By Seaside Studios
For technical support, feedback and requests write to:
seasidegamestudios@gmail.com
Twitter of the creator: https://twitter.com/GerardBelenguer

Index

Overview

Welcome to the All In 1 Spring Toolkit! This comprehensive toolkit is designed to breathe life into your Unity projects, enhancing the game polish and adding an extra layer of visual polish through dynamic spring animations.

Whether you're an experienced Unity developer or just starting out, this toolkit offers a versatile solution to elevate your projects. For beginners, it provides a user-friendly set of ready-to-use components that simplify the process of incorporating engaging spring animations. With just a few clicks and a couple lines of code, you can infuse your game elements with lively movements and captivating visual feedback.

For advanced users, the All In 1 Spring Toolkit offers a solid foundation to build upon. Leveraging its extensible architecture, you can seamlessly create custom spring behaviors tailored to your unique project requirements. Dive into the toolkit's codebase and unlock a world of possibilities, crafting intricate animations and game mechanics that truly set your projects apart.

With its intuitive workflow and comprehensive documentation, this toolkit empowers you to effortlessly add that extra layer of polish and visual flair. Transform static elements into dynamic, responsive components that react naturally to user interactions, enhancing the overall immersion and engagement of your Unity projects.

Thank you for choosing the "All In 1 Spring Toolkit" as your go-to solution for game polish and juice. Get ready to unleash your creativity and elevate your projects to new heights!

I'm always open for questions, feedback and suggestions. The best way to contact me is by email. Please don't write questions in the Unity Forums, Youtube videos, Twitter or wherever else since I will probably miss them. I always reply much faster (in 24h or less) by email.
When reaching out please attach your invoice number too and make sure you have read this document. The email address is:

seasidegamestudios@gmail.com

If you like the asset please make sure to drop a review on the Asset Store page. It helps out a ton.

Setup

The asset requires no setup. Simply import the package into any Unity version with any render pipeline, and it's ready to use. The asset folder can be moved anywhere in your project, and the included Demo folder can be safely deleted if desired.

**If you want to see the Demo example as showcased in the Trailer, Screen Shots and WebGL Demo you'll need to set the Project Color Space to Gamma (in Project Settings -> Player). Of course this is completely optional and won't affect at all the behavior of the asset. But if the Demo looks different and very white is because you are using the Linear Color Space instead of Gamma:*

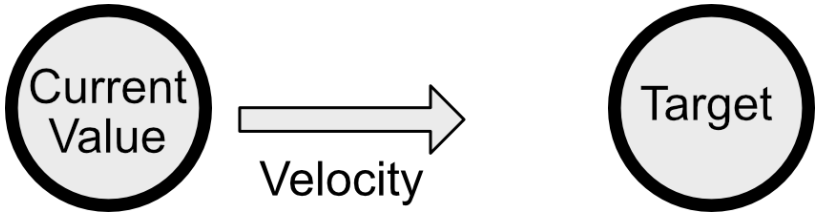
What is a spring?

A "spring" is an ongoing efficient system that moves a Current Value towards a Target in a spring-damped motion, creating configurable natural, bouncy animations. This versatile mechanism can be applied to any Unity property like position, scale, rotation, color, and more, allowing you to easily add responsive, lively animations to your game elements.
Every single moving thing in this Demo is using springs and it will always be the best place to see how springs can be used.

Spring Main Elements

All springs, regardless of type, have three core values:

1. **Current Value**: The value of the spring at the current frame.
2. **Target**: The desired value that the spring aims to reach. The spring will gradually move the Current Value towards the Target value, frame by frame.
3. **Velocity**: Depending on the state of the spring, the velocity will vary. Generally, the further apart the Current Value and Target are, the higher the velocity will be. When both values are equal, the velocity is zero, and the spring is at rest.

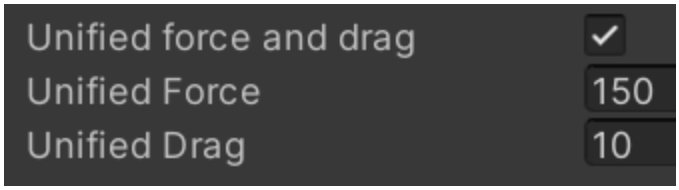


To fine-tune the spring behaviors, you can adjust two parameters:

1. **Force**: This determines how much force is applied to move the Current Value towards the Target. A higher Force will cause the Current Value to approach the Target more quickly, resulting in a faster-growing and larger Velocity.
2. **Drag**: This parameter slows down the spring. The higher the Velocity, the more the Drag will pull back on the spring. A higher Drag value results in a more stable and predictable spring with less bouncing. When the Drag is low, the Current Value will bounce back and forth around the Target until eventually reaching equilibrium.

