Aim Assist Pro Documentation

February 2026

version 2.1.0

Contents

1 Greetings

Thank you for downloading my asset Aim Assist Pro. The asset's intent is to make fast paced shooters more accessible to controller players by providing functionality that modifies look input or adjust the aim.

2 About The Asset

The asset's actual content are the scripts that make up the aim assist system. A practice range is also present to test things out - one flavour for cinemachine 2, one for cinemachine 3 and one for no cinemachine. These use the new input system.
There's also a scene for the old input manager, that doesn't use cinemachine.

The package expects the camera to be used for aiming. What that means is, if your game has a camera present, and aiming is completely different to that (e.g. you eyeball shots with the player that has a laser pointer to show where he aims, but the camera is top down like a security camera, and is fixed), then the package is not going to work.
Important - for convenience, an `.md` version of this documentation is also available so you can have an LLM process it to be asked for information. The source code's Doxygen docs are also present, to make this more effective.

2.1 Unity dependencies

For the actual asset: none.
For the practice range scene:

- shader graph (any render pipeline - lighting may vary)
- input system (on certain scenes)
- text mesh pro (writing on walls)
- ai navigation (navmesh for moving enemies)

As these are marked in the `package.json` file, Unity should prompt to download the necessary dependencies.
Important - if the practice range is all pink for you, you either don't have shader graph in your project yet, or you need to restart Unity to get rid of its shader graph glitches.

2.2 Supported Input Systems

The asset itself does not rely on any input system, and as such will work with either. However, with the Unity's new Input System, integrating the input modifying assists happen through an input processor, without code.
If you're using the old Input Manager, or a third party tool, you'll need to use a single line of code to integrate the input modifying assists, more on that later.

If your project wants to use the new Input System but you haven't enabled it yet, Unity will prompt a dialog to do so. Alternatively, you can do it manually.

2.3 Supported Camera setups

The asset supports Cinemachine 2.x, Cinemachine 3.x and no cinemachine (manual camera-transform control pairs, more on that later in the notable components section).

For Cinemachine 2.x, the following rigs are supported:

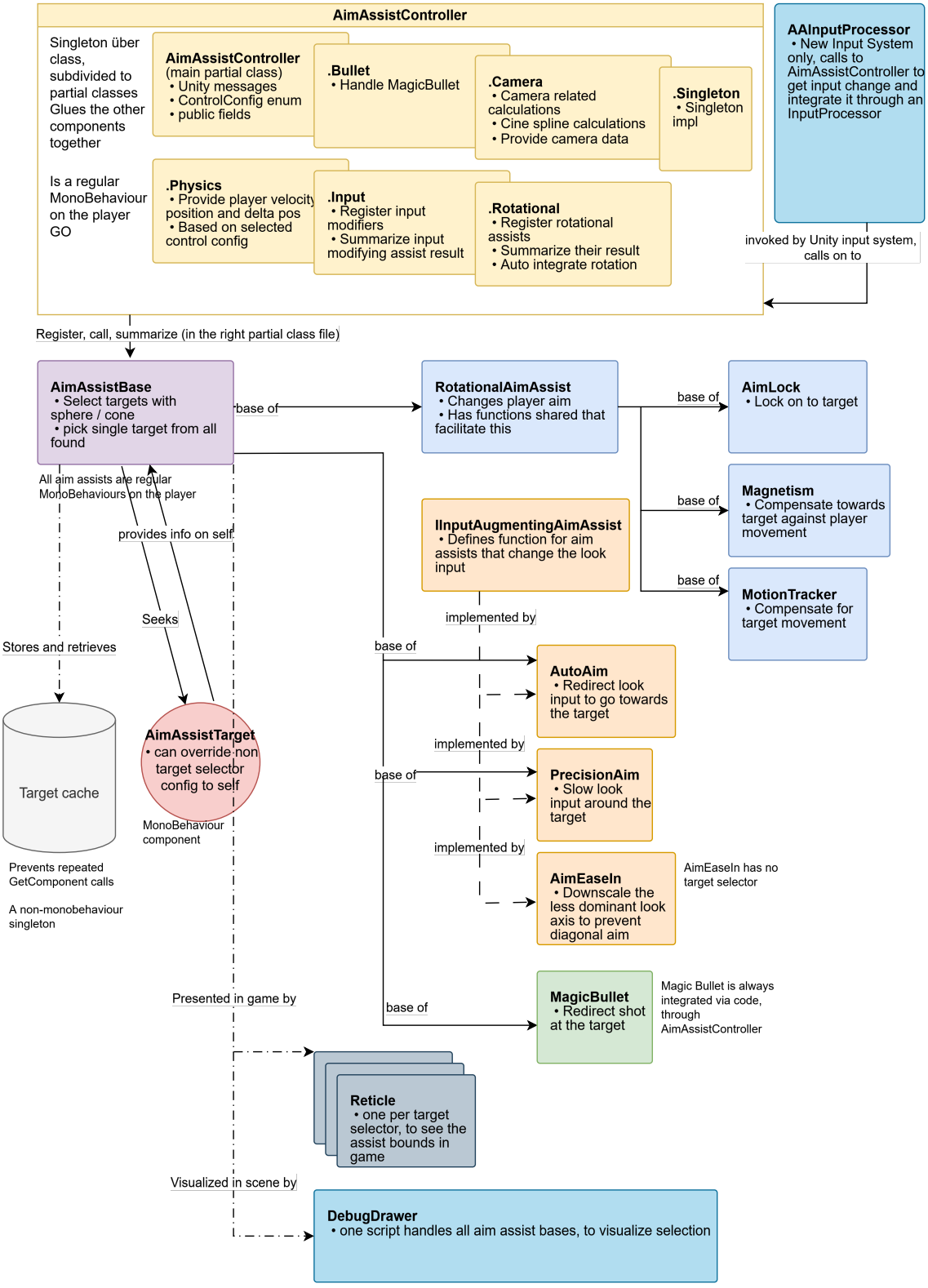
- VirtualCamera with no Body
- Cinemachine3rdPersonFollow (at least either Follow or LookAt set)
- CinemachinePOV
- CinemachineHardLockToTarget

For Cinemachine 3.x, the following rigs are supported:

- CinemachineCamera with no Body
- CinemachineThirdPersonFollow
- CinemachineOrbitalFollow (either 3 rig or sphere, again either Follow or LookAt set)
- CinemachineHardLockToTarget

Any other setup within cinemachine, the asset should prompt a warning and not do anything. There's also a no-Cinemachine solution implemented that can be used for scenes with a simple camera setup.

3 High Level Overview



Above see an overview of how the components interact that make aim assist work. The system works on a per-frame basis, with the target selectors working on a per-physics update basis.

- `AimAssistController` is a singleton class, subdivided into partial classes for ease of understanding, that holds all functionality together, serves as a scaffolding for the whole system, and can be used to ask for the end result.
  The controller also serves as default implementation for providing the camera data as well as player data. This can be overwritten easily, more on that later.

- Each `AimAssistBase` behaviour selects its own target, and has its own `targetSelectorConfig` that can configure how it does so.

- A target is defined by the `AimAssistTarget` component and must have a collider that is considered by the aim assist's target selector.

- If a target was found, the aim assist scripts do their calculations, and

  1. Rotational Aim Assists like `Magnetism`, `AimLock` or `MotionTracker`, generate a `Rotation`, that has its `Yaw` and `Pitch` in signed degrees. It is best **not** to use these separately, just use the controller exposed method to get the assist result.
  2. Input Modifying Aim Assist like `PrecisionAim`, `AutoAim` and `AimEaseIn` generate a modified *look input delta* that will be used as the player look input.

- The calculation are then applied -

  1. For rotations, several implementations are made and come with the package. These depend on `ControlConfig`s that will be detailed in the section below, and the rotation will be applied based on what control config is selected, and this include an option for a custom script made by the developer downstream. Whether any of this is applied, is behind a flag `autoApplyRotation`.
  2. For modified input, with the new Input System package, there's an `AimAssistInputProcessor` that uses the controller to return the assisted input, so no tinkering with code is needed.
     With the old input manager or a custom solution, the controller's `AssistInput` method is called to get the modified look input that replaces the raw one.

- To visualize and debug, `AimAssistReticle`s show the effective area for each aim assist (with a target selector), and these change color based on whether a target is found. the `DebugDrawer` can also be added to the player, that will visualize all aim assists' target selector in the scene view, and these also change color in play mode when a target is found.

4 Migration 2.0.4 to 2.1

If you owned a copy of the asset before this version releases, please follow the instructions below.

4.1 Obsolete components

More details on each of the mentioned components can be found in the *Components detailed* section below.

The following components are now obsolete - see their replacement added

- `LegacyPlayerPhysicsInfo` - completely replaced by `AimAssistController`, which makes this manual selection completely automatic.
- `PlayerControlType` - was used by the legacy player physics info. Not needed anymore
- `AimAssistCameraProvider` - was collapsed into `AimAssistController`, and got simplified. Also look for `RefreshCinemachineComponents` within the new `AimAssistController`.
- `CameraControlOverride` - was used by `AimAssistCameraProvider`, not needed anymore. There's no single or multi camera override, there's now a `Control Config` you can pick, and one of them is `Manual Override` where you have to set your yaw and pitch controls, and optionally a viewport camera that is considered *the* camera during calculations. If you use a multi camera setup, reassign the references when you switch cameras.
- `AutomaticPlayerPhysicsInfo` - replaced by `AimAssistController` and its `.Physics` partial class which also got more accurate. If you need your own custom overrides, implement `IPlayerPhysicsInfo`, which the controller also implements.
- `CameraProviderBase` - replaced by `AimAssistController`, that implements `ICameraProvider` which you can implement yourself with a behaviour script, and assign to the player, to provide your own cameras (if the controller default isn't right)
- `PlayerPhysicsInfo` abstract class - with `AimAssistController` this is not needed anymore and if you want manual control, implement `IPlayerPhysicsInfo` instead, which is also implemented by the controller.
- `InputAimAssistManager` - replaced by `AimAssistController` and its `.Input` partial class. The controller is also a singleton, so you just need to call `AssistInput` on the controller, similarly how you called `AssistAim` on this one.
- `AimAssistBulletManager` - replaced by `AimAssistController` and its `.Bullet` partial class. The controller is a singleton as well, call its `GetMagicBulletCameraForward` that returns whether the assist has a target, and sets the camera forward. **Difference**: if there's no target found, it returns the active camera forward instead of `Vector3.zero`.
- `YawTarget` - replaced by `ControlConfig` in `AimAssistController`, that defines an entire control rig, not just the yaw part. Pick the one that suits your project, or if you integrated rotation via code, pick `Script` and implement `IAimAssistRotationApplier` and put it on the player, where `AimAssistController` is.
- `AutomaticRotationalIntegrator` - replaced by `AimAssistController` and its `.Rotational` partial class. As mentioned in the previous point, select your control config or `Script` and implement the rotation yourself

- `CameraWithControl` and `CameraTransformControl` - replaced by `AimAssistController` as mentioned.
- `RotationalAimAssistManager` - its auto registering is replaced by `AimAssistController` and its `.Rotational` partial class, the rest were completely removed (more on that in this section a bit later).
- `AimAssistResult` - while not since 2.0.4, still, it's replaced by `Rotation` that is far more straightforward to understand (got `Yaw` and `Pitch` instead of an explanation) and has built-in support to normalize the angles to be the steepest, can be defined from directions and now has implicit conversion to, and from `Quaternion`s.
- `AimAssistBase`'s `queryTriggerInteractions` is now part of `TargetSelectorConfig`
- `AimAssistBase`'s and `TargetSelectorConfig`'s `useExtraSpheres` option for the cone selector has been removed. Zone scaling is introduced that doesn't sacrifice reticle accuracy and is much more effective than the extra spheres.

4.2 Changed behaviour

See the changed behaviour below. For more details on each, check out the *Components detailed* section.

- Unlike `YawController`, `ControlConfig` affects the player physics component in how it calculates the player position. This is because previously 'position' was just the player body's position, which may not be the point the camera orbits around in all scenarios. It is important to set a *position* that turns in place, as opposed to orbiting around another one.
- `AimLock` - it has a few changes
  - Now the system disables `Magnetism` and `MotionTracker` when `AimLock` is active. The reasn for this is that `AimLock` now fully compensates against target movement and player movement, making the other two unnecessary while this one is active. This is done to emphasis the *lock* effect.
  - Angular velocity is not dependent on target distance anymore, it's an outright number in degrees per second, affected only by the optional curve. The distance-dependent velocity was an unnecessary overcomplication that was rightfully brought up by some of my customers.
  - There is no separate angular velocity per axis, there's one that is shared, and it is measured by total target distance to crosshair in camera space.
  - Now it's an option to disable auto rotation on look input, and set a re-engage cooldown. Note that *look input* is acquired through using the input modifying part of the `AimAssistController`, either in the input processor or directly. If that isn't in use, then this feature won't take effect, lacking the necessary look input. Also, look input won't disable `AimLock`'s newly added player and target movement compensation, just the auto rotation.
  - It's possible to stick to the previously found target, if it was found again. This is possible because the system now doesn't exit early, but seeks all targets within scope it can find, and has new functionality to pick *the* target later.

