Aim Assist Pro Documentation

April 2025

version 2.0.4

Contents

1 Greetings

Thank you for downloading my asset Aim Assist Pro. The asset intends to help developers of the shooter genre enhance their games' player experience with an assisted aiming system, helping the player feel more powerful and less stressed about practicing fine adjustments while still feeling in complete control over their aim. The system helps developers serve their players with better gameplay and enemies instead of making the AI stand still so they can be actually hit.
The documentation contains all the information needed to integrate Aim Assist Pro into your project.

If the asset has been useful to your project, please consider giving it a good review to help others find it.

If you found that the asset is lacking in some areas, do not hesitate to drop me an email with some feedback.

2 About The Asset

2.1 Quick Overview

The asset consist of C# scripts that calculate rotation / pitch addition on top of your player's input. It is also equipped with a practice range so you could try out the settings and tailor them to your game's profile.
The asset is intended for the shooter genre, especially for controllers to help players aim. See the general descriptions for the type of aim assists below to get a feeling of how they would serve you to take your FPS game's experience to the next level. The documentation also contains considerations that were made to make the asset perform better by reducing wasteful calls to Unity's API.

2.2 Key Features

This short section will list out the package's key features, so you do not miss out on one that's needed for your project and you don't use it because it isn't communicated properly. Each of these will be detailed in this document in a later section.

- Provides the system with the currently in-use camera, this component can be extended
- Supports layers for aim assist targets and the map separately
- AimAssistTarget components are cached and only queried from their own layer (boost performance)
- AimAssistTargets can define their 'position' multiple ways
- AimAssistTarget components can overwrite and use their own unique aim assist configs on a per-instance basis
- Magnetism: compensate against player movement with counter-strafing in mind
- AimLock: An aimbot that rotates towards the center of the target

- MotionTracker: compensates against the enemy movement
- PrecisionAim: Slow look input around the target with gradual regain (prevent abrupt popping back)
- AutoAim: Gravitate the player look input towards the target, as if the player had a perfect input
- AimEaseIn: prevent unwanted diagonal aim
- MagicBullet: calculates the direction towards the target when shooting
- TargetSelection using spheres of progressively smaller size: useful around cover
- TargetSelection with a cone (raycasts with optional sphere casts in a cone shape)
- Events that trigger when a target is found or lost
- Integration with Cinemachine 2 and 3
- Interpolation of aim assist rotation for a smoother but less accurate experience if needed
- Reticles that change color if a target is found
- Options to override camera transforms (player point of view), and camera controls (what the script rotate with the result)
- Debug drawers that show the target selection in the scene, with a low-clutter overview and a detailed drawing mode
- Automatic integration for rotations (Magnetism, AimLock, MotionTracker), if the character controls allow it
- Input Processor for Input System that automatically handles the Input Modifying assists (PrecisionAIm, AutoAim, AimEaseIn)
- Custom interactive editor to equip a GameObject with aim assist
- Custom interactive editor to make a GameObject an aim assist target
- Editor-only log messages that warn if an aim assist cannot work, and detail the cause
- Practice Range with in-game aim assist settings UI to tweak configuration and find what works best

2.3 Supported Input Systems

The asset works by calculating the necessary rotations for turn or pitch, or scaling the magnitude of the player input. The outputs are either the changed input values or rotation adjustments in degrees for the sideways turn and the pitch, respectively. The asset itself **does not** contain input handling, but the test scenes contain examples.

Depending on your input setup, pick the appropriate test scene based on which folder it is in - the ones requiring input manager or input system will be named accordingly.
As of v2.0.4, the old Input Manager is also supported and there is a test scene for it. The new input system has various test scenes and is fully supported since an earlier version,

including an InputProcessor that makes integration a lot easier. An example of manual integration will be presented in this documentation later.

the code was written **using preprocessor guards, so the code shouldn't break** when the package is imported into a project that uses either the old input manager, or the new input system.

2.4 Supported Camera Control Solutions

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
- CinemachinePOV
- CinemachineHardLockToTarget

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

- CinemachineCamera with no Body
- CinemachineThirdPersonFollow
- CinemachineOrbitalFollow
- CinemachineHardLockToTarget

With the *LookAt* target set, Cinemachine hard tracks the given target. If the camera cam orbit freely around it, it can still somewhat be used for assisted aim, although you'll always look at the aim assist target through your character (the cine LookAt target). If you use a top-down game with your character as the LookAt target however, obviously aim assist will have no meaningful use as your camera looks towards the ground.
There's also a no-Cinemachine solution implemented that can be used for scenes with a simple camera setup. The camera selection will be detailed in a later section.

2.5 Demo Scenes

If you use the new Input System, but do not have it enabled yet, Unity will prompt to import the package. Click 'yes' which will restart the editor, so you'll have to import the package again. After that, several test scenes are available in **Scenes** folder and its subfolders. There are separate scenes for Cinemachine 2, 3 and no cinemachine in their respective folders, so open the appropriate scene.

the scene without Cinemachine shows an example on using a multi-camera rig with overwrites: The first and third person cameras have their own control transform in pair and they are set as a Multi override.
The Practice Range has the following utility:

- A standing still target
- An A-D strafing target
- A simple GridShot that can be configured from the inspector
- Wandering targets that aimlessly go on within their dedicated space - this uses NavMesh
- A projectile shooter that shoots boxes
- An A-D strafing crouch spamming target
- Targets around cover
- A target that wanders but when it sees the player within its range, it follows the player.
- Reticles for all aim assists that use a target selector
- Debug Drawers to show the target selectors in edit mode
- UI to customize aim assist settings on the run

Test scenes are included using Unity's Input System.

2.6 Supported Character Control Methods

The asset works with either `RigidBody`, `NavMeshAgent`, `CharacterController` or a barebone `Transform`. Again, detailing these will be discussed in the appropriate subsection of section Notable Components.

2.7 Enabling the Input Method for your Project

Test scenes are implemented using the new Input System. Unity should prompt you to enable the new Input System when you import Aim Assist Pro. In case it doesn't, follow the steps below to enable it.

While all test scenes require input system at this time, the code was written using preprocessor guards, so the code shouldn't break when the package is imported into a project that uses the old input manager.

1. Go to File - Build Settings - Player Settings
2. Scroll down to configuration
3. Under Active Input Handling, select `Input System Package (new)`
4. Restart your editor.

