For UI Toolkit

Requirements

I recommend using Unity version 2020.1 and above. If possible, I suggest upgrading to the latest LTS version. All animated elements are prepared as CustomElements in C# without using third-party libraries. All elements inherit from the `VisualElement` class. This allows them to be created directly in a C# script or via the UI Builder. More details here.

How It Works

All elements in this package operate on the same principle and share a consistent or extended API as needed. Each element is designed to be invoked through a C# script to determine when to update its animation state.

Below are the public methods available for each element. You can choose which method to call for updating the animation based on your preference:

`TimeUpdate(float deltaTime)`
This method is intended to be called in every `Update()` method from a runtime script, with `Time.deltaTime` as the input parameter in seconds. It utilizes the element's `FrameTime` parameter to update itself according to the predefined frame length.
For example, if `FrameTime = 0.04f` (40ms), the animation frame rate corresponds to 25fps (1000 ms / 40 ms = 25).

`AnimationTick()`
This method updates the animation directly and can be called in the `Update()` method from a runtime script at a frequency determined by the user according to their own logic.

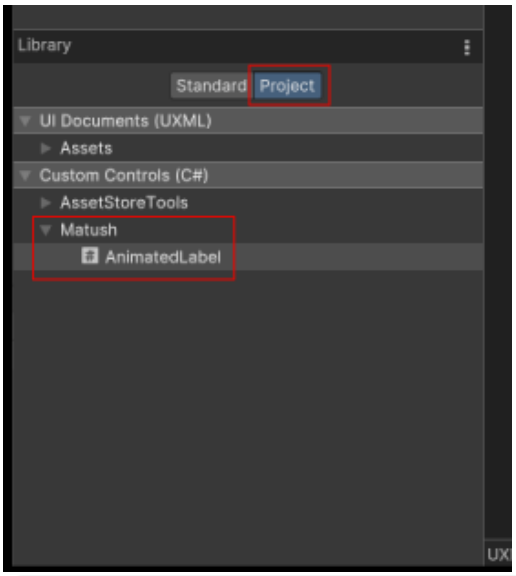
Each element includes the following public parameters:

- **`Speed (float)`**
  Defines the playback speed of the animation, independent of the animation frame rate.
  Default value: `1f`
- **`FrameTime (float)`**
  Defines the length of the animation frame when using the `TimeUpdate()` method.
  Default value: `0.04f` (25fps)

(Each element may contain additional public parameters as needed for its specific animation requirements.)

Setup & Use

General UIElements Setup



First, the user needs to have a `UIDocument` and the corresponding `VisualTreeAsset` (.uxml) created. This document can be modified using the UI Builder editor. All Animated Elements are under the `Matush` namespace.

It's also possible to create elements directly in a C# script. Like other UIElements, these elements are publicly accessible for declaration:

```C/C++
AnimatedLabel animatedLabel = new AnimatedLabel();
sampleDocument.rootVisualElement.Add(animatedLabel);
```

Additionally, you can style the element either directly through the `.uxml` file or by using `.uss` stylesheets.

Animated Label

API

- *Speed (float)*
- *FrameTime (float)*
- *PauseTIme (float)*
- AnimationText (String)
- AnimationCurve (UnityEngine.AnimationCurve)
- AnimationSize (float > 0)
- Colors (List<UnityEngine.Color>)

AnimationText
The text of the label must be defined as the first parameter before all other parameters.

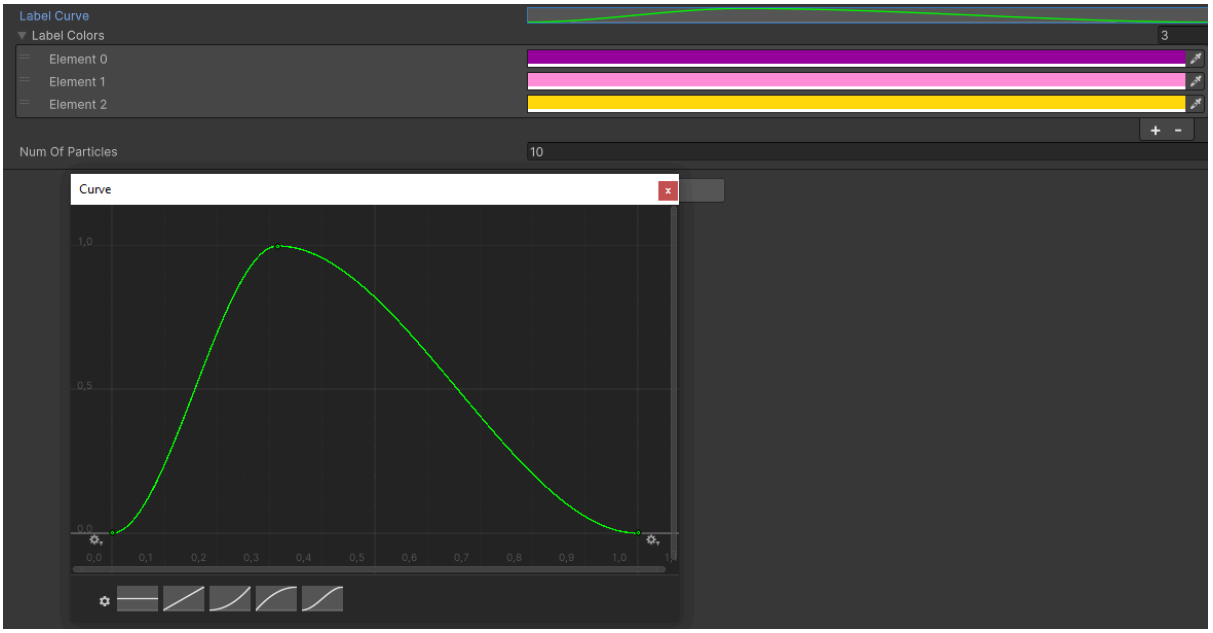
AnimationCurve
The curve that defines the progression of the color transition in the label animation.