In summary, Force controls how quickly the Current Value reaches the Target, while Drag determines how gently it does so. Understanding the interplay between these elements is crucial for tuning the springs to achieve the desired feel.

In general Force and Drag will be setup in the spring component, in the editor, using the Inspector:



Although of course you can change it at any point in code.

Current Value, Target and Velocity are usually interacted with in code. As we'll see in the next section.

Spring Types

The toolkit offers various spring types that can be utilized by any Spring Component (see the next section). These spring types consist of an array of one-dimensional springs, which serve as the foundation of the asset, and a collection of helper functions

to interact with the core elements discussed in the previous section. The available spring types are:

1. **Float Spring**: A one-dimensional spring. This is the most fundamental and intuitive type, guiding the Current Value towards the Target in a spring-damped motion. It serves as the core of the toolkit, as all other spring types build upon this base.

2. **Vector2, Vector3, Vector4 Spring**: These springs have two, three, and four dimensions, respectively. They function similarly to the Spring Float but with additional dimensions, offering the same functionality and methods.

3. **Color Spring**: A four-dimensional spring. It operates like the Vector4 spring but includes helper functions and additional features to handle Vector4 to Color conversions.

4. **Rotation Spring**: Up to 10 dimensions, although not all are used simultaneously. We employ an axis-angle representation of a quaternion, along with additional dimensions that assist in handling overshoots beyond 180 degrees in the same direction, properly adding velocity, and more. While the implementation details are intricate, we recommend using this spring like the others, as the exposed public methods and workflow are consistent across all spring types.

The main functionality and functions of these Spring Types are:
1. **SetValue**: Sets the Current Value.
2. **GetCurrentValue**: Returns the Current Value.
3. **SetTarget**: Sets the Target.
4. **GetTarget**: Returns the Target.
5. **AddVelocity**: Adds a specific value to the velocity, used to nudge the spring in a particular direction (see the Nudges and Punches section for more information).
6. **SetVelocity**: Sets the Velocity to a desired value.
7. **GetVelocity**: Returns the Velocity.
8. **SetApplyClamping**: Activates or deactivates clamping on the spring.
9. **SetMinValues** and **SetMaxValues**: Changes the bounds of the clamping.
10. **StopSpringOnClamp**: Activates or deactivates the Stop On Clamp feature, which stops the spring when it overshoots the minimum or maximum clamping values. When this occurs, the OnClampingApplied event Action is triggered.

Basic Usage

To get started with the All In 1 Spring Toolkit, follow these steps:

1. Add a Spring Component: Choose the appropriate Spring Component for your needs (e.g., TransformSpringComponent, CamFovSpringComponent, AudioSourceSpringComponent) and add it to your GameObject.
2. Configure Force and Drag: In the Inspector, adjust the Force and Drag parameters to fine-tune the spring behavior. Higher Force will make the spring react more quickly, while higher Drag will make it more stable with less bouncing. You can also use the Springs Debugger to adjust these values in real-time during play mode, which is helpful for quick iteration and fine-tuning.
3. Set up Auto Update or Manual Control:
    - For components like TransformSpringComponent, enable "Use Transform As Target" and assign a Target Transform in the Inspector for automatic updates.
    - If "Use Transform As Target" is disabled, you'll need to call SetTarget() yourself in code.
    - Most springs have similar ways of deciding if they are automatically updated or not within their General Properties
    - Some springs, such as FloatSpring, don't have auto-update properties. Interact with these through code by calling SetTarget() manually and using GetCurrentValue() to retrieve and apply the spring's value where needed.
4. (Optional) Additional Configuration: Depending on the specific Spring Component, you may have additional options to configure, such as clamping values or specific properties to animate.

With these steps, most spring animations will be ready to use with minimal or no code required. The component will handle spring calculations and apply the results to the appropriate properties of your GameObject.