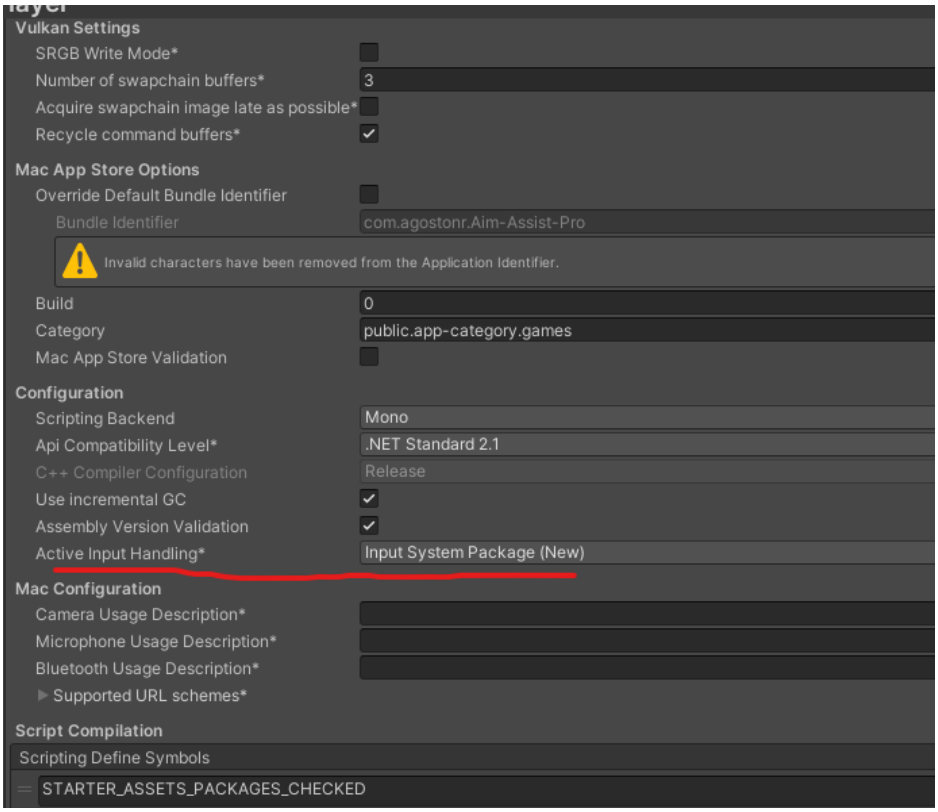


Figure 1: Enable the new input system

3 High Level Overview

This section will details how the package's component connect and work together. If your project requires you to tailor the asset to your needs, start here.
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.

1. Each `AimAssistBase` behaviour has its own separate `TargetSelector` that it uses to find an aim assist target. These TargetSelectors need to know the player's viewpoint, the currently active camera. This camera transform is provided by `AimAssistCameraProvider`, a mandatory component.

2. Using the camera viewport location, the TargetSelector finds a target (detailed later).

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

   (a) Rotational Aim Assists like Magnetism, AimLock or MotionTracker, that generate a `Rotation`, send their results to the `Rotational Aim Assist Manager`, a mandatory component for these aim assists, to summarize the deltas and return a total delta rotation for the given frame.

   (b) Input Modifying Aim Assist like PrecisionAim, AutoAim and AimEaseIn generate a modified *look input delta* that will be used as the player look input.

4. The calculation are then applied -

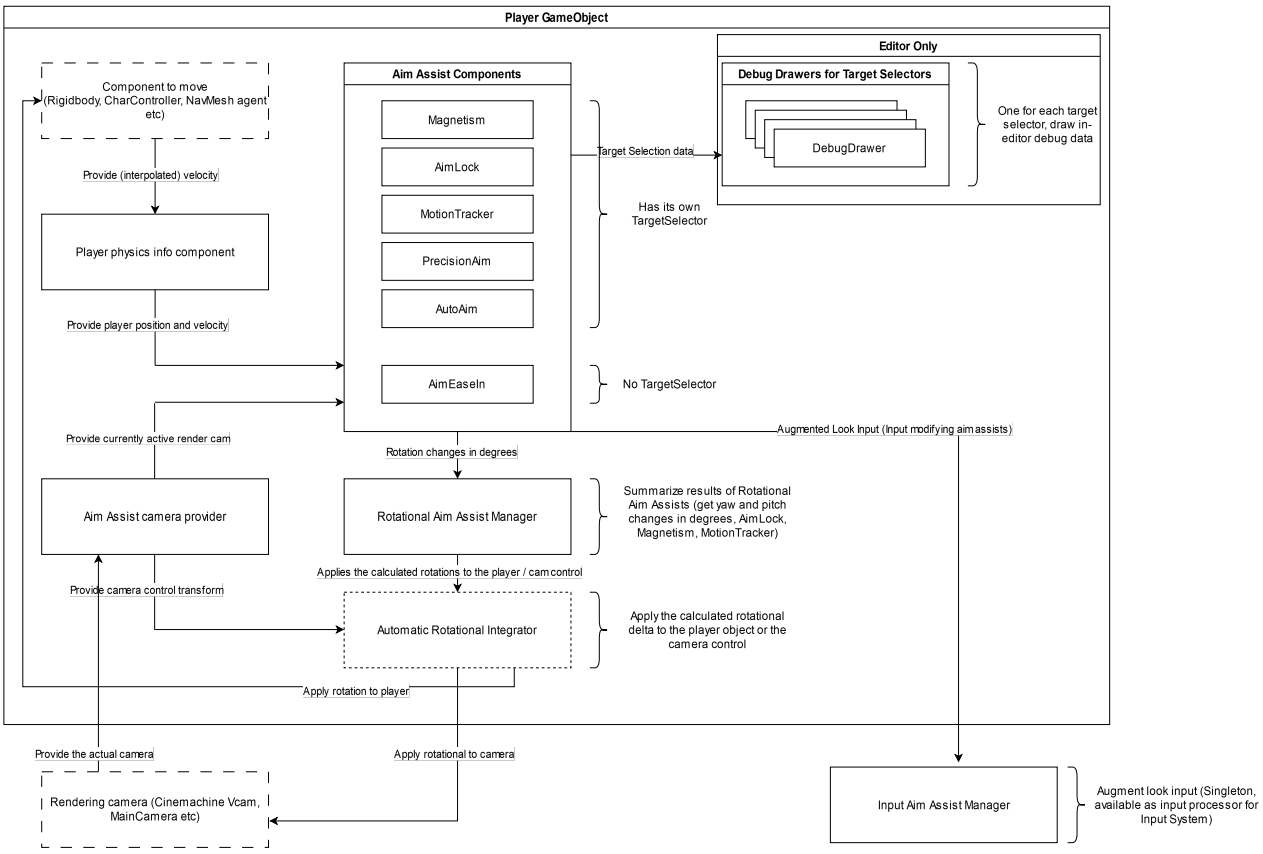


Figure 2: Aim assist components diagram

(a) `Automatic Rotational Integrator` uses the Rotational Aim Assist Manager's result and applies the rotation deltas to either the player body, or the camera control that is returned by the `AimAssistCameraProvider`. This component is optional as there are use cases it cannot be used, for example a third party component handling the camera orientation, that overwrites any external value coming from outside. More on that later in the integration section.

(b) `Input Aim Assist Manager` singleton is used to modify the look input using a simple method, and the end result can be used either through the `AimAssistInputProcessor` that is added as an input processor to any Input System input action, or used manually in your code that handles the look input.

4 Notable Components

As mentioned before, all aim assist types contain calculations that adjust the rotation and pitch of the player to get their aim closer to the target. See some considerations below to better understand how the assisting code works.

