
Welcome to the FPS Animation Framework documentation!

by Kinemation

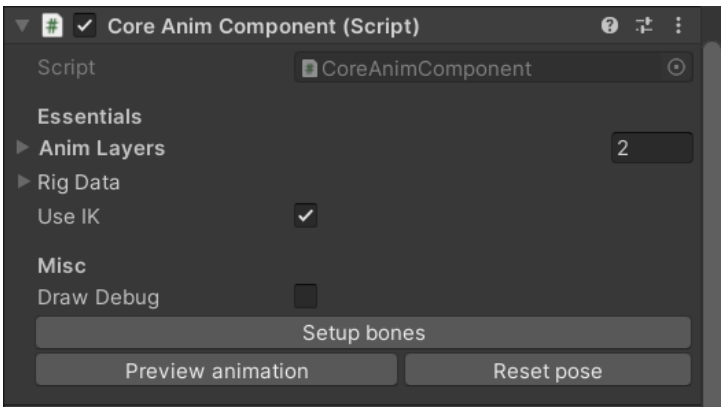
How it works

FPSFramework consists of 2 major parts:

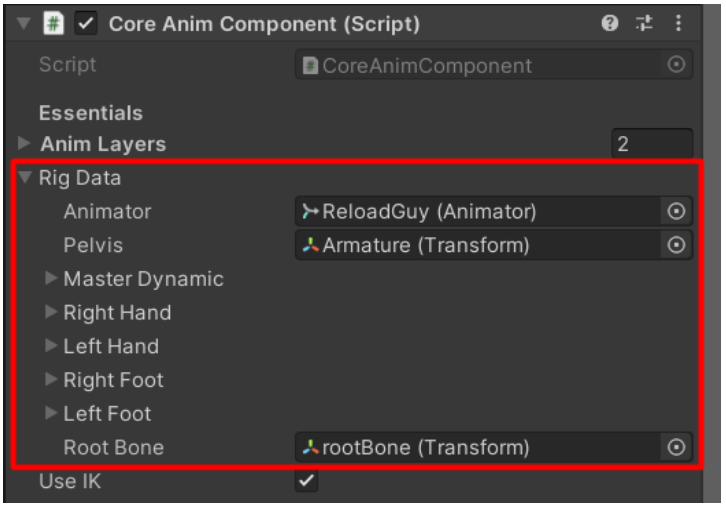
- CoreAnimComponent
- Animation layers

CoreAnimComponent applies animation layers and contains essential information about the character rig.

Animation layers modify the character pose in runtime. You can create your own layers, by creating a new class and deriving it from AnimLayer abstract class.



You can also preview the animator in the editor




```
public void SetMasterIKTarget(Transform t)
{
   rigData.masterDynamic.target = t;
}
```

This method is essentially important, as it defines the bone or object transform, which MasterIK will copy rotation/location from.

Master Dynamic, Right Hand, Left Hand, Right Foot, and Left Foot are DynamicBones. DynaimcBone consists of:

- Bone target transform is used as a tip bone in the IK solution
- Hint target is used as a hint target in the IK solution
- Obj is the actual IK target

```
public abstract class AnimLayer : MonoBehaviour
 {
     [Header("Misc")]
     public bool runInEditor;
     protected DynamicRigData rigData;

     public virtual void OnRetarget(ref DynamicRigData data)
     {
         rigData = data;
```

```
        }

        public virtual void OnPreAnimUpdate()
        {
        }

        public virtual void OnAnimUpdate()
        {
        }

        public virtual void OnPostIK()
        {
        }
    }
```

Execution order:

1. OnRetarget(`ref DynamicRigData data`)
2. OnPreAnimUpdate()
3. OnAnimUpdate()
4. OnPostIK()

LookLayer

This layer is used for runtime aim offset, leanin. Here're the methods you should use to work with this layer (make sure you add refs to this layer in your player controller class):

```
public void SetAimRotation(Vector2 newAimRot) // newAimRot.x delta MouseX, newAimRot.y delta MouseY
public void SetLeanInput(int direction) //1 right, 0 no lean, -1 left
public void SetLookWeight(float weight)
```

Settings

Layer Blending

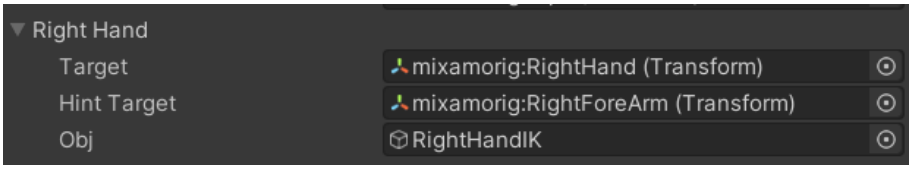
Layer Alpha defines the weight of the applied procedural layer: 1 - 100% effect, 0 - no effect. *Hands Layer Alpha* - 0 means base animation layer, 1 means with a procedural offset applied. By default, hands IK targets are parented to the head, this can be a problem when you have an unarmed motion, like this one:

Goblin lookin' 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.