For more dynamic control, reference the Spring Component in your scripts and use its public methods like SetTarget(), AddVelocity(), or GetCurrentValue(). This approach allows you to create responsive behaviors while leveraging the spring system's power.

You'll find numerous examples of these methods and ways to interact with Spring Components in the Demo Examples. All Prefabs in the demo have the main script attached to the parent object, providing practical implementations for reference.

Spring Components

Spring Components, such as TransformSpringComponent, CamFovSpringComponent, AudioSourceSpringComponent, ShaderFloatSpringComponent, UiSliderSpringComponent and others are concrete implementations of the SpringComponent class.

Each of these components exposes the same or very similar public functions as the spring types, which we explained in the previous section. However, we recommend double-checking the available methods when using these components, as they often provide additional useful functions on top of the base spring functionality.
If you find that a particular method from the base spring is missing, you can access it through the public spring reference inside the SpringComponent.
Some examples of Spring Components and their functionalities include:

TransformSpringComponent: Applies spring animation to a GameObject's position, rotation, and scale.
CamFovSpringComponent: Animates the field of view of a camera component with a spring-like motion.
AudioSourceSpringComponent: Applies spring animation to an AudioSource's volume and pitch.
ShaderFloatSpringComponent: Animates float properties in shaders using spring behavior.
UiSliderSpringComponent: Adds spring-like animation to a UI slider's value.
AnchoredPositionSpringComponent: Applies spring animation to the anchored position of a UI element. Ideal alternative for Transform Spring Component position spring when using it on UI.
LightIntensitySpring: Animates the intensity of a light source using spring behavior.
RigidbodyPositionSpring: Applies spring animation to the position of a Rigidbody component.

These components provide a convenient way to integrate spring animations into various aspects of your Unity project, such as visual elements, audio, and shaders. By leveraging the power of Spring Components, you can effortlessly enhance the interactivity and responsiveness of your game, creating a more engaging and polished experience for your players.

SpringComponent Public Methods

The SpringComponent partial Public Methods classes provide enhanced usability and easier access to spring functionality for each SpringComponent. While all methods are accessible through the springs directly (for example: FloatSpringComponent.springFloat.SetTarget), these Public Methods classes offer shorter, more convenient calls and serve as self-contained documentation.

Each SpringComponent has a corresponding partial class that exposes key methods for manipulating and querying the spring's state. This approach allows for more intuitive and type-safe interaction with the springs from your scripts. These partial classes are located in: Assets\AllIn1SpringsToolkit\Scripts\SpringsAPI

These methods provide direct access to the spring's properties, allowing you to easily get or set values such as the target, current value, velocity, force, and drag. You can also manipulate clamping behavior and add velocity for more dynamic animations.

Similar partial classes exist for all SpringComponents, each tailored to the specific spring type they represent. This consistent API across all spring types ensures a uniform interaction model, making it easier to work with different spring components in your projects.

The methods contained in these classes are the basic methods we'll use to interact with the Target, Current Value, Velocity, Drag, Force and Clamping. These are the main methods all partial classes contain:

- **GetTarget() and SetTarget()**: Get or set the target value of the spring.
- **GetCurrentValue() and SetCurrentValue()**: Get or set the current value of the spring.
- **GetVelocity() and SetVelocity()**: Get or set the velocity of the spring.
- **AddVelocity()**: Add velocity to the spring.
- **GetForce() and SetForce()**: Get or set the force applied to the spring.
- **GetDrag() and SetDrag()**: Get or set the drag applied to the spring.
- **SetMinValues() and SetMaxValues()**: Set the minimum and maximum values for clamping.
- **SetClampCurrentValues()**: Enable or disable clamping of the current values.

By using these public methods, you can create more readable and maintainable code when working with springs, as well as benefit from IDE auto-completion and type

checking. They offer a more streamlined way to interact with the springs while also serving as easily accessible documentation for the available operations on each spring type.

Transform Spring Component Breakdown

This is probably the main component you'll be using, and the one that will yield the most noticeable game polish results. Therefore, we recommend getting very familiar with this component. Also, getting familiar with this component will make using the other SpringComponents much easier, as this is the most complex one.

