This document will provide you with a quick overview of the workflow process in the FPS Animation Framework. **It is highly recommended to visit the online documentation.**

👋 Welcome!

Scriptable Animation System is a brand-new animation system for the Unity, which introduces a new way of implementing procedural animation features in your project.

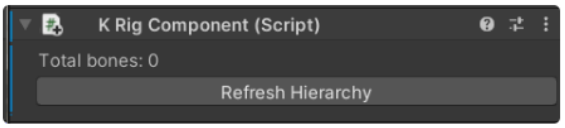
In this document, we will cover the main features of the system, as well as differences with the previous version.

Character Rig

In this section we will set up the charcater rig.

Adding IK Objects

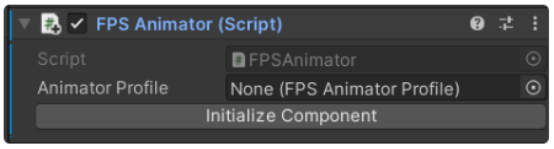
Add your character to the scene. Locate the root bone of the hierarchy and add the **Rig Component** to it:



Rig Component.

This component contains the information about the character skeleton. It is used in runtime to re-trive the Transform bone references.

Now, go back to your charcater Game Object and add the **FPSAnimator** component:



FPS Animator Component.

Now press the *Initialize Component* button. This action will add the IK Objects to your character:

IK Objects.

All **IK Objects** follow the same naming conveniton in the project: "IK_ObjectName", where "IK" stands for Inverse Kinematics, and "ObjectName" defines the actual name.

All **IK Objects** have a **KVirtualElement** attached to them - this component is used to dynamically copy the animation data from the real bone to the **IK Object**.

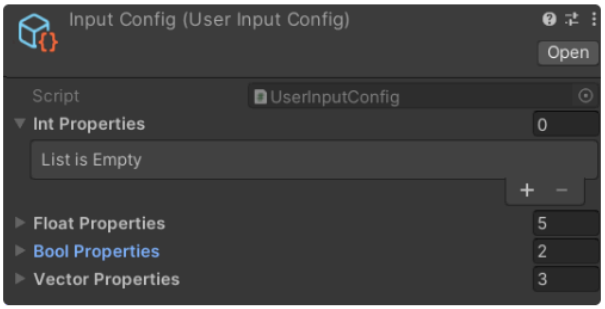
> **Example:** the IK RightHand object will use the Right Hand bone as a targetBone in the Virtual Element component. This will make the IK RightHand copy the transform of the Right Hand bone in runtime, which is essential for procedural animation.

> **Note:** if there are any issues with your skeleton, like non-standard naming convention for bones, you will see warning and error messages in the log. You will have to manually add those Game Objects, as well as Virtual Element components.

At this point, we have prepared the character skeleton. In the future, if you make any changes to the hierarchy, make sure to *Refresh Hierarchy* in the **Rig Component**. This will make the system aware of your newly added Game Objects, so you can use them in the Animator Layers.

Create Input Config

Now we need to create an **UserInputConfig**. Right click in any folder and go to **KINEMATION/Input Config**:



Input Config.

Input Config contains what runtime properties will be used in our system. It is essential for communication between the FPS Animation Framework entites and your custom code.

So far the **Input Config** supports 4 types of properties: Int, Float, Bool and Vector4.

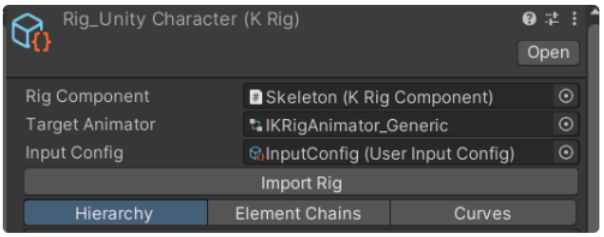
> **Tip:** Floats have a special interpolation feature, which allows the system to automatically interpolate the value once it's changed. This feature is useful when you want a smooth transition from one value to another.

> **Example:** let's say you want to disable FPSAnimator upper body override. To do this, you only need to grab a reference to the **IUserInputController** and call the `SetValue(string, object)` method to set the value for *"StabilizationWeight"* and *"PlayablesWeight"*.

Create Rig Asset

> **Tip:** On of the groundbreaking features implemented in the Scriptable Animation System is the new way of storing the information about the character skeleton - **Rig Assets**. They contain all the crucial information about the character, and they are used to dynamically select *Rig Elements*, *Rig Chains* and *Curves* in the Editor.