Rig

Right Hand, Left Hand, Right Foot, Left Foot - are dynamic bones:



Target is the actual bone the bone whose rotation&location is going to be copied by the dynamic bone.
Hint target is a pole target used for the Two-Bone IK function. By default, it's the parent of the Target
Obj is an empty GameObject, so you can easily drag or reparent it easily in the inspector.

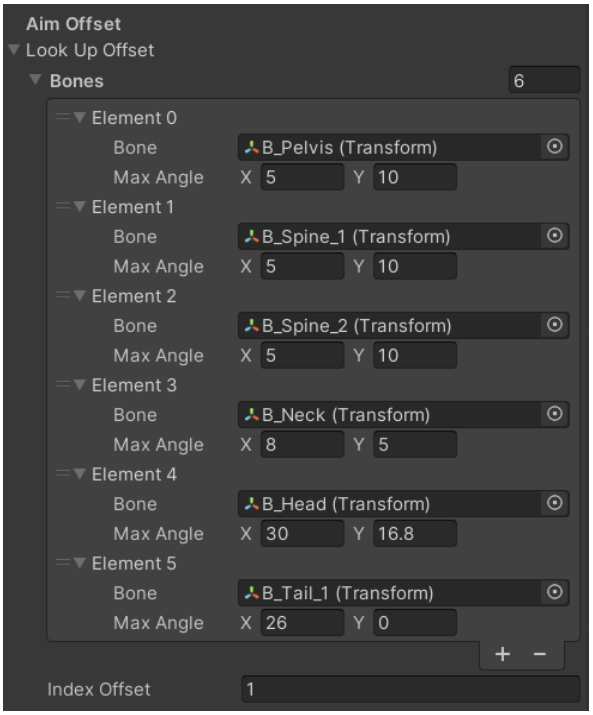
You can disable IK, if you want to modify spine bones only. However, **using IK is strongly recommended to achieve more natural results!**

RootBone is an empty GameObject used for bones rotation/translation. All operations are done in the space of the RootBone, which makes it possible to apply an additive offset to arms, spine, and hips without affecting the base animation layer.

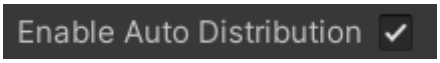
By default, RootBone is created as a child of the object LookComponent is attached to. The Offsets category allows you to add a translational offset to hips and arms.

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, 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.

Enable Manual Spine control is useful only in Play mode.

AdsLayer

The system uses 2 types of aiming:

- Additive: animations are fully applied
- Absolute: entirely overrides animation so sights are aligned

The absolute approach is fully automatic and doesn't require additional setup.
The additive approach requires aim pose data calculation as it simply adds translation&rotation when aiming.

Methods:

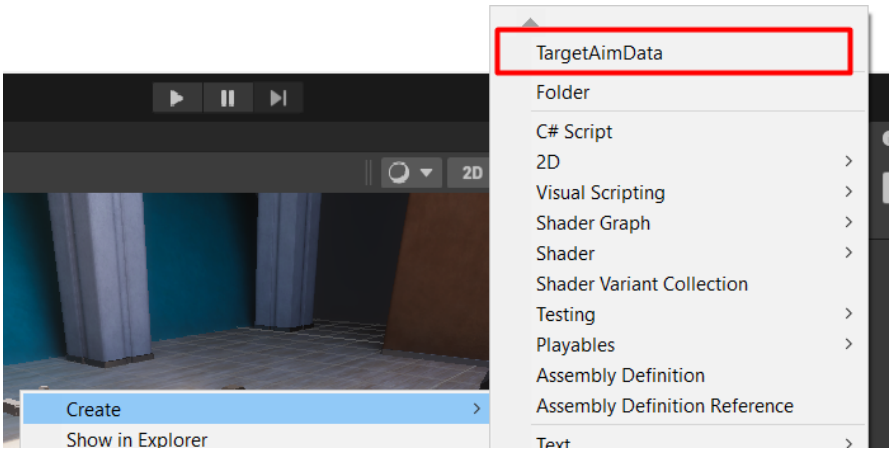
```
public void InitLayer(GunAimData gunAimData)
public void SetHandsOffset(Vector3 offset)
```

```
public void CalculateAimData()
```