AnimationSize
Defines the sampling for individual letters in the text.
The default value of 1 sets the size of the animation curve to match the total number of characters in the text.
For example, a value of 2 defines the curve progression for twice the number of characters in the text.
Similarly, a value of 3 applies the curve to three times the number of characters, and so on.

Colors
A list of colors used to define the transition. It must contain at least two items.

Example

```C/C++
public class SampleScenePresenter : MonoBehaviour {
        [SerializeField]
        UIDocument _sampleDocument;

        [SerializeField]
        private AnimationCurve _labelCurve;

        [SerializeField]
        private Color[] _labelColors;

        private AnimatedLabel _animatedLabel;

        void Start() {
            if (_sampleDocument != null) {
                _animatedLabel = _sampleDocument.rootVisualElement.Q<AnimatedLabel>();
                 // Set the text of the animated label to "Hello World!".
                _animatedLabel.AnimatedText = "Hello World!";
                 // Set the Curve property of the animated label to _labelCurve.
                _animatedLabel.Curve = _labelCurve;
                 // If _labelColors is not null and contains more than one color,
                 // convert the colors to a list and assign it to the Colors property of the
                 animated label.
                if (_labelColors != null && _labelColors.Length > 1) {
                    _animatedLabel.Colors = _labelColors.ToList();
                }
            }
        }

        void Update() {
         // Get the time elapsed since the last frame.
            var deltaTime = Time.deltaTime;
         // Update the time for the animated label using the elapsed time.
            _animatedBackground.TimeUpdate(deltaTime);
        }
    }
```



Animated Scrolling Background

This element behaves as an infinitely scrolling background. Therefore, it requires a tiled texture that seamlessly loops. Examples of such textures are included in the project. You can find a guide on how to create these textures, for example, here.

API

- *Speed (float)*
- *FrameTime (float)*
- TiledTexture (Texture2D)
- Direction (Vector2)
- Scale (float)

TiledTexture
The texture suitable for infinite scrolling **must be set as the first parameter** of this element.

Direction
Specifies the scrolling direction. The default value is set as follows:

```
C/C++
private Vector2 _direction = new Vector2(1, 1);
```

This corresponds to a direction moving right and down at a 45-degree angle.

Scale
Specifies the size of the texture within the element.

Example

C/C++

```
    public class SampleScenePresenter : MonoBehaviour {
        [SerializeField]
        UIDocument _sampleDocument;

        [SerializeField]
        private Texture2D _tiledTexture;
        private AnimatedScrollingBackground _scrollingBackground;


        void Start() {
            if (_sampleDocument != null) {
                _scrollingBackground =
_sampleDocument.rootVisualElement.Q<AnimatedScrollingBackground>();
                if (_scrollingBackground != null) {
                 //Setting tiledTexture as first parameter
                    _scrollingBackground.TiledTexture = _tiledTexture;

                    var randomX = UnityEngine.Random.Range(-1f, 1f);
                    var randomY = UnityEngine.Random.Range(-1f, 1f);
                    _scrollingBackground.Direction = new Vector2(randomX, randomY);

                 //Cale and Speed are set manualy, depending on your texture size
                    _scrollingBackground.Scale = 0.1f;
                    _scrollingBackground.Speed = 1.5f;
                }
            }
        }

        void Update() {
            var deltaTime = Time.deltaTime;
            if (_scrollingBackground != null) {
                _scrollingBackground.TimeUpdate(deltaTime);
            }
        }
    }
```

Animated Star

This element is essentially a pulsating image, named "Star" due to its use of enlarging stars on items in a store to draw attention and encourage purchases. The asset pack includes a star image suitable for this purpose, but users are free to use their own image through `.uss` or the API (Sprite).

