time, then down again; indefinitely
    float val = Mathf.PingPong(context.animatorContext.PassedTime * 50, 125);
    cData.SetPosition(cData.InitialPosition + Vector3.up * val);
}
```

Rotating the character

To rotate the character, use `CharData's SetRotation(Quaternion rotation)` method. If you want to rotate around a specific pivot, you may set it using either the `SetPivot(Vector3 pivot)` method or the

`AddPivotDelta(Vector3 delta)` method.

If you don't set a pivot for the rotation, it will rotate around the center of the character.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Rotate the character indefinitely in the z axis over time
    float angle = context.animatorContext.PassedTime * 50 % 360;
    cData.SetRotation(Quaternion.Euler(0, 0, angle));

    // And by adding this line, it will use the pivot you set;
    // in this case the character will rotate around the point 150 units from its
    // center on the x axis
    cData.AddPivotDelta(Vector3.right * 150);
}
```

Scaling the character

To scale the character, use the `SetScale(Vector3 scale)` method.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Ping-pong the scale between (0, 0, 0) and (1, 1, 1) over time
    float val = Mathf.PingPong(context.animatorContext.PassedTime, 1);
    Vector3 scale = Vector3.one * val;
    cData.SetScale(scale);
}
```

Modifying a character's vertices

Setting the vertices' positions

You can set the positions of the character's vertices using the `SetPosition(int i, Vector3 value)` method on the `VertexData` type. The integer specifies the vertex to modify (again, see CharData) while the vector specifies the new position.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Pingpong the magnitude of the offset between 0 and 125 over time,
    // then add that offset to the two top vertices
    float val = Mathf.PingPong(context.animatorContext.PassedTime * 50, 125);
    for (int i = 1; i < 3; i++)
    {
        cData.mesh.SetPosition(i, cData.initialMesh.GetPosition(i) + Vector3.up * val);
    }
}
```

Setting the vertices' colors

Using the `SetColor(int i, Color32 value, bool ignoreAlpha)` method, you can set the color value of each vertex.
If you pass `true` for `ignoreAlpha`, only the RGB color channels are overwritten; the alpha channel will remain unchanged.
If you want to do the opposite of this and only set the alpha channel, then you can use the `SetAlpha(int i, float alpha)` method.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Set each vertex color to red and set the alpha dependent on passed time.
    Color32 color = Color.red;
    float alpha = Mathf.PingPong(context.animatorContext.PassedTime * 125, 255);
    for (int i = 0; i < 4; i++)
    {
        cData.mesh.SetColor(i, color, true);
        cData.mesh.SetAlpha(i, alpha);
    }
}
```

Setting the vertices' UVs

Using the `SetUV0(int i, Vector2 uv)` and `SetUV2(int i, Vector2 uv)` methods, you can set the `UV0` and `UV2` values of each vertex respectively.

These properties are more niche compared to the other ones, and you will likely use them much less; `char` is the only built-in animation to utilize this property.

```
public void Animate(CharData cData, IAnimationContext context)
{
    // Pan the UV0 of the character over time
    Vector2 delta = Vector2.right * context.animatorContext.PassedTime;
    for (int i = 0; i &lt; 4; i++)
    {
        cData.mesh.SetUV0(i, cData.initialMesh.GetUV0(i) + delta);
    }
}
```

Making your animation fancier

The above character transformations and vertex modifiers are all you really need to animate your character!
Combined with easing functions (see: AnimationCurveUtility, Waves), even these really simple animations above can look quite nice already.

For example, the animation code for the built-in `jump` animation is hardly more complicated than that for the first animation on this page (at least once you've looked at Waves 😉).

```
public override void Animate(CharData cData, IAnimationContext context)
{
    Data data = (Data)context.customData;