- In cinemachine's OrbitalFollow or the old FreeLook, the system now converts more accurately between degrees and spline interpolation time. This eliminates wobbling in these configurations, enabling snappy lock onto the target with maxed out settings.
- Low effort scenes like the one for Orbital Follow and POV are removed. There's only the practice range, implemented for Cinemachine 2 and 3, No cinemachine (for the input system) and No cinemachine (for the input manager). The Cinemachine scenes switch cameras with `Tab`, between FPS, TPS aim, TPS free look and POV (camera) view.
- `MotionTracker` has an option `Stickiness` that makes movement compensation complete if the target center is near the reticle's edge. This is useful for fast moving targets as the reticle will stick to them more readily.
- `MotionTracker` also does not turn off when you're strafing in the target's direction. That decision made it hard to track a target that's faster than the player. Strafe compensation is fully handled by `Magnetism`, in the absence of `AimLock`.
- `Magnetism` had a problem with unnecessary compensation while moving back and forth, but not to the side, while aiming at a target. This was due to a calculation error against the target position, and is now fixed - only A-D strafing and jumping / falling is compensated against.
- In `AimAssistController.Physics` default implementation, in the absence of `Rigidbody`, `CharacterControl` or `NavMeshAgent`, manual velocity calculation has changed. *Velocity* isn't calculated with a running average anymore, because it was floaty but didn't reduce jitter from movement nearly good enough. The jitter was caused by inaccurate player position input and the lack of player movement compensation, so no manual smoothing is needed anymore.
- `RotationalAimAssistManager`'s 'interpolation' shenanigans are removed completely because they were an incorrect fix to a problem. Previous jitter during movement while having `AimLock` active was caused not by rotation *not smooth enough* but an inaccurate player location input, and Unity's own movement interpolation clashed with the unnecessary manual one.
- In `AimAssistBase` as mentioned before, `useExtraSpheres` has been removed, because at long range, it had to be extremely overtuned to be effective, and that made it subpar on short range. Both the sphere and the cone can now scale depending on distance.
- The aim assist bootstrapper has been simplified greatly, due to the underlying functionality getting simplified as well.
  As seen in the documentation's quick start guide, the dialog now only offers to pick the desired aim assists, their layers (as a reminder), and whether you want to have a debug drawer as well on the player.
- `DebugDrawer` was referred to as singular because now a single component is responsible to any aim assist you have on the player. It should also auto detect the player's assigned aim assists. Its options are also simplified, but the core functionality of detailed and overview drawing is kept.
- Target selection no longer exits on the first found target. The selector goes all the way, then picks the one target to assist against based on the strategy you choose - first to find (legacy), closest to crosshair (the one the player aims the closest to), and closest to player (in world space).

`AimAssistInputProcessor` still supports the `InputAimAssistManager`, but it's better to do a complete migration to `AimAssistController`.

5 Quick start

5.1 Setting up the player

Right click on the player game object, select *AimAssist*, then *Add Aim Assist*.
This will bring up a dialog where you can pick which aim assists to add.

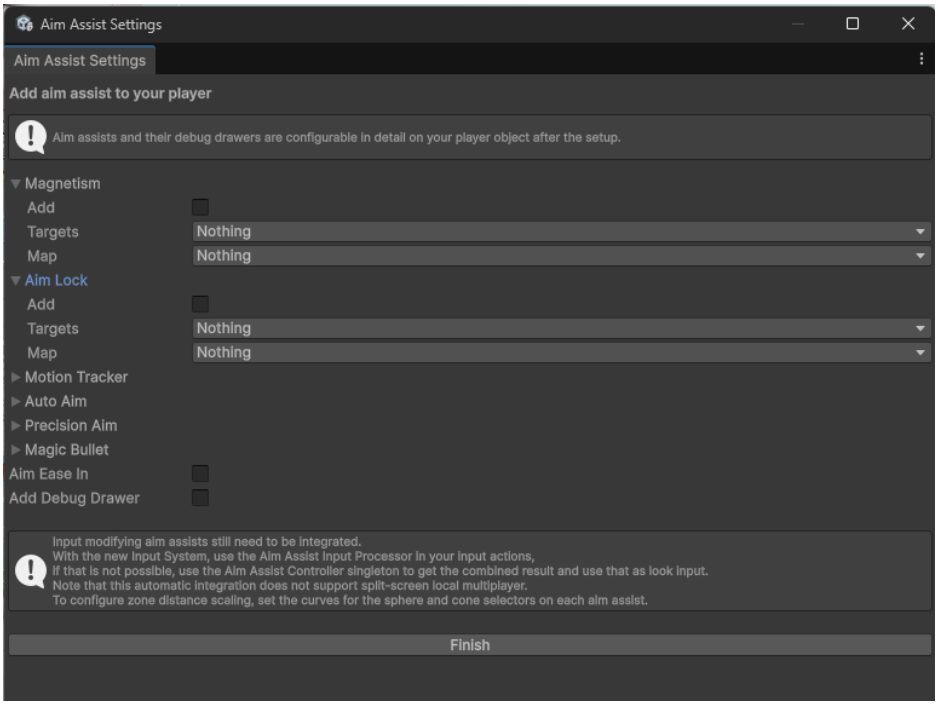


Figure 1: Add aim assist dialog

Since *v2.1*, the dialog is greatly simplified. You pick which aim assist to add, for each of them set the layers that targets can be found on, and set the layers that the aim assist considers, besides the targets. The reason of this separation is that objects on the `Map` layer will not be searched for an `AimAsssitTarget` through the `GetComponent` call, reducing repeated slow calls.
`AimEaseIn` does not have a layer setup because it isn't derived from `AimAssistBase`, and as such, has no target selection.

Below the aim assists is the option to add a debug drawer alongside the aim assist, which should automatically pick and populate aim assists automatically.
`AimAssistController` will be added automatically, and that contains all the default implementations, which were spearated into various classes in previous versions.

Ideally, the aim assists and the controller goes where the rigidbody / character controller (if any) is.

If you set up your target manually, make sure to add the desired aim assist scripts, set the relevant layers (both target and map), and also add an `AimAssistController`.

5.2 Setting up the target

Right click the target game object, *AimAssist*, then *Make target*. This will bring up another dialog, where you pick which layer the target will be on, and if there's no collider on the target already, you can select a type of collider to assign.

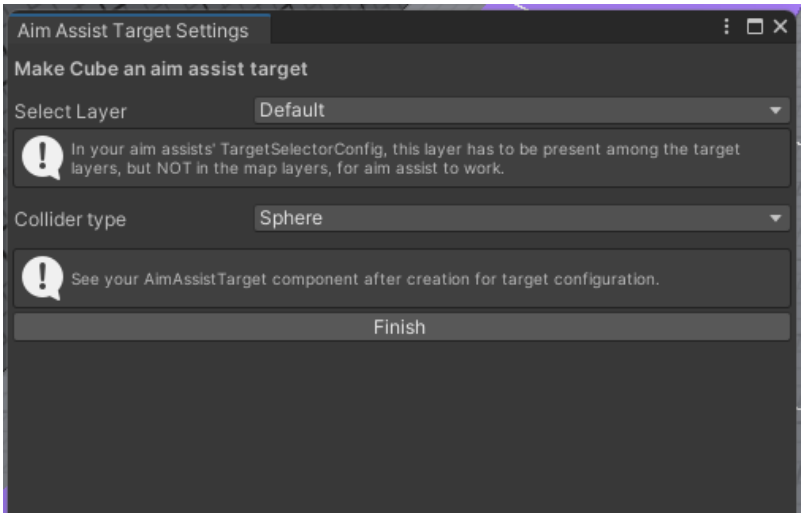


Figure 2: Make target dialog

5.3 Integration

5.3.1 Rotations

Several common configurations are implemented out of the box for the asset. If your setup matches any of these, pick that config in `AimAssistController`'s `ControlConfig` option.
Unlike in previous versions where only the yaw config was selected, now you choose an entire rig. Use the tooltips for details on how each of these work. **It is important to note** that changing this will also change how the physics component of the `AimAssistController` determines the player position (needed for calculations), because it's ideal if the position is the rotation point, that will eventually get the rotations applied to.
If your project, or another asset you use makes it impossible for automatic integration, select `Script` from `ControlConfig` (keep the auto apply flag on!), and implement `IAimAssistRotationApplier` with a behaviour script that you attach to the player. This will contain an `Apply` method with the frame's pitch and yaw rotation (in quaternion) that you can apply yourself.

A common cause for this is that the character control solution in use fully controls the target yaw and pitch, and just sets the player control to be the tracked numbers, which makes external rotation addition impossible.

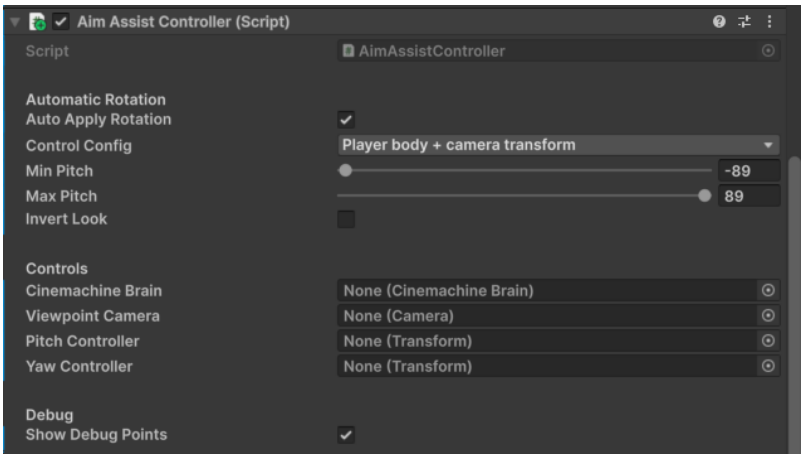


Figure 3: Aim assist controller

Outside of this, you can get access to the rotations using the `AimAssistControlle.Instance` singleton:

- `Rotation RawDelta` - this frame's calculated total rotational assist
- `Quaternion RawDeltaYaw` - this frame's rotation's yaw component, converted to `Quaternion`
- `Quaternion RawDeltaPitch` - this frame's rotation's pitch component, converted to `Quaternion`
- `Rotation CalculateTotalAssistRotation()` - force calculate the rotation. This is called by the system internally anyway to get `RawDelta`.

5.3.2 Input modifications

With the new Input System, just use `AimAssistInputProcessor` in your actions, and you're set.

Without the new Input System, e.g. either a third party solution or the old Input Manager, you'll need to use the `AimAssistController` singleton again, and call `AimAssistController.Instance.AssistInput(rawLookDe` to get the modified (assisted) look input. With the result, you continue with the player look implementatin as before.

It is also important to note that `AimLock` can now stop its auto rotation while there's look input, to avoid fighting the intentional player look. This will take effect only if `AimAssitController.AssistInput` is used, either through `AimAssistInputProcessor` automatically, or through code manually, because this is how the system knows what was the current LookInput for this frame. Use the `AssistInput` even in the absence of input modifying assist, to enable this feature of AimLock.

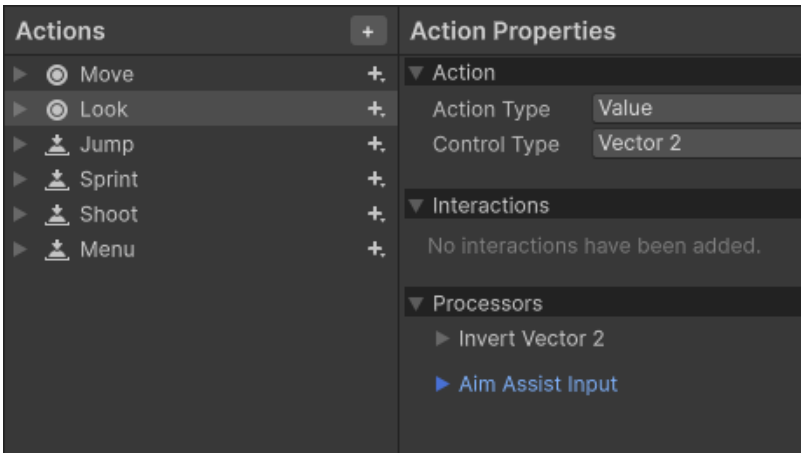


Figure 4: Input processor on Look action

5.3.3 Magic Bullet

A standalone assist that concerns the gun, not the player. As such, unfortunately it's integrated by code always. Call `AimAssistController.Instance.GetMagicBulletCameraForward` that returns whether the assist has a target to shoot at, and sets the incoming `assistedBulletDirection`, that defines a direction from camera forward to aim assist target. Use this in your gun handling script to get the assisted direction. **Example usage**: Practice Range / `BulletController` line **81**.

