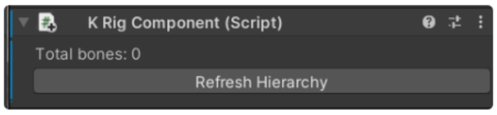
Character Rig

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:



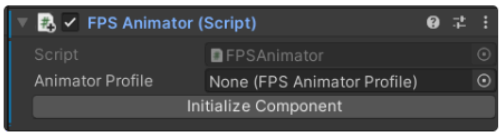
K Rig Component (Script)

Total bones: 0

Rig Component.

> **Tip:** 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 character Game Object and add the **FPS Animator** component to it. Then, press the *Initialize Component* button:



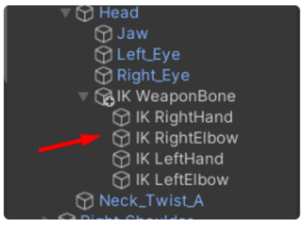
FPS Animator (Script)

Script: FPSAnimator

Animator Profile: None (FPS Animator Profile)

FPS Animator Component.

This action will add the IK Objects to your character:



```
Head
  Jaw
  Left_Eye
  Right_Eye
  IK WeaponBone
    IK RightHand
    IK RightElbow
    IK LeftHand
    IK LeftElbow
Neck_Twist_A
```

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**.

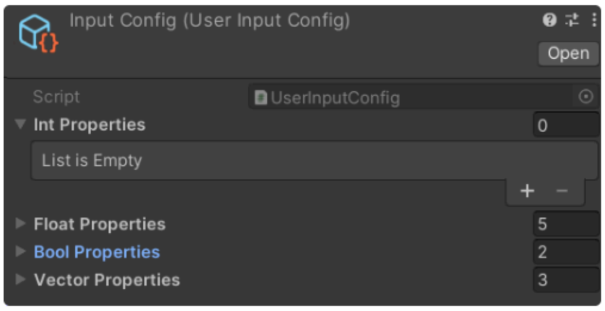
Example: the IK RightHand object will use the Right Hand bone as a targetBone in the Virtual Element component. This will make the IK RightHand copy the transform of the Right Hand bone in runtime, which is essential for procedural animation.

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 (User Input Config)

Open

Script	UserInputConfig
Int Properties	0
List is Empty	
Float Properties	5
Bool Properties	2
Vector Properties	3

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.

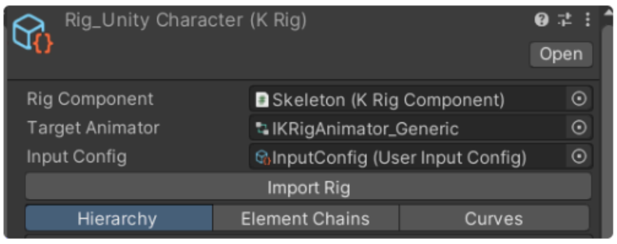
Tip: Floats have a special interpolation feature, which allows the system to automatically interpolate the value once it's changed. This feature is useful when you want a smooth transition from one value to another.

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_Unity Character (K Rig)

Rig Component: Skeleton (K Rig Component)
Target Animator: IKRigAnimator_Generic
Input Config: InputConfig (User Input Config)

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.)

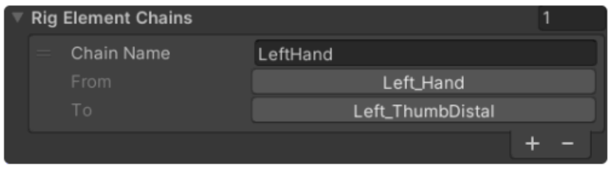
Now we need to set the key properties:

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**:


Rig Element Chains 1
Chain Name: LeftHand
From: Left_Hand
To: Left_ThumbDistal

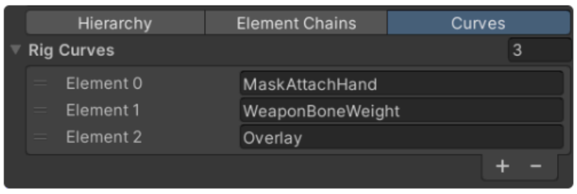
Left Hand chain includes the left hand and its fingers.