API

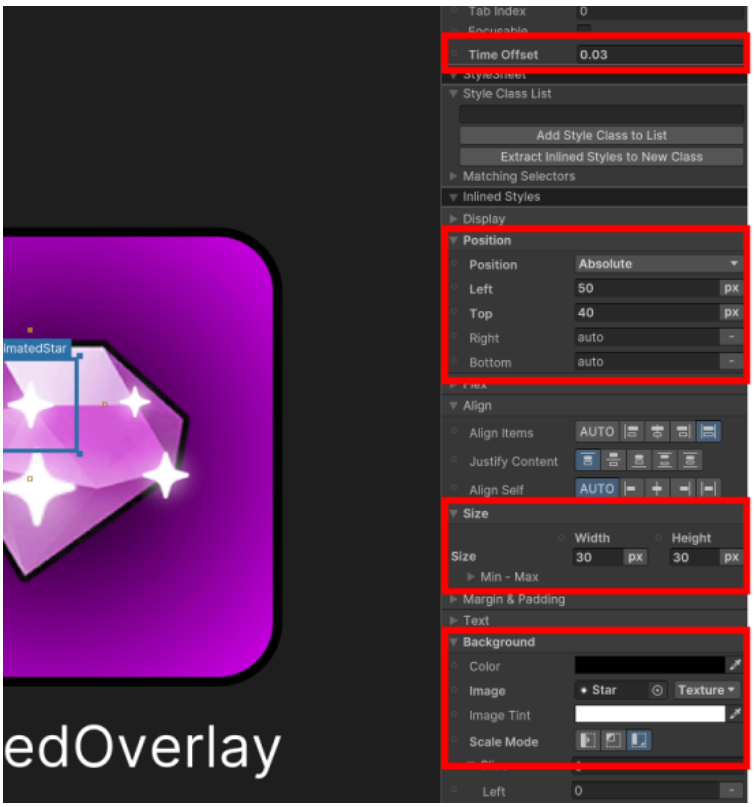
- *Speed (float)*
- *FrameTime (float)*
- AnimationCurve (UnityEngine.AnimationCurve)
- TimeOffset (float)
- Sprite (Sprite)

AnimationCurve
The curve that defines the progression of the color transition in the label animation.

TimeOffset
Defines the time delay before the animation starts. It is recommended to set a random value in seconds for each `AnimatedStar` when animating multiple stars simultaneously.

Sprite
The image used for the pulsating element.

Example



Here you can see an example of parameter setup in UIBuilder. I set the element to `Absolute` and position it to align with the background image to achieve the desired placement. Next, I configure the size and set the `background-image` to "Star" for animation.

C/C++

```
public class SampleScenePresenter : MonoBehaviour {

    // Serialized field for the UI Document, allowing it to be set in the Unity Inspector.
    [SerializeField]
    UIDocument _sampleDocument;

    // Serialized field for the animation curve used to scale the stars, set in the Unity
Inspector.
    [SerializeField]
    private AnimationCurve _starsScaleCurve;

    // Private list to store references to all AnimatedStar elements in the UI.
    private List<AnimatedStar> _stars;

    // Called when the script instance is being loaded.
    void Start() {
        // Check if the UI document is assigned.
        if (_sampleDocument != null) {
            // Query all AnimatedStar elements in the root visual element and store them in the
_stars list.
            _stars = _sampleDocument.rootVisualElement.Query<AnimatedStar>().ToList();
            // Assign the animation curve to each star in the list.
            _stars.ForEach(star => star.AnimationCurve = _starsScaleCurve);
        }
    }

    // Called once per frame to update the animations.
    void Update() {
        // Get the time elapsed since the last frame.
        var deltaTime = Time.deltaTime;
        // Update the time for each star's animation using the elapsed time.
        _stars.ForEach(star => star.TimeUpdate(deltaTime));
    }
}
```

Animated Warp

Custom animated component that emits particles from its center outward, creating a dynamic "warp" effect. As the particles move outward, they also scale, enhancing the warp illusion. A pre-configured sprite suitable for this effect is included in the project, but users can customize the particle scale, quantity, and animation speed to fit their needs..

API

- *Speed (float)*
- *FrameTime (float)*
- ParticleSprite (Sprite)
- NumOfParticles (int)
- ParticleScale (float)

ParticleSprite
Specifies the sprite to be used for the particles that are animated.

NumOfParticles
Defines the number of particles to be emitted from the center of the element in random directions.

ParticleScale
Sets the scale of the particles within the element, allowing users to adjust their size.

Example

C/C++

```
    public class SampleScenePresenter : MonoBehaviour {
        [SerializeField]
        UIDocument _sampleDocument;

        [Header("Warp Settings")]
        [SerializeField]
        private int _numOfParticles;

        [SerializeField]
        private Sprite _warpParticleSprite;

        private AnimatedWarp _animatedWarp;

        void Start() {
            if (_sampleDocument != null) {

                _animatedWarp = _sampleDocument.rootVisualElement.Q<AnimatedWarp>();
                if (_warpParticleSprite != null && _animatedWarp != null) {
                    _animatedWarp.ParticleSprite = _warpParticleSprite;
                    _animatedWarp.NumOfParticles = _numOfParticles;
                    _animatedWarp.Speed = 0.1f;
                }
            }
        }

        void Update() {
            var deltaTime = Time.deltaTime;

            if(_animatedWarp != null)
                _animatedWarp.TimeUpdate(deltaTime);
        }
    }
```