It is time to create a **Rig Asset**. Right click in any folder and go to **KINEMATION/Rig.**



Rig Asset.

The **Rig Asset** contains all the information about your character skeleton, including:

- Hierarchy - actual hierarchy of your character.
- Element Chains - an alternative to Avatar Masks.
- Curves - Playables curves for dynamic animations (e.g. reloading, grenade throw, etc.)

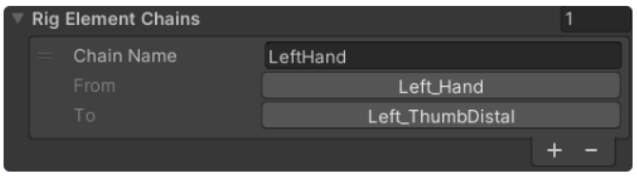
Now we need to set the key properties:

1. Set the **Rig Component** reference to your character's Rig Component.
2. Set the **Target Animator** to the desired Animator Controller.
3. Set the **Input Config** to the asset we created a step earlier.

Finally, click the *Import Rig* button. If you are making any changes to the character skeleton, always make sure to update the **Rig Asset** - this is crucial for the system to retrieve Transform bone references in runtime.

Add Element Chains

Element Chains is a new feautre in the FPS Animation Framework. It represents a sequence of elements (usually bones), which is used by Animator Layers in runtime. For example, if we want to attach the left hand to the weapon, we will need to add a respective **Element Chain**:



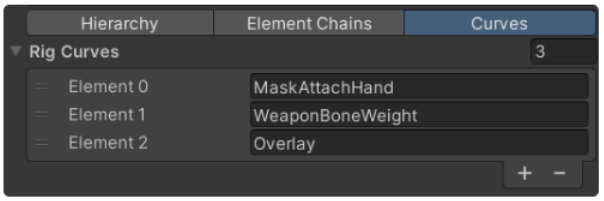
Left Hand chain includes the left hand and its fingers.

The main benefit of **Element Chains** is that it's easier to set them up, unlike Avatar Masks. Plus, you have all the chains in the Rig Asset, which makes it even more convenient.

Add Curves

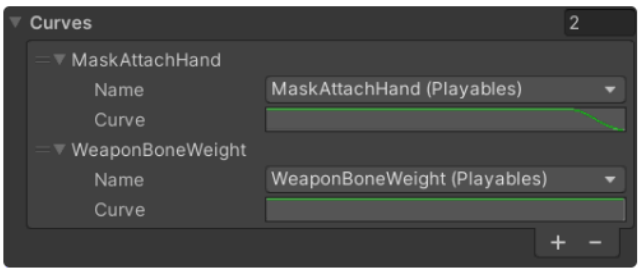
Now let's go to the Curves section in our **Rig Asset**. By default, the list will be empty. Let's fix that by adding:

- MaskAttachHand
- WeaponBoneWeight
- Overlay



You can add custom curves here.

These curves will be used in the Playables system for custom animations, like reloading, grenade throw or any other dynamic fire-and-forget type of animation. Here is an example from the demo project:



AA_AK12_ReloadEmpty Example.

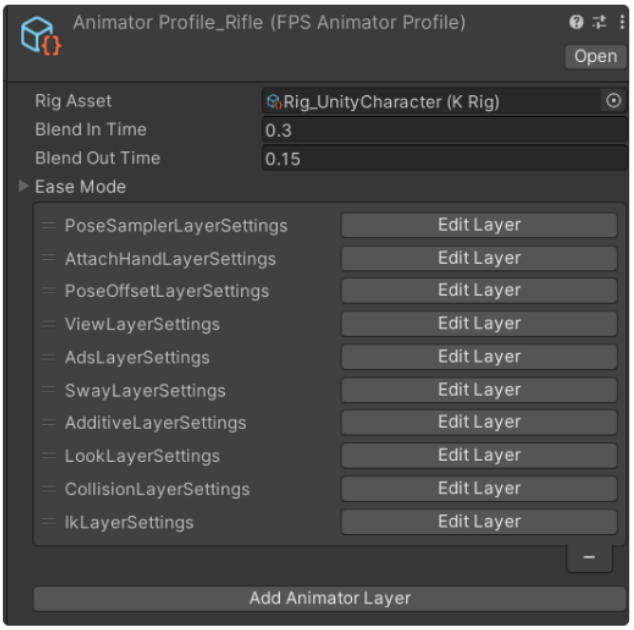
The character skeleton is now ready, in the next section we add Animator Profile and Layers to our system!