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:

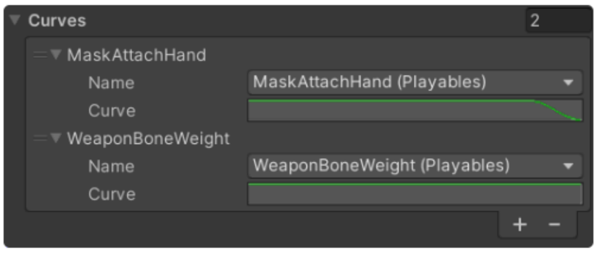
- MaskAttachHand
- WeaponBoneWeight
- Overlay



Rig Curves 3
Element 0: MaskAttachHand
Element 1: WeaponBoneWeight
Element 2: 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:


Curves 2
MaskAttachHand
Name: MaskAttachHand (Playables)
Curve
WeaponBoneWeight
Name: WeaponBoneWeight (Playables)
Curve

AA_AK12_ReloadEmpty Example.

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

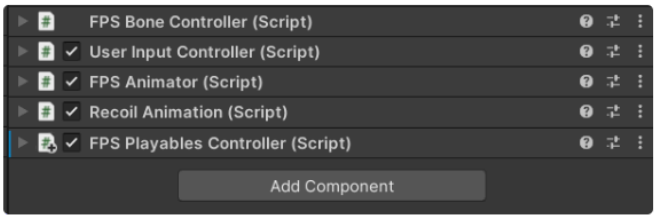
Components

In this section we will quickly discuss the core components.

Add Components

FPS Animator will automatically add the **FPS Bone Controller** and **User Input Controller**. Now you need to add the **FPS Playables Controller** and the **Recoil Animation** components.

Your character inspector should look something like this:


FPS Bone Controller (Script)
User Input Controller (Script)
FPS Animator (Script)
Recoil Animation (Script)
FPS Playables Controller (Script)
Add Component

Make sure all of these components are added.

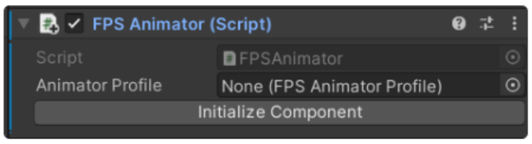
Now let's see roles of each component we have just added.

FPS Animator

FPS Animator is a central hub component, which controls the flow of other entities in the framework. This component initiailizes, updates and disposes Bone Controller, User Input Controller and Playables Controller.

Later in this document we will cover the **Linking**, which is an extremely important part of this system.

You can specify the default Animator Profile, but this is optional. It is only useful for debugging purposes, in any other case it is recommended to dynamically link layers:


FPS Animator (Script)
Script FPSAnimator
Animator Profile None (FPS Animator Profile)
Initialize Component

Just leave it empty for now.

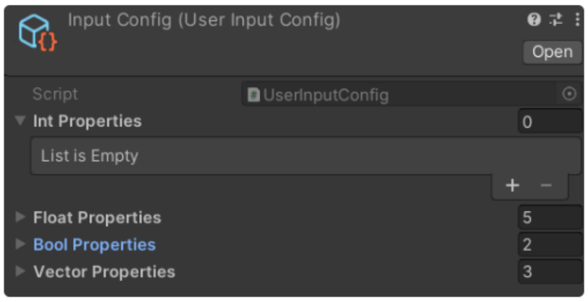
FPS Bone Controller

This component manages the procedural animation features. It is responsible for initialization, update, disposal and pose caching of procedural animations in realtime.

You do not have to directly reference this component in your code, as its flow is fully controlled by the FPS Animator.