5.4 Manual Integration examples

5.4.1 Integration to BR200 by Photon Engine

Photon Engine released an excellent asset that's also free at the time of writing: BR200.
The reason no automatic integration works is that it doesn't use Input System, and the character controls directly set and overwrite a value that's replicated over the network from the input authoritative local player.

To resolve the first problem, `AgentInput` has a method `ProcessStandaloneInput`:

```
1  public sealed partial class AgentInput {
2      partial void ProcessStandaloneInput(bool IsInputPoll)
3      {
4          ...
5          // original:
6          // Vector2 mouseDelta = mouse.delta.ReadValue() * 0.075f;
7
8          // modified:
9          Vector2 mouseDelta = AimAssistController.Instance.AssistInput(mouse.delta.ReadValue() * 0.075f);
10
11         ... // rest of the method and class
12     }
13 }
```

In this example, as `Input Processors` and exposed methods weren't an option, I had to find the part of the code that handles the player input in a local player authoritative manner (so the network will respect the result that it's not a bot input), and run it through the Aim Assist Controller's assist code.

```
public class BR200RotationalIntegrator : NetworkBehaviour
{
    private KCC _kcc;
    private AimAssistController _aimAssist;

    private void Awake()
    {
        _kcc = GetComponent<KCC>();
        _aimAssist = AimAssistController.Instance;
    }

    public override void FixedUpdateNetwork()
    {
        base.FixedUpdateNetwork();
        if (!_kcc)
        {
            Debug.LogWarning("KCC not found for aim assist integration.");
            return;
        }

        if (!_aimAssist)
        {
            Debug.LogWarning("Rotational aim assist manager not found.");
            return;
        }

        UseAimAssist();
    }

    private void UseAimAssist()
    {
        Rotation delta = _aimAssist.RawDelta;

        if (delta == Rotation.Identity)
        {
            return;
        }

        _kcc.AddLookRotation(delta.Pitch, delta.Yaw);
    }
}
```

As for the rotational aim assists, Br200 exposes `NetworkBehaviour` where in `FixedUpdate` changes can be made that will be interpolated by their framework. In this example I use KCC's `AddLookRotation` to set the end result of my `RotationalAimAssistManager`, so once again, compared to previous versions, version 2.0.0 needs no direct reference to each aim assist.

5.4.2 Integration to ECM2 by Oscar Garcián

Another excellent package is the Easy Character Movement 2 for character movement. This package also has its own internal mechanisms to handle look input and as such, internally set values are overwritten to store the current rotation for a frame. Luckily, this package also exposes an easy to use API to get around this.

For input modifying aim assists, ECM2 uses Input System too, so `AimAssistInputProcessor` can be used. For rotational assists, I made the following component:

```
public class ECM2RotationalIntegrator : MonoBehaviour
{
    // or use your own character based on the Character base class
```

```
4       private ThirdPersonCharacter _tpsCharacter;
5       private FirstPersonCharacter _fpsCharacter;
6       private AimAssistController _aimAssist;
7
8       private void Awake()
9       {
10          _tpsCharacter = GetComponent<ThirdPersonCharacter>();
11          _aimAssist = AimAssistController.Instance;
12
13          if (!_tpsCharacter)
14          {
15              _tpsCharacter = GetComponentInParent<ThirdPersonCharacter>();
16          }
17
18          _fpsCharacter = GetComponent<FirstPersonCharacter>();
19
20          if (!_fpsCharacter)
21          {
22              _fpsCharacter = GetComponentInParent<FirstPersonCharacter>();
23          }
24      }
25
26      private void LateUpdate()
27      {
28          Rotation delta = _aimAssist.RawDelta;
29
30          if (_tpsCharacter)
31          {
32              _tpsCharacter.AddControlYawInput(delta.Yaw);
33              _tpsCharacter.AddControlPitchInput(delta.Pitch);
34          }
35          else if (_fpsCharacter)
36          {
37              _fpsCharacter.AddControlYawInput(delta.Yaw);
38              _fpsCharacter.AddControlPitchInput(delta.Pitch);
39          }
40      }
41  }
```

Here I use the exposed `AddControlYawInput` and `AddControlPitchInput` with the calculated angles, and ECM2 handles the rest.

6 Components detailed

6.1 AimAssistTarget

`AimAssistTarget` defines a target from the perspective of the aim assist system.

To make a GameObject an aim assist target, you can right click it, and from the GameObject context menu select `AimAssist / Make Target` that will bring up a dialog window to equip the object.

In any case, a GameObject needs the following to be considered an eligible target:

- Have a component `AimAssistTarget`
- Have a collider on the same gameobject as the target component. If you want this to be a trigger, enable it in each aim assist, using its configuration `QueryTriggerInteraction`.
- Assign it to a layer that is part of an aim assist script's **TargetSelectorConfig / Target Mask**

Any target may override an aim assist's configuration concerning itself, and this can be done by

- Ticking the **enable aim assist override** checkbox
- Checking whether or not to use the aim assist - if an aim assist is disabled on the player, this check will not have an effect. If the player's aim assist is enabled, this check can disable it.
- Setting the configuration for the aim assist, but not its target selector (target selector are always active, this is active only when a target is found).

`AimAssistTarget`'s *position resolver* means a method to let the aim assist system know what *Target Position* means. It is useful for aim assists that rely on a precise position as they directly use that in their calculations. `AimLock` and `AutoAim` for example will turn the player to look at this point.

- **Object Transform** means the `transform.position` of the target game object
- **Collision Center** means the center of the collision geometry of the target game object
- **Proxy Transform** means a proxy transform location that has to be set in the inspector. If it isn't set, the script defaults to `transform.position`.

Besides marking *GameObject*s as targets, it also contains events that notify the subscribers when an aim assist acquired that target.

- `TargetSelected` event is called once, when an aim assist finds the given target. It isn't called repeatedly in the assist loop.
- `TargetLost` event is called once, when an aim assist doesn't see the target anymore. After this, `TargetSelected` can be invoked again.

And lastly, it provides info on the target, like its (Unity interpolated) velocity or collider volume.

6.2 Cache

To save performance, caching is enabled to reduce unnecessary `GetComponent` calls that look for the target script. This cache is built on the fly - when the aim assist finds a collider, it looks for the aim assist component. To further optimize this, a separate layer is assigned for targets and the map itself, and the map layer's objects will **not be queried** for an aim assist target component. The reasoning behind this is that there should be a handful of targets / enemies in a game, but there may be hundreds of other objects on the map, that still should block aim assist as they block the line of sight.

This also means that if an object on the target layer doesn't have an AimAssistTarget script on it, but later during runtime gets one, and the cache has already stored that it has no target script, then even after assigning the script to it runtime, the system won't recognize this game object as a target. To fix this, call `Cache<AimAssistTarget>.Instance.Purge()` to clear the cache and have it rebuild itself.

- If an assist script is already stored, it will just return that and won't call `GetComponent`.

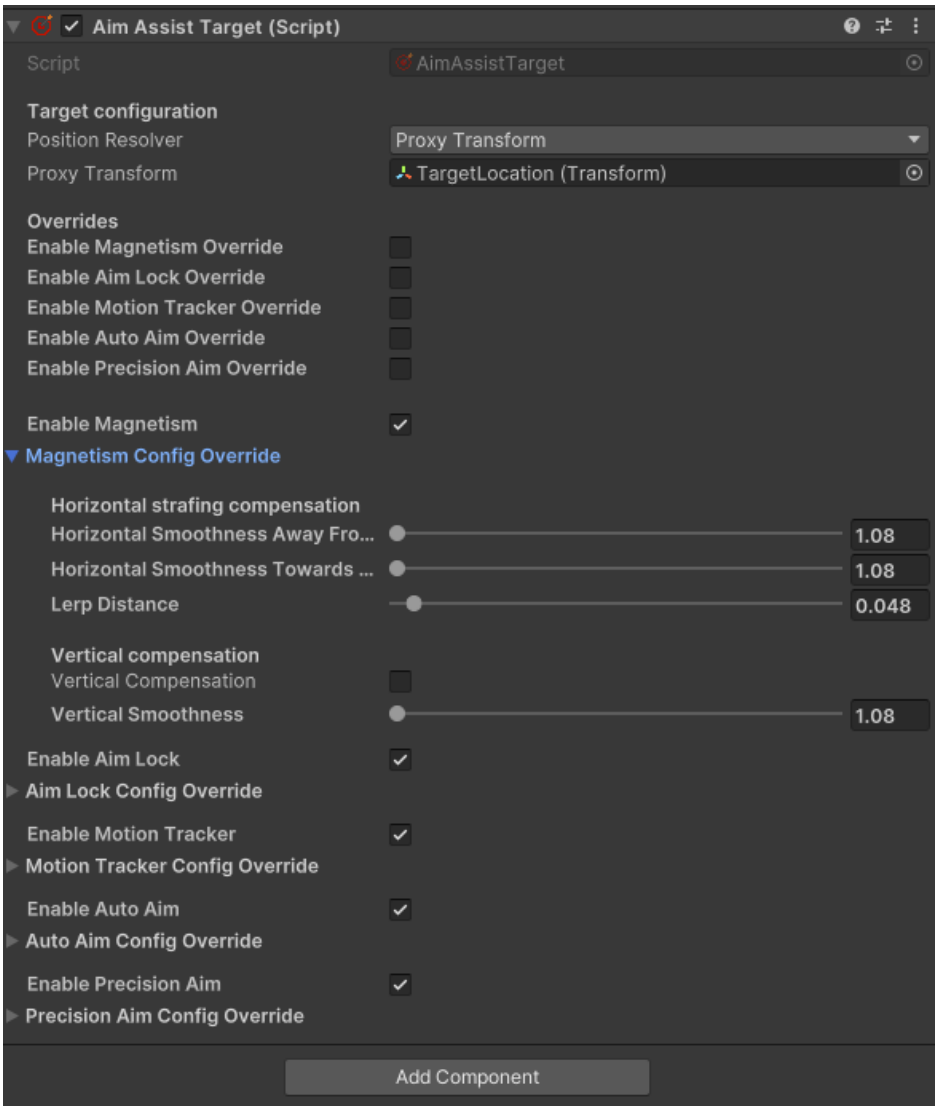


Figure 5: Configuration for the aim assist target

- If it doesn't find it, it's going to cache null and won't try to look for it again. **When assigning these target scripts to *GameObject*s runtime, make sure to either add it to the cache as well or just purge the cache so it can rebuild itself**.
- If it finds it on the *GameObject* then it stores the target script and also returns it. Next time it will just find it in the cache.

6.3 AimAssistController

Whenever you want to access something within the asset, this is the go-to component to use. Being a singleton, you can just call `AimAssistController.Instance` to get a reference to it. A partial class, its functionality is subdivided into separate files for ease of maintenance.

6.3.1 Main file

Contains the interfaces to implement, contains the public fields to inspector settings, contains the Unity Messages that glue together the methods that are implemented mostly by the rest of the partial class.

6.3.2 .Bullet

Only responsible for handling the `Magic Bullet` component, that is, registering it into the `AimAsssitController` and giving a simple access to it for the actual project this asset is used in.

6.3.3 .Camera

Contains logic related to any camera functionality, including Cinemacine.

