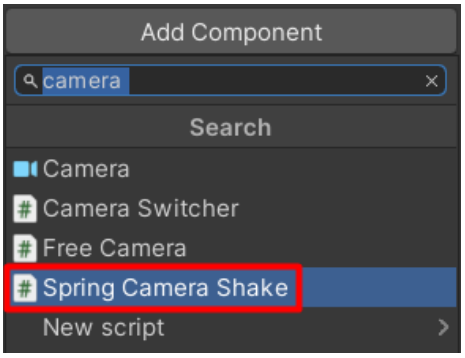
Welcome to the FPS Animation Framework documentation!

by Kinemation

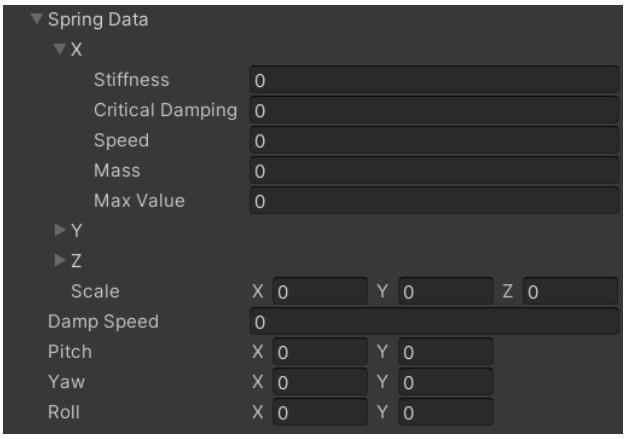
New: Camera recoil

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 "SpringCameraSake" 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.

IK Weight control

IK weights can be adjusted in code, using these methods:

```
public void SetRightHandIKWeight(float effector, float hint)
public void SetLeftHandIKWeight(float effector, float hint)
public void SetRightFootIKWeight(float effector, float hint)
public void SetLeftFootIKWeight(float effector, float hint)
```

where effector - alpha for the target bone, and hint - for the pole target.

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)*

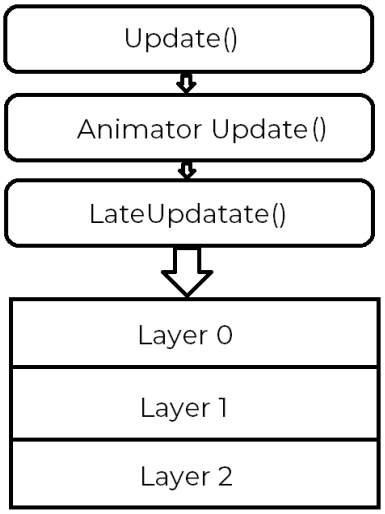
Dynamic retargeting

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.

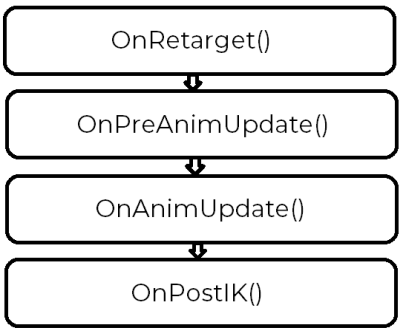
DynamicRigData is a collection of **DynamicBones**, used for Arms and Leg IK. It also contains Character and Weapon information, as this is used in the Animation Layers.

Update structure

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



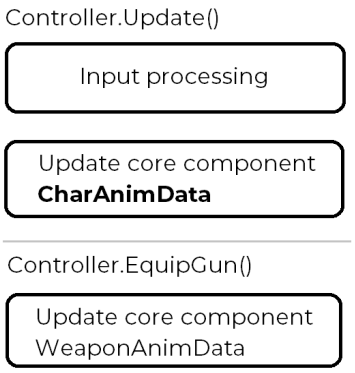
Animation Layers are applied in LateUpdate(), so it doesn't affect animator and animation constraints.
The **Anim Layer** update cycle is quite simple:



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;
    public Vector2 moveInput;
    public int leanDirection;

    public LocRot recoilAnim;
}
```

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;
}
```