User Input Controller

The **FPS Animation Framework** introduces a standalone input property system. The idea is that you can define custom properties in the **Input Config** asset and then use them in runtime:


Input Config (User Input Config)

Open

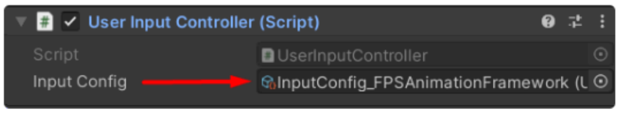
Script	UserInputConfig
Int Properties	0
List is Empty	
Float Properties	5
Bool Properties	2
Vector Properties	3

This asset contains your custom properties.

This system is used to establish communication between the framework entities and your project code-base.

> **Example:** let's say you want to disable aiming when jumping. You can create a new float parameter, set it to 1 or 0 in code based on the jumping condition, and then specify that parameter as a control value in the Ads Layer.

User Input Controller allows you to work with the property system, by getting or setting the values. This component only has a reference to the Input Config - make sure to assign the Input Config we created in the previous section:


User Input Controller (Script)

Script	UserInputController
Input Config	InputConfig_FPSAnimationFramework (U

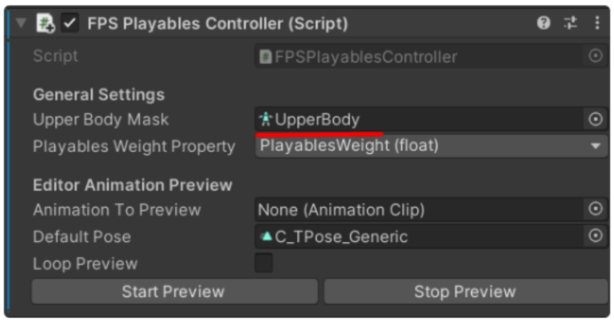
Specify the input asset.

> **Tip:** the User Input Component inspector has a reall neat feature - it displays all the active properties and their values in the playmode.

Playables Controller

This component is responsible for playing custom animations in realtime. As the name suggests, it utilizes Unity Playables API to override standard Animator Controller behaviour, by adding custom animation layers. These layers override the character upper body to play your reloading, grenade throwing and other types of motions.

You must specify the upper body Avatar Mask in the component inspector - this will tell the system what bones will be affected by custom animations:



FPS Playables Controller (Script)

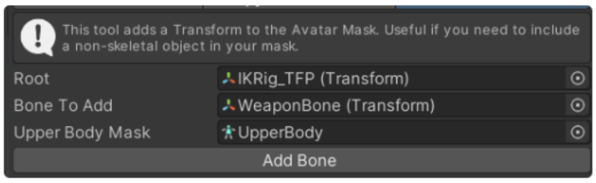
Script: FPSPlayablesController

General Settings
Upper Body Mask: UpperBody
Playables Weight Property: PlayablesWeight (float)

Editor Animation Preview
Animation To Preview: None (Animation Clip)
Default Pose: C_TPose_Generic
Loop Preview: ☐

Playables Controller inspector.

Your upper body mask needs to be adjusted in order to play custom animations. To do this, go to **Window/FPS ANIMATOR/Tools/Avatar Mask Modifier.** Now we need to add the WeaponBone object to the avatar mask. Specify the *Root* and the *Upper Body Mask*:



This tool adds a Transform to the Avatar Mask. Useful if you need to include a non-skeletal object in your mask.

Root: IKRig_TFP (Transform)
Bone To Add: WeaponBone (Transform)
Upper Body Mask: UpperBody

Press the Add Bone button.

Tip: the WeaponBone will be used later in the animation workflow, essentially it contains the weapon movement.

The **Playables Weight Property** is used to control the influence of the Playbles Controller. You can specify here any float value from the input system, but it is recommended to leave the PlayablesWeight as a default value.

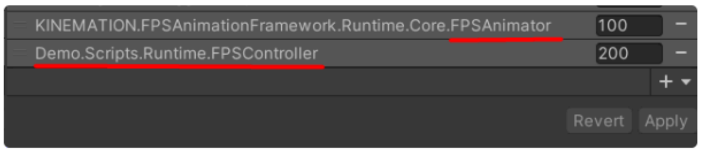
You can also use this component to preview animations in the editor. Just select your animations and hit the preview button!