    float eval = data.wave.Evaluate(context.animatorContext.PassedTime, GetWaveOffset(cData,
context, data.waveOffsetType)).Item1;
    cData.SetPosition(cData.InitialPosition + Vector3.up * eval);
}
```

AnimationUtility

`AnimationUtility` is a static utility class to be used with all animation types. The full API docs can be found here.

Raw Positions & Deltas

One of TMPAnimator's settings is a toggle that decides whether animations should be scaled or not (see TMPAnimator Settings).
In some cases, you will want to ignore this scaling in your animation though. For example, the built-in `spread` animation sets the individual vertices of the character to the center point of the character, to make it invisible at first and then over time spread out from the center point to the original vertex positions. If this was scaled, then the vertices would in many cases move either not enough, or too far, to make the character invisible.
These methods allow you to get and set positions and deltas that ignore the scaling of the animator:

Getters

`Vector3 GetRawVertex(int index, Vector3 position, CharData cData, ref IAnimationContext ctx)` - Calculate the raw version of the passed in vertex position, i.e. the one that will ignore the animator's scaling.

`Vector3 GetRawPosition(Vector3 position, CharData cData, ref IAnimationContext ctx)` - Calculate the raw version of the passed in character position, i.e. the one that will ignore the animator's scaling.

`Vector3 GetRawDelta(Vector3 delta, CharData cData, ref IAnimationContext ctx)` - Calculate the raw version of the passed in delta, i.e. the one that will ignore the animator's scaling.

Setters

`void SetVertexRaw(int index, Vector3 position, CharData cData, IAnimationContext ctx)` - Set the raw position of the vertex at the given index. This position will ignore the animator's scaling.

`void SetPositionRaw(Vector3 position, CharData cData, IAnimationContext ctx)` - Set the raw position of the character. This position will ignore the animator's scaling.

`void AddVertexDeltaRaw(int index, Vector3 delta, CharData cData, IAnimationContext ctx)` - Add a raw delta to the vertex at the given index. This delta will ignore the animator's scaling.

`void AddPositionDeltaRaw(Vector3 delta, CharData cData, IAnimationContext ctx)` - Add a raw delta to the position of the character. This delta will ignore the animator's scaling.

AnchorToPosition

Given a Vector2 that represents an anchor (see ParameterTypes), you can calculate the actual position on the character using the `Vector2 AnchorToPosition(Vector2 anchor, CharData cData)` method.

GetValue

A simple wrapper method that allows you to evaluate an AnimationCurve in any WrapMode.

Wave Utility

The `AnimationUtility` class contains a `Wave` type; for more information about it as well as the `WaveOffsetType` enum, see the next section.

These are the utility methods for the `Wave` type:

Converting functions

There are a few simple, general converting functions (that are not specific to the `Wave` type, but to waves in general):

`float FrequencyToPeriod(float frequency)` - Get the period of a wave from its frequency
`float PeriodToFrequency(float period)` - Get the frequency of a wave from its period
`float WaveLengthVelocityToFrequency(float wavelength, float wavevelocity)` - Get the frequency of a wave from its wavelength and velocity
`float WaveLengthFrequencyToVelocity(float wavelength, float frequency)` - Get the velocity of a wave from its wavelength and frequency
`float WaveVelocityFrequencyToLength(float wavevelocity, float frequency)` - Get the wavelength of a wave from its velocity and frequency

GetWaveOffset

When evaluating a `Wave`, you have to pass in an offset, which is dependent on the current character you are animating as well as the `WaveOffsetType` you are using; the `float GetWaveOffset(CharData cData, IAnimationContext ctx, WaveOffsetType type)` calculates the correct offset for you.

Wave

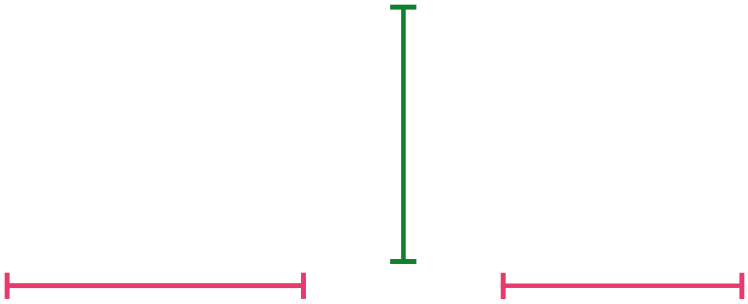
The `Wave` class significantly streamlines the process of creating periodic animations.

Defining a wave

`Waves` are defined by, and created using, eight different parameters, which are as follows:

- **Upward Curve**: Defines the "shape" of the half of the curve that moves up
- **Downward Curve**: Defines the "shape" of the half of the curve that moves down
- **Up Period**: Defines the amount of time it takes to travel the upward curve
- **Down Period**: Defines the amount of time it takes to travel the downward curve
- **Amplitude**: Defines the maximum value the wave reaches; this is `1` by default
- **Crest Wait**: Defines the amount of time the wave maintains its peak
- **Trough Wait**: Defines the amount of time the wave maintains its minimum
- **Uniformity**: Defines how much the offset passed in when evaluating the wave is considered

The following diagram illustrates the first five parameters. On the left you see the upward curve, on the right the downward curve.



The second diagram illustrates crest and trough wait.
Both essentially insert a period of time during which the wave's value does not change.

The uniformity is hard to visualize in a diagram in this way; I find it easier to just imagine it abstractly as the uniformity the animation will show across the animated text segment. Below, you see the same animation with the exact same wave, except that the first animation uses uniformity = 0, the second uniformity = 0.25, the third uniformity = 1.

Creating a wave

You create a wave simply by passing in these eight values into `Waves` constructor.
