Welcome to the FPS Animation Framework documentation!

by Kinemation

- Tutorial
  - Prerequisites
  - Basic setup
  - Integration
    - Core Component
    - Recoil Animation
    - Animation layers
  - Summary and quick guide
  - Camera recoil
  - Camera and stabilization
  - Dynamic motion
- Animation layers
  - Ads layer
  - Blending Layer
  - Left-Hand IK Layer
  - Locomotion Layer
  - Look Layer
    - Blending
    - Aim Offsets
  - Recoil layer
  - Sway layer
  - Leg IK
  - Weapon collision
  - Slot layer
- Troubleshooting
  - Spinning character in the editor or play mode
  - Look layer doesn't work
  - Strange recoil behavior
  - Aiming doesn't work with custom models, while it works fine with the demo weapons
  - Left-Hand IK works when reloading
  - The camera has an extra rotation
- Main Concepts
  - Dynamic retargeting
  - Update structure
  - Input
  - Character Information

Weapon Information

Tutorial

Prerequisites

Download the latest Example demo project **(Note:** the project will have compilation errors, ignore them and import the asset)
Join our Discord community server

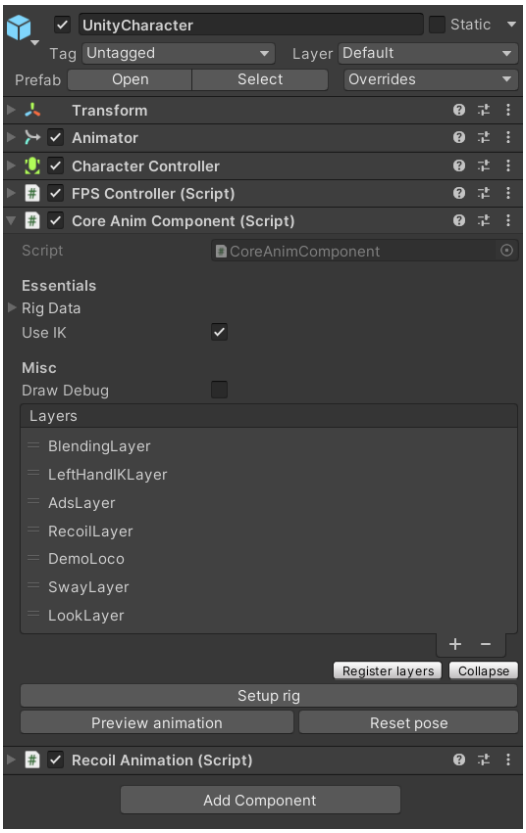
Note: the framework doesn't include the controller class, as it's supposed to be integrated into your own controller. This document provides full coverage of the integration process.

Basic setup

To get started, you need to add 2 components to your character:

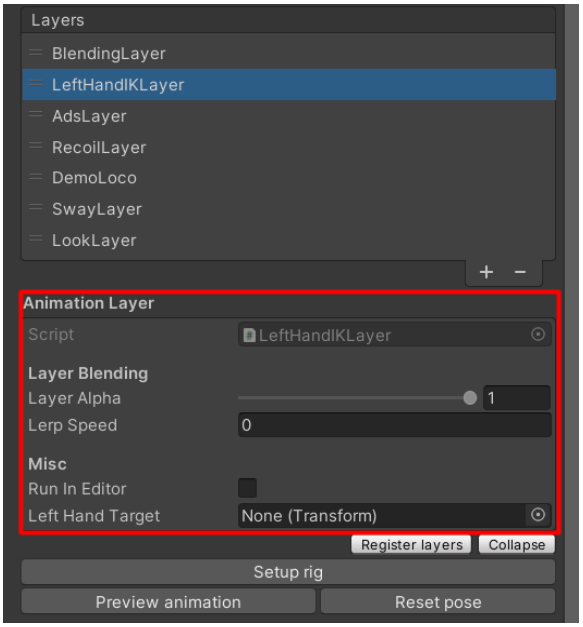
1) **Core Anim Component** - it applies bone modifications in runtime, by calling update functions from the animation layers
2) **Recoil Animation** - it's a standalone component used for procedural recoil

Here's an example of how your character inspector will look like:



Note: FPSController is an example controller class, you don't have to use it.
For more check the demo project.