CoreToolkitLib.cs

Methods used to update Weapon Information:

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

CoreComponentLib.cs

Let's break down each struct used in **WeaponAnimData**

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;
```

```
}
```

LocRotSpringData

```
public struct LocRotSpringData
{
    public VectorSpringData loc;
    public VectorSpringData rot;
}
```

This struct is used for spring weapon sway.

FreeAimData

```
public struct FreeAimData
{
    public float scalar;
    public float maxValue;
    public float speed;
}
```

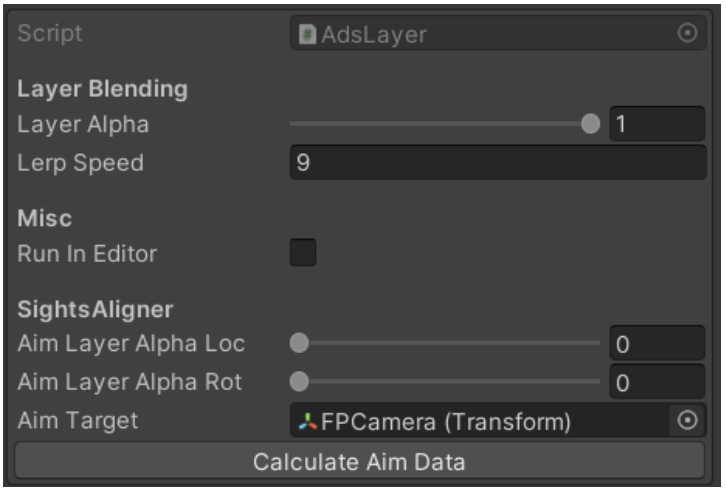
Used for free aiming (dead zone rotation)

MoveSwayData