Profiles and Layers

In this section we will work with Animator Profiles and Layers.

Unlike the previous version of the **FPS Animation Framework**, the **Scriptable Animation System** does not depend on weapons or items directly, instead it operates in **Animator Profiles** - a collection of animation features.

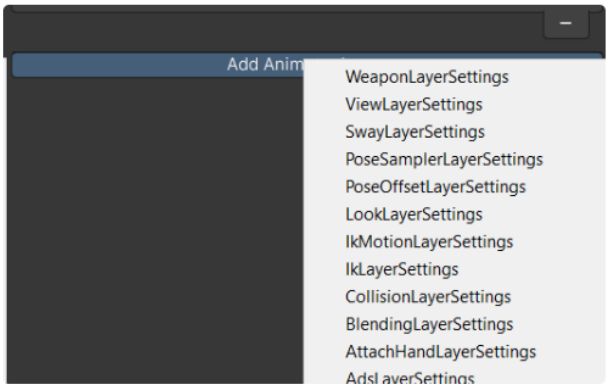
To create a new profile, right-click in any folder and go to **KINEMATION/FPS Animator General/Animator Profile:**



An example asset from the demo.

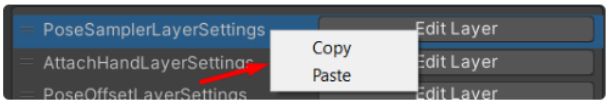
- **Rig Asset** - assign previously created Rig Asset. It will be used for all Animator Layers.
- **Blend In/Out Time**- defines the blending time for this Profile in seconds.
- **Ease Mode** - the easing function for the blending.

To add a new **Animator Layer**, click on the *Add Animator Layer* button and select the desired type from the list:



Animator Layer selection.

You can copy/paste layer settings by right-clicking on the layer:



Context menu.

Now you can start adding Animator Layers to the profile. There are no mandatory layers for the system, which makes it incredibly flexibly not just for items or weapons, but game play situations as well.

You can find out more about each layer and its settings in **Animator Profiles** section. You will also find the use-cases and examples there.

In the next section we will find out how to link Animator Profiles.

Linking

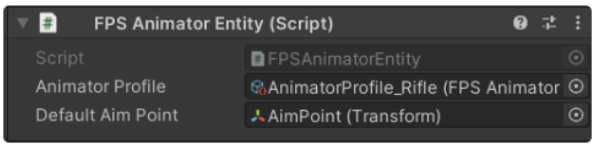
In this section we will learn how to link layers.

The linking process is used to smoothly blend in our desired Animator Profile in runtime.

Let's take at the linking functionality in the source code:

```
public void UnlinkAnimatorProfile() { ... }
public void LinkAnimatorProfile(GameObject itemEntity) { ... }
public void LinkAnimatorProfile(FPSAnimatorProfile newProfile) { ... }
```

The system has a way to link a Game Object, if it contains the Animator Profile. This is only possible, if your Game Object has an **FPSEntityComponent**:



Entity can be a weapon, item or a game play scenario.

Then, in your controller code you can just pass the weapon Game Object to the **LinkAnimatorProfile** method, and it will work as expected.

The **UnlinkAnimatorProfile** is used to blend out the currently active profile, which might be useful when you do not want to use the system at all.

Code example

Let's see how we can apply the linking in practice. Here's a code snippet from the demo:

FPSController.cs

```
private void EquipWeapon()
{
    ...
    var gun = _instantiatedWeapons[_activeWeaponIndex];
    _fpsAnimator.LinkAnimatorProfile(gun.gameObject); // Linking a new profile.
    _animator.CrossFade("CurveEquip", 0.15f);
    ...
}
```

In the demo project, when we are changing weapons, we effectively link a new animator profile from the active weapon.

You can link different animator profiles depending on the game play. For example, in the demo it is possible to achieve the unarmed state by pressing the T key:

FPSController.cs

```
if (Input.GetKeyDown(KeyCode.T))
{
    _isUnarmed = !_isUnarmed;

    if (_isUnarmed)
    {
        GetGun().gameObject.SetActive(false);

        _animator.SetFloat(OverlayType, 0);
        _userInput.SetValue(FPSAConstants.PlayablesWeightProperty, 0f);
        _userInput.SetValue("StabilizationWeight", 0f);
        _fpsAnimator.LinkAnimatorProfile(unarmedProfile);
    }
    else
    {
        GetGun().gameObject.SetActive(true);

        _animator.SetFloat(OverlayType, (int) GetGun().overlayType);
        _userInput.SetValue("PlayablesWeight", 1f);
        _userInput.SetValue("StabilizationWeight", 1f);
        _fpsAnimator.LinkAnimatorProfile(GetGun().gameObject);
    }
}
```

As you can see, here we link an unarmedProfile whne toggling our feature. Additionally, we use the *SetValue* method of the **UserInputController** to adjust the Playables influence over the character upper body, so our Animator has a full control over the character pose.