Animation layers can be added/removed by clicking the **“+”/”-”** icons. The Layers list allows you to change the order of layers by dragging them, works pretty much like an ordinary list. The order is **crucial**, keep it in mind when adding new layers.
To see the layer, you need to select the item in the list, and it will pop up in the inspector:

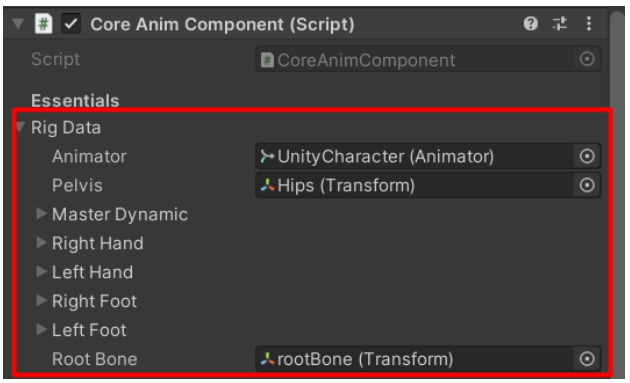


To hide the currently selected layer, you can press the **“Collapse”** button.
“Register layers” button is used when you manually added an animation layer without using **“+”**. It’s used only for compatibility with the older versions, so the new GUI can work with them.

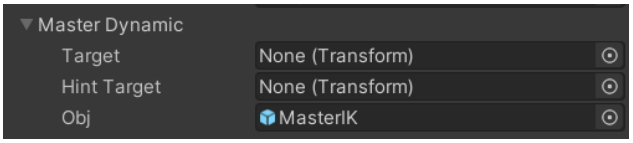
Tip: you can copy and paste animation layers by right-clicking on a layer. You can also copy/paste the whole layer list by clicking on the header.

You can preview the animation in the editor by hitting **“Preview animation”**, and **“Reset pose”** to stop previewing.

Before adding layers, you need to click on the **“Setup rig”** button. Pay attention to the log, if it says that some bones are missing, you will need to manually assign bones references in the **Rig Data**:



Master Dynamic, Hands, and Feet are **IK Dynamic Bones**. Let's take a look at their properties:



Target is an actual bone we want to copy rotation/position from. For example, for the RightHand it will be the actual right-hand bone.
Hint Target is a pole IK target, elbows for the arms and knees for the legs.
Obj is the game object that represents the **Dynamic Bone** in the game world, it's created automatically when you press the **"Setup rig"**. By default, MasterIK, RightHandIK, and LeftHandIK are parented to the head bone of the character. But you can easily change it if you want.

You can find more about **Rig Data** in the Main Concepts section.

Once the rig is set up, we can start adding animation layers. If you press **"+"** you will see the list of all the animation layers in this assembly.

The setup and description of all animation layers can be found in the Animation Layers section.

Integration

First of all, it's important to understand what kind of data is used in the framework:

- **WeaponAnimData** defines the properties of each weapon. For example, weapon sway data, pose positioning, etc
- **RecoilAnimData** is specific for each weapon and defines how the recoil should be generated
- **CharAnimData** defines the properties of the character, such as player input (e.g. mouse delta, movement input, leaning direction, etc.)

Add **CharAnimData** to your controller class and modify its properties based on the gameplay actions, such as sprint pressed, aim released, etc.

You will need to add **WeaponAnimData** and **RecoilAnimData** to your weapon class. The demo also has an example **Weapon.cs**.

Now, we are going to cover how to integrate the framework into your custom controller. For this example, we will use the **FPSController** script from the demo project.

Add references to the animation layers, core component, and recoil animation component to your controller class. Now let's see what methods to use during the gameplay:

Core Component

There’re 3 methods we are interested in:

- **public void** OnGunEquipped(WeaponAnimData gunAimData) // Call when equipping a new weapon. Note: it requires a WeaponAnimData, so you need to add this property to your weapon class.
- **public void** OnSightChanged(Transform newSight) // Call when cycling sights.
- **public void** SetCharData(CharAnimData data) // Call to update player input. CharAnimData contains general user input, such as the mouse delta, or movement keys input. It's used by all animation layers and should be updated every frame.

Recoil Animation

Call these methods to play the recoil animation in runtime:

- **public void** Init(RecoilAnimData data, **float** fireRate, FireMode newFireMode) // Call when equipping a new weapon. To update the fire mode, directly access fireMode property of the RecoilAnimation component. Also, you will need to add a RecoilAnimData property to your weapon class.
- **public void** Play() // Call in OnFire event.
- **public void** Stop() // Call when stop firing.

Animation layers

With animation layers, it’s slightly complicated, as it’s all up to the layer implementation. You need to directly access the layer and call its methods/modify properties based on the gameplay. For example, when pressing the sprint button, we need to disable the aiming layer, so we need to call *adsLayer.SetLayerAlpha(0f)*.

Generally, you will almost always use the **SetLayerAlpha(float weight)** method to control how much effect the layer has on the output pose. For example, if the weight is 0, this means the layer is disabled, whereas 0.5 means that only 50% of the layer is applied to the pose. The pose in this context means the output pose from the Animator. The framework works as a post-processing for the animation.

You can find more about animation layers and methods in the Animation Layers section.