Workflow

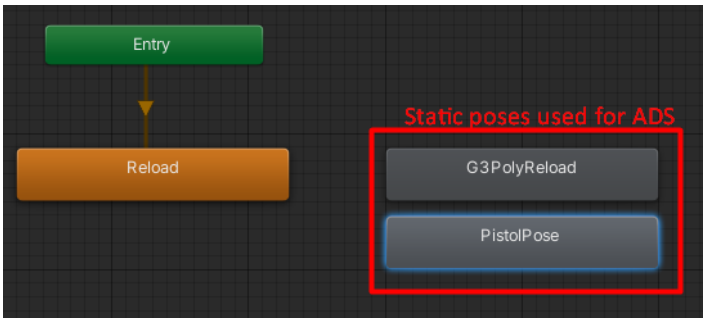
Step 1

Create a new **TargetAimData** and assign the static pose clip - only the first frame of the animation will be used, so you can even assign a reloading animation here.



Note: you need to create this SO[1] for each gun

Then add an empty state to your animator: when calculating bone data it's important to play the static pose.



Also, make sure that the name of the state and animation clip is **THE SAME!**
You can specify a custom name by editing a string field: if the **stateName** string is empty, the animation clip name will be used instead.

Step 2

Add **AdsLayer** reference to your weapon system class.

Make sure that InitLayer() is called when a new gun is equipped:

---

[1] Scriptable object

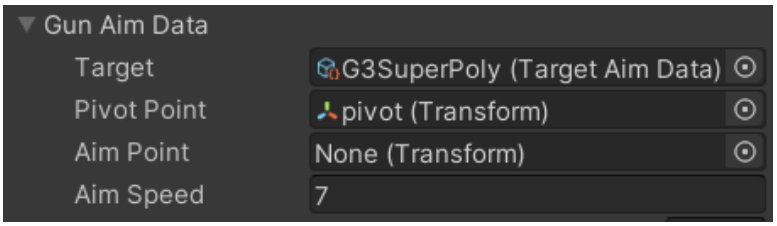
```
private void EquipWeapon()
    {
        ...
        adsLayer.InitLayer(gun.gunAimData);
    }
```

Then, make sure that you update the current aim point transform when cycling sights:

```
private void Update()
    {
        ...
        //
        if (Input.GetKeyDown(KeyCode.V)) <- changing sights/scopes
        {
            adsLayer.aimData.aimPoint = GetGun().GetScope(); // update scope
        } …
```

The demo project contains examples of weapon and weapon system classes, so you will have a practical example of how the sights aligner can be applied.

Add **GunAimData** to your weapon and set it up:

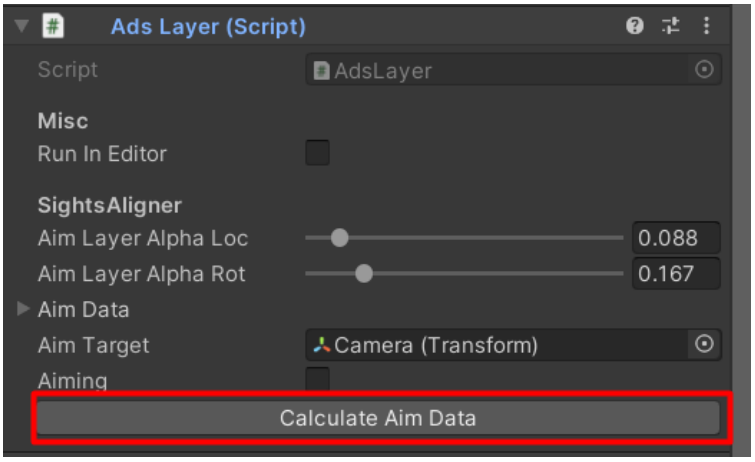


Target - TargetAimData (ScriptbaleObject, contains Pos&Rot and static pose anim clip)
PivotPoint - an empty object, that can serve as a default aim point. Make sure it has the right rotation: X is the forward, Y is the up etc.
AimPoint - should be none as default. This is modified in a runtime.

Step 3

Note: This step is only useful if you want to preserve the base anim layer

Calculate aim target data in play mode when a weapon is equipped:



If all the above steps were followed correctly, this will write aim translation and rotation based on the static pose clip.