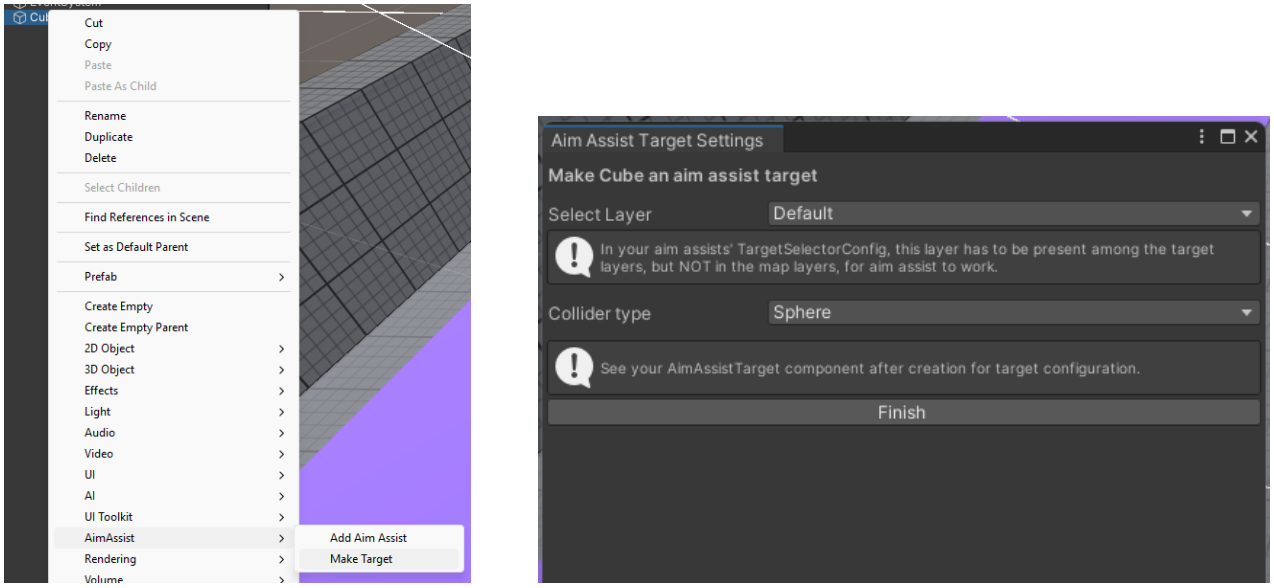
4.1 AimAssistTarget

`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**



(a) Context menu (b) Dialog

Figure 3: Aim assist target context menu and setup

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 gameobject
- **Collision Center** means the center of the collision geometry of the target gameobject
- **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`.

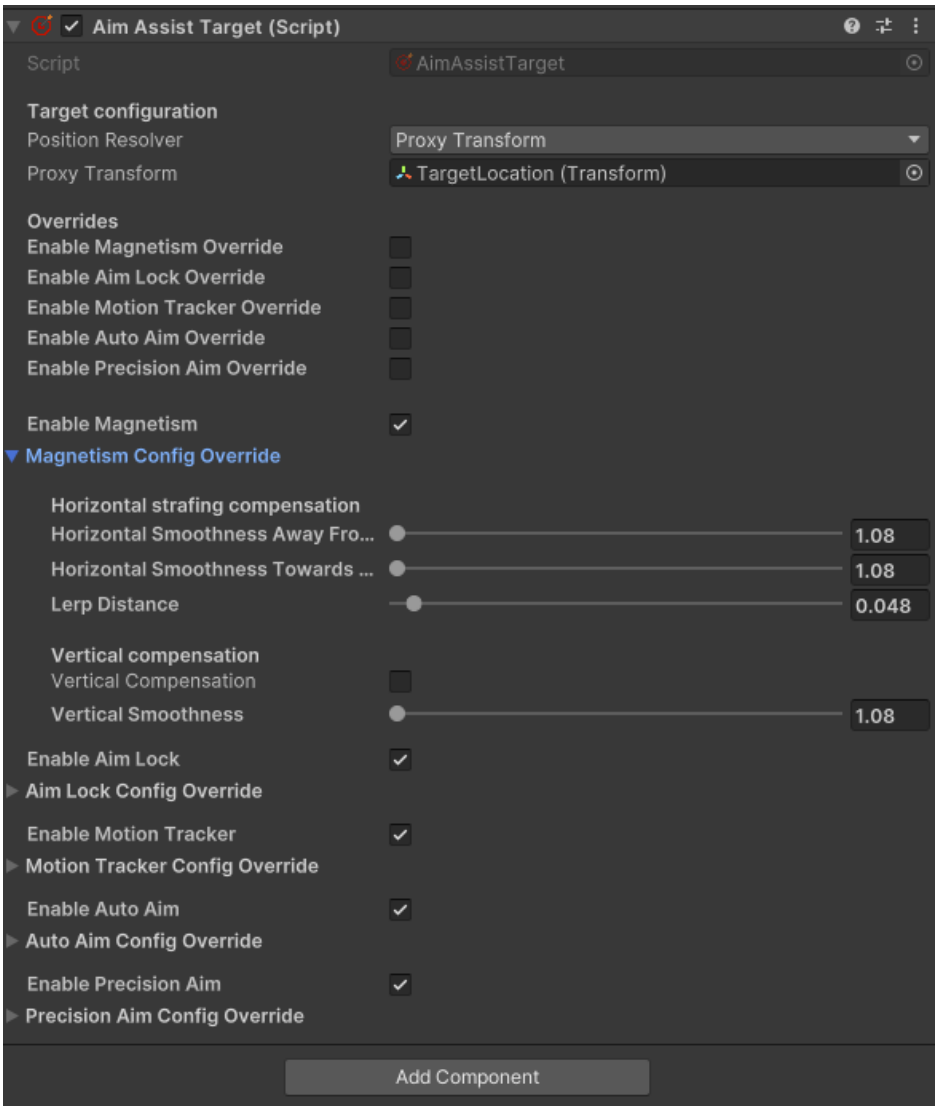


Figure 4: Configuration for the aim assist target

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.
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 not target script, then even after assigning the script to it runtime, the system won't recognize this gameobject 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`.
- 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.