Note: the **Playables Controller** is not added automatically, unlike other components. This is because of the **IPlayablesController** interface. This one is particularly useful if you have a custom system, like Animancer, which you would like to integrate directly with the framework.

All you have to do is to implement the **IPlayablesController** interface, and you can use your custom component in the **FPS Animation Framework.**

Execution order

Lastly, we need to slightly adjust the *Script Execution Order* in the *Project Settings*. Go to the **Edit/Project Settings/Script Execution Order** and make sure to put the **FPS Animator** before your controller class:



KINEMATION.FPSAnimationFramework.Runtime.Core.FPSAnimator 100
Demo.Scripts.Runtime.FPSController 200

FPSController should be your controller class, where you use the FPS Animator component.

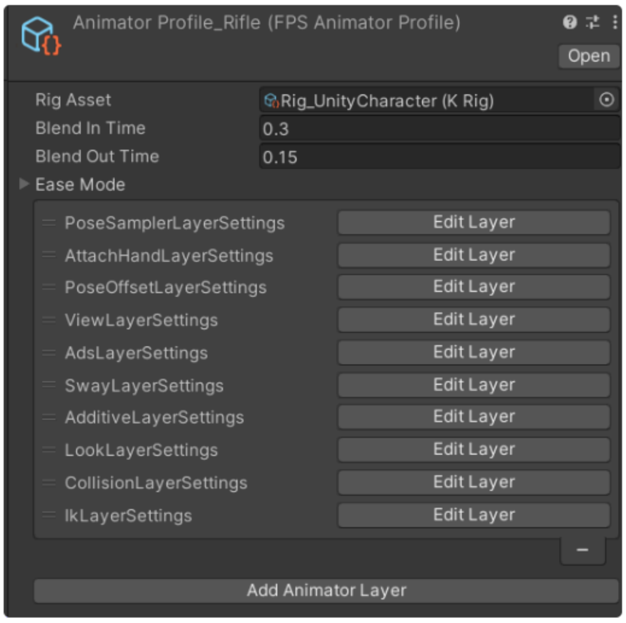
Tip: this step is required for the proper initialization process, so the **FPS Animator** can process all the mentioned above components.

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:**



Animator Profile_Rifle (FPS Animator Profile)

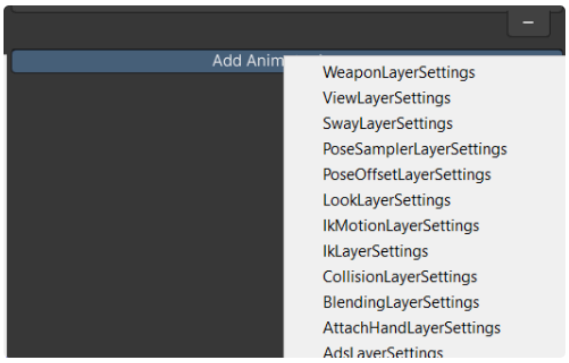
Rig Asset	Rig_UnityCharacter (K Rig)
Blend In Time	0.3
Blend Out Time	0.15
Ease Mode	
PoseSamplerLayerSettings	Edit Layer
AttachHandLayerSettings	Edit Layer
PoseOffsetLayerSettings	Edit Layer
ViewLayerSettings	Edit Layer
AdsLayerSettings	Edit Layer
SwayLayerSettings	Edit Layer
AdditiveLayerSettings	Edit Layer
LookLayerSettings	Edit Layer
CollisionLayerSettings	Edit Layer
IkLayerSettings	Edit Layer

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.

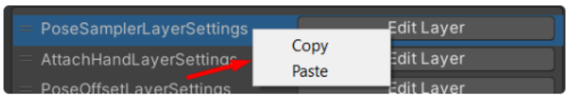
To add a new **Animator Layer**, click on the *Add Animator Layer* button and select the desired type from the list:

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

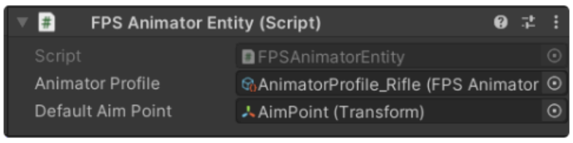
In this section we will learn how to link layers.