Summary and quick guide

Here’s a quick summary of everything above:

Animation layers setup:

1) Add **CoreAnimComponent** and **RecoilAnimation** to your character
2) Press the **“Setup rig”** button, and if the log says some bones are missing, assign bone references manually
3) Add animation layers to the Core component by using the **“+”** icon
4) To preview the animator press **“Preview animation”**
5) If something doesn’t work correctly, check the **“Draw Debug”** option. Green spheres must follow the hands and feet positions
6) Drag list items to change the order of layers

Controller and weapon setup:

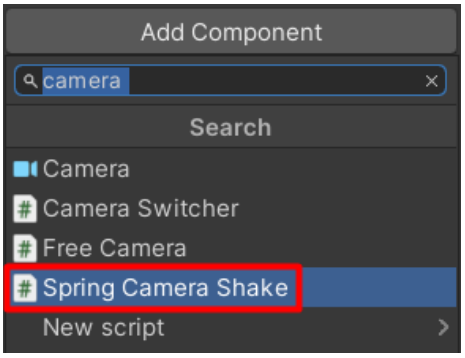
1) Add references to the **CoreAnimComponent**, **RecoilAnimation**, and animation layers you want to modify to your controller class
2) Add **WeaponAnimData** and **RecoilAnimData** to your weapon class
3) Add **CharAnimData** to your controller class. Modify its properties based on the gameplay code. The examples can be found in the FPSController script
4) Update **WeaponAnimData** and **RecoilAnimData** by calling *coreComponent.OnGunEquipped(gunData)* and *recoilComponent.Init(RecoilAnimData, fireMode, fireRate)* respectively
5) Update **CharAnimData** based on your gameplay code. Example: *_charAnimData.AddAimInput(new Vector2(deltaMouseX, deltaMouseY))*. Finally, call *coreComponent.SetCharData(_charAnimData)* to apply changes.
6) Call *coreComponent.OnSightChanged(newSightTransform)* when cycling sights
7) Directly access animation layers in your controller and modify them based on the gameplay code.

Note: all the examples can be found in the demo project.

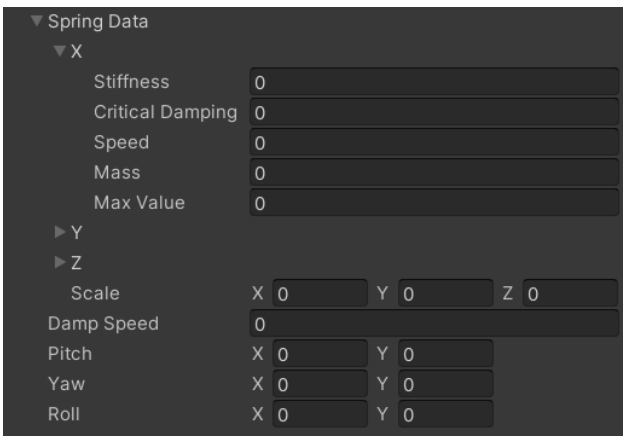
Camera recoil

Camera recoil is quite similar to the weapon sway, it's implemented using spring interpolation. It's super easy to use and set up.

First off, add a **"SpringCameraShake"** component to your camera object:



SpringCameraShake has a Profile property, this one defines the behavior of the camera shake. It's similar to the weapon sway, the same spring interpolation logic is used:



X, Y, and Z are Pitch, Yaw, and Roll rotation axes respectively.
Damp Speed defines blending out speed.
Pitch, Yaw, and Roll properties define the min and max target values for each rotation axis.

To play camera shake, add a reference to the **SpringCameraShake** component and call **PlayCameraShake()** method:
To update the shake profile, access the shakeProfile property directly:

```
// Update shake profile when equipping a new weapon
springCameraShake.shakeProfile = weaponCameraShake;
```

Finally, make sure that the shake script is executed after the camera stabilization script:

In FPSController camera is stabilized using the root bone of the character.

Camera and stabilization

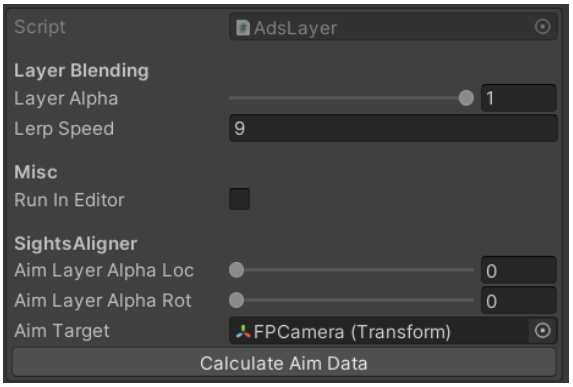
The camera rotation logic is handled in **FPSController** script, and it's pretty straightforward:

1) Update bones' transforms
2) Set the camera rotation to the root bone rotation. By doing so, we stabilize it
3) Add mouse input to the camera rotation

The first step is essential, make sure that **FPSController** or your custom camera logic runs after the **Core Anim Component**.