4.2 Camera Provider

A `Camera Provider` is a component that provides the cameras that are the player's currently active viewing camera (to be used as a 'player eye' for the aim assist calculations), and controls to actually implement the changes by the automatic integrator.
If no camera override is set, Cinemachine is not in the assembly or is not present in the scene, then it will default to `Camera.main`.
The built-in component for the project is `AimAssistCameraProvider`, but if this doesn't suit your needs, you can use its base class `CameraProviderBase` that exposes the following properties:

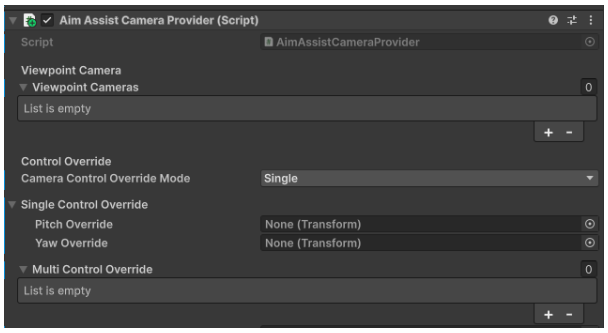
- For Cinemachine 3:
  - CinemachineBrainEvents - in Cine 3 events are exposed on the cine brain and one of them is dedicated to fire when the brain has been updated. It's an option for the automatic rotational integrator to use this for updates, instead of LateUpdate
  - CinemachinePanTilt - in Cine 3 this is a regular component. Can be null.
  - CinemachineOrbitalFollow - in Cine 3 this is a regular component. Can be null.
- For either Cinemachine 2 or 3:
  - ActiveCinemachineCamera - the currently active cinemachine camera
- ActiveCameraTransform - the transform of the currently active camera
- ActiveCamera - the currently active and enabled camera's Camera component
- CameraControlPitch - the transform whose localRotation's Pitch will be adjusted with the rotational aim assist calculations
- CameraControlYaw - the transform whose rotation's Yaw will be adjusted with the rotational aim assist calculations

See the component's settings:
There's a *camera transform override* that, when set, will be the transform that represents the camera's viewpoint. There are two options for *camera control overrides*,

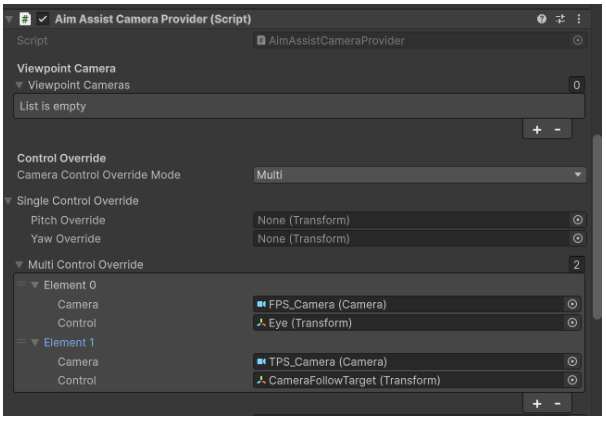
- Single: provide a separate yaw and pitch (may be the same gameobject), that will be rotated by the automatic integrator,
- Multi: provide a list of cameras with the transforms they control. If this is set, the first camera that is active and enabled, will be used from the cameras, and the transform that is associated with it will be controlled. Make sure that this transform isn't overwritten by another script.

The component settings explained are:

- `Viewport cameras`: In the abscence of Cinemachine, this list of cameras will be used as the player viewport. The first enabled and active camera will be the aim assist camera. If Cinemachine is absent, and this isn't set, `Camera.main` will be used.
- `Control Override` as explained before, the camera controls



(a) Camera Provider with single override



(b) Camera Provider with multi override

Figure 5: Camera Provider settings

- `Pitch and Yaw overrides` are only used in `Single` control override mode. These two will be sent to the Automatic Rotational Integrator to add the rotation delta to.
- `Multi Control Override` is only used in `Multi` camera mode, and works as explained before.

If this camera setup doesn't suit your project, implement `CameraProviderBase` and provide the accurate camera information.

4.3 AimAssistBase - formerly TargetSelector

The `AimAssistBase` uses the `Camera Provider` as 'player transform' to find targets in the game. In update 2.0.0, **each aim assist script has its own, dedicated target selector** that makes the system a lot more versatile than before.

- `Query Trigger Interaction` isn't directly part of the target selector config. It's present to enable or disable interacting with trigger colliders at the developer's discretion.
- `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.
- `Aim Assist Radius` is used with sphere type target selection, and is the outer spherecast's radius.
- `Max Sphere Cast Count` means the maximum number of progressively smaller spheres to shoot until a target is found, or this number is reached.
- `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.

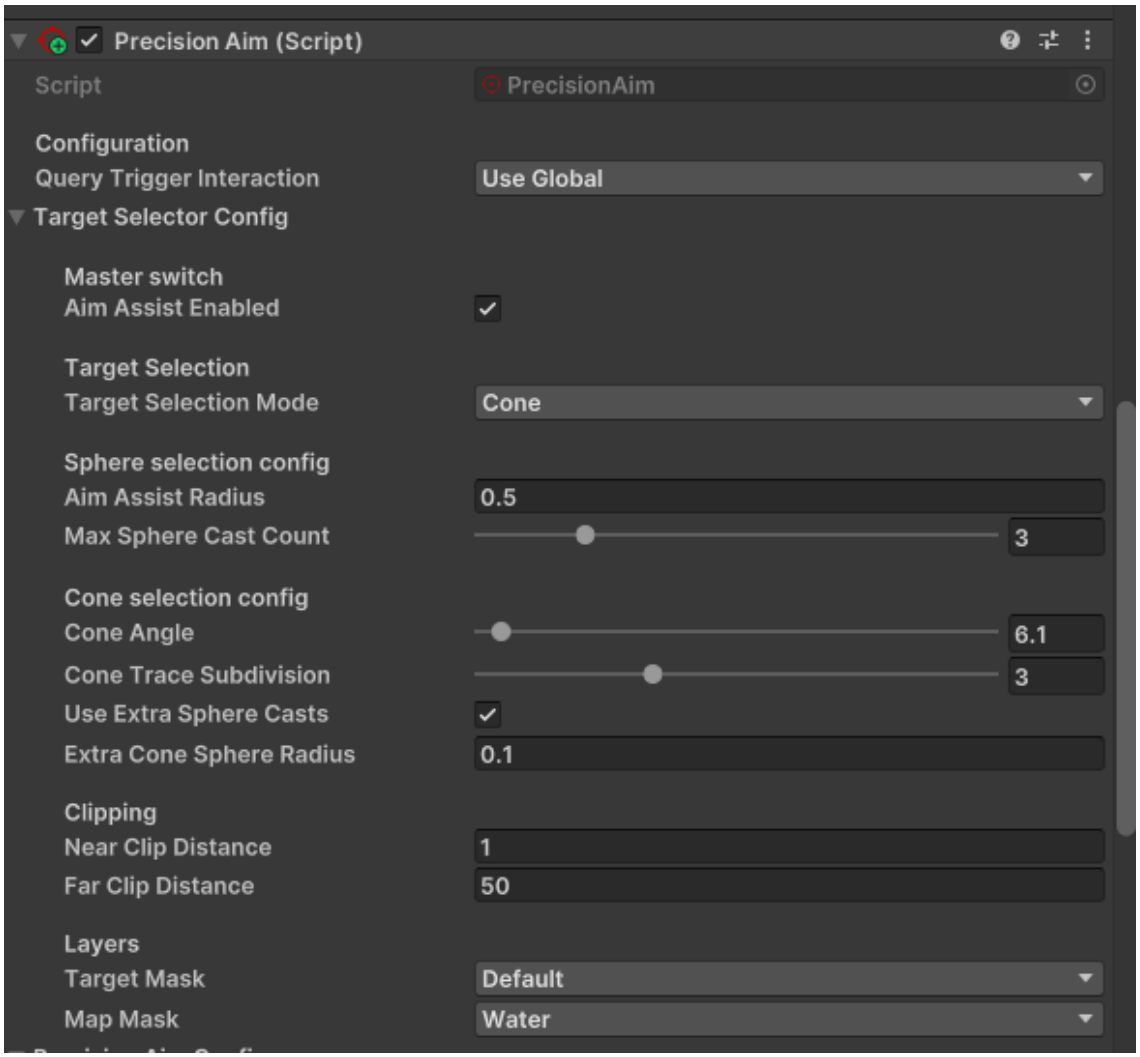
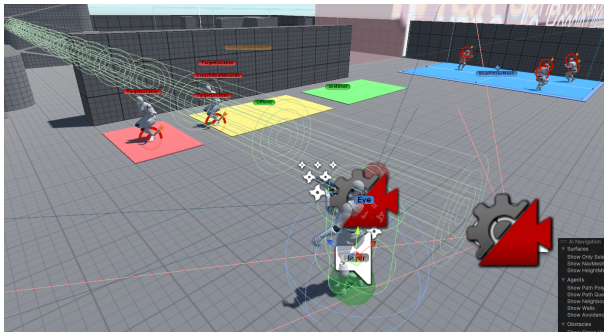