```
public struct MoveSwayData
{
    public Vector3 maxMoveLocSway;
    public Vector3 maxMoveRotSway;
}
```

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.

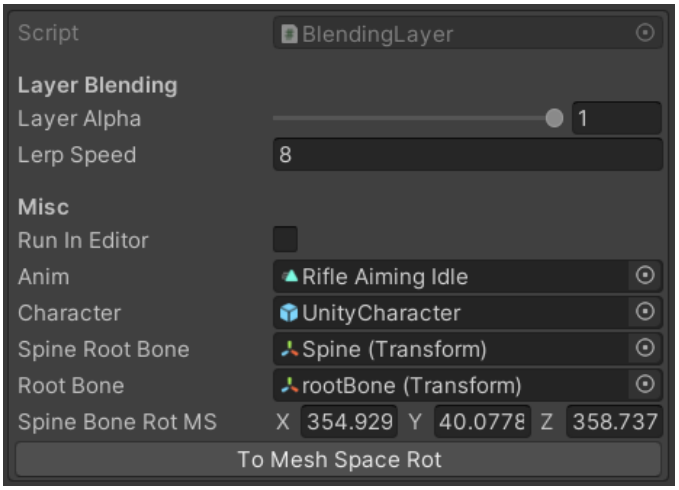
The Additive approach doesn't affect base animation and allows playing them normally when aiming, whereas the Absolute mode perfectly aligns sights, and overrides base animation.

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".

Blending Layer

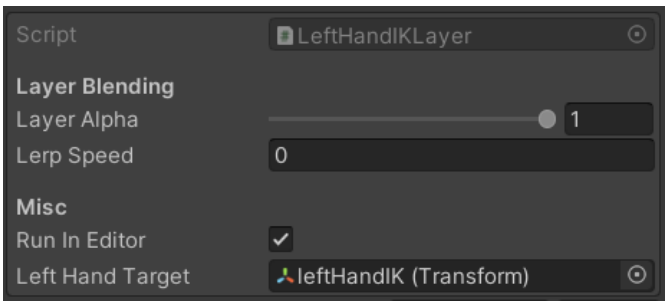
This is an experimental feature, used to override the character's spine bone in mesh space.
The problem with the Unity Animator is that it overrides animation layers in local space, which is a big problem when it comes to full-body characters.



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

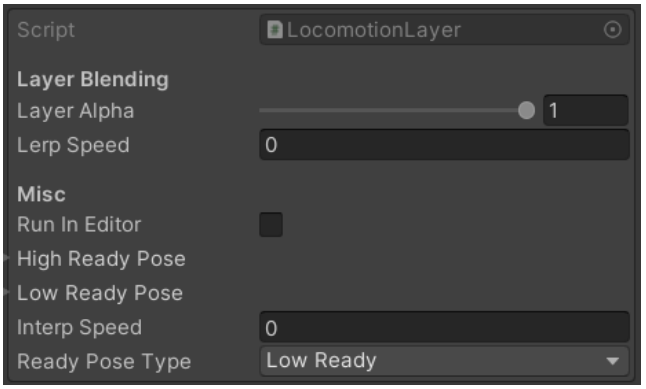
Left-Hand IK Layer

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

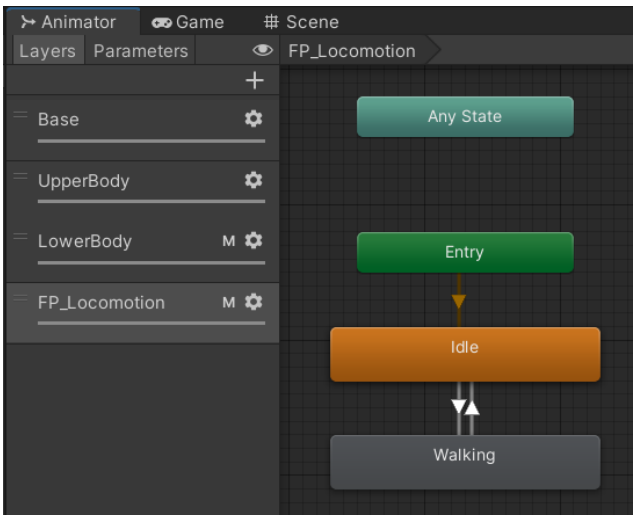


Locomotion Layer

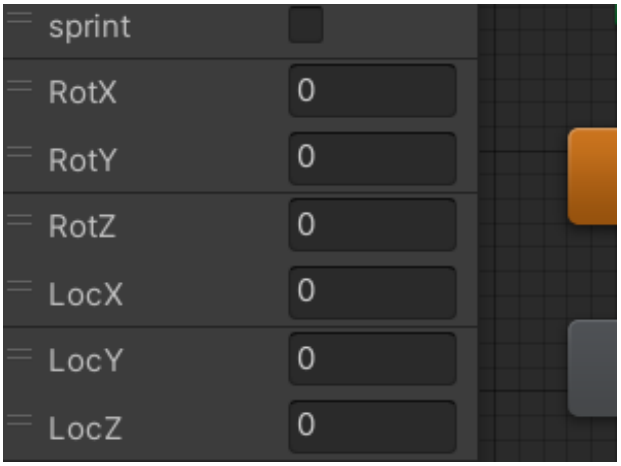
This layer applied ready poses and procedural idle/walk animations.



If you open the UnityCharacter animator in the demo project, you will notice that there's a procedural layer in the Animator:

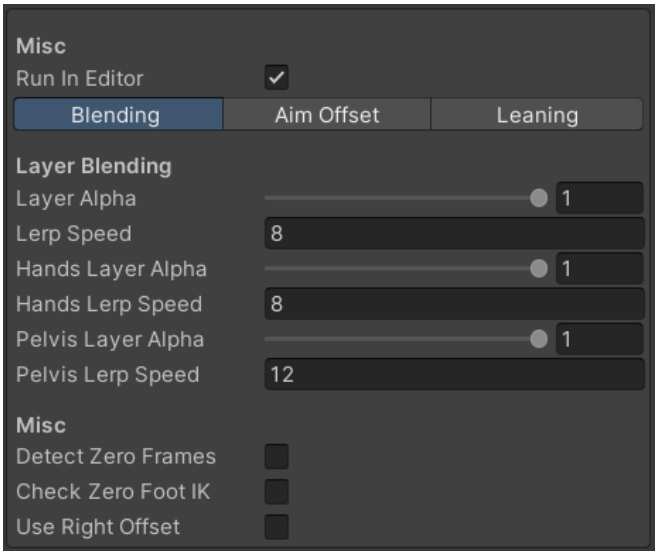


Why is that? The reason is simple: procedural idle/walk animations are driven by animation curves from the animator.