Animation layers

Ads layer



The aiming is performed fully procedurally, and no keyframed animation is required.
There're 2 types of aiming used:

- *Additive*
- *Absolute*

The Additive doesn't work out of the box and requires translation/offset calculation.
The Absolute works automatically and doesn't require any pre-calculations.

The Additive approach doesn't affect base animation and allows playing them normally when aiming, whereas the Absolute mode perfectly aligns sights, and overrides base animation.

You can easily blend between these 2 modes by using Aim Layer Alpha sliders.

AdsLayer is also used for pre-calculation for the Additive approach. To calculate aim data, make sure that you are in play mode and click on **"Calculate Aim Data"**.

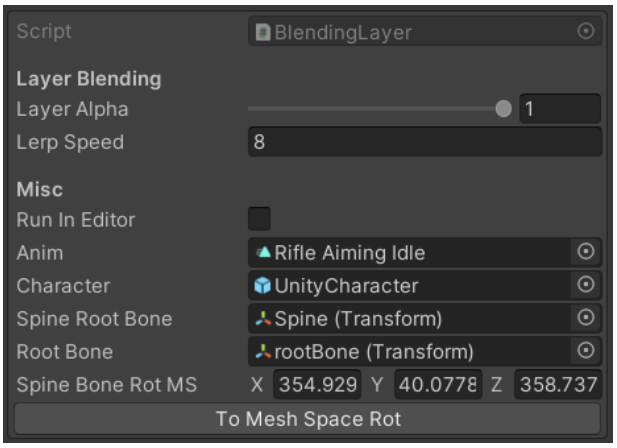
Aim Target is a transform we want sights to align with, it can be a camera object for instance.

Useful methods:

```
public void SetAdsAlpha(float weight) // Controls weight of the procedural aiming
public void SetPointAlpha(float weight) // Controls weight of the point aiming
```

Blending Layer

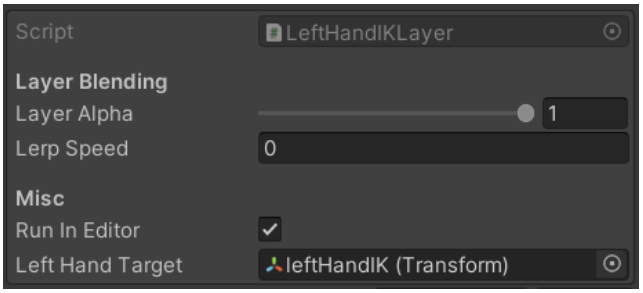
This is an experimental feature, used to override the character's spine bone in mesh space. The problem with the Unity Animator is that it overrides animation layers in local space, which is a big problem when it comes to full-body characters.



Anim - the base pose animation from which bone data will be extracted.
Keep in mind, that this layer is still under development.

Left-Hand IK Layer

This layer allows attaching the left hand to the gun barrel. The target for the left hand is specified in the **WeaponAnimData**.



Locomotion Layer

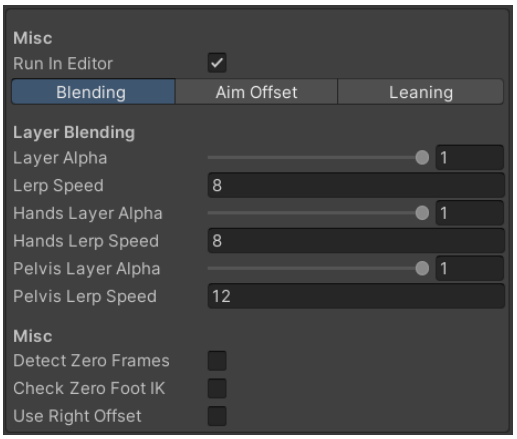
This layer applies procedural high/low-ready poses.

Useful methods:

- `public void SetReadyPose(ReadyPose poseType) // Sets current ready pose`
- `public void SetReadyWeight(float weight) // Controls ready pose weight. Currently better to use SetLayerAlpha`

Look Layer

This layer modifies the character's spine bones to look around. It also handles the zero-keyframe check.