Figure 6: Target Selector settings

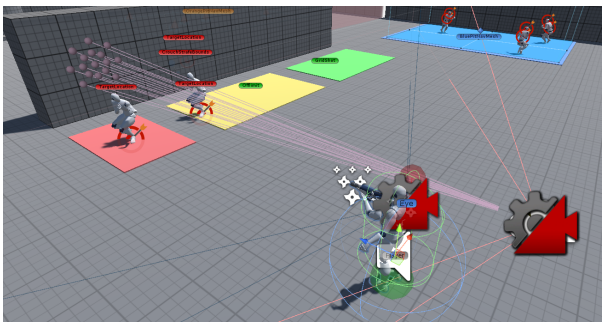
- `Use Extra Sphere Casts` will enable shooting a sphere on top of every line for the cone selector. With this enabled the selector will be less accurate but you can get away with shooting less lines
- `Extra Cone Sphere Radius` is the radius of these extra spheres
- `Near Clip Distance` is the near 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.



(a) Sphere Selection



(b) Cone Selection

Figure 7: Target Selection Modes

Using the new `DebugDrawer`'s `Detail` drawing mode, on the left you see the progressively smaller spheres of the sphere target selection, and on the right you see the lines distributed in a cone volume, with the spheres showing the size of the extra sphere casts.

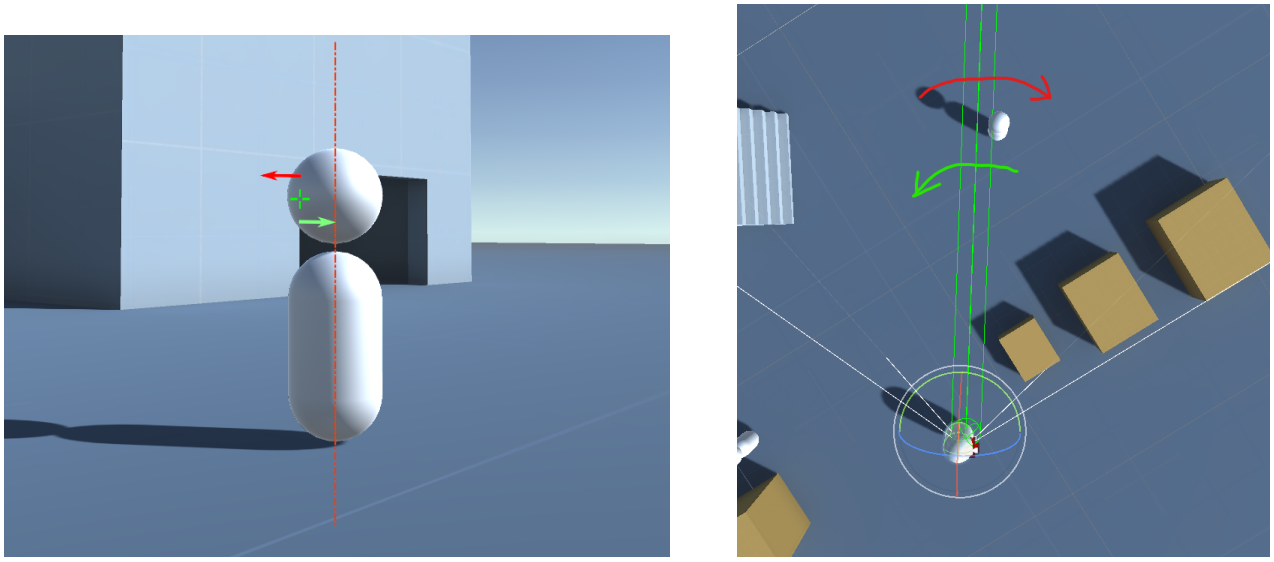
4.4 Rotational Aim Assists

Rotational aim assists rotate the player / camera control by a given degree.

A breaking change compared to previous versions is that they have their own base class now, and return a `Rotation` struct that has a `Yaw` and `Pitch` value, and can be created from quaternions or directions. `Rotation` also represents the shortest route to rotate to the target as it is normalized on creation, and it never contains a `NaN` value as it is sanitized.
All rotational assists need a `Rotational Aim Assist Manager` that will be discussed later.

4.4.1 Magnetism

Dubbed by the community of a famous shooter game, Magnetism works by smoothly compensating for the player's strafing movement to make it easier to keep track of the enemy. It returns the rotation adjustment in degrees that you add to your rotations.
The screenshots are from a previous 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.
Magnetism no longer needs a movement input delta, it reads this value from the velocity provider.



(a) Movement to and away from target

(b) Turn compensation for movement

Figure 8: 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 `PlayerPhysicsInfo` for player velocity. This component will be detailed separately.
See the settings below.

- `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.

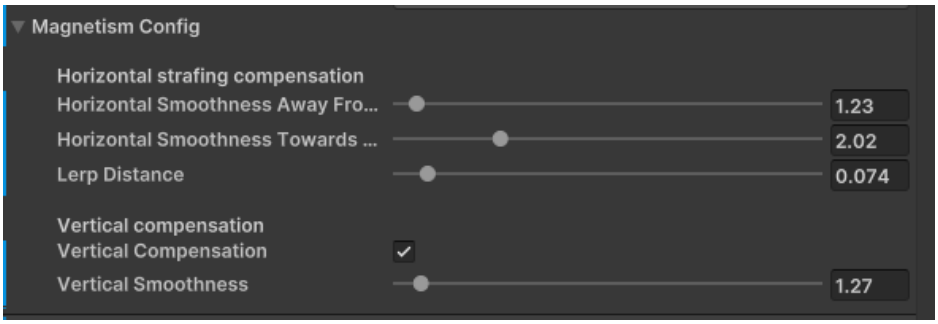


Figure 9: Magnetism settings

- `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). 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 smoothnesses 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 highground - 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.