Look Layer

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



Layer Blending

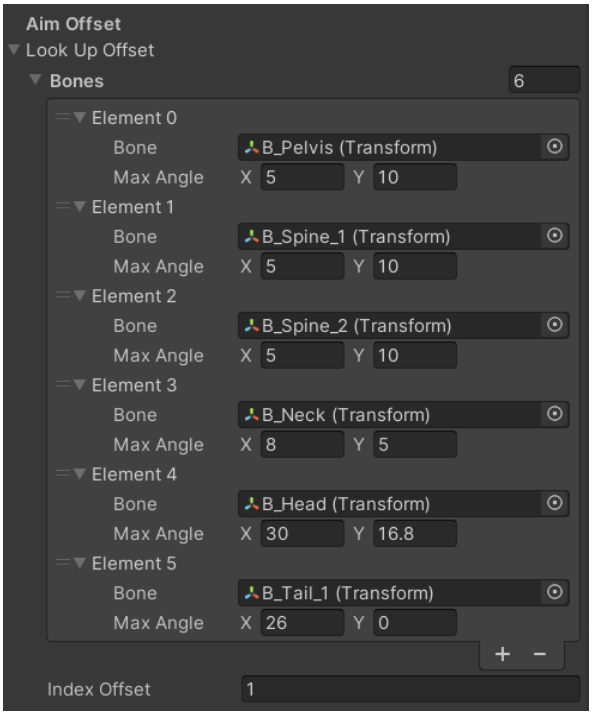
Hands Layer Alpha - 0 means base animation layer, 1 means with a procedural offset applied. By default, hands IK targets are parented to the head, this can be a problem when you have an unarmed motion, like this one:

Goblin looking up

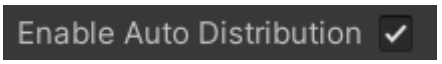
So, in this case, you can decrease the Hands Layer Alpha to something like 0.15-0.2 so you still have some effect on the arms.

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.

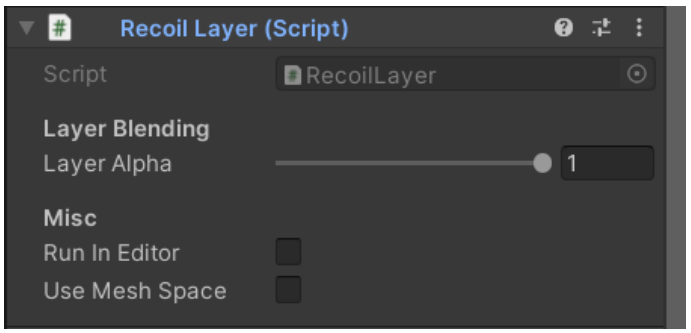
Example from above: Goblin character has a tail, so we want it to be affected by aim offset, so just add the tailbone to the list and set Index Offset to 1 - now it's not going to be changed by auto distribution.

Enable Manual Spine control is useful only in Play mode.

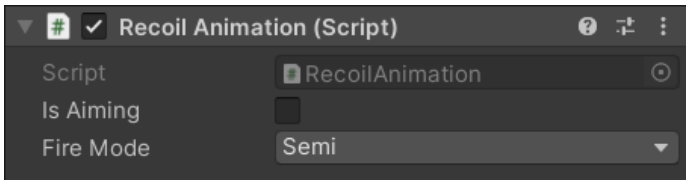
Detect Zero Frames - defines if should check for empty frames.
Check Zero Foot IK - should be enabled if the character feet don't have animation data.
Use Right Offset - defines if the AimRight rotation should be applied.

Recoil Layer

Applies procedural recoil animation.



However, the actual animation values are generated in the RecoilAnimation component:



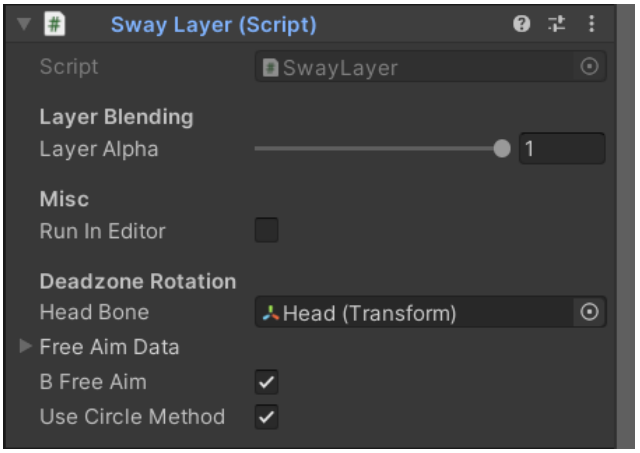
Essential methods of **RecoilAnimation.cs**:

```
public void Init(RecoilAnimData data, float fireRate, FireMode newFireMode)
public void Play()
public void Stop()
```

Init is called whenever a weapon is equipped

Sway Layer

This layer handles weapon sway when moving/looking, and free aim mechanic.



Integration

Basic setup

To integrate the framework with your project you will need to modify your Controller and Weapon class.

Let's start with the Weapon class. You will need to add **WeaponAnimData** and **RecoilAnimData** fields. These properties will be passed to the **Core Anim Component** and **Recoil Animation Component.**

Now let's move to the Controller class: add the **CharAnimData** struct, a reference to the **Core Anim Component, RecoilAnimation component,** and **Anim Layers**.

Now we need to update **CharAnimData** and **WeaponAnimData** in the Controller class. Here're the example methods from the FPSController class (Example demo project)

Equipping a weapon:

```
private void EquipWeapon()
{
    ...
    _recoilAnimation.Init(gun.recoilData, gun.fireRate, gun.fireMode);
    coreAnimComponent.OnGunEquipped(gun.gunData);
    ...
}
```

Changing sights:

```
private void ChangeScope()
{
    coreAnimComponent.OnSightChanged(GetGun().GetScope());
}
```

Firing:

```
private void Fire()
{
    _recoilAnimation.Play();
}
```

StopFiring:

```
private void Fire()
{
    _recoilAnimation.Stop();
}
```

To update CharAnimData:

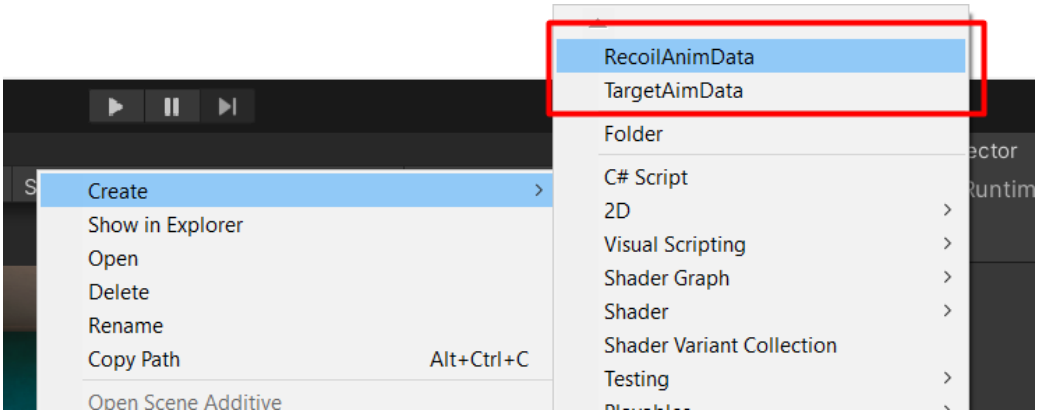
```
private void UpdateAnimValues()
{
```