The spring will always need a FollowerTransform. By default, the current Transform will be assigned as the FollowerTransform. This component will take this FollowerTransform towards the different targets of the component using three different springs:

1. **Position Spring**: This spring animates the position of the FollowerTransform, allowing it to move towards the target position with a spring-like motion. It utilizes the Spring Vector3 type to handle the three-dimensional position.
2. **Scale Spring**: This spring animates the scale of the FollowerTransform, enabling it to interpolate smoothly between different scale values using a spring-like behavior. It utilizes the Spring Vector3 type to handle the three-dimensional scale.
3. **Rotation Spring**: This spring animates the rotation of the FollowerTransform, allowing it to rotate towards the target rotation with a spring-like motion. It utilizes the Spring Rotation type, which can handle complex rotational behavior.

Like all the Springs of the toolkit, these three springs in the TransformSpringComponent will have Force, Drag, Clamping, and all the expected features this document describes.

The target of this TransformSpringComponent can either be set by code if Use Transform As Target is false or managed automatically by the component if Use Transform As Target is true and a Target Transform is assigned in the Inspector.

Nudges and Punches (Free Game Polish!)

The term "nudge" refers to pushing a spring to trigger a reaction from it. As we learned earlier, springs don't have a fixed state; we can alter the Current Value or Velocity at any time, and the spring will react and adapt to the change without issues. This ability to

nudge the springs gives us the opportunity to add a lot of polish and liveliness to our projects with just a single line of code.

We typically consider two different ways of nudging or punching (we'll use these terms interchangeably):

1. **SetValue**: By instantly changing the Current Value using the SetValue method, we can ensure that the Target is different from the Current Value, thereby guaranteeing that an animation will occur. There are some examples of this in the Demo. For instance, in an action game where characters can get hit, we could set the current value of the object's scale on the frame it's hit, causing the object to grow and then animate back to its resting state.
2. **AddVelocity**: Similarly, using the AddVelocity method will cause the spring to move. If the spring is at rest with a velocity of 0, it will animate out of the resting state and eventually return to rest. If the velocity was non-zero, it will still influence and nudge the spring in a satisfying way.

Exploring the Demo examples is the best way to see all of these concepts in action and gain a deeper understanding of how nudges and punches can enhance the game juice of your projects.

Create your own springs (extend the asset)

The toolkit allows advanced users to extend its functionality by creating custom SpringComponents tailored to their specific needs. If the existing SpringComponents don't meet your requirements, you can create your own by inheriting from the SpringComponent class.

To create your own SpringComponent, follow these steps:

1. **Create a new script**: Start by creating a new script that inherits from SpringComponent. This will be the foundation of your custom component.
2. **Declare your Springs**: In your custom SpringComponent class, declare the Spring(s) that you will use in the component. For example, you might declare a SpringFloat like this: `public SpringFloat mySpringFloat;`
3. **Register Springs**: Implement the RegisterSprings method and call RegisterSpring(spring) for each spring you declared. This ensures that the asset manages the lifecycle and update of every spring.
4. **Implement IsValidSpringComponent()**: This method should return true or false depending on whether the component is properly set up. It's used to ensure that

all necessary references are present and that all configurations are correct. If false is returned, the SpringComponent is automatically disabled.

5. **Set Initial Values**: In the SetInitialValues() method, set the initial values of your springs. This step is crucial, as otherwise, the springs will start at 0. You can use any logic to handle the initial values here. You can also make use of hasCustomInitialValues and hasCustomTarget that are part of SpringComponent. These booleans are used in the toolkit included implementations and you may want to use them too to branch the logic and set some other custom value such a serialized field when true. This approach provides flexibility in initializing your springs, allowing you to adapt to different scenarios and requirements within your custom SpringComponent.
6. **Update Logic**: In the Update() method (or any other method you prefer), implement the logic to use your springs. This is where you'll update values based on the registered springs' current values and apply them to your game objects or components.

By following these steps, you can create custom SpringComponents that integrate seamlessly with the toolkit. This allows you to leverage its powerful spring functionality while tailoring it to your specific project requirements.

For a practical example, you can refer to the NewSpringComponentExample.cs class included in the toolkit, which demonstrates these steps in action and documents the process.

Add Components at runtime

The asset wasn’t really designed with AddComponent in mind. Adding a SpringComponent in the Inspector, setting it up and optionally referencing it from some other script is always the ideal recommended use case.