- Implements `ICameraProvider`, so it provides the active camera transform, the active camera, the active Cinemachine camera and the raw camera transform.
- Out of these, `RawCameraTransform` is an outlier because it doesn't return a Unity construct. It returns an `FTransform`, that got its name from Unreal Engine's same class, and it is present so a `transform` construct can be made up from code, instead of always having to represent an actual Unity managed `Transform`. Its functionality is very similar to Unity's `Transform` otherwise, besides the fact that it doesn't handle scales, due to Aim Assist not having to deal with those.
- `.Camera` contains the implementation `Degrees2Interpolation` to convert between angles in degrees and cinemachine spline interpolations, that were needed for both Orbital Follow and Free Look to work properly. This is updated every frame, so the proeprty has a private setter.
- `RefreshCinemachineComponents` is put here as well, a public method to find the relevant cinemachine stuff after a change was made to the scene. This **won't** overwrite `cinemachineBrain` if it was set from the inspector.
- This refresh components is called on either version of Cinemachine's camera activated event, just for good measure.
- For Orbital Follow and Free Look again, to get an accurate aim with cinemachine, in spite of using a spline interpolation rather than actual degrees, an interpolation value needs to be calculated.
  - Spline points are evaluated for the rig's local space. This is done with the exposed functionality in Cinemachine 2, which was unfortunately made private / internal in Cinemachine 3. So for that, the functionality was copied from the official source code, and made into `Cinemachine3OrbitalSpline`, that gives the same results as the official cinemachine 3 spline's resolver.
  - These rig-local points are then converted into world space.
  - During the use of `AimLock`, the *at-target* rotation is then calculated through an intersection of the target and the control point (e.g. `LookAt`), and the cinemachine spline (approximated), which gives a spline interpolation that defines where the camera would be, to aim perfectly at the target.
  - The above needs to be calculated per frame for smooth results, but it is not active unless the specific rig is selected and is in use.

6.3.4 .Input

Contains logic related to input modifying aim assists, that is, registering them and providing a method that can be called to calculate, summarize and return the final assisted look input.
Call `public Vector2 AssistInput(Vector2 rawLookDelta)` with your raw look input, to get the modified look input delta, which can be used in your project as before.

With the new Input System, you do not have to call this manually, the `AimAssistInputProcessor` will call this method for you. But with a custom third party input handler, or the old Input Manager, this is the method you'll use.

Also important that `AimLock` needs to have this used for its `disable on look input` functionality, because this is how the actual look input is registered for the aim assist system.

6.3.5 .Physics

Contains logic related to providing info for the assist calculations about the player. This is the file that contains the default implementation of `IPlayerPhysicsInfo`.

- `Velocity` delegates to Unity's `Rigidbody` or `CharacterController` first, and if none is present, it returns the position difference over the previous frame. For smoother results, it's better to have a Unity component in place because internally it interpolates movement and smooths over micro collisions.
- `Position` shows the position of the player, which needs to be precise because this provides the basis of the calculations, to determine *where* the player has to start rotation from.
  I have found it to be ideal that this value is the exact point that is going to be rotated, when look input is applied. For example, if you rotate your player body for horizontal look input, then the horizontal `(x and z)` coordinates of the position should be the player body position, while the `y` coordinate should be the camera position, assuming the camera orbits around itself.

- **Important to note** that this value is affected by your selected `ControlConfig`, for the reason mentioned above.
  - `ControlConfig.ManualOverride` returns a mix of the pitch- and yaw controller positions, using `x` and `z` of the yaw controller, and `y` from the pitch controller.
  - `ControlConfig.Cinemachine2FreeLook` attempts to find the `Follow` or `LookAt` transforms if they are set, and return either of them, first the `LookAt`, and if absent, then the `Follow` position. If neither are present, it returns the camera position. For best results, it's advised to have these components set for the control rig, because the camera will orbit around these points.
  - `ControlConfig.Cinemachine3OrbitalFollow` attempts to return the `LookAt` + `Offset` position sum, because that's the effective look at position of the rig. If these are not set, then the regular camera position is returned. Again, for best results, the `LookAt` should be set.
  - `ControlConfig.Cinemachine2POV` or `ControlConfig.Cinemachine3PanTilt` just returns the camera position as the camera will rotate in palce for either.

  - Any other configuration will just return the mix of the player game object's horizontal position (x and z), and the camera's vertical position (y).

- Besides these, a raw velocity is calculated based on position difference, but as stated, it is best to rely on a Unity component for smooth results.

6.3.6 .Rotational

Registers the rotational assists, holds the framely calculated aim assist delta rotations, and also has the methods to calculate those on demand.

- `RawDelta` is the current frame's `Rotation` aim assist rotation.
- `RawDeltaYaw` is the current frame's `Quaternion` aim assist yaw rotation.
- `RawDeltaPitch` is the current frame's `Quaternion` aim assist pitch rotation.
- You can get a calculation result with `Rotation CalculateTotalAssistRotation()`.
  This isn't going to overwrite the previous values, and handling the mix of the 3 rotational assists is present in this method. What this means is, when `AimLock` is active, `Magnetism` and `MotionTracker` will be disregarded.
- The rest of the code is about applying the pre-defined implementations that are selected using `ControlConfig`.
  Even thought these are private methods, all of them are commented and will be part of the exported doxygen file. This means that you can use your favourite LLM to provide implementation level details about each of these without you having to look into the methods themselves.

6.3.7 .Singleton

Contains the code that makes `AimAssistController` a singleton.

6.4 AimAssistBase

The foundation for all aim assists that use a target selector (all except `AimEaseIn`, which does not work on targets). This also means that each aim assist can have its own target selection config and choose different targets based on that.
To configure how a target is selected, a `TargetSelectorConfig` struct is used.

- `Aim Assist Enabled` enables or disables the aim assist, including its target selector. If this is disabled, then per-target overrides cannot enable the aim assist.
- `Target Selection Mode` picks whether to shoot spheres or lines in a cone shape to select a target. If you use spheres, then the selection radius will get progressively smaller over distance, as you'd expect.
  With the cone, you get a uniform spread from the player's point of view.
- `Multi Target Resolution` can be selected for a strategy on how to pick *the* target from all the found targets. Options are *First to find*(legacy), *Closest to crosshair* (player aims closest to) and *Closest to player*(in world space).

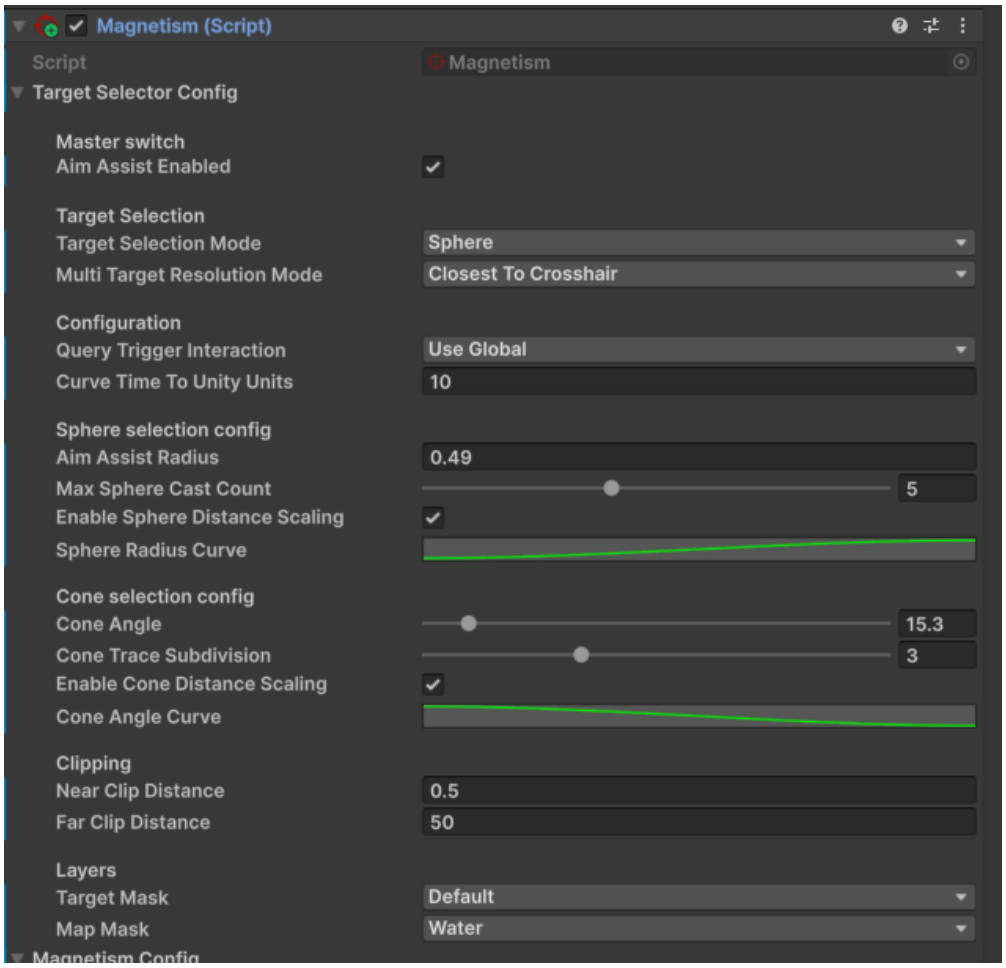


Figure 6: Target Selector settings

- `Query Trigger Interaction` Enable or disable interacting with trigger colliders at the developer's discretion.
- `Curve time to unity units` is a multiplier for the distance scalers - 1 unit on the curse is this amount of unity units. Having it 10 means that if you set the curves 1 unit on the x axis, the end of it means 10 unity units.
- `Aim Assist Radius` is used with sphere type target selection, and is the outmost sphere cast's radius.
- `Max sphere cast count` sets the number of concentric but progressively smaller spheres to cast in sphere mode, that help with precision around corners.
- `Enable sphere distance scaling` enables the curve that is applied to the sphere radius when the player looks at a distance. With this, you can for example make the sphere larger at long distances, so it casts a wider net.

- `Sphere radius curve` is the curve whose value will be applied to the sphere radius, at a distance defined by the curve time multiplier by the curve time conversion value.
- `Cone Angle` is only used with the cone selector, and sets the angle of the cone shape in which the lines are shot. This also means that a narrow cone angle has more dense lines that find targets more accurately.
- `Cone Trace Subdivision` sets the density of the lines that are shot in the cone shape. Increasing this means shooting **exponentially** more lines.
- `Enable cone distance scaling` enables the curve modification to the cone angle
- `Cone angle curve` defines the curve that is applied to the cone angle at a given distance. The curve value is the multiplier, and is selected by the curve time and the unit conversion that yield unity unit distances.
- `Near Clip Distance` is the near clip distance of the target selector. It works as an offset to the origin of the lines / spheres, so instead of shooting from the camera, increasing this means putting the start of the lines / spheres increasingly in front of the camera
- `Far Clip Distance` is the farthest point where the aim assist can reach
- `Target Mask` are the layers that could contain targets. Objects of this layer are queried for an AimAssistTarget when found by the target selector. For best performance, most objects on this layer should have an aim assist target component
- `Map Mask` are the layers that can block out the aim assist, e.g. cover and the map itself, but will **not** be queried for AimAssistTarget components. This is separated from the target mask as there are a hundred times more non targets that have to block the aim assist line of sight than actual targets. You should not put any AimAssistTarget object on this layer or the aim assist will not find it.

Once there's a target, If we weren't already looking at it, the `TargetSelected` event is fired. If we found no target but used to see one (one frame ago) then the `TargetLost` event is fired.

New feature that regardless of target selection method, the selector will actively seek out the previous target, to see if it's still in scope. It'll be then added to the found targets, and this fact can be used by certain aim assists to prioritize these old targets over new ones.
Using the new `DebugDrawer`'s `Detail` drawing mode, you can see the example with two assists enabled, one using the cone selector with a broader nier clip distance, and one using the sphere selector at a right near clip distance.
The way reticle distance scaling works is, the system shoots a single ray forward from camera center to find the actual distance (cannot use target distance, this has to work when there is no target acquired too), and uses that, divided by the unity to curve unit conversion value, to sample the curve, which will return its value, that is going to be a multiplier on the reticle, be it sphere or cone. So in essence, the curve conversion value multiplied by the curve time is going to define a distance, over which the curve's Y axis value is applied to the cone angle / sphere radius as a multiplier.