Since this class is designed specifically to be used with animations though, likely you will mostly be creating it using the `CreateWave` method from the `ParameterUtility` class. For info about how to use wave as a parameter in your animations, see ParameterUtility Wave.

Evaluating a wave

Once you have created a `Wave`, you can evaluate it using a time value and an offset value: `(float, int) Evaluate(float time, float offset)`.
As you can tell by the signature, `Evaluate` returns two values. The float is the actual value of the curve for the given time and offset. The integer indicates whether the curve is currently travelling up or down; if it is negative, the curve is moving down, if it is positive, the curve is moving up. This value is useful for when you want to switch the behavior of your animation depending on what curve the wave is currently travelling.
For example, the built-in `fade` animation can use different anchors depending on whether the character is fading in or out. The animation below visualizes how `fade` uses both values of the `Evaluate` method.

Checking wave extrema

You can check if the wave passed an extrema (so either the crest or the trough) during the last update, using the `int PassedExtrema(float time, float deltaTime, float offset)` method. Once again, you need the time value and the offset, and additionally here you need the deltaTime since you last checked. Essentially, the method checks whether the wave passed an extrema during the time interval of [`timeValue - deltaTime`, `timeValue`]. A positive return value indicates that a maximum was passed, a negative return value indicates that a minimum was passed, and 0 indicates neither was.

An additional, optional parameter is `PulseExtrema`. If the checked wave has a crest or trough wait, this parameter defines whether an extremum is passed once the wait time begins, or once it ends. `PulseExtrema` can also consider both to be an extrema, but of course be aware that it will then notify you of each extremum twice.

If multiple extrema were passed during the specified interval, it will notify you of the latest one.

TMPWriter

TMPWriter is one of the two main components of TMPEffects, along with TMPAnimator. Primarily, it allows you to

as well as execute commands and invoke events when specific indices are reached.

Getting started with TMPWriter

After adding TMPEffects to your project, add a TMPWriter component to a GameObject with a TMP_Text component (either TextMeshPro - Text or TextMeshPro - Text (UI)).

Writing your first text

Write some placeholder text in the TextMeshPro's component textbox. Hit the play button in the preview section of the TMPWriter inspector, or start playing the scene. Your text should be being written, and it should look something like this, with each character appearing one after the other, with a short delay in between:

⚠ If your text is instantly showing all at once, or is writing too slowly, ensure the `Delay` field in the inspector, which defines the delay after showing a character in seconds, is set to a sensible value. The example above uses a delay of 0.075 seconds.

Adding command tags

TMPWriter allows you to easily execute commands when a specific index is reached. You may add them using command tags, prefixed by a '!'. For example, `<!wait=1.5>` will pause the writer for 1.5 seconds before continuing.

A full overview of all built-in command tags is given in the next section.

Adding event tags

In addition to command tags, TMPWriter also supports event tags. TMPWriter exposes multiple UnityEvents, to which you may subscribe in the inspector or through code. One of these events is `TextEvent`. Whenever the writer reaches the index of an event tag, `TextEvent` is raised with the parsed tag as parameter. Unlike command tags or animation tags, you may use any name for event tags, as well as any parameters. Typically, you would use the tag name in the event callbacks to check whether to process the event / tag.
Here are a few example tags: `<?myevent>`, `<?characterspeaking="Faust">`, `<?alert message="*Your message*" priority="warning">`

Animating text appearances

The examples above look pretty boring; you can change the way the text is shown, as well as hidden, by using a TMPAnimator component along with TMPWriter. The example below shows you a few variations.

For info about how to set up the TMPAnimator, see the sections on TMPAnimator.

Getting started with TMPWriter

This section gives you a complete overview of all built-in commands. All of the built-in commands modify the TMPWriter's writing behavior (with the exception of `<!debug="">`). `SceneCommands` allow you to call any method you want.

Wait - Pause the writer for the given amount of time

Parameters: `name` : time in seconds

Example: `I WILL NOW <!wait=1.5>WAIT`

Show - Show the text block from the very start

Parameters: None

Example: `THIS WILL ALWAYS BE <!show>SHOWN</!show>, FROM THE VERY START`

Delay - Set the delay between showing characters

Parameters: `name` : delay in seconds

Example: `<!delay=0.25>I START SLOW... <!delay=0.05>BUT NOW IM FAST`

Skippable - Set whether the text is skippable

Parameters: `name` : `true` / `false`

Example: `<!skippable=true>WHEN IM SKIPPED, I WONT <!skippable=false>GO ALL THE WAY`

Debug - Print a message to the Unity console; you hopefully wont need this much but I decided to leave it in anyway 😄

Parameters: name : Your message type : l(og) / w(arning) / e(rror)

Example: PRINTING TO THE CONSOLE NOW:
<!debug="Test Message" type="warning">

TMPWriter overview

This section gives an overview of the actual TMPWriter component, both for the inspector and scripting. The full API documentation can be found here.

Preview