4.4.2 Aimlock

A smooth aimlock that tracks the target for you. You can use a curve to smooth out the tracking, giving it a more natural, spring-like feel. The aimlock is going to end up aiming at the position of the target, which is why this point can be configured at the `AimAssistTarget`. It returns the adjustments that you add to your rotation in degrees.

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.

As of v2.0.4, AimLock also 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).

- `Horizontal time to aim` The time it should take in seconds for the aim assist to move from the edge of the aim assist radius to the center of the target on the horizontal axis.

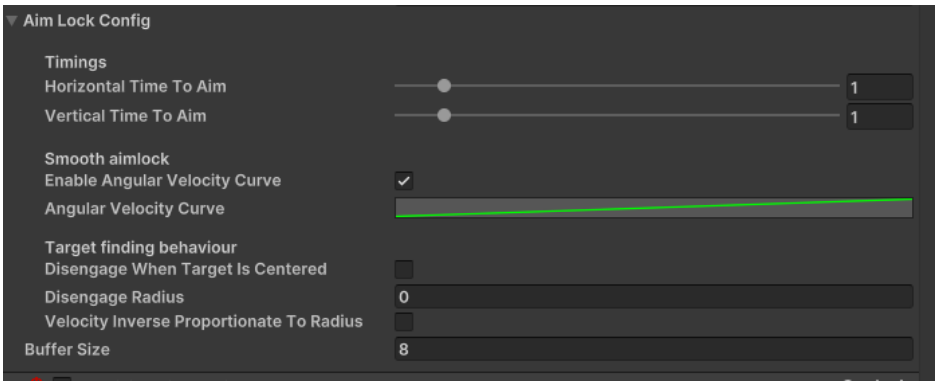


Figure 10: Aimlock settings

- `Vertical time to aim` The time it should take in seconds for the aim assist to move from the edge of the aim assist radius to the center of the target on the vertical axis.
- `Enable angular velocity curve` enables or disables the smoothing curve for the aimlock's angular velocities.
- `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. You can set any curve you prefer, though keep the following in mind for the best results. The example curve presented doesn't lock your aim at all when you're already looking at the center of the target and it also falls rapidly. This allows for some wiggle room when already aiming at the target and also prevents unnecessary stutter due to the aimlock assisting back and forth for every small adjustment.
- `Buffer Size` is the size of the buffer where the player positions are collected, and will be used to average them out. Having a high value makes for a smoother but more floaty motion and lower values are snappier but may occasionally present jitter at high velocity values with no smoothing curve.
- `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. Target movement and player movement can then be followed using Magnetism and MotionTracker. This is reset after a new target is found, or a target is lost.
- `Disengage radius` is the radius that defines how close the aim has to be to the target position to disengage.
- `Velocity inverse proportionate to radius` - Velocity used to be inversely related to the aim assist radius, meaning that a wide radius was slower than a narrow one. It was implemented to avoid an abrupt shift in camera position in FPS mode. Now the default is a consistent speed regardless of radius, and the legacy implementation can be enabled with this flag.
- `Buffer Size` - the size of the player position buffer. It is used to interpolate the player position, to reduce any possible jitter.

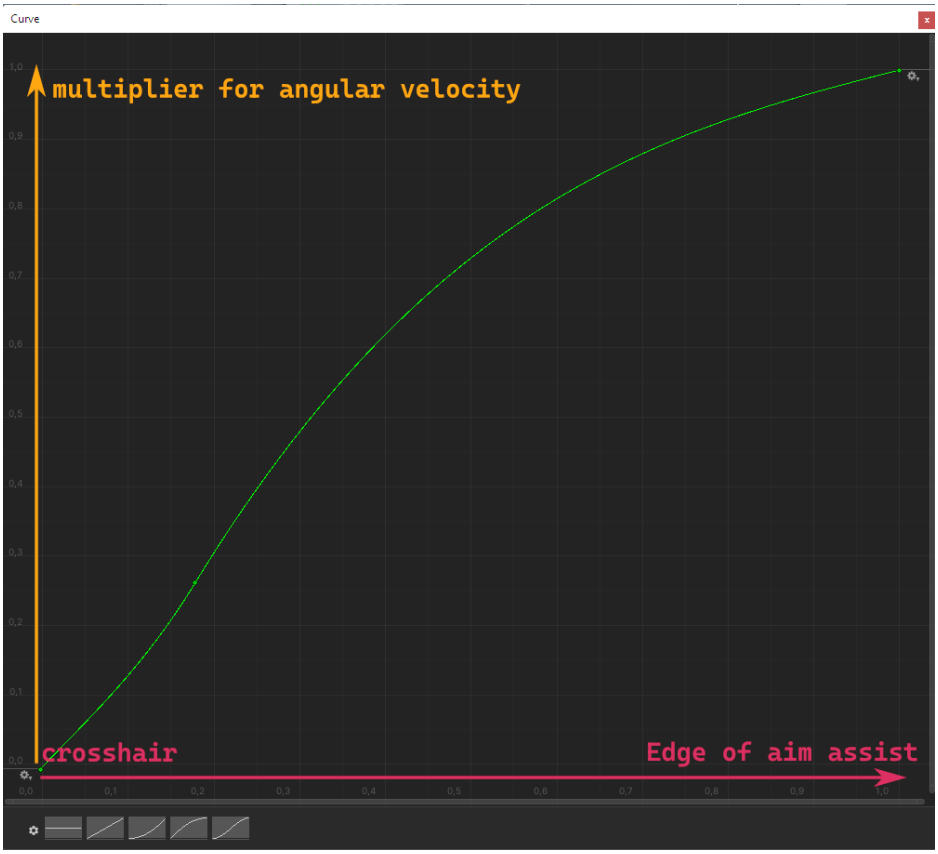


Figure 11: Aimlock angular velocity curve

4.4.3 Motion Tracker

A new rotational aim assist added in version 2.0.0 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.

It also turns itself off when the player's movement direction matches with the enemy's movement direction to enable counter-strafing. To have a smooth support for counter-strafing, use Magnetism.