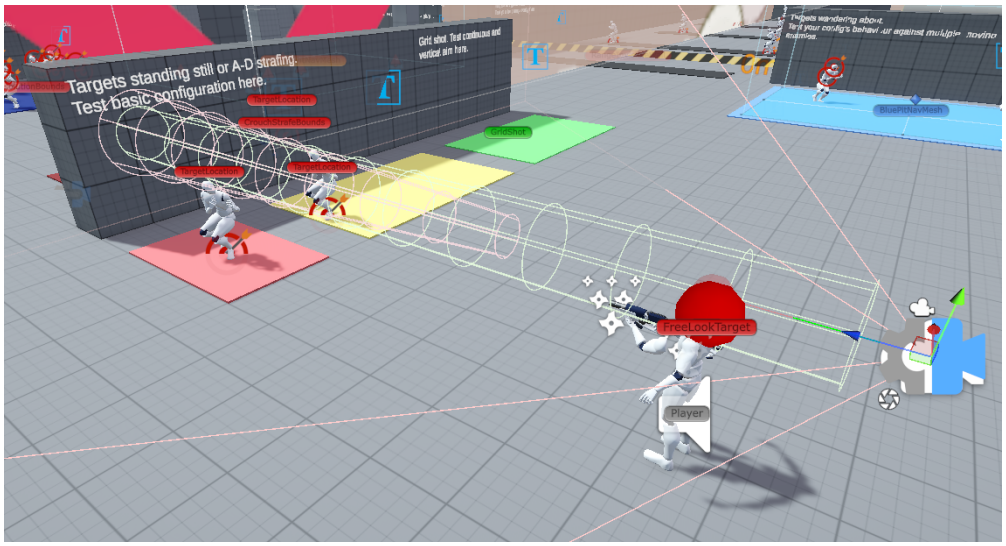


Figure 7: Sphere and cone selector

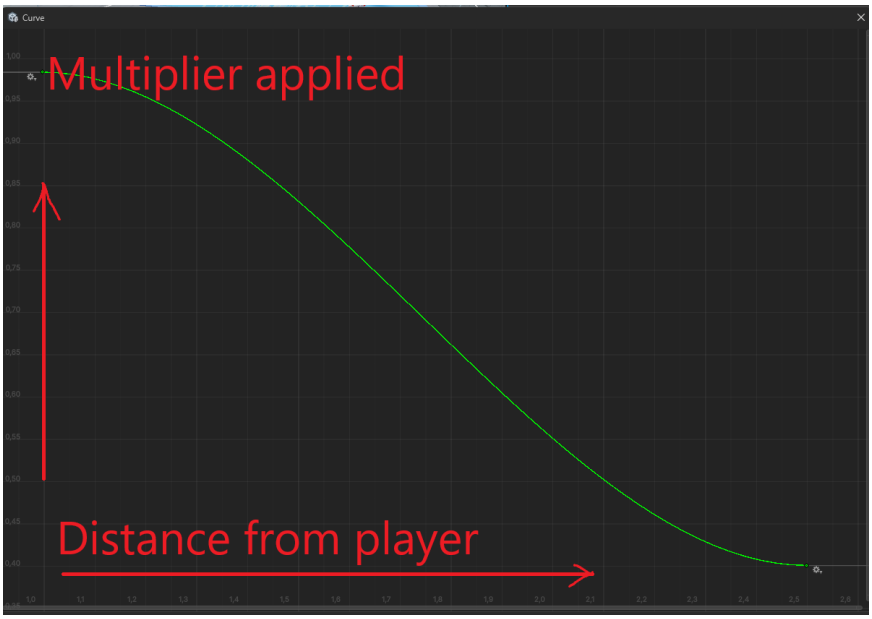


Figure 8: Reticle distance scaling

In the previous version, the cone selector could shoot extra spheres on top of its lines but that is removed - this reticle scaling is way more effective for long distance aiming.

6.5 Rotational Aim Assists

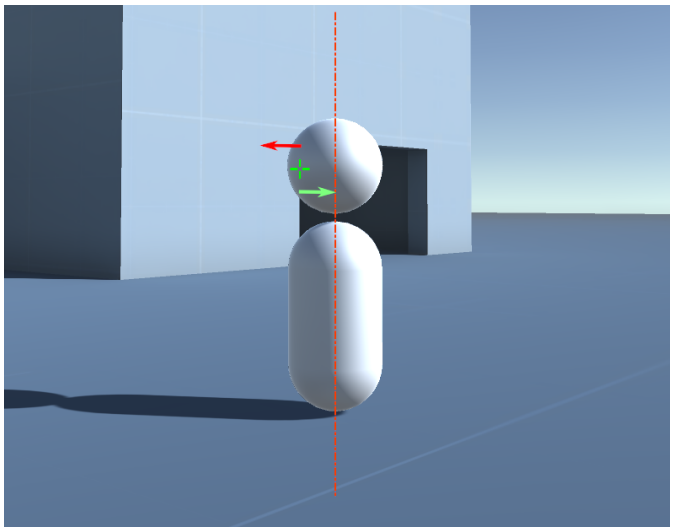
Rotational aim assists rotate the selected `ControlConfig` by a given degree, or in Cinemachine free look / orbital follow's case, a given spline interpolation that is translated by a degree.
The end result of the calculation is going to be a `Rotation`, that is a delta yaw and pitch value that

represents the shortest route to rotate an angle.

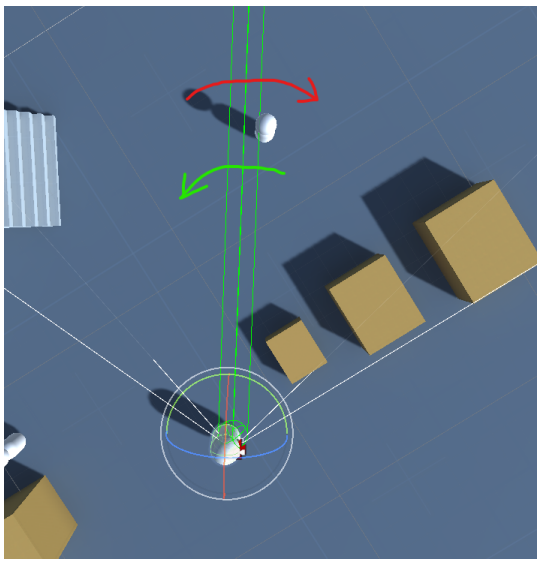
In any case, you'll add the resulting rotation to your control config, either using a pre-defined method and apply it automatically, assuming that your character controls don't keep track their own internal state, which is then locked every frame, or through one of your own scripts, as described in *quick start*.
Important - in Cinemachine free look or orbital follow configuration, the `pitch` component of this rotation is not an angle in degrees, but a delta spline interpolation that moves the camera pitch by a given degree. The conversion comes implemented as part of the package, and is assigned at the end of the calculations in `RotationalAimAssist`, to take place in all three rotational assists.

6.5.1 Magnetism

Works by loosely compensating for the player's strafing movement to make it easier to keep track of the enemy.
The screenshots are from one of the earliest version's example scenes that are no longer in the asset as they are replaced with a much higher quality practice range. They however still show the idea accurately so they serve as a good example.
Since as of *v2.1*, `AimLock` now also compensated for player movement, as well as target movement, whenever `AimLock` is active, `Magnetism` will not take effect.



(a) Movement to and away from target



(b) Turn compensation for movement

Figure 9: Magnetism

To understand what the comments mean in the script and documentation let's walk you through a couple of terms for the sake of clarity.

- Moving **towards** the target is the movement represented by the **green** arrow on figure **a**. It basically means that the player is strafing sideways in the target's direction and is not yet facing the target,

thus moving towards its center. Figure **b** shows with a **green** arrow how Magnetism will compensate that strafe to prevent dropping the crosshair right to the target.

- Moving **away** from the target is the movement represented by the **red** arrow on figure **a**. It means that the player is strafing sideways and is moving the opposite direction, away from the target's center. Figure **b** shows with a **red** arrow how Magnetism will compensate for strafing by rotating the camera back towards the target.

So strafe compensation means turning the camera the opposite direction of the player's strafe direction. Magnetism does this but slowed down by a given factor, and that factor differs whether the player moves towards, or away from the target. This separation allows for keeping track of the target when the player moves away while still allowing for a smooth mirror strafing to follow the target's movement.

Magnetism, like any rotational aim assist, relies on `IPlayerPhysicsInfo` for player velocity. `AimAssistController` provides the default implementation as detailed in its section, but you can implement your own if needed, but more on that later. See the settings below.

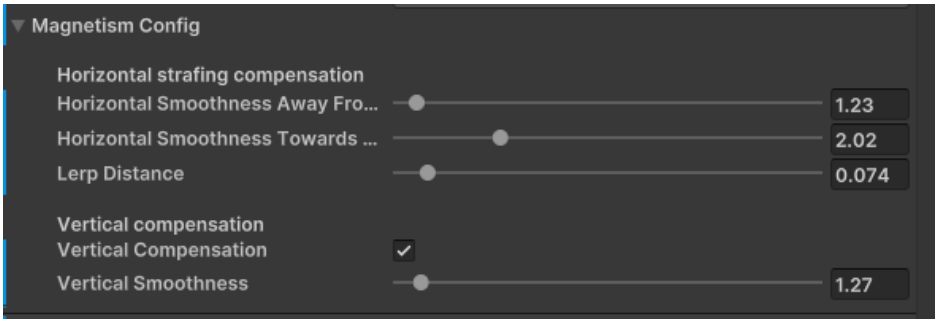


Figure 10: Magnetism settings

- `Horizontal smoothness away from target` smoothness, as a divisor for the rotation that compensates the strafing of the player while they are moving *away* from the target (defined above). The larger this number the less the compensation is for the strafing. It's capped above 1, but if the value would be set to 1, you would "lock" your aim next to the target and hard track that spot instead of letting the crosshair reach the target.

- `Horizontal smoothness towards target` smoothness, as a divisor for the rotation that compensates the strafing of the player while they are moving *towards* the target (defined above). The larger this number the less the compensation is for the strafing. It is ideal to set it a bit larger than `Smoothness away from the target` to allow for mirror strafing, that is, tracking the target's movement with your own movement.

- `Lerp distance` is the distance in the middle of the target that interpolates between the *towards* and *away* smoothness. It is a percent of the aim assist radius (for the sphere selection, and for the cone it's the cone's radius at the distance of the target, in both cases, adjusted for zone scaling). Without this when your speed matches the target's while mirror strafing there would be a noticeable stutter present at lower frame rates due to the aim assist switching between the two smoothness values abruptly.

- `Vertical compensation` whether the aim assist should compensate for the player's vertical movement e.g. jump. Useful when walking on stairs or dropping from a high ground - it keeps track of the height of your aim so you don't end up aiming at the ground.
- `Vertical smoothness` smoothness of the vertical aim, similar to the others defined above.

6.5.2 Aimlock

Locks on to the target with automatic rotation that can be further customized with a curve, and fully compensates for player movement, as well as target movement, to make the *lock on* feel accurate.
When this assist is active, `Magnetism` and `MotionTracker` will not be taken into account, due to the reason mentioned above. Do **not** access either of these assists on its own, it's best to leave that to the `AimAssistController` where this selection logic with the active asssits is present. The controller already calculates and sets the resulting `Rotation` and `Quaternion` values for each frame, detailed in the controller's section.
AimLock interpolates the player's position but uses an accurate look forward vector to help incidental jitter due to rendering timing issues with the position of the player.

Fires an event called `notifyTargetCenterReached`, that notifies when the target's center, within the given Disengage Radius, is reached. This is active regardless whether disengage is enabled or not, and fires once for any maintained target (so it can fire again when a new target is found, or this one is lost).

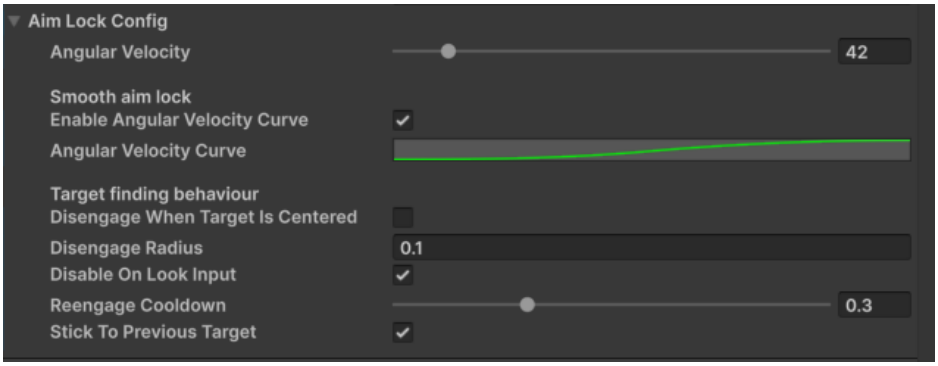


Figure 11: Aimlock settings

- `Angular velocity` - how fast the automatic rotation happens, in degrees per second. Goes from 1 to 360.
  Unlike previous versions of the asset, this isn't going to scale the value based on target distance, it'll apply that outright.
- `Enable angular velocity curve` enables or disables the smoothing curve for the aimlock's angular velocity.
- `Angular velocity curve` a curve that serves as a multiplier based on how close the player's crosshair is to the center of the target. The X axis is the aim assist's radius with 0 being the crosshair and 1 being the outer edge. The Y axis is a multiplier that is applied to the angular velocity. It is advised to keep both axes between 0 and 1 as it'll be clamped anyway.