To preview the writer in editor mode, you can hit the play button in the `Writer preview` section at the top of the TMPWriter inspector. Right next to it, are the buttons for resetting, stopping, and skipping the writer respectively. The progress bar lets you freely skip to any point of the writing process.

The two toggles above the player decide whether events and commands are executed in the editor preview. There is a few things to consider with them:

- Event toggle
  - You will also have to set the actual events you want to raise to `Editor and Runtime`.
  - ⚠ Be careful about which events you allow to be raised in preview mode. Generally I'd recommend setting the event toggle to false completely.
- Command toggle
  - `SceneCommands` are never raised in preview mode.
  - If you create any new commands, you can decide whether it should be raised in preview mode through its `ExecuteInPreview` property.

Controlling the writer

TMPWriter supplies multiple methods to control the writer.

- **StartWriter()**: Starts (or resumes) the writing process
- **StopWriter()**: Stops the writing process
- **ResetWriter()**: Stops the writing process and resets it
- **ResetWriter(int index)**: Stops the writing process and resets it to the given index (must be smaller than the current index of the writer)
- **SkipWriter(bool skipShowAnimations)**: Skips the current text until the next unskippable section, or until the end of the text. Does nothing if the current section is unskippable
- **RestartWriter(bool skipShowAnimations)**: Stops the writing process, resets it and then starts it again

There are also a few method that let you modify the writing process in a more subtle way:

- **Wait(float seconds)**: Wait for the given amount of time until showing the next character; behavior is equivalent to the `wait` tag

- **SetDelay(float seconds)**: Sets the delay used after each character; behavior is equivalent to the `delay` tag
- **SetSkippable(bool skippable)**: Sets whether the current text section is skippable; behavior is equivalent to the `skippable` tag
- **WaitUntil(Func condition)**: Wait until the given condition evaluates to true; ⚠ There is no built-in timeout. It is up to you to ensure the condition wont be false forever / for too long

Default values for the delay as well as the "skippability" of the text can be set in the TMPWriter inspector.

Command databases

The TMPWriter inspector has a foldout labeled `Commands`. There, you may choose the command database that is used to process command tags from the TextMeshPro component's text. If you toggle `Use default database` to true, the default command database is automatically selected for you. The default database is defined in the TMPEffects preferences file. You can also set the database used by the TMPWriter component through script, using the `SetDatabase(TMPCommandDatabase db)` method.

Below the database, there is another field, `SceneCommands`, which is simply a dictionary that allows you to map tag names to SceneCommands. Tags defined here are also parsed by the TMPWriter.

For more about databases, see Databases. For more about SceneCommands, see SceneCommands.

Writer events

Besides the OnTextEvent (see Getting started), there are the following events you may listen to:

- **OnCharacterShown(CharData cData)**: Raised whenever the writer shows a new character; passes the newly shown character
- **OnStartWriter()**: Raised whenever the writing process is started
- **OnStopWriter()**: Raised whenever the writing process is stopped
- **OnResetWriter(int index)**: Raised whenever the writing process is reset; passes the index that was reset to
- **OnResetWriter(int index)**: Raised whenever the writing process is skipped; passes the index that was skipped to
- **OnFinishWriter()**: Raised whenever the writing process is finished, and the whole text is shown

Adding & removing tags through script

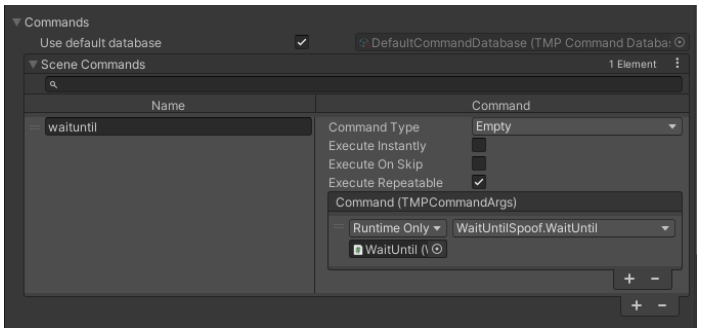
The TMPWriter class exposes three different TagCollections: `CommandTags`, which contains all parsed command tags and `EventTags`, which contains all parsed event tags. Additionally, `Tags` is the union of the other two collections.

For each of the `TagCollections`, you may freely add and remove tags at any point.

Scene commands

In addition to `TMPCommand`, which the commands we've seen so far derive from and which are stored on disk, there is also the `SceneCommand` type. It allows you to call any method of your scene objects.

Adding scene commands



When you add a new element to the `SceneCommands` dictionary found in TMPWriter's `Commands` foldout, you will see an empty field `Name` on the left side. Here you can assign a name to reference the command with. On the right, in UnityEvent field, you can simply drag any GameObject and choose the method to call when this command is invoked. The example creates a `SceneCommand` called "waituntil", which calls a method "WaitUntil(TMPCommandArgs args)".

Using scene commands

`SceneCommands` are applied in the exact same way as their respective counterpart: `SceneCommand` tags can be directly inserted into the text, with a '!' prefix.

Creating commands

This section walks you through creating your own commands. First, create a new C# script in the Unity editor.

Creating the class

Add the following using statement at the top of your class: `using TMPEffects.TMPCommands;`.
Then, make the created class derive from `TMPCommand`. In order to be able to create the command object in the Unity editor and add it to your database, make sure to decorate the class with the CreateAssetMenu attribute.

Members

You will have errors due to `TMPCommand's` abstract members not being implemented. Auto-implement them using your IDE, or add them manually. When you are done, your class should look something like this:

```
using UnityEngine;
using TMPEffects.TMPCommands;

[CreateAssetMenu(fileName="new YourFirstCommand", menuName="Your/Path/YourFirstCommand")]
public class YourFirstCommand : TMPCommand
{
    public override TagType TagType => throw new System.NotImplementedException();

    public override bool ExecuteInstantly => throw new System.NotImplementedException();

    public override bool ExecuteOnSkip => throw new System.NotImplementedException();

    public override void ExecuteCommand(TMPCommandArgs args)
    {
        throw new System.NotImplementedException();
    }

    public override bool ValidateParameters(IDictionary<string, string> parameters)
    {
        throw new System.NotImplementedException();
    }
}
```

Let's go over each member individually.

Properties

TagType: Defines whether the tags for this command should operate on an index, a text block, or either option. For example, the built-in `wait` command operates on an index, and the built-in command `show` operates on a text block (see Built-in commands).

ExecuteInstantly: Commands where this property is true are executed the moment the TMPWriter begins the writing process, instead of when their opening tag index is reached. From the built-in tags, only `show` is executed instantly.

ExecuteOnSkip: Commands where this property is true are executed even when their index is skipped over by the writer (i.e., when TMPWriter.SkipWriter() is called). This should be true for commands that need to ensure they are being called even if skipped over, for example a command that starts a quest or adds an item to the player's inventory.

Optional properties

There are a few optional properties. If you don't override them, they are set to false by default. In both cases, this is to protect you from yourself 😉
Only set these to true if you are sure it is safe for your case!

ExecuteRepeatable: Commands where this property is true may be executed multiple times, specifically if the writer is reset / restarted at any point (i.e., when TMPWriter.ResetWriter() is called). This should be false for commands that need to ensure they are only ever raised once, for example a command that starts a quest or adds an item to the player's inventory.

ExecuteInPreview: Commands where this property is true are executed in the editor preview. ⚠ Note that you must wrap this property in a `#if UNITY_EDITOR` preprocessor directive if you want to override it; otherwise your builds will fail.

Methods

ValidateParameters(IDictionary<string, string> parameters): This method is called during tag processing. It allows you to specify whether a given tag for this command has valid parameters. ParameterUtility will come in handy here. Return true if the parameters are valid, return false if not. If false, the tag will not be processed.

ExecuteCommand(TMPCommandArgs args): The meat of your command. This executes the actual command you are implementing.

TMPCommandArgs

The sole argument for the `ExecuteCommand` method. It's kept relatively simple: it provides access to the actual `EffectTag`, through which you may get the tag's parameters, the `EffectTagIndices`, and the executing TMPWriter.

Full example

As complete example, the class below is the implementation of the built-in `delay` command.

```
using System.Collections.Generic;
using UnityEngine;

namespace TMPEffects.TMPCommands.Commands
{
    [CreateAssetMenu(fileName = "new DelayCommand", menuName = "TMPEffects/Commands/Delay")]
    public class DelayCommand : TMPCommand
    {
        public override TagType TagType => TagType.Index;
        public override bool ExecuteInstantly => false;
        public override bool ExecuteOnSkip => true;
        public override bool ExecuteRepeatable => true;

#if UNITY_EDITOR
        public override bool ExecuteInPreview => true;
#endif

        public override void ExecuteCommand(TMPCommandArgs args)
        {
            if (ParameterUtility.TryGetFloatParameter(out float delay,
args.tag.Parameters, ""))
            {
                args.writer.SetDelay(delay);
                return;
            }

            // Since validate parameters ensures the parameter is present and float,
            // this state should be impossible to reach
            throw new System.InvalidOperationException();
        }

        public override bool ValidateParameters(IDictionary<string, string> parameters)
        {
            if (parameters == null) return false;
            if (!parameters.ContainsKey(""))
                return false;