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

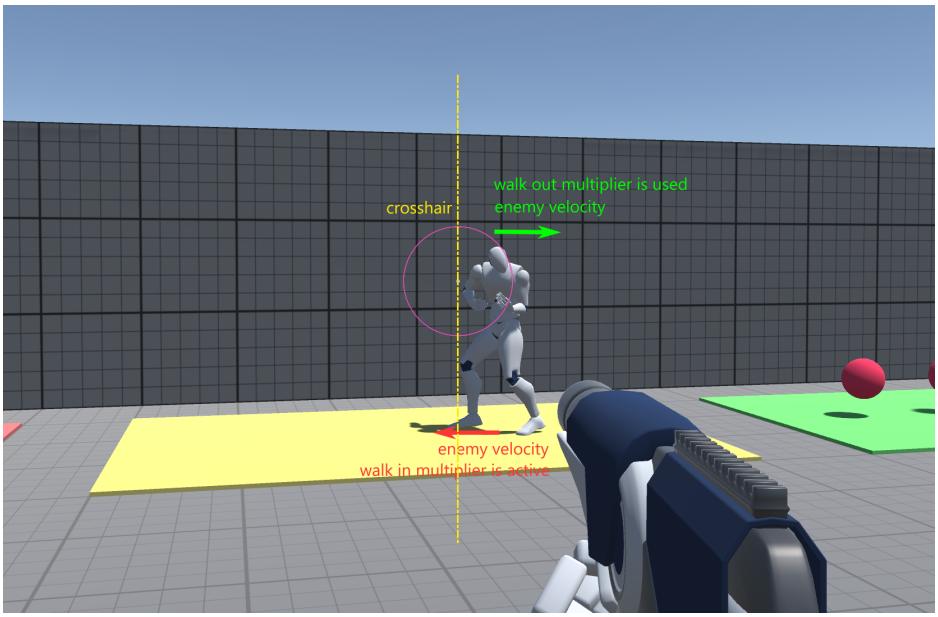


Figure 12: Motion Tracker

Figure 13: Motion Tracker settings

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.

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

4.5 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 the `Input Aim Assist Manager` that will be detailed later.

4.5.1 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.

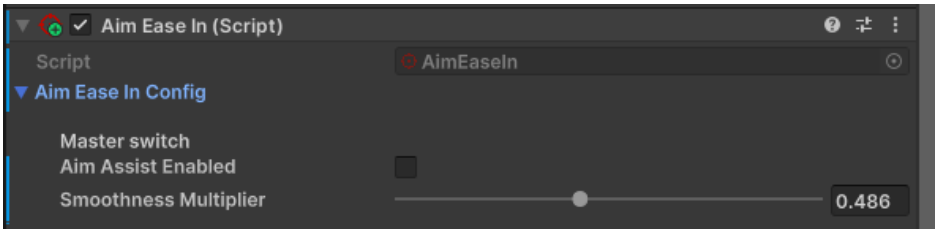


Figure 14: Aim ease in settings

- `Aim assist enabled` a master switch to enable the aim assist.
- `Smoothness multiplier` a multiplier for the less dominant axis.

4.5.2 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 its bounding sphere.

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.

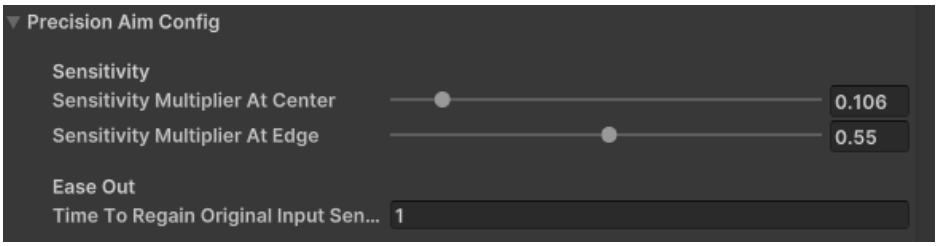


Figure 15: Precision aim settings

- `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.

4.5.3 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 flickshots against targets.

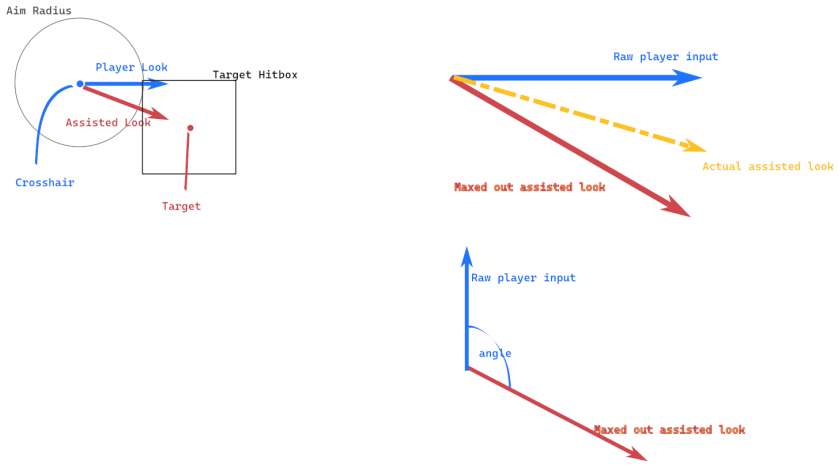


Figure 16: Auto aim explained

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.
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.

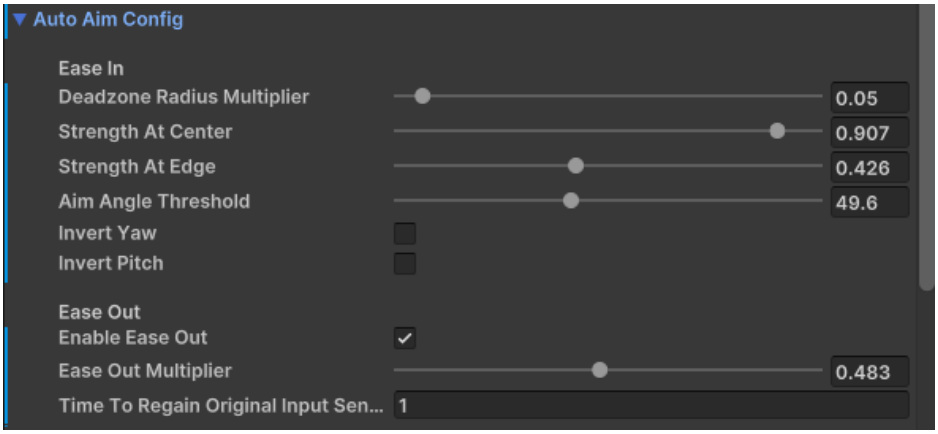


Figure 17: Auto aim settings

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

4.6 Bullet Modifying Aim Assists

4.6.1 Magic Bullet

As of v2.0.4, the asset includes a new aim assist, 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.