- `Disengage when target is centered` will stop AimLock when the target's center is reached, within a certain radius. This is useful to make AimLock only force to look at the target, but auto-disable afterwards. This is reset after a new target is found, or the current target is lost.
- `Disengage radius` is the radius that defines how close the aim has to be to the target position to disengage.
- `Disable on look input` disables the *automatic rotation component* of the aim assist, when there's any look input present. Look input presence is determined by using `AimAssistController.AssistInput`, either on its own or through the input processor, as detailed in the controller's section. This option will **not** disable the player and enemy movement compensation.
- `Reengage cooldown` sets the time that it takes for the auto rotation to engage again, if the `disable on look input` option is active.
- `Stick to previous target` will choose the previous frame's target, if there was any, over anything new, if that target was found again by the selector.

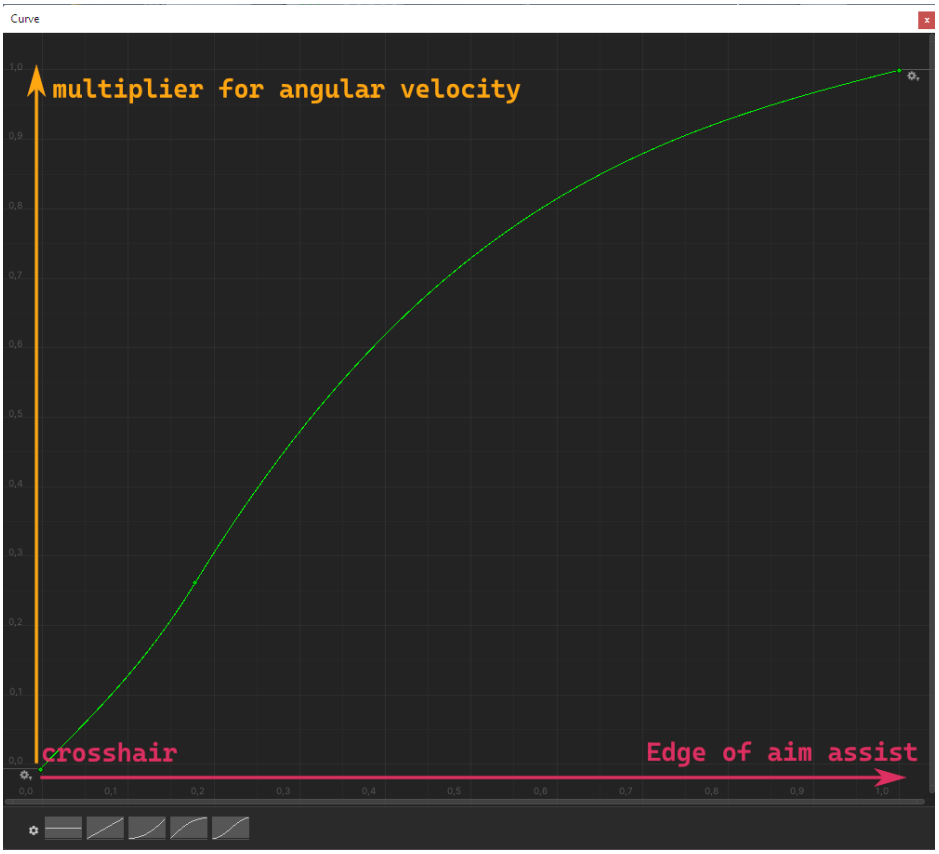


Figure 12: Aimlock angular velocity curve

As stated, the aim assist fully compensates for enemy movement, when the enemy is walking into the crosshair (otherwise the auto rotation and the enemy movement compensationm would just fight each

other). The assist also fully compensates for player movement, but only when the player is looking directly at the center of the target, where the auto rotation wants to take it. Towards the edges, the player movement compensation fades. This is useful to help break free of the target through movement if the player already has look input, which disables automatic rotation.

6.5.3 Motion Tracker

A new rotational aim assist that loosely tracks the enemy through its movement, but does not look at it on its own like `AimLock` does. `MotionTracker` also enables pre-aim and lets the enemy walk into the crosshair due to having two different strength values for keeping up with the enemy when it's past the crosshair, and when it's moving towards the crosshair. Think of it like `Magnetism` but for enemy movement.
This assist is **not** active when `AimLock` is enabled, because `AimLock` already fully compensates for enemy as well as player movement, on top of its automatic rotation.

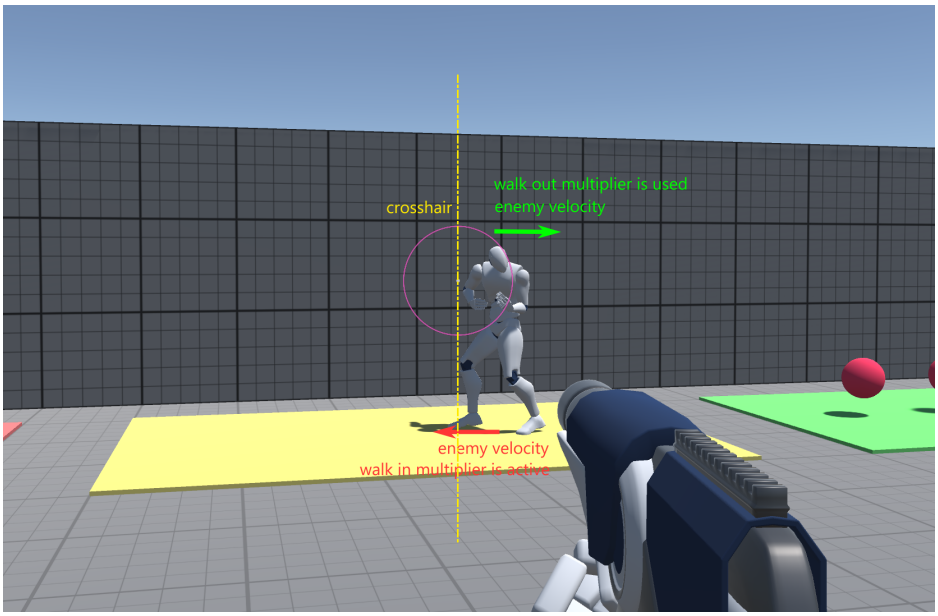


Figure 13: Motion Tracker

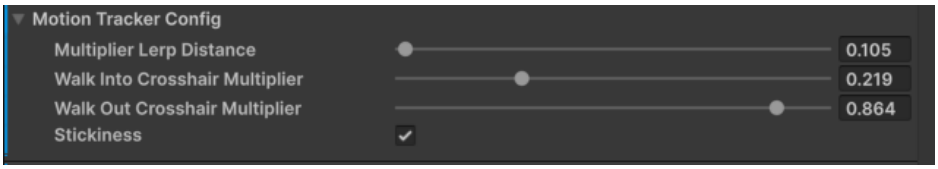


Figure 14: Motion Tracker settings

Its settings are

- `Multiplier Lerp Distance` is similar to `Magnetism`'s lerp distance, a percent of the sphere radius with sphere selection, or the cone radius at target distance with cone selection along which

line the interpolation will take place between the two smoothness values. A wide value means smoother transition, and a narrow value means a sudden transition. Without this, a large change between two values would result in jitter.

- `Walk Into Crosshair Multiplier` is the multiplier that will be applied to the direct 1 to 1 tracking of the enemy velocity while it is walking into the crosshair. Lower values let the enemy step into the crosshair faster.
- `Walk Out Crosshair Multiplier` is the multiplier that will be applied to the direct 1 to 1 tracking of the enemy velocity while it is walking out of the crosshair, and the crosshair has to be dragged after it. Higher values mean a more aggressive follow of the enemy.
- `Stickiness` means sticking to the target, when it's about to leave the crosshair. it is done by the assist compensating 1:1 against enemy movement, when the enemy is far enough to the edge of the crosshair. Because of this, using `MotionTracker` in tandem with `Magnetism` is a great, non invasive way to keep the player crosshair roughly on the target, without completely outsourcing all mechanical abilities.

Unlike Magnetism, these values are present for both horizontal and vertical axes.

6.6 Input Modifying Aim Assists

Input modifying aim assists augment look input, and their result should be used as the previous raw look input for handling player input. This is simplified through `AimAssistController` as detailed already. As a quick recap, to use these, with the new Input System, just use `AimAssistInputProcessor` on your look input action, and with a third party solution or the old Input Manager, you'll integrate manually, through running your raw look input through `AimAssistController.AssistInput`, and using the resulting assisted input instead of the raw one.

6.6.1 Aim ease in

A very simple aim assist which selects a dominant axis (the one where your delta is higher) and downscales the other. This allows for a convenient horizontal or vertical aim instead of going diagonally. It takes in the look input delta `Vector2` and returns a modified look input delta `Vector2` so you just have to run your look input through this before using it in your script for look rotations. Differs from other aim assists that it doesn't even require a `TargetSelector` so it's not a subclass of `AimAssistBase` - its behaviour is present always and is not tied to a target.

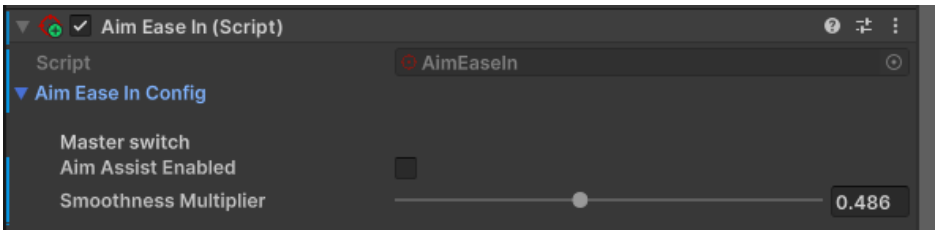


Figure 15: Aim ease in settings

- `Aim assist enabled` a master switch to enable the aim assist.

- `Smoothness multiplier` a multiplier for the less dominant axis.

6.6.2 Precision aim

Simply scales down the input delta using 2 values - the multiplier at the center of your aim, and a multiplier at the edge of the aim assist radius.

From version 2.0.0, the target's volume is also taken into account - the aim assist radius' edge doesn't have to touch the center of the target to begin working, giving it a more generous working space. The 'target volume' is approximated with the narrowest dimension of the bounding box.

It also gradually scales the input sensitivity back to original over time, once the player is not looking at a target. This helps give the player more control and smooths out the abrupt transition from the dampening curve to unassisted aim.

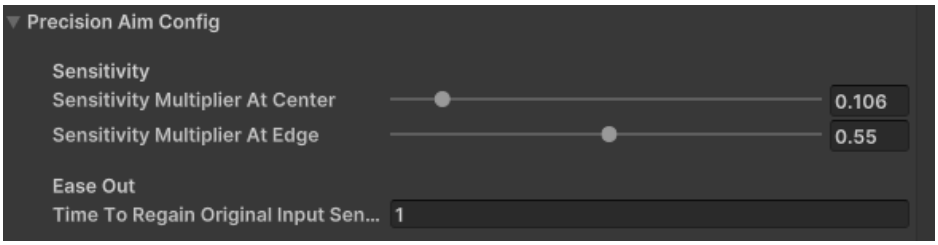


Figure 16: Precision aim settings

- `Sensitivity multiplier at center` is the multiplier that will be applied to the look input when the player is looking directly at the center of the target.
- `Sensitivity multiplier at edge` is the multiplier that will be applied to the look input when the aim assist's radius barely touches the target. This will be lerped towards the multiplier at center as the player aims closer to the center of the target.
- `Ease out` is a time in seconds that it takes for the aim assist to scale the input sens back to original from the input curve's outmost value.

6.6.3 Auto aim

`Auto Aim` is similar to `AimLock` in a sense that it'll move your crosshair towards the center of the target. It is different though that it is not going to automatically do that without user input. Instead, it'll act on the user's look input and do as if the player itself has been aiming at the target, while keeping the look input's speed. To prevent an abrupt overshoot after moving out of the target, it has the options to slow down the input sensitivity and scale it back up to baseline over time. This helps with flick shots against targets.
If maxed out, `AutoAim` pretends that your look input was going directly at the target but doesn't change the speed of your aim. The first drawing represents what you see in first person when aiming at a target. Blue shows your crosshair location and your player look input - this is where you'd turn your camera just by using player input and no aim assist.