            return ParameterUtility.HasFloatParameter(parameters, "");
        }
    }
}
```

Adding the command to a database

To actually use the command in your text, you will have to follow these steps:

1. Create a command object: Right click in your project view and create it (it will be in the path you specified in the CreateAssetMenu attribute).
2. Add that object to the database you want to use and give it a name
3. Use that database in the TMPWriter component

Done! You can now use your custom command like any of the built-in ones.

Creating scene commands

See Scene commands on how to add scene commands.

Parameters

The following three sections introduce you to the different parameter types, and show you how to make your life easier when writing animations (or commands).

Parameter types

TMPEffects supports a variety of different parameter types that come with built-in parsing utilities (see the next section).

Supported types

This is the full list of currently supported parameter types:

- float
- int
- bool
- Color
- Vector2
- Vector3
- Vector2Offset
- Vector3Offset
- Anchor
- TypedVector2
- TypedVector3
- AnimationCurve
- WaveOffsetType
- Array

Type formatting

This is an overview of how to correctly format the different parameter types in your tags, most with at least one example.

- float: Must use `.` as decimal, not `,`
  - `2.54`, `3`
- int: Just a plain integer number
  - `12`
- bool: Either `true` or `false`
- Color: Colors may be defined either in HEX format, HSV format or RGB(A) format. In addition to that, there are bunch of keywords
  - `#DEADBEEF`, `hsv(0.3,64,52)`, `rgb(0,0.5,0.5)`, `indigo`
- Vector2: Two bracketed floats, separated by comma
  - `(0.3, 22.4)`
- Vector3: Three bracketed floats, separated by comma (or two; third float automatically set to zero)
  - `(0.3, 22.4, 0)` = `(0.3, 22.4)`
- Vector2Offset: Same as Vector2, but with leading `o:`

    - o:(0.3, 22.4)
- Vector2Offset: Same as Vector3, but with leading `o:`
    - `(0.3, 22.4, 0)` = `o:(0.3, 22.4)`
- Anchor: Same as Vector2, but with leading `a:`
    - `a:(0.3, 22.4)`
- TypedVector2: Format of either Vector2, Vector2Offset or Anchor
    - `(0.3, 22.4)`, `o:(0.3, 22.4)`, `a:(0.3, 22.4)`
- TypedVector3: Format of either Vector3, Vector3Offset or Anchor
    - `(0.3, 22.4, 0)`, `o:(0.3, 22.4, 0)`, `a:(0.3, 22.4)`
- AnimationCurve: Generally keywords; though you may also construct custom curves by specifying one of the predefined methods (`cubic`, `quadratic`, `linear`) or just a raw vector sequence. See AnimationCurveUtility for more info.
    - `easinoutsine`, `quadratic((0,0),(0.2,0.7),(1,1))`, `(0,0),(0.2,0.7),(1,1)`
- WaveOffsetType: Keywords
    - `index / idx`, `segmentindex / sindex / sidx`, `x / xpos`, `y / ypos`
- Array: Multiple of the desired type, separated by a semicolon
    - `0.3;4.82;1`, `red;green;blue`, `true;false;false`

ParameterUtility

`ParameterUtility` is a static utility class for parameter validation and parsing, to be used with all types of animations and commands.
The full API docs can be found here.

For each of the supported parameter types listed in the previous section, there is a `HasXYZParameter` method, a `HasNonXYZParameter` method, a `GetXYZParameter` and a `TryGetXZYParameter` method. Each of these methods are explained individually below.

HasParameter

`bool HasXYZParameter(IDictionary<string, string>, string name, params string[] aliases)` checks whether the given set of parameters contains a parameter of the given name, or any of the aliases, that is of type `XYZ`.

Example: `HasFloatParameter(parameters, "duration", "dur", "d")` returns true if `parameters` contains a parameter named either "duration", "dur" or "d", and the value could be converted to type float. Otherwise, it returns false.

HasNonParameter

`bool HasNonXYZParameter(IDictionary<string, string>, string name, params string[] aliases)` checks whether the given set of parameters contains a parameter of the given name, or any of the aliases, that is NOT of type `XYZ`.

Example: `HasNonFloatParameter(parameters, "duration", "dur", "d")` returns true exactly when `parameters` contains a parameter named either "duration", "dur" or "d", but the value could NOT be converted to type float. Otherwise, it returns false.

GetParameter

`XYZ GetXYZParameter(IDictionary<string, string>, string name, params string[] aliases)` returns the parameter defined by the given name, or any of the aliases, converted to type `XYZ`. Otherwise, it will throw an exception.

Example: `GetFloatParameter(parameters, "duration", "dur", "d")` throws an exception if `parameters` does not contain a parameter named either "duration", "dur" or "d", or, if the parameter is defined, it could not be converted to type float. Otherwise, it will return the parameter converted to type float.

TryGetParameter

`bool TryGetXYZParameter(out XYZ value, IDictionary<string, string>, string name, params string[] aliases)` wraps `GetParameter` in a try-catch statement, returning true if it was successful, otherwise it

returns false. If successful, you can get the value of the converted parameter from the `out XYZ value` parameter.

Example: `TryGetFloatParameter(out float value, parameters, "duration", "dur", "d")` returns false if `parameters` does not contain a parameter named either "duration", "dur" or "d", or, if the parameter is defined, it could not be converted to type float. Otherwise, it will return true and `value` will be set to the parameter converted to type float.

Array

For array parameters, each of these four methods have an additional required parameter: `ParseDelegate<string, T, IDictionary<string, T>` ,
where `ParseDelegate` is defined as: `public delegate W ParseDelegate<T, U, V, W>(T input, out U output, V keywords)` .

Essentially, this delegate is used to parse the individual elements of the array.
You can use the `ParsingUtility.StringToXYZ` methods for this (they are not further explained here, but you can look at the API docs for them).

Example: `TryGetArrayParameter<float>(out float[] value, parameters, ParsingUtility.StringToFloat, "numbers", "nums")`

ParameterDefined

In addition to these type-specific methods, there are also generic methods for checking whether a parameter is defined, without performing any type checks.
These are:
`bool ParameterDefined(IDictionary<string, string>, string name, params string[] aliases)` : Checks whether a parameter of the given name or any of its aliases is present in the dictionary. EXACTLY one must be defined for this to return true; if for example two aliases are present it is considered not defined.

`string GetDefinedParameter(IDictionary<string, string>, string name, params string[] aliases)` : If the parameter is defined according to `ParameterDefined` , this will return the value of that parameter. Otherwise, it will throw an exception.

`bool TryGetDefinedParameter(out string value, IDictionary<string, string>, string name, params string[] aliases)` : Wraps `GetDefinedParameter` in a try-catch statement. If successful, the parameter value will be stored in the `out string value` parameter.

Waves

If your animation uses Waves, you can use the pre-defined wave parameters set by using `ValidateWaveParameters(IDictionary<string, string> parameters, string prefix = "")` and `GetWaveParameters(IDictionary<string, string> parameters, string prefix = "")` in `ValidateParameters`

and `SetParameters` respectively. The passed in `prefix` lets you use multiple waves with differently prefixed parameter names.

If you have a default wave, you can combine it with the set parameters like this:

```
[SerializeField] Wave wave;

public void SetParameters(object customData, IDictionary&lt;string, string&gt; parameters)
{
    Data d = customData as Data; // Cast custom data to whatever type it is
    d.Wave = CreateWave(wave, GetWaveParameters(parameters));
}
```

This will create a new wave from the set parameters and the values defined in the default wave as fallback values for the non-set parameters.

For the full list of parameters that are part of the wave parameters set, see the API documentation.

AutoParameters

AutoParameters is a plugin for TMPEffects that minimizes any parameter-related boilerplate for animations by automatically implementing the `ValidateParameters`, `SetParameters` and `GetNewCustomData` methods for you. For more information, see the page on it.

AnimationCurveUtility

`AnimationCurveUtility` is a static utility class that allows easy creation of various AnimationCurves. The full API docs can be found here.

Predefined curves

All of the easing functions presented at this link are implemented: https://easings.net/
You can create the corresponding AnimationCurve like this: `AnimationCurveUtility.EaseInOutSine()`.
You can also get the Bézier points (don't worry about what this means if you don't know 😄) that define the AnimationCurve, so you can manipulate them to easily create slightly modified versions of the existing curves using the Bézier constructors.

Bezier constructors

You can create AnimationCurves using Bézier points.
Simply call `Bezier(params Vector2[] points)` with your points Bézier points.
The method will automatically infer whether you are creating a linear, quadratic or cubic Bézier curve based on the amount of points.
If the amount of points does not clearly indicate one specific type, higher degree Bézier curves are preferred.
There are also the `LinearBezier`, `QuadraticBezier`, and `CubicBezier` methods, if you want to make sure the correct degree Bézier curve is created.

⚠ When creating your own AnimationCurves like this, always keep in the back of your mind that Unity's AnimationCurves use time as input; this means the Bézier curve must at all times advance on the X axis, or you will get an invalid AnimationCurve.
For example, imagine the quadratic curve defined by the points (0,0), (0,1), (1,1):

This will yield an invalid curve! Consider the very beginning of this curve. At the very beginning, the curve moves perfectly straight up; that is not possible in AnimationCurves.
Something even more extreme, like the curve moving "back" / to the left, is of course not possible either:

Huge props to qwe321qwe321qwe321 on GitHub for his `BezierToAnimationCurve` implementation, as well as the optimized curve points!

TMPEffectTag

`TMPEffectTag`, together with `TMPEffectTagIndices`, is the data structure used to represent a tag in the code, for example animation tags or command tags.
The API docs can be found here.

Properties

The `TMPEffectTag` consists of the following properties:

- Name : `string`
- Prefix : `char`
- Parameters : `ReadOnlyDictionary<string, string>`

Example: the show tag `<+fade anc=zero dur=0.55>` would be:

- Name : "fade"
- Prefix : '+'
- Parameters : {{ "anc", "zero" }, { "dur", "0.55" }}

Indices

The `TMPEffectTagIndices` struct consists of:

- StartIndex : `int`
- EndIndex : `int`
- OrderAtIndex : `int`

The indices are a half open interval; meaning a tag with a StartIndex of 5 and an EndIndex of 12 will "contain" the indices 5, 6, 7, 8, 9, 10, 11 and 12.

The OrderAtIndex is used to maintain an order if there are multiple tags at the same index. Generally speaking, the tags' OrderAtIndex won't be sequential (i.e. 1, 2, 3, and so on), but may skip around. You will notice this if you iterate, for example, over the `BasicTags` property of a TMPAnimator. The only invariant OrderAtIndex it guaranteed to follow is that they are sorted from smallest to largest.

TagCollections

`TagCollection` is a collection of `TMPEffectTagTuples`, which combine TMPEffectTags and TMPEffecTagIndices.
It maintains an order for the tags based on their indices, and exposes multiple functions to get contained tags based on their indices, and vice versa.

When creating a `TagCollection`, you can pass a ITMPTagValidator, to guarantee you may only add specific tags to the collection.

For normal use cases, you won't be creating `TagCollections` though; generally, you will use the `TagCollections` exposed by the TMPAnimator and TMPWriter to add or remove tags from script.

Be aware that if you add tags to a `TagCollection`, if there is already a tag present with the exact same `StartIndex` and `OrderAtIndex`, that the `OrderAtIndex` of the contained tags will be adjusted.

CharData

The `CharData` class holds information about a character, which is primarily used by the TMPAnimator and its animations.