Blending

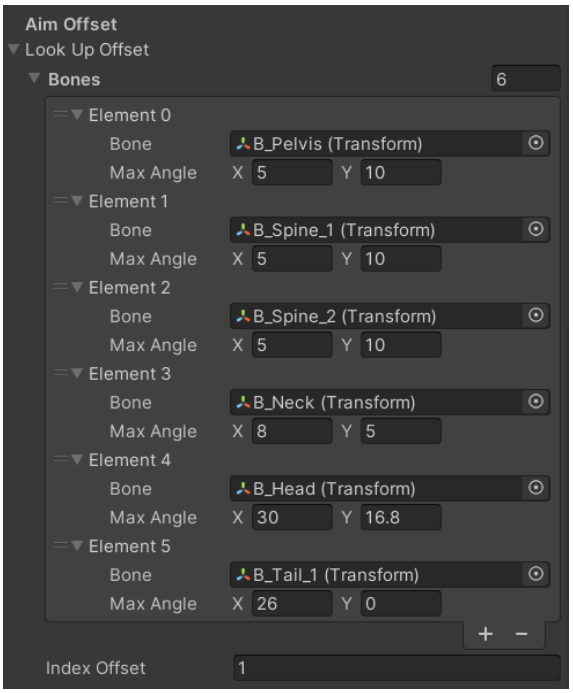
Hands Layer Alpha - defines how much look rotation should affect hands positions. By default, hands IK targets are parented to the head, this can be a problem when you have an unarmed motion, like this one:

Goblin looking up

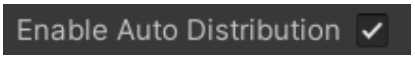
So, in this case, you can decrease the Hands Layer Alpha to something like 0.15-0.2 so you still have some effect on the arms.

Aim Offsets

AimOffset contains 2 lists of bones that are used for aiming.



Every element contains a reference to the bone transform, and maximum look angles: X is the max look Up/Right, and Y is the max look Down/Left.
Editing each bone is a very boring task, so you can automize it by enabling this flag:



So whenever you edit the higher element, the rotation of the other lower elements will be adjusted automatically. If there're bones, which you don't want to adjust you can use the Index Offset property. This offset defines the number of elements (from the end) that won't be automatically adjusted.

Example from above: Goblin character has a tail, so we want it to be affected by aim offset, so just add the tailbone to the list and set Index Offset to 1 - now it's not going to be changed by auto distribution.

Detect Zero Frames - defines if should check for empty frames.
Check Zero Foot IK - should be enabled if the character feet don't have animation data.
Use Right Offset - defines if the AimRight rotation should be applied.

Useful methods:

- `public void SetPelvisWeight(float weight) // Controls pelvis displacement alpha`
- `public void SetHandsWeight(float weight) // Controls how much the look rotation should affect hands`

Recoil layer

First off, create a RecoilData scriptable object by clicking the right mouse button.
You need to manually update the recoilAnim property in the controller class:

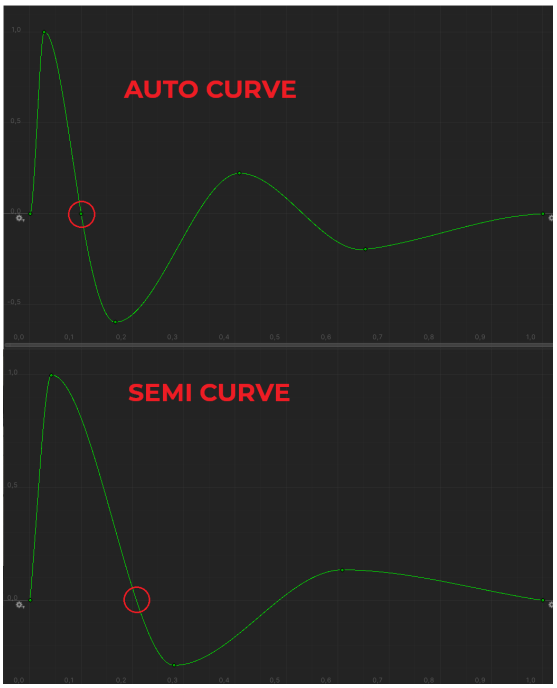
```
_charAnimData.recoilAnim = new LocRot(_recoilAnimation.OutLoc,
Quaternion.Euler(_recoilAnimation.OutRot));
// Should be updated every frame
```

Recoil is based on Unity Animation Curves. The formula is pretty simple:

Animation Value = LerpUnclamped(0, Randomized Value, AnimationCurveValue)
LerpUnclamped is used to achieve a bouncy effect (curve value less than 0).
All recoil curves must start and end with zero.