Step 4

How to enable point aiming: make sure that you update **pointAiming** flag in AdsLayer. The point-aiming offset can be specified in the GunAimData struct.

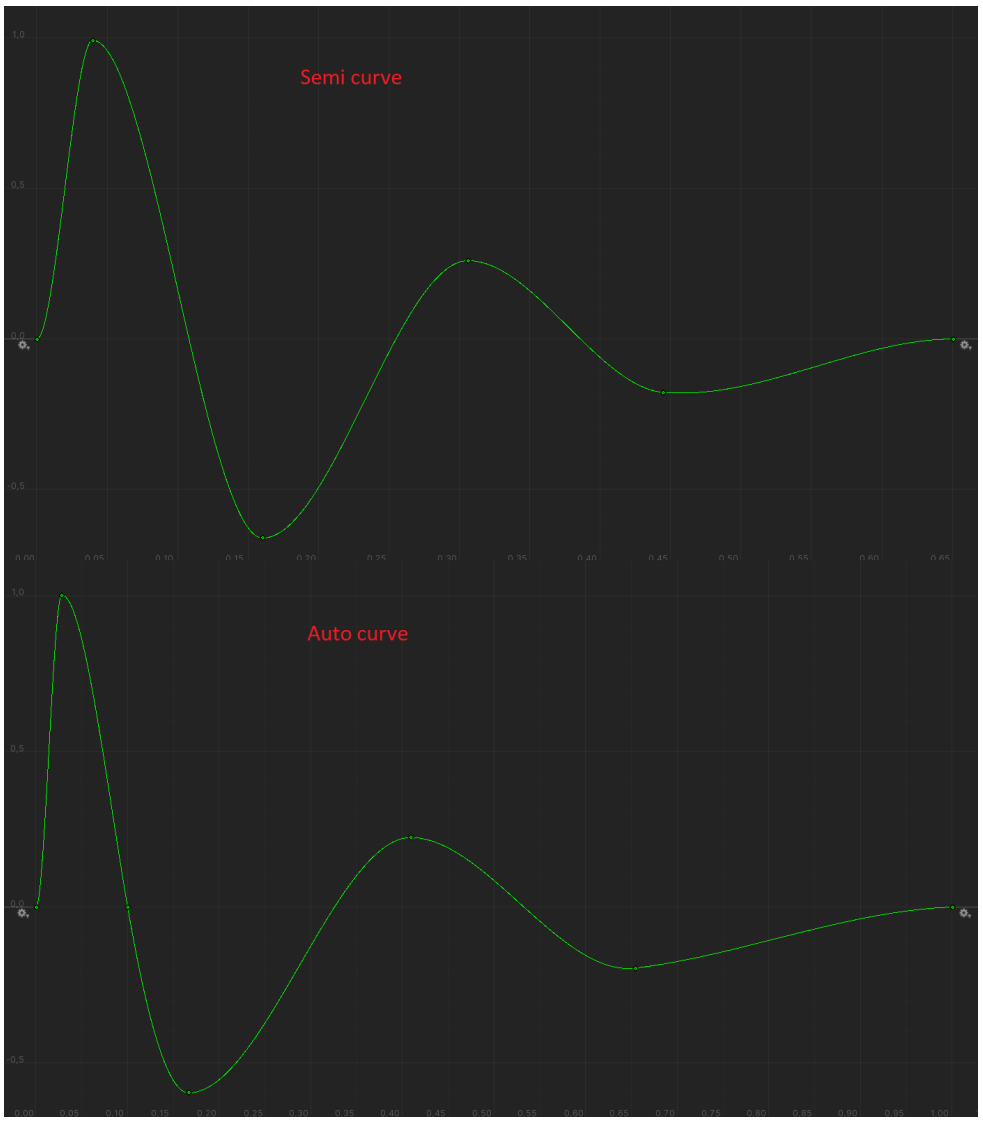
RecoilLayer

RecoilLayer 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.