In addition to holding a selection of data supplied by the respective TMP_CharacterInfo, accessible through the `info` field, also holds TMPEffects-specific data and methods to manipulate said data.

Modifiable properties

Each `CharData` has a `position`, `rotation` and `scale` property. You may modify all of those properties using the respective setter methods.
`CharData` also exposes the initial, readonly value of each of those properties.

Through the `mesh` field, you can access the character's `VertexData`.
In TextMeshPro, each character consists of a rectangular mesh.
`VertexData` allows you to modify the properties of each of the four vertices of the character mesh.
These properties are:

- Position
- Color
- UV0
- UV2

`VertexData` also exposes a `ReadOnlyVertexData` object through its `initial` field.
It contains the initial, readonly `VertexData`.

Animating CharData

For an explanation and examples as to how you can animate characters by modifying the mentioned properties, see Animating a character.

IAnimationContext

An instance of the `IAnimationContext` type serves as the context for all animations.

Properties

`IAnimationContext` exposes various properties that will be useful for your animations:

- **CustomData**: The custom data object created in `GetNewCustomData` (see Creating Animations).
- **SegmentData**: Contains information about the animation segment the current character belongs to.
  - **StartIndex**: The first index of the segment within the containing text
  - **Length**: The lenght of the segment
  - **FirstVisibleIndex**: The index of the first visible character (i.e. non-whitespace character)
  - **LastVisibleIndex**: The index of the last visible character (i.e. non-whitespace character)
  - **FirstAnimationIndex**: The index of the first character will actually be animated (i.e. not whitespace, not excluded by TMPAnimator)
  - **LastAnimationIndex**: The index of the last character will actually be animated (i.e. not whitespace, not excluded by TMPAnimator)
  - **Max**: The maximum vertex positions of text in this segment
  - **Min**: The minimum vertex positions of text in this segment
  - **SegmentIndexOf(CharData)**: Get the index within this segment for the passed in `CharData`
- **State**: Exposes multiple readonly properties about the current state of the `CharData` (with previous animations already applied). Generally, to be used with the `late` tag parameters (see Getting started with TMPAnimator).
  - **CalculateVertexPositions()**: Calculate the current vertex positions. Results can be read from `BL_Result`, `TL_Result`, `TR_Result`, `BR_Result`.
- **Finished(int), Finished(CharData)**: Check whether the given `CharData` is done animating. To be used with show and hide animations (see Creating show / hide animations).
- **FinishAnimation(CharData)**: Finish the animation for the given `CharData`. To be used with show and hide animations (see Creating show / hide animations).
- **AnimatorContext**: The `IAnimatorContext` exposes properties about the animator state.
  - **PassedTime**: The time that has passed since the animator began animating. Generally speaking, you should use this as the time value for basic animations
  - **DeltaTime**: The current delta time used by the animator to update the animations.
  - **UseScaledTime**: Whether the animator uses scaled time (=> whether `PassedTime` is scaled)

- **ScaleAnimations**: Whether the animator scales the animation. Used by various AnimationUtility methods.
- **StateTime(CharData)**: How long the given `CharData` has been in its current visibility state for. Generally speaking, you should use this as the time value for show and hide animations
- **VisibleTime(CharData)**: How long the given `CharData` has been visible for. Can use this alternatively to `PassedTime` for basic animations

Databases

In TMPEffects, the set of animations a `TMPAnimator` can use (or the set of commands a `TMPWriter` can use) is defined by the database it uses.

Databases, like animations and commands, are ScriptableObject assets.
You can create a new database by right-clicking in your project view, then Create -> TMPEffects -> Database, again, just like animations and commands.
You can then add any animation / command to your database, and assign it to a TMPAnimator / TMPWriter component in the inspector.
Of course, you can also modify the built-in default databases any way you want, or assign different databases to be used as default database in the TMPEffects settings (in the top bar, Edit -> Preferences -> TMPEffects).

SceneAnimations and SceneCommands are separate from databases and are instead added to a dictionary in the component's inspector; see the individual sections on them.