But if, for some reason, you want or need to add a SpringComponent at runtime this is how you do it:
1. Add a SpringComponent in runtime
2. Setup the SpringComponent
3. Initialize the SpringComponent
4. Use it normally

Here 2 examples:

```
colorSpring = gameObject.AddComponent<ColorSpringComponent>();

colorSpring.SetAutoUpdateToTrue(isRenderer: false,
uiImage.gameObject);

colorSpring.Initialize();

colorSpring.SetCurrentValue(Color.white);

colorSpring.SetTarget(Color.yellow);
```

And:

```
transformSpringComponent =
uiImage.gameObject.AddComponent<TransformSpringComponent>();

transformSpringComponent.useTransformAsTarget = true;

transformSpringComponent.followerTransform = uiImage.transform;

transformSpringComponent.targetTransform = uiImageTarget.transform;

transformSpringComponent.SpringScaleEnabled = false;

transformSpringComponent.Initialize();
```

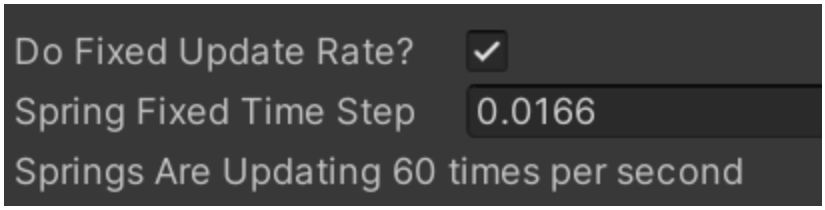
You can find the full example from above in:
Assets\AllIn1SpringsToolkit\Demo\Scripts\ExampleAddComponent.cs

Asset Window

You can access this Window in Tools/AllIn1/SpringsWindow, in the top bar of the Unity editor: Tools -> AllIn1 -> SpringsWindow

Once opened you'll see something like this. You have 3 tabs. The first one is the Springs Debugger that is explained in the next section.

In the Spring Settings tab you'll find the Fixed Update rate. When enabled, the springs, similarly to how Unity physics work, will update a fixed amount of times per second. Meaning that some frames, if the framerate isn't stable, it may update more than once and that some others it may not even update at all.

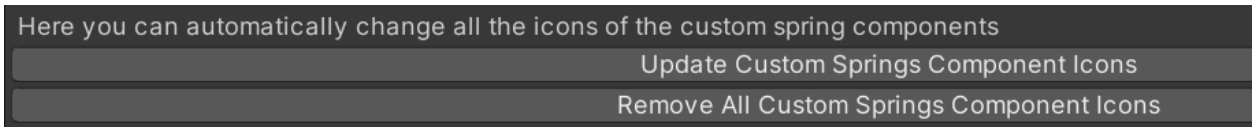


This feature makes sure that the springs are always stable, even in very low frame rate situations. And it also ensures that the springs animations always look and feel exactly the same in all frame rates.

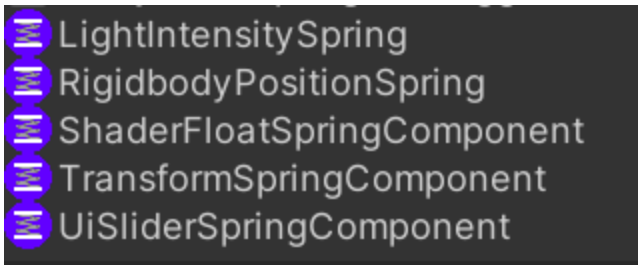
Fixed Time Step can be changed according to your project needs, you can even dynamically change it if you really want to. But for most projects the default value will work great.

The feature can be disabled if you wish to, but if the game dips below 10-15 frames you may get some strange behaviors.

In the Other tab (only in Unity version 2021.2 onward) you'll see 2 buttons to automate the addition and removal of the asset custom icon to the Spring Components found in the project.