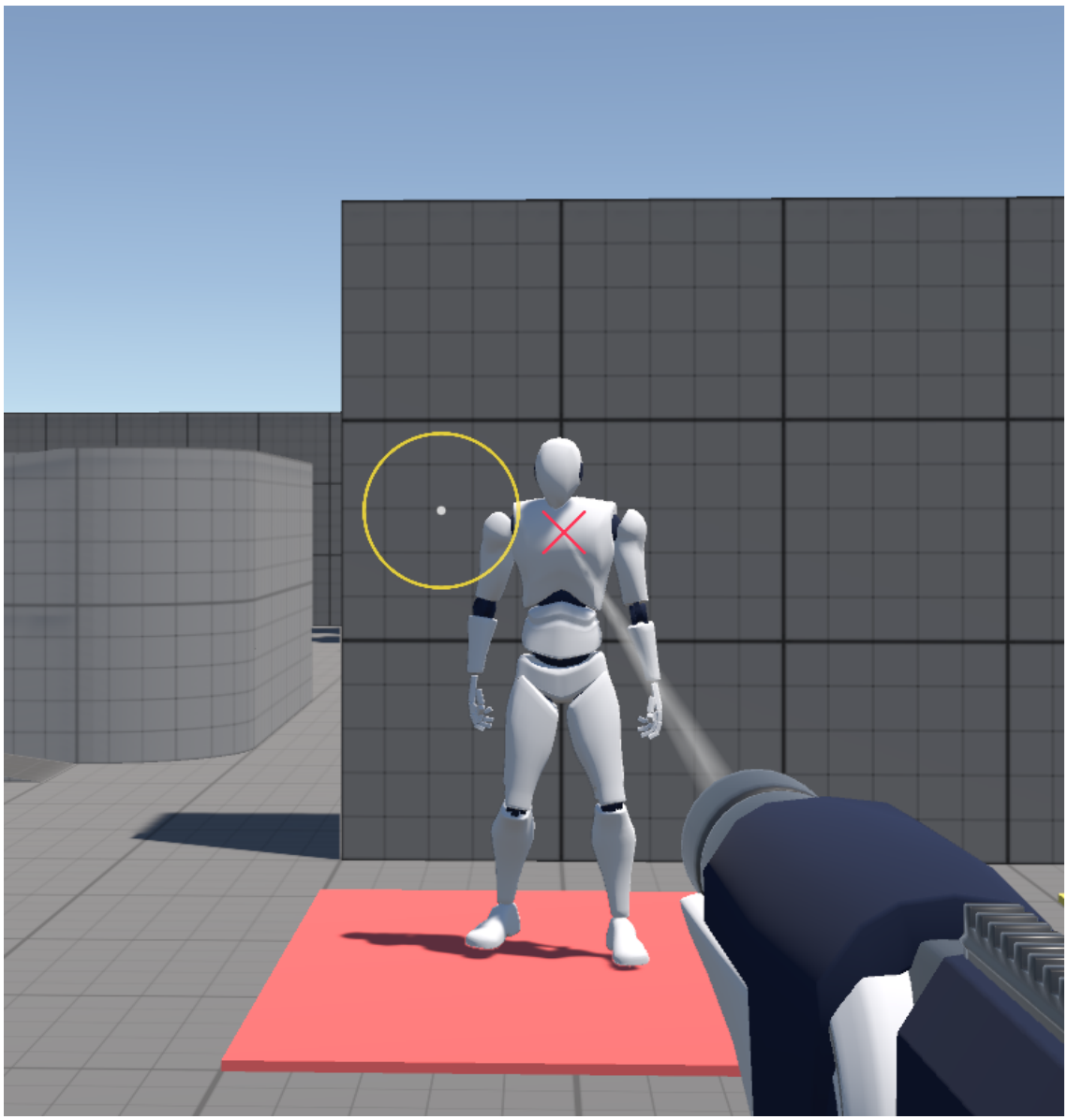


Figure 18: Magic Bullet in action

To make its usage more convenient, in spite of it being called by code, a Singleton accessor is defined, named `AimAssistedBulletManager`. More on that in the notable components section.

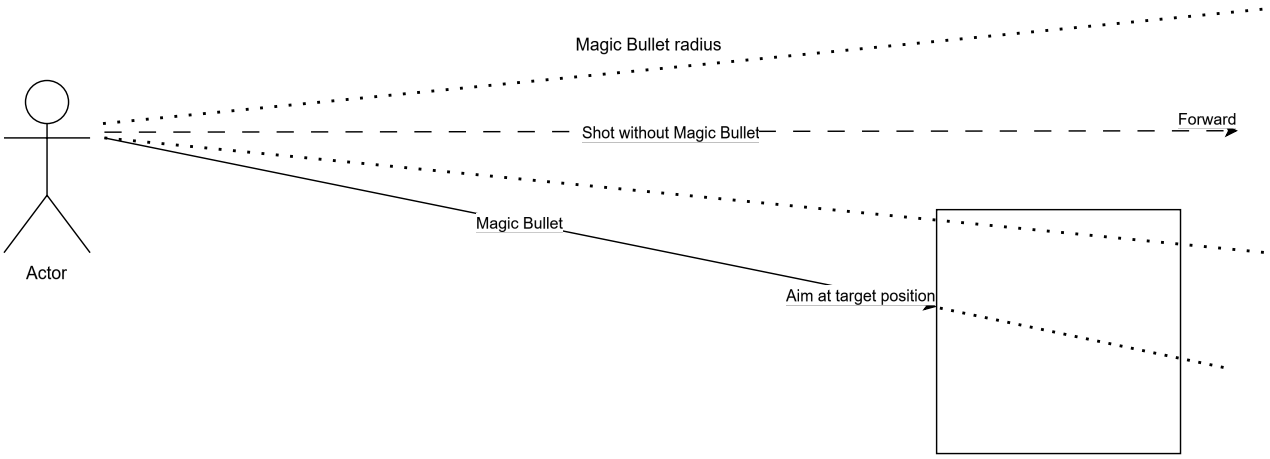


Figure 19: Magic Bullet explained

4.7 Rotational Aim Assist Manager

This component aggregates all the registered rotational aim assists and returns a `Rotation` delta, that is the total aim assist rotation for the given frame. If no target was found or no aim assist was active, this will return `Identity`, equal to `Quaternion.identity`. `Rotation` represents a total yaw and pitch different that takes a transform from the origo to the desired orientation. This delta contains the `yaw` angle in degrees that either turns the player or the camera control, and the `pitch` angle in degrees that changes the camera pitch as a local rotation.

`Rotation` always contains the shortest route to the desired rotation and its values are never `NaN`.

All Rotational aim assists are self-registered on their `Start` unity method into this manager. This means the on first frame all aim assists that are present, will be available in the rotational aim assist manager.

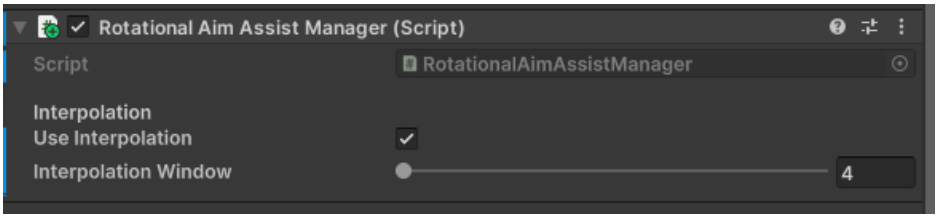


Figure 20: Rotational Aim Assist Manager Settings

It also enables interpolating the result for a smoother but more floaty experience. Exception of this is AimLock that has to be accurate due to always getting input by itself (seeking the target center and turning to it), but it has its own smoothing built-in. Magnetism and MotionTracker use this interpolation when enabled.

- `Use interpolation` enables using the interpolation, which is the average of the previous window's results.

- `Interpolation window` defines how many previous results are averaged out.

In brief, Rotational Aim Assist Manager is the component to get the rotational aim assists from so since version 2.0.0, even if manual integration is necessary, there's no need to reference the aim assists one by one.

4.8 Automatic Rotational