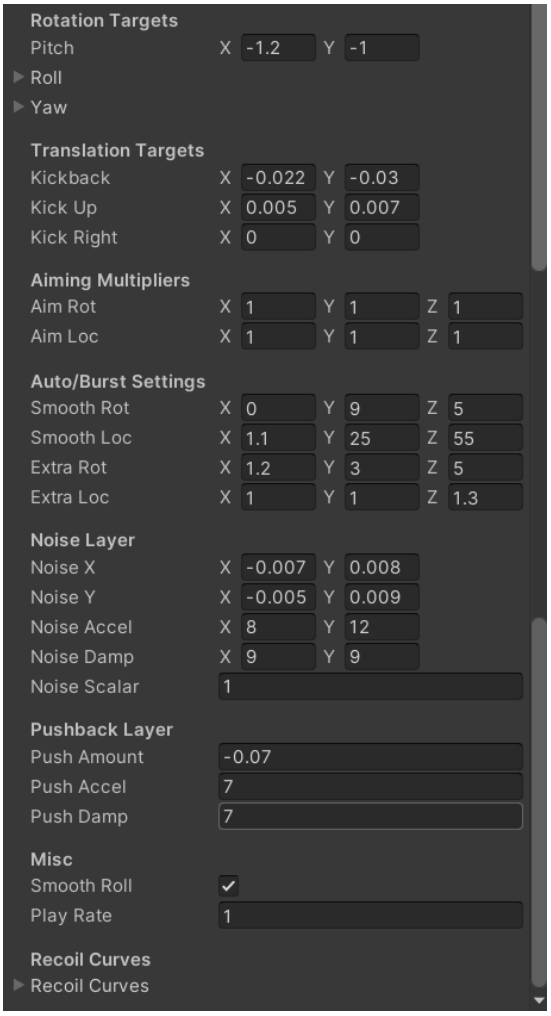
Curves for auto/burst weapons

Auto curves are actually just modified semi-curves, here's an example:



Curves are pretty much the same, **BUT** the Auto curve value is zero at some point - this point is the delay between shots. Let's say the fire rate is 600 RPM, this means a 0.1s delay between each shot. Consequently, our auto curve value should be zero at 0.1s, otherwise, glitches might be expected.

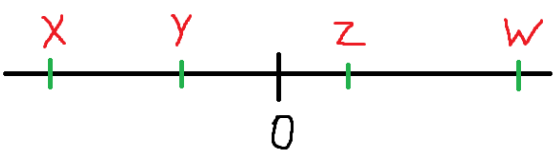
Why is that? Because Auto/burst animation gets looped and the animation max time is set to the delay between shots in seconds.



Recoil data example

Pitch defines the min and max values of look up/down rotation. Roll defines the rotation around the Z(forward) axis. Yaw defines the rotation around the Y (left/right) axis.

Roll&Yaw are Vector4, here's why:



This is done in order to prevent getting a random value very close to 0 because 0 means no animation effect => or strange results.

Translation targets define maximum and minimum values.
Aiming multipliers are applied when the aiming flag is set to 1.

Smooth Rot/Loc defines interpolation speed when firing in auto/burst mode.
Extra Rot/Loc are multipliers applied when firing in auto/burst mode.
Noise layer performs a smooth movement in the YX plane (left/right-up/down).
Noise scalar is used when aiming.
Pushback layer is a strong kickback after the first shot in full-auto burst mode.

Sway layer

This layer handles weapon sway when moving/looking, and free aim mechanic.
To enable/disable free aim use this method:

```
public void SetFreeAimEnable(bool enable)
```

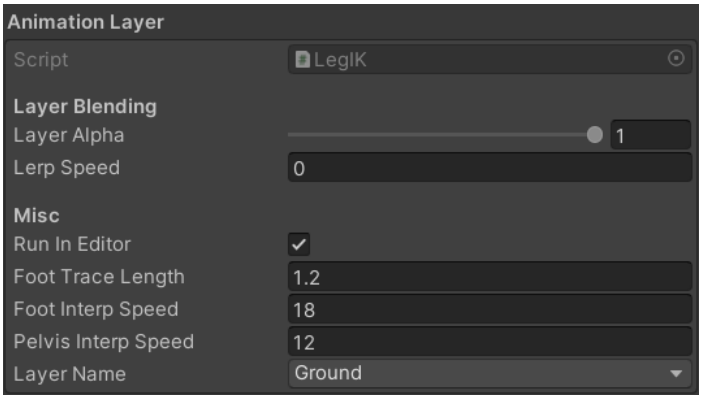
This layer uses these fields of the WeaponAnimData:

```
public LocRotSpringData springData;
public FreeAimData freeAimData;
public MoveSwayData moveSwayData;
```

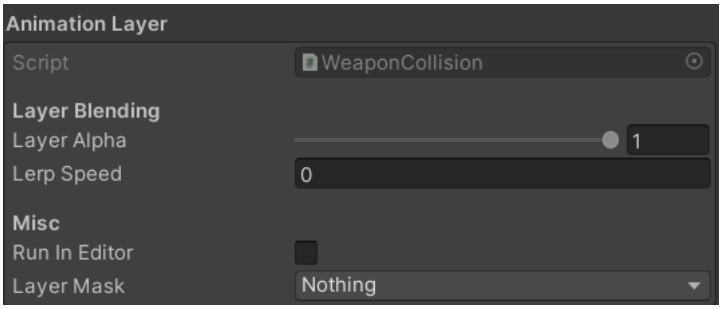
These are updated in *CoreAnimComponent.OnGunEquipped(WeaponAnimData)* method.