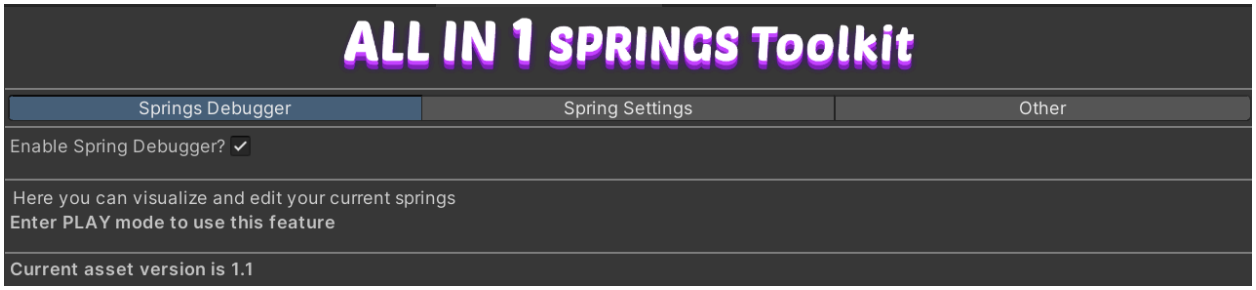
You'll notice that all SpringComponents have a custom icon:



This purple icon will also show in the Scene view when Gizmos are enabled. If you want to remove all purple icons or you want to add them to the new SpringComponents you created you can use these shortcut buttons.

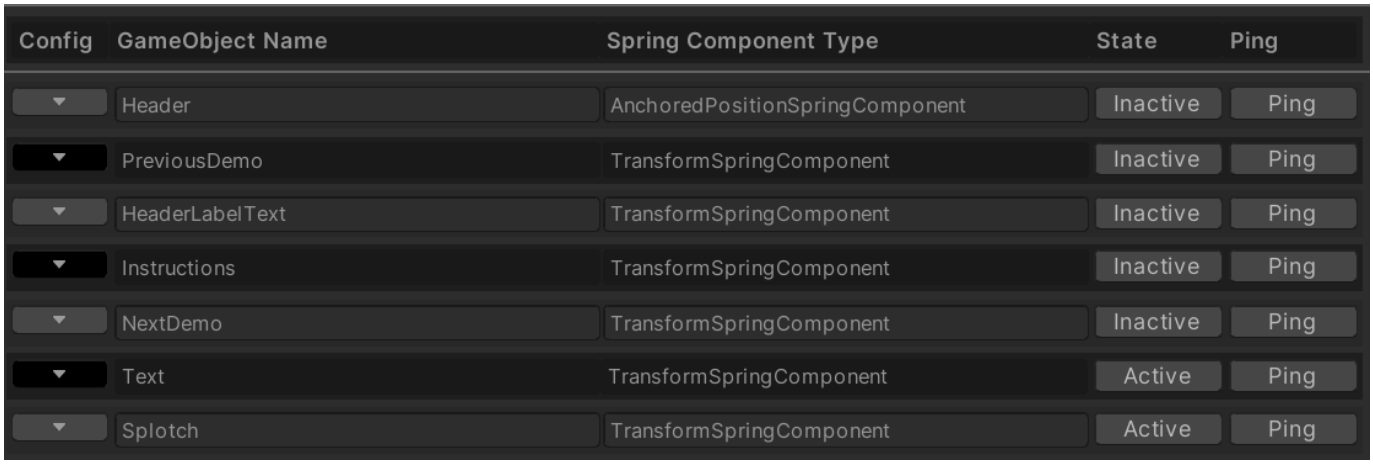
Spring Debugger

To enable the Spring Debugger, once the Asset Window is open, simply set "Enable Spring Debugger" to true:



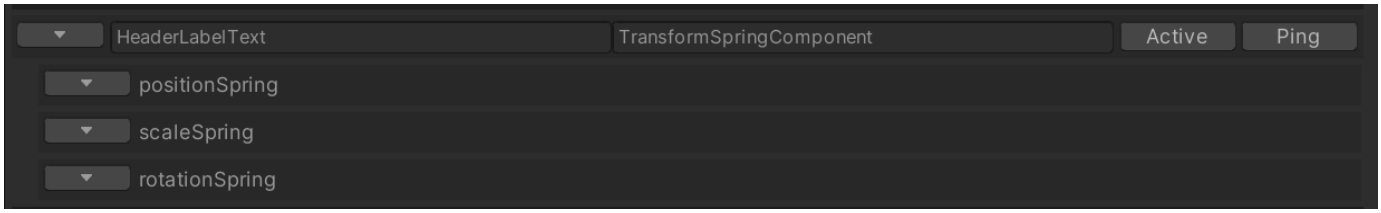
When the Debugger is enabled, springs will be registered and unregistered from the debugger automatically, allowing you to visualize, inspect, and interact with all the springs in the scene from this window. The Debugger has a tiny performance overhead, which is why you can choose to have it enabled or disabled. However, in all cases, it only runs in the Unity Editor. Even if the Debugger is enabled, it will never run or cause any overhead during a final build.