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**:


FPS Animator Entity (Script)

Script	FPSAnimatorEntity
Animator Profile	AnimatorProfile_Rifle (FPS Animator
Default Aim Point	AimPoint (Transform)

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 UserInputController _userInput; // Dynamic input system.
private FPSCameraController _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<UserInputController>();
_fpsCamera = GetComponentInChildren<FPSCameraController>();
_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(FPSANames.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.

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(FPSANames.StabilizationWeight, 0f);
    _userInput.SetValue(FPSANames.PlayablesWeight, 0f);
}

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

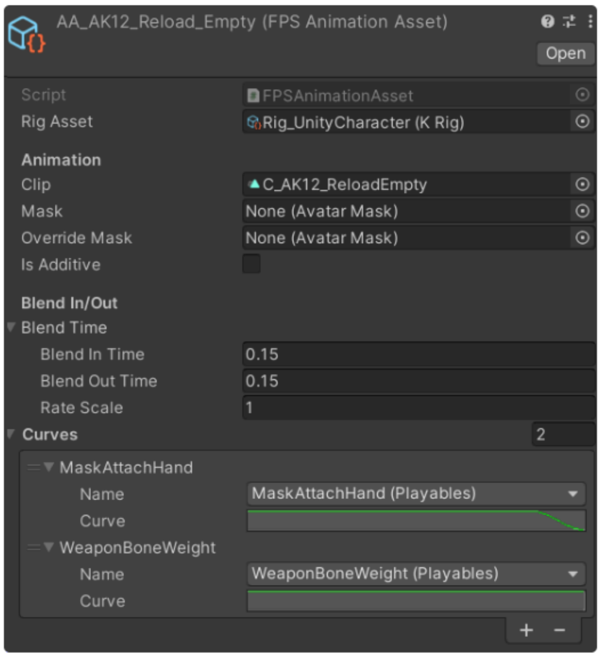
In the example above, we toggle the Playables systems and stabilization based on the sprinting status. You can adjust custom properties in a similar way, depending on the requirements of your project.

Playing Animations

In this section we will find out how to play custom animations in runtime.

Animation Asset

The **FPS Animation Framework** uses Animation Assets to play custom animations on your character from code. To create a new asset, right click and go to **KINEMATION/FPS Animator General/Animation Asset**:



AA_AK12_Reload_Empty (FPS Animation Asset)

Script	FPSAnimationAsset
Rig Asset	Rig_UnityCharacter (K Rig)
Animation	
Clip	C_AK12_ReloadEmpty
Mask	None (Avatar Mask)
Override Mask	None (Avatar Mask)
Is Additive	
Blend In/Out	
Blend Time	
Blend In Time	0.15
Blend Out Time	0.15
Rate Scale	1
Curves	2
MaskAttachHand	
Name	MaskAttachHand (Playables)
Curve	
WeaponBoneWeight	
Name	WeaponBoneWeight (Playables)
Curve	

AK12 reloading example.

First, you must specify the **Rig Asset** - it will be used to pick the curve names via a popup widget. Then, make sure to assign the actual Animation Clip.

Mask property is optional, but you can use it if you need to affect a specific part of the body.

Override Mask and **Is Additive** are used for shared animations, like emotes, grenade throws, etc. The **Override Mask** will make the left arm use the absolute animation, while the right arm will be only using additive motion.

Blend Time controls the transition for your animation. The **Rate Scale** is a playback speed multiplier.

The **Curves** list defines custom curves, which will be used by this animation. This is an extremely useful feature, when you want custom animations to control animation features.

> ✓ **Example:** let's say we want to disable left-hand IK when reloading. Add the MaskAttachHand curve to the animation, set its value to 1. Now we can use this curve as a mask parameter in the left-hand IK layer, which will blend it out when the animation is playing.