Leg IK

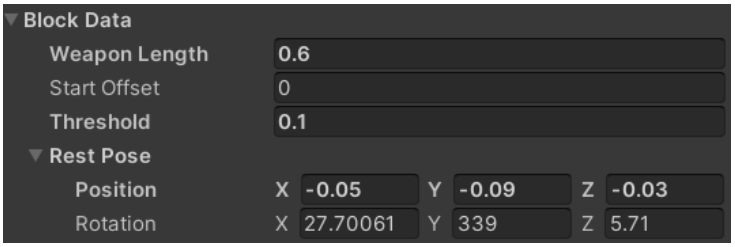
This layer uses a simple downtrace method to prevent leg clipping. This layer is usually added as the last layer in the list.
Foot Trace Length - the length of the ray
Layer Name - layer masks used for collision detection.



Weapon collision



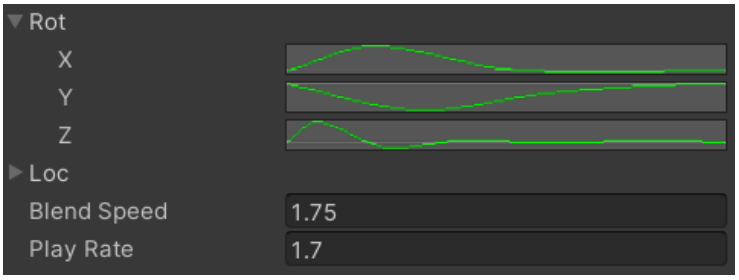
This layer uses **GunBlockData** struct:



Weapon Length - can be seen as a green line in the editor.
Start Offset - forward offset of the trace starting point.
Threshold - the maximum length of the collision line
Rest Pose - a procedural pose applied to MasterIK when the threshold is exceeded.

Slot layer

Sometimes it's required to play a procedural animation from code. This can be easily achieved by using **Slot layer** and **Dynamic motions**. DynamicMotion itself is a struct that contains curves and blending parameters. The animation is defined by those curves:



Example of a DynamicMotion

The actual motion is applied in root bone space, not in local space.
To play a dynamic motion, use this method in the **Slot laye**r:

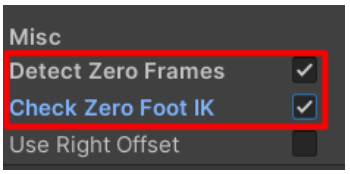
```
public void PlayMotion(DynamicMotion motionToPlay)
```

Troubleshooting

In this section, we are going to cover the most known issues and how to fix them. If you faced a problem that is not described here, please, share your problem on our Discord server.

Spinning character in the editor or play mode

This issue happens when there're non-keyframed bones in your animation. It's strongly recommended to have all bone tracks in your animation clips. However, you can fix this problem by checking these properties in the Look layer:



Look layer doesn't work

There could be 3 main reasons why this layer doesn't work:

1) EnableManualSpine control is checked in the AimOffset tab. Don't forget to uncheck it
2) CharAnimData mouse delta is not updated. Make sure that charAnimData.deltaAimInput is properly updated in your controller class
3) The layer alpha is set to 0

Strange recoil behavior

If your recoil animation looks strange, there could be 2 reasons why it happens:

1) The character pose doesn't have bone tracks for the hands - make sure that every bone is keyframed in your animation/pose.
2) The pivot point is not set up - to solve this, make sure that the pivot point is not null in **WeaponAnimData.GunAimData** struct

Aiming doesn't work with custom models, while it works fine with the demo weapons

There could be 2 reasons why the aiming doesn't work with your custom models:

1) Make sure that aim point and pivot point objects have the right orientation: Z - forward, Y - up, and X - right
2) Make sure that the aim point and pivot point are not affected by the scale of the weapon. For this, instead of scaling the whole weapon prefab, scale the mesh only. Alternatively, you can create an empty game object, parent the weapon to it, and then add aim and pivot points to that game object

Left-Hand IK works when reloading

You need to disable the Left-Hand IK layer when playing a reloading animation. The easiest way to do this is to call *leftHandLayer.SetLayerAlpha(0f)* when the reload starts. Make sure to enable it back when the reload is over.

The camera has an extra rotation

This issue might happen in the demo project if the script execution order is incorrect. Make sure that FPSController runs after the CoreAnimComponent. This is important, because when looking around we rotate the camera parent bone, and then apply look rotation to the camera itself, which results in the double rotation amount.

Main Concepts

FPS Animation Framework introduces 2 main concepts: **Core Component** and **Animation Layer**.

Animation Layer modifies the character's bones in runtime, that's all it does - all animation logic is handled in this class. However, it's quite useless on its own, as the Core Component is required to apply the procedural modifications.