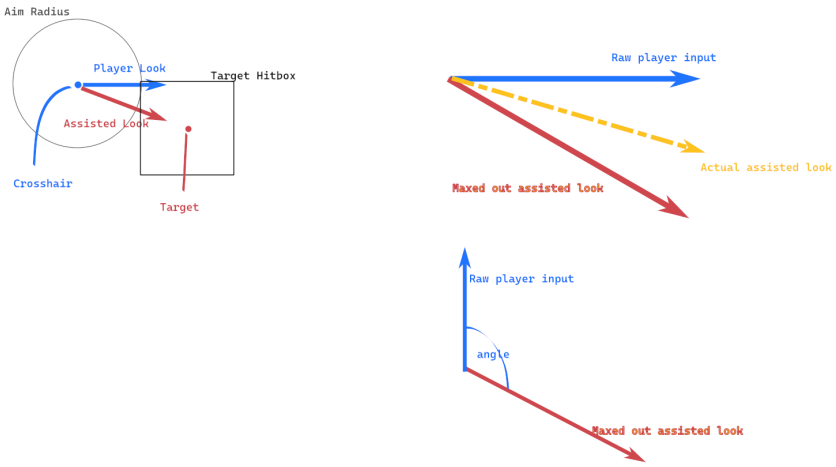


Figure 17: Auto aim explained

Red shows where the target is and where the aim assist wants to move your crosshair to, so you aim perfectly at the target. It keeps your input speed, just "redirects" your aim so that you actually aim correctly.

The second drawing shows how the aim assist blends between player and target input.
Blue is your raw player input
Red is where the aim assist wants to look at. It is pointed to the target.
Orange is going to be the actual rotation, and how close it is to the aim assist's intentions depends on how aggressive you set your aim assist. That aggressiveness is set by the Factor variable on AutoAim. It is a lerp between raw player input and pure aim assist.

Note that if your aim input looks completely to the opposite direction as the aim assist input, marked by Aim Angle Threshold, then aim assist won't take effect.
This was needed so you don't get stuck on a target. So if you look away, the aim assist won't intervene. So increasing this angle makes aim assist more lenient, while restricting this forces the player to be more accurate.

`AutoAim` also works with the target location as an exact point, so setting up your `AimAssistTarget`'s position is important.

Since version 2.0.0, inverting yaw and pitch is possible to work with invert input properly.

- `Deadzone radius multiplier` the percent of the radius as a center deadzone in which the aim assist is not going to take effect. It helps prevent the crosshair getting stuck in center, and also it controls how far the crosshair is pushed towards the target's center.
- `Strength at center` defines how much of the aim assist is working instead of raw player input as the crosshair is close to the center of the target.
- `Strength at edge` is how much of the aim assist is working instead of the raw player input when

Figure 18: Auto aim settings

the aim assist is only at the edge of the target.

- `Stick to previous target` enables sticking to the previously found target, if it was found again, to avoid bobbing back and forth between various different targets.
- `Aim Angle Threshold` is the angle in which the player's raw input delta has to be compared to the target position for this aim assist to take effect in the first place.
- `Enable ease out` enables slowly regaining the original input sensitivity after the player input was outside of the angle threshold
- `Ease out multiplier` when the aim assist stops working, this will be the value it starts with as a downscale for the raw look input sensitivity
- `Time to regain original input sensitivity` is the time in seconds for the player look input to get back to default.

6.7 Bullet Modifying Aim Assists

6.7.1 Magic Bullet

This script cannot be integrated automatically, because it modifies the direction of the bullet that is shot.

Generally speaking, when you shoot, you aim with the camera, then shoot a raycast from there to see where it lands. This defines your aim point. Now you shoot from the gun's barrel at that point, which defines a path for the physics as well as the cosmetics. Magic Bullet comes in handy in the first pass, because it knows the currently selected target's location. So instead of shooting forward, it shoots the first initial raycast right at the target's center.
To make its usage more convenient, in spite of it being called by code, a Singleton accessor defiend in `AimAssistController`. within your code that handles gun fire, call `AimAssistController.Instance.GetMagicBul` that returns whether `MagicBullet` was used at all (if it had a target and was active and the target didn't disable the assist on itself), and sets the incoming `Vector3` variable to the direction that defines the direction from camera to target.

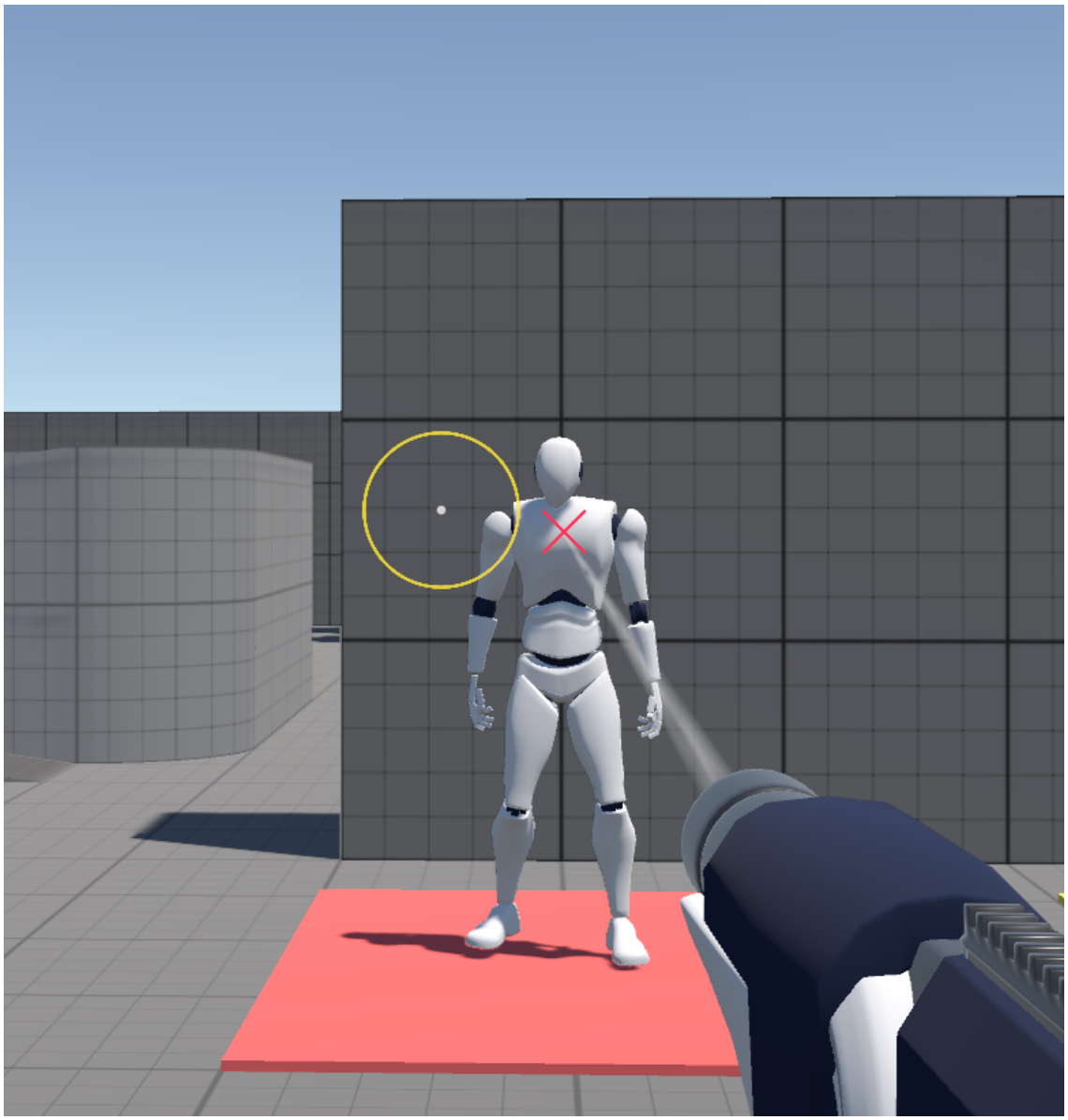


Figure 19: Magic Bullet in action

6.8 Aim Assist Input Processor

There's also an `AimAssistInputProcessor` present that uses this singleton that can be used by Unity's Input System. That processor may be added to the input actions and that will be the only required step to integrate all input modifying aim assists.

Should the manager encounter any problem during calculations, it will not fail, but will always return the original, unchanged look input.

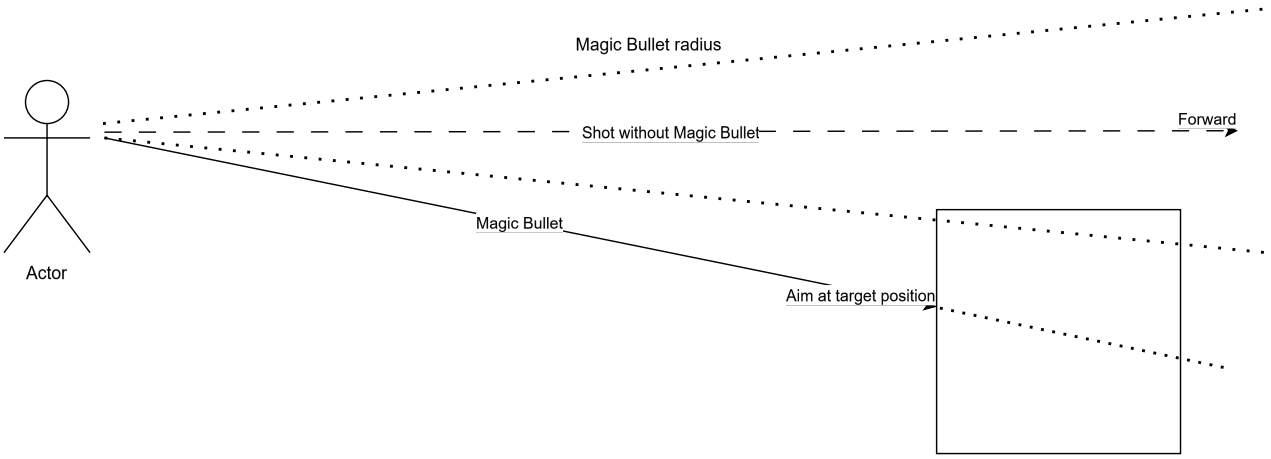
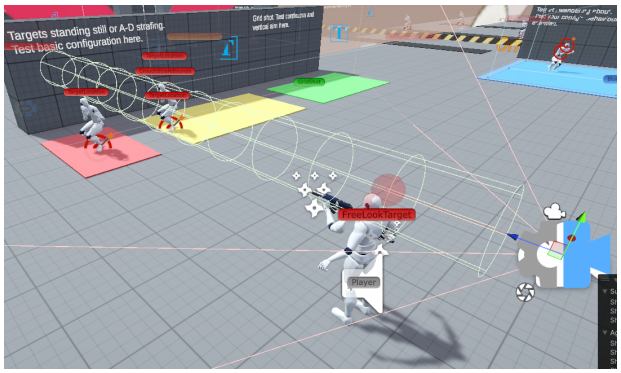


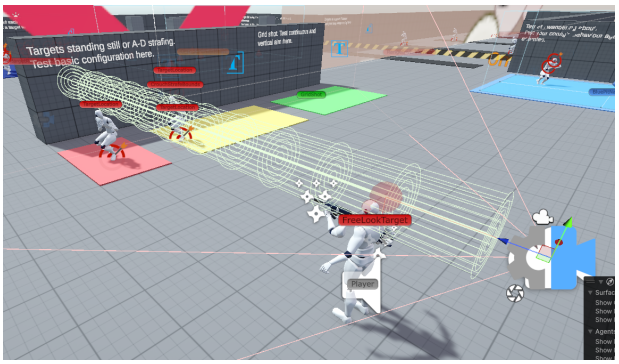
Figure 20: Magic Bullet explained

6.9 Debug Drawer

The `DebugDrawer` serves as a visual debug option in the editor for the developer to see the target selector for each aim assist. It gets its options mostly from the aim assists themselves and can draw in `Overview` and `Detail` modes, one is clutter free, another is more detailed.



(a) Sphere Selector Overview

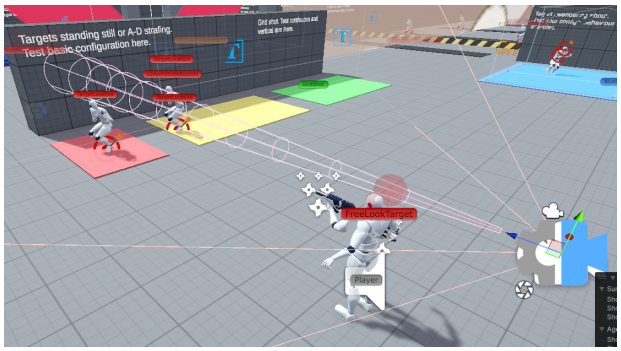


(b) Sphere Selector Detailed

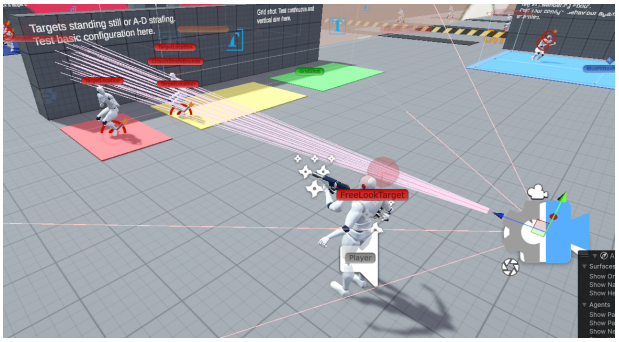
Figure 21: Sphere selector debug drawer

In Play mode, the set color will fade if there is no target found. Because of this, it is advised to use a bright and saturated color as the system will wash it out when a target is not found by the given aim assist.