Integration

In this section we will learn how to integrate the system.

Scriptable Animation System offers a streamlined integration process. Let's start with the custom controller integartion. We will use the FPSController script from the demo project as an example.

Initialization

First, you need to add references to the framework core components:

```
private FPSAnimator _fpsAnimator; // Central management component.
private IUserInputController _userInput; // Dynamic input system.
private IFPSCameraController _fpsCamera; // Camera system.
private IPlayablesController _playablesController; // Dynamic animation system.
private RecoilAnimation _recoilAnimation; // Recoil effect component.
```

Then, make sure to initialize all the components:

```
_fpsAnimator = GetComponent<FPSAnimator>();
_userInput = GetComponent<IUserInputController>();
_fpsCamera = GetComponentInChildren<IFPSCameraController>();
_playablesController = GetComponent<IPlayablesController>();
_recoilAnimation = GetComponent<RecoilAnimation>();
```

Weapon change

When switching weapons, we need to link a new Animator Profile:

```
private void EquipWeapon()
{
    ...

    // Where gun is your active weapon/item.
    _fpsAnimator.LinkAnimatorProfile(gun.gameObject);
    _animator.CrossFade("CurveEquip", 0.15f);
}
```

Additionally we play an Equip animation, which is optional and depends on your project implementation specifically. A similar logic is applied to the UnEquip method.

Aiming

```
private void ToggleAiming()
{
    if (_aimState != FPSAimState.Aiming)
    {
        _aimState = FPSAimState.Aiming;
        _userInput.SetValue("IsAiming", true);
        _fpsCamera.UpdateTargetFOV(60f);
    }
    else
    {
        DisableAim();
        _fpsCamera.UpdateTargetFOV(90f);
    }

    _recoilAnimation.isAiming = IsAiming();
}
```

Additionally, we access the **FPSCameraController**, and adjust the target FOV. We also adjust the recoilAnimation aiming status, which is important as it will adjust the animation accordingly.

Playing animations

To play an animation, you need to call the:

```
// Where yourAnimation is an Animation Asset.
_playablesController.PlayAnimation(yourAnimation, 0f);
```

You can also specify the start time of the animation as a second parameter, which might be useful for mechanics like staged reloads.

Recoil

The recoil is implemented via Recoil Animation component and Camera Shakes. First we need to initialize the Recoil Animation component when a gun is equipped:

```
_recoilAnimation.Init(gun.recoilData, gun.fireRate, gun.isAuto ? FireMode.Auto :
FireMode.Semi);
```

You only need to add the Recoil Data to your custom weapon class. The Recoil Data is a Scriptable Object, that contains the information about the procedural recoil animation.

Next, we need to apply the recoil effect in runtime:

```
// Called every shot.
private void Fire()
{
    // Make sure to add an FPSCameraShake public field to your weapon class.
    _fpsCamera.PlayCameraShake(GetGun().cameraShake);
    if (_recoilAnimation != null) _recoilAnimation.Play();
}

// Called when firing key is released.
private void StopFiring()
{
    if (_recoilAnimation != null) _recoilAnimation.Stop();
}
```

Making adjustments

Sometimes it is necessary to adjust our system's behavior in runtime, depending on gameplay situations.

In the **Scriptable Animation System** it is implemented primarily via the Input System:

```
private void OnSprintStarted()
{
    _userInput.SetValue("StabilizationWeight", 0f);
    _userInput.SetValue("PlayablesWeight", 0f);
    _userInput.SetValue("LookLayerWeight", 0.3f);
}

private void OnSprintEnded()
{
    _userInput.SetValue("StabilizationWeight", 1f);
    _userInput.SetValue("PlayablesWeight", 1f);
    _userInput.SetValue("LookLayerWeight", 1f);
}
```

In the example above, we toggle the Playables systems and stabilization based on the sprinting status. You can adjust custom properties in a similar way, depending on the requirements of your project.