Workflow

Step 1

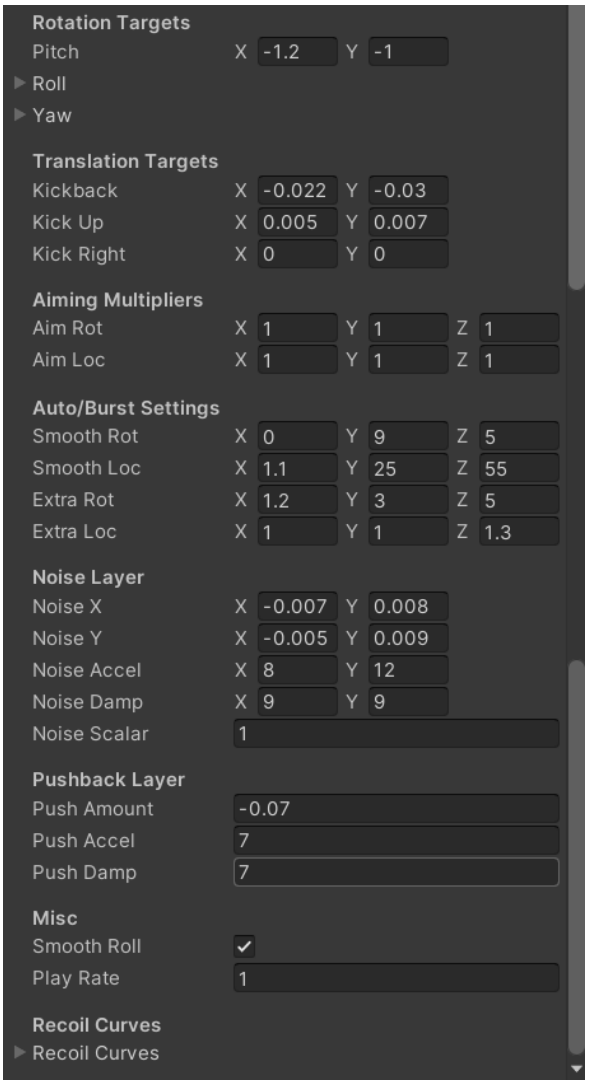
Add RecoilAnimation to your character. RecoilAnimation handles all the procedural animation logic based on the input parameters. Here're the most important methods:

```
        public void Init(RecoilAnimData data, float fireRate) // This should be called whenever a new weapon is equipped

        public void Play()
        public void Stop()
```

Step 2

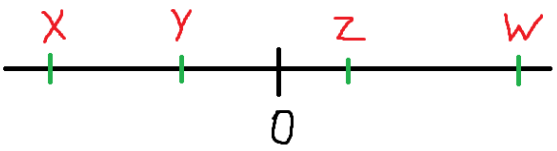
Add RecoilAnimData to your Weapon class. To create a new data asset: Left click/Create/RecoilAnimData.



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.

Step 3

Call Init(), Play() and Stop()
This is when you need to integrate the package logic into your weapon system. Add a reference to the RecoilAnimation in your weapon system class (or any other class where you handle shooting/equipping).

Add Init(), Play() and Stop() to your code.

Step 4

Add RecoilAnimation functionality to your weapon system class (or any other class where you handle shooting/equipping).

```
    private void EquipWeapon()
     {
         var gun = weapons[_index];

// Start
         _recoilAnimation.fireMode = gun.fireMode;
         _recoilAnimation.Init(gun.recoilData, gun.fireRate);
// End
```

```
        _bursts = gun.burstAmount;

        _animator.Play(gun.poseName);
        gun.gameObject.SetActive(true);
    }
```

Check the demo project for more info.

Sway Layer

Working with SwayLayer is incredibly easy.

Workflow

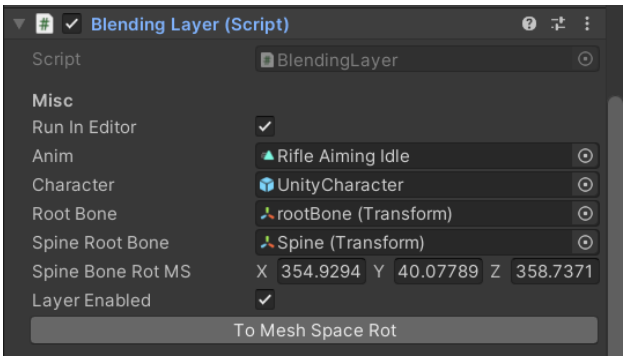
This layer has only one method:
`public void SetTargetSway(LocRotSpringData swayData)`

It just sets spring sway data for the current weapon, call this method when a new weapon is equipped.

BlendingLayer

Blending layer calculates the rotation of the spine-root bone (the first spine bone) in root bone space and then overrides it in runtime.
This allows to the upper body, as the Unity animator overrides animation layers in local space, which is such a problem when it comes to a full-body character.

Workflow



Select the animation which will act as a base pose, then select the character root animated object and press the button.

Keep in mind, that this layer is still under development.