```
        coreAnimComponent.SetCharData(_charAnimData);
    }
```

You will also need to create 2 scriptable objects for each weapon:

- TargetAimData
- RecoilAnimData

Right click => Create

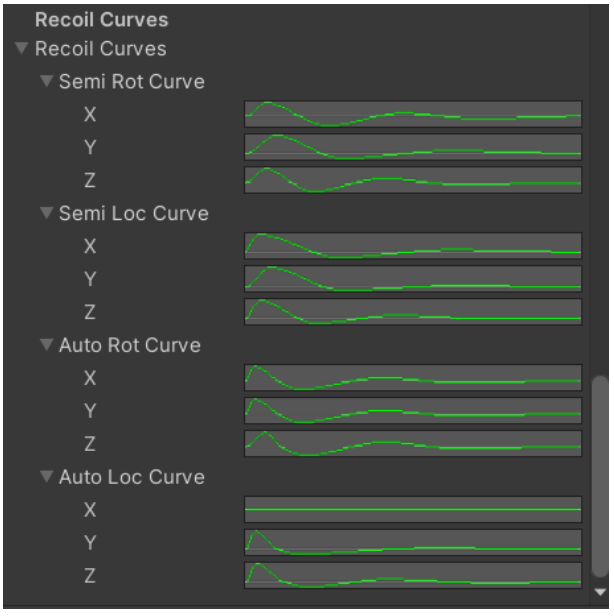


You can find samples of such SOs in the demo project.

Recoil setup

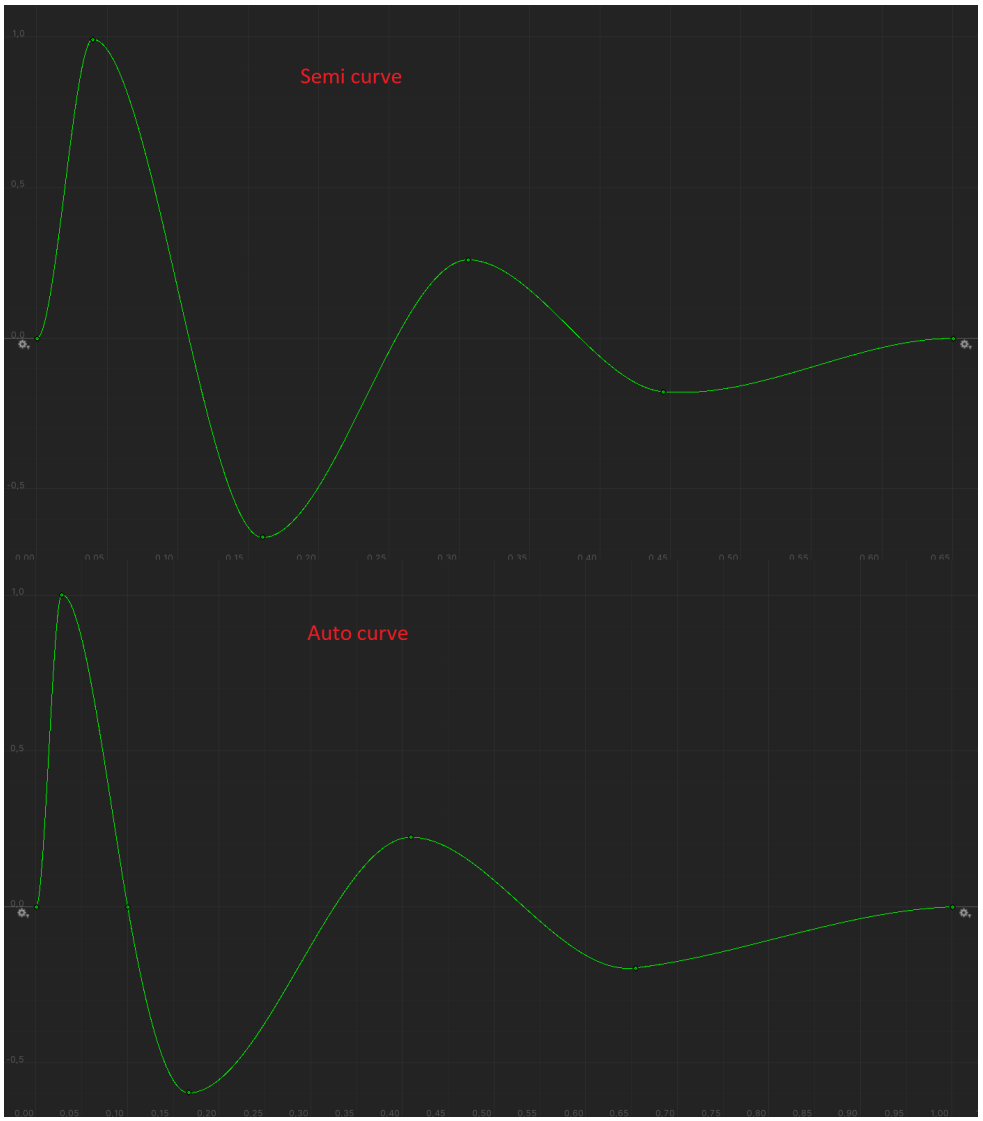
Recoil is based on Unity Animation Curves. The formula is pretty simple:

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.



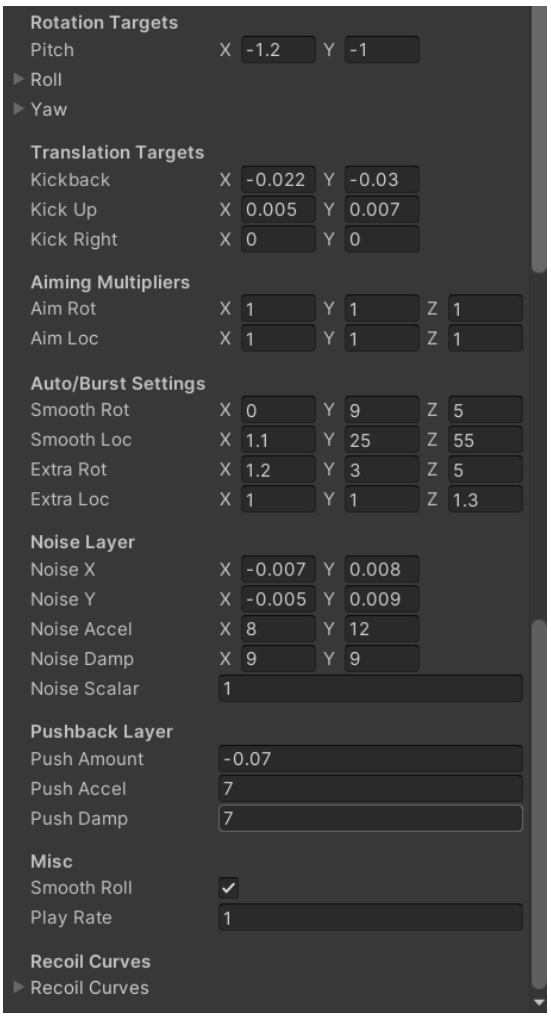
Curves for auto/burst weapons

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.

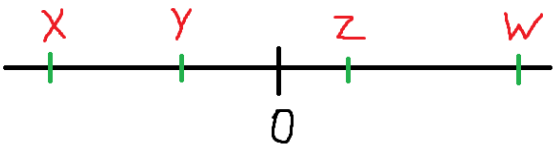
Why is that? Because Auto/burst animation gets looped and the animation max time is set to the delay between shots in seconds.



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.

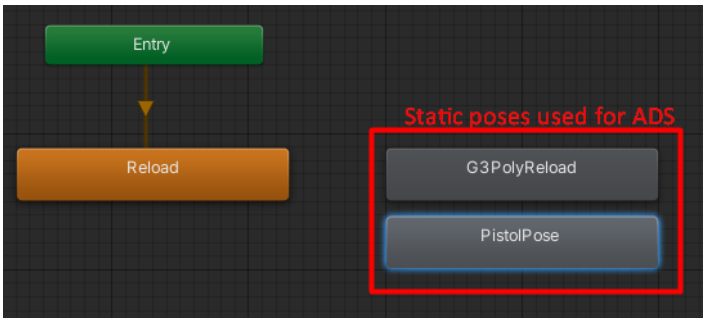
Translation targets define maximum and minimum values.
Aiming multipliers are applied when the aiming flag is set to 1.

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.

Aiming Setup

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.