When the game is running and the Debugger is enabled, you'll see something like this:



In these headers, you can see the name of the object, the Spring Component Type, the State button to toggle the object on and off, and the Ping button to highlight the GameObject in the Hierarchy.

When expanding any item with the left button with the down arrow you'll see the springs registered in the SpringComponent:



In this case, we can see all the position, scale, and rotation springs of the TransformSpringComponent. If we open any of these with the down arrow, we'll see the same UI that the TransformSpringComponent shows. We can use this to monitor the active springs in the scene and, most importantly, to change their values or quickly iterate on Nudge values. Instead of creating a script and changing the values from there, we can open a spring in the Debugger and try different Nudge values, which we can then bring over to any script.

Finally, if you press Alt when pressing any arrow button, you'll affect the whole hierarchy, and all other elements will perform the alternate command of the button you pressed. So if you do Alt+Click to open a spring, all other springs in the list will close.

The Spring Debugger is a powerful tool for visualizing, inspecting, and interacting with the springs in your scene, allowing you to fine-tune and iterate on your spring animations efficiently.

Springs vs Tweens

While springs are a powerful tool for creating lively and responsive animations, they are not always the best solution. In this section, we'll dissect both springs and tweens to understand when it's best to use one or the other.

Tweens, short for "in-betweening," are a popular yet rudimentary solution that involves interpolating between two values over time. This interpolation can be done based on a curve or equation to control its motion. However, tweens have a rigid set of parameters and a state that needs to be tracked. If an object is being affected by a tween, we must ensure to stop it before starting a new one. Additionally, given the heavily parameterized nature of tweens, it may be challenging to code complex, elaborate behaviors.

On the other hand, springs have no state; they update every frame based on some parameters and a target value. They don't care about any changes that happen; they

will keep running and updating the state, trying to reach equilibrium. This lack of state allows for very organic and elaborate results with minimal effort.

Tweens are better suited for:

1. **Simple Scenarios**: Going from point A to B, executing a single, uninterrupted animation.
2. **Performance**: In some cases, tweens may have a slight performance advantage. However, this is generally not a concern unless you're developing for low-end platforms and abusing the number of springs. Always profile your projects when evaluating and making performance decisions; you'll be amazed at how many springs you can use with almost no performance cost.

On the other hand, springs are better for:

1. **Organic, Lively Animations**: Springs excel at producing natural, lifelike motion that feels responsive and organic.
2. **Stateless, Highly Interactive**: Springs can react dynamically to user input or external forces, making them ideal for creating responsive and engaging interactions.
3. **Continuous Transitions**: Spring animations can seamlessly transition between different states without abrupt changes or interruptions, creating a more cohesive and polished experience.
4. **Game Polish and Juiciness**: The responsiveness and natural motion of springs contribute significantly to the overall "game feel" and juiciness of a project, enhancing player satisfaction and engagement.

While tweens offer simplicity and predictability, springs provide a level of liveliness, responsiveness, and organic motion that can elevate the user experience of your Unity projects.

Next Update Pending Features

While version 1.0 includes all essential functionalities, some additional features are currently in development. These upcoming enhancements will be released in a future update:

1. **Rotation Spring Clamping**: Currently, the rotation spring follows any target without the ability to define clamping. We're working on implementing customizable clamping options to give users more control over rotation limits.

2. **Spring Bone Chains**: Single bone chains are functioning well, but we recognize the need for more complex setups. We're developing a solution to effectively chain multiple bones and make the feature more robust.

Credits

Gerard Belenguer (https://twitter.com/GerardBelenguer): Direction and engineering.
José Javier Delgado: Engineering.

A few assets under CC0 1.0 Universal license are sourced from Kenney, Kay Lousberg and Quaternius:
https://www.kenney.nl/assets (Demo buttons and other UI textures)
https://kaylousberg.itch.io/prototype-bits (Dummies demo mesh)
https://quaternius.com/packs/ultimatespacekit.html (Spaceship demo meshes)

The font used is Poetsen One (Open Font License from Google):
https://fonts.google.com/specimen/Poetsen+One

Special thanks to my friend Antón Miranda for putting together an amazing cover art image, trailer and screenshots for the storefront.