Code

To play an Animation Asset from code, you need to call the PlayAnimation method on the **IPlayablesController**:

```
// Where yourAnimation is an Animation Asset.
// Second parameter is the start time.
_playablesController.PlayAnimation(yourAnimation, 0f);
```

💻 Extending the System

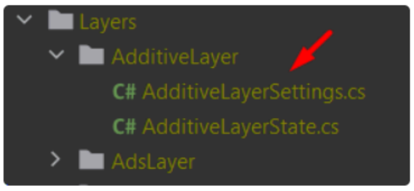
In this section you will learn how to create custom animation modules.

Overview

Animator Layer represents 2 classes:

- **FPSAnimatorLayerSettings:** a Scriptable Object that contains all the data.
- **FPSAnimatorLayerState:** a class that contains all the animation logic.

If you open up the source code of the framework, you will notice that every layer has its own folder with 2 classes:



```
Layers
  AdditiveLayer
    C# AdditiveLayerSettings.cs
    C# AdditiveLayerState.cs
  AdsLayer
```

AdditiveLayer example.

FPSAnimatorLayerSettings class is responsible not just for containing the data, but for instantiating the **FPSAnimatorLayerState** as well:

AdditiveLayerSettings.cs

```
public class AdditiveLayerSettings : WeaponLayerSettings
{
    ...
    public override FPSAnimatorLayerState CreateState()
    {
        return new AdditiveLayerState();
    }
}
```

CreateState() method must return a new instance of the desired **FPSAnimatorLayerState**-type. When you are linking a new **Animator Profile**, the system iterates over all animation features (*Layer Settings*) and then invokes the *CreateState()* method to create the *Layer State*.

Now let's actually create a custom animator layer.

Example

First, create new **FPSAnimatorLayerSettings**- and **FPSAnimatorLayer**-derived classes:

YourFeatureLayerSettings.cs

```
public class YourFeatureLayerSettings : FPSAnimatorLayerSettings
{
    //Define your settings here, including the KRigElements.
    public KRigElement myRigElement;

    public override YourFeatureLayerState CreateState()
    {
        return new YourFeatureLayerState();
    }

#if UNITY_EDITOR
    public override void OnRigUpdated()
    {
        base.OnRigUpdated();

        // (!) Always update KRigElements here.
        // Called when a rig asset is updated/changed.
        UpdateRigElement(ref myRigElement);
    }
#endif
}
```

YourFeatureLayerState.cs

```
public class AdditiveLayerState : FPSAnimatorLayerState
{
    private YourFeatureLayerSettings _settings;

    public override void InitializeState(FPSAnimatorLayerSettings newSettings)
    {
        base.InitializeState(newSettings);
        _settings = (YourFeatureLayerSettings) newSettings;
    }

    public override void OnEntityUpdated(FPSAnimatorEntity newEntity)
    {
        // Called when a weapon/item is equipped.
        // Use this callback for weapon/item-related features.
    }

    public override void OnGameThreadUpdate()
    {
        // Use this to collect general, game-thread data.
        // It is safe to access character references here.
        // Called on Update() before the main animation pass.
    }
```

```
// Use this callback to re-initialize values.
// It is only called when the Link Dynamically is true.
public override void OnLayerLinked(FPSAnimatorLayerSettings newSettings)
{
    _settings = (YourFeatureLayerSettings) newSettings;
}

// Make sure to manually register all the bones affected by this state.
// This is required for smooth blending and pose caching.
public override void RegisterBones(ref HashSet<int> registeredBones)
{
    registeredBones.Add(_settings.myRigElement);
}

// Make sure to cache the poses for active bones.
public override void CachePoses(ref List<KPose> cachedPoses)
{
    cachedPoses.Add(new KPose()
    {
        element = _settings.myRigElement,
        modifyMode = EModifyMode.Add,
        pose = KTransform.Identity,
        space = ESpaceType.ComponentSpace
    });

    // Add other poses here.
}
```