Core Anim Component is used to apply Animation Layers and acts as an interface for your controller class to communicate with Animation Layers. In this context, it means that **CoreAnimComponent** is fed with the player input and weapon data, and all this information is used by the **Animation Layers**.

The actual layer control is performed **by accessing the layer directly**. This means that you will need to add layer references to your controller class. For example, when pressing RMB we want to aim, so we need to change Ads Layer alpha to 1: *adsLayer.SetAdsAlpha(1f)*

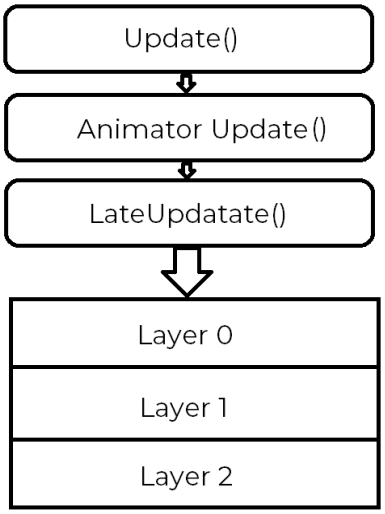
Dynamic retargeting

This is represented by **DynamicBone** and **DynamicRigData.**
The idea of **DynamicBone** is that some empty object (let's call it RightHandIK) is copying bone (let's say right hand) transforms in runtime. This allows us to preserve base keyframed animation, and apply procedural modifications smoothly.

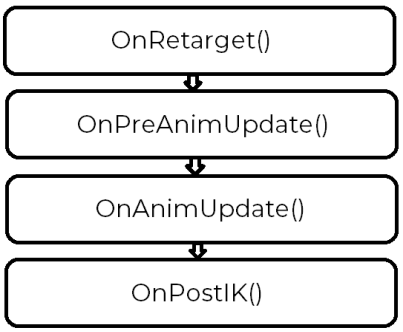
DynamicRigData is a collection of **DynamicBones**, used for Arms and Leg IK. It also contains Character and Weapon information, as this is used in the Animation Layers.

Update structure

Now let's take a look at how the **Animation Layers** are applied in runtime:



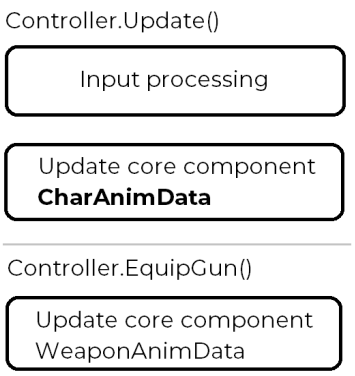
Animation Layers are applied in LateUpdate(), so it doesn't affect animator and animation constraints.
The **Anim Layer** update cycle is quite simple:



Input

All player input is defined in the **CharAnimData** struct - this struct is used by all animation layers. To update the **CharAnimData**, call **SetCharData**:
coreAnimComponent.SetCharData(yourCharData);

All the information related to the weapon (e.g. pose offset, sway data, etc.) is defined in the **WeaponAnimData** struct - this one is updated in the *OnGunEquipped()* method.



Character Information

In order to feed the Core Component with input data, **CharAnimData** struct is used:

```
public struct CharAnimData
{
    // Input
    public Vector2 deltaAimInput; // it is used for weapon sway
    public Vector2 totalAimInput; // it is used for look layer
    public Vector2 moveInput; // player input movement direction
    public int leanDirection;

    public LocRot recoilAnim;

    // Use this method to add aim input to the component
    public void AddAimInput(Vector2 aimInput)
    {
        deltaAimInput = aimInput;
        totalAimInput += deltaAimInput;
        totalAimInput.x = Mathf.Clamp(totalAimInput.x, -90f, 90f);
        totalAimInput.y = Mathf.Clamp(totalAimInput.y, -90f, 90f);
    }
}
```

CoreToolkitLib.cs

Weapon Information

WeaponAnimData defines the properties specific to each weapon.

```
public struct WeaponAnimData
{
    public Transform leftHandTarget;

    public GunAimData gunAimData;
    public Vector3 handsOffset;
    public LocRotSpringData springData;
    public FreeAimData freeAimData;
    public MoveSwayData moveSwayData;
    public GunBlockData blockData;
}
```

CoreToolkitLib.cs

Methods used to update Weapon Information:

```
public void OnGunEquipped(WeaponAnimData gunAimData)
public void OnSightChanged(Transform newSight)
```

CoreComponentLib.cs

GunAimData:

```
public struct GunAimData
{
    public TargetAimData target; //Scriptable object, contains additive location/rotation for additive aiming
    public Transform pivotPoint; //Physical pivot of the weapon
    public Transform aimPoint; // Default sight
    public LocRot pointAimOffset; //Additive offset used for point aiming
    public float aimSpeed;
}
```