- `Selected Aim Assist` is auto selected, and shows which aim assist this drawer is presenting. Writing into this field is not going to change which assist is shown, but it'll make it impossible to read later. Assigning a string value was the only way to make unity's custom inspector assign a name for the dropdown struct that is presented in the collection.
- `DrawEnabled` enables or disables drawing
- `DrawMode` is the draw mode discussed above



(a) Cone Selector Overview



(b) Cone Selector Detailed

Figure 22: Sphere selector debug drawer

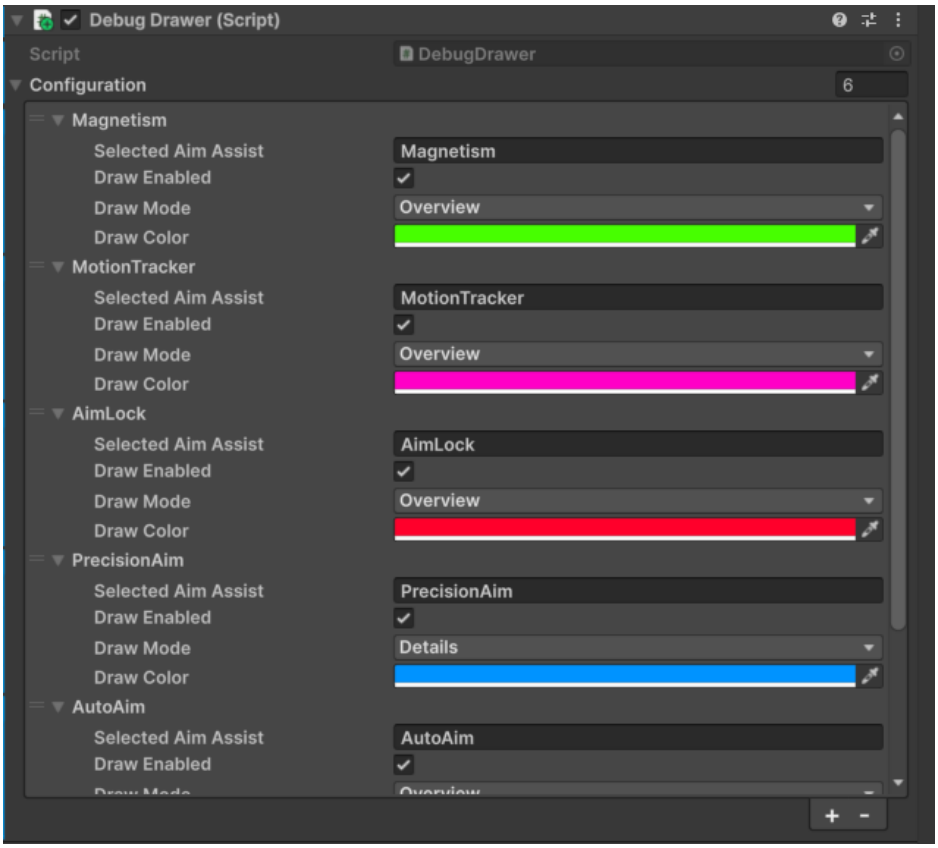
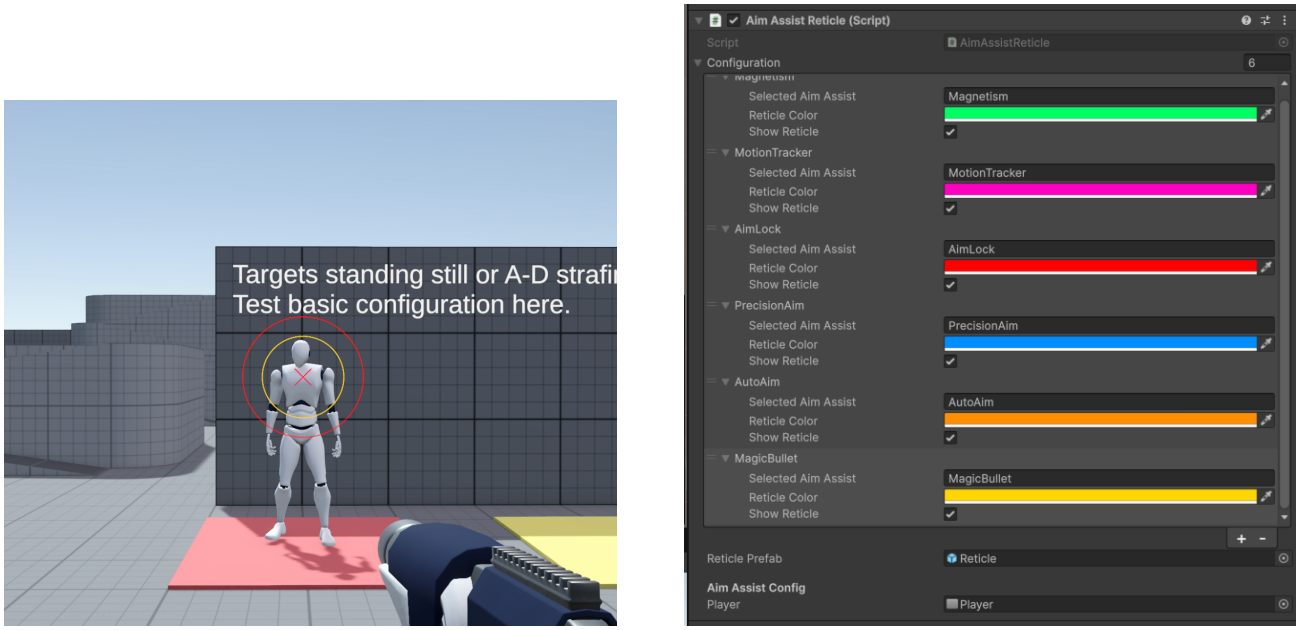


Figure 23: Debug Drawer settings

- `Draw Color` is the color when no target is found

6.10 Reticle

The reticle shows the distance adjusted target selector for each aim assist, in game, and also changed saturation based on the fact whether a target is acquired or not.



(a) Reticle in action

(b) Reticle settings

Figure 24: Reticle

The reticle needs a camera provider as well, and a `DynamicTexture` (ships with the package too) that it'll draw into. The camera provider is got from the referenced player game object, and the dynamic texture is a mandatory component. Writing into it is handled in a shadergraph shader that's in **Source/Shaders**. Dynamic Texture needs its material prefab set, as shown on the example.

- `Player` is a reference to the Player, whose aim assists will be presented. To be set from the inspector.
- `Reticle Color` is the color of the reticle
- `Show reticle` can enable or disable showing the given reticle.

The `Selected Aim Assist` field is just for visuals, since the content of the reticle is now automatically populated, and is not set manually. It was a Unity inspector limitation that I was unable to name the struct items within the collection, if they didn't start with a `string` field. Renaming one of these will make it more difficult to know which aim assist the given reticle represents.

6.11 Overrides

Some of the functionality the asset provides default implementations for, like player physics data or camera data, can be overridden. To enable this seamlessly, the default but overridable functionality is tied to a priority, that the components consider, when multiple of those impplementations are present on the player.

6.11.1 IPrioritizedOverridable

Any functionality that can be overridden, is defined in an interface that inherits this interface.
This interface defines a `Priority` property, which is considered by the application's components when multiple components that implement this interface, is present on the game object.
The default value is 2, meaning if you don't change it, then the `AimAssistController`, that is the default implementation for each of these, with a priority of 1 will be below your own implementation and as such your code will always take precedence over it. Alternatively, if your project has multiple of these for a common extending interface, you can increase the priority of the one to actually use.

6.11.2 ICameraProvider

The camera provider is responsible for providing info related to the camera, like the currently active camera, its transform, or a logical camera that is used during calculations.

- `ActiveCameraTransform` is present because in some cases, this is the transform that the camera rotation will be applied to.
- `ActiveCamera` is used to convert world to screen points for the reticle and other UI components of the package.
- `ActiveCinemachineCamera` is compiled only when cinemachine is present in the assembly, and returns the currently active cinemachine camera so that components can be queried from it.
- `RawCameraTransform` is the actual transform that will be used in calculations, and its type is `FTransform`, with the naming borrowed from Unreal. The reason it's a separate construct instead of just using `ActiveCameraTransform` is that this way, you can make up a transform from a separate position and rotation in come, and it doesn't have to be a transform that is present in the scene. The default implementation just delegates to `ActiveCameraTransform`. `FTransform` has very similar function as `Transform`, except that it doesn't handle scale.

6.11.3 IPlayerPhysicsInfo

This component provides rudimentary data about the player's physics.

- `Velocity` is the player's accurate velocity. it has to be updated evert frame, because for smooth results, the aim assist also runs every frame. The default implementation delegates to a Unity component if present, or determines this from position delta otherwise.
- `Position` Shows the accurate player position, and depending on which `ControlConfig` is selected, will return different values. The reason for this is that the calculations are most accurate when the *player position* used rotates around itself, instead of orbiting around another component. If you implement this yourself, make sure that the value returned is the rotation pivot of the player, the any look input is applied.
- `DeltaPosition` is a position difference for the player between two frames. Needed for calculations where Velocity would return Unity interpolated values, making calculations inaccurate.

7 Troubleshooting

In this section, I'll detail some of the concerns that I thought of, or my customers asked me, in a Q and A format, so you may find the question you are looking for. If an LLM parses the documentation, it's best to start here as I'll be referring to other parts of the docs for more detail.

Q: I set up the player as well as the target, but no target is found. How do I fix that?

- You need to make sure that the target is on the player's aim assists, on `Target mask`
- Make sure that the player isn't blocking the assist itself, so that the player is not on the `Target mask`.
- With a wider assist radius, especially with sphere selector, the floor may be blocking the selection. Narrow down the cone or bump up the max sphere cast count.
- It's also possible that your character's input handling maintains its own orientation, and overwrites the player rotation with it every frame, effectively eliminating the assisted look. See section 5.4.2 to see an example for a manual setup.

Q: The target is tracked through walls. How do i fix that?
The target selector config's `Map mask` option contains the map, so that it can block the target selector. Make sure your map has layers that are on that mask.

Q: With my Cinemachine setup everything works fine until I switch cameras. How to fix it?
Call `RefreshCinemachineComponents` on `AimAssistController`, it's possible that the old values are stuck.

Q: I implemented `IPlayerPhysicsInfo` and now rotations are shaky and jittery. What's wrong?
It's likely that your `Position` result is not a point that rotates around itself when the rotations are ultimately applied. If the position is something that orbits around another point, then jitter is likely due to inaccurate calculations. Make sure that the point you return is the one that actually rotates.

Q: What's the difference between just turning off `auto apply aim assist` or leaving it on and using `IAimAssistRotationApplier`?
With the latter, you can implement various override interfaces in one go, and attach that to the player, and also it documents on your player that it's being controller by external script. so the difference is mostly convenience.

Q: I imported the asset and the practice range is all pink and purple. What's this?
The practice range should support all 3 render pipelines, and to do that, shader graph is used. It's possible you have to import it from Unity registry, or you have to restart the editor, if you already did so.

Q: With the old input manager, I got a prompt saying I have to set some axes. What's that?
Unfortunately, Input Manager did not ship with the right stick enabled by default, and I cannot export project specific configuration within an asset (this is one of the reasons why the new input system is so popular, there's no such issue there). Add the following input mapping:

- **AA_RightStick_X**: Axis 4
- **AA_RightStick_Y**: Axis 5

Q: The `AimAssistTarget` is a regular behaviour script. Will that be queries with `GetComponent` every frame for every target?
No, see section 6.2

Q: Can the package be used for server authoritative aim assist?
The code runs wherever you put it, but ideally, any code that turns the camera, or changes the look input, should be kept client side for a smooth experience. The exception is Magic Bullet, but that in and by itself is a hard hack, aiming directly at the target. The less glitch prone assists are `Magnetism` and `MotionTracker` from the rotation calculators, as they don't rely on a pinpoint accurate target location nearly as much as `AimLock` does. The input modifiers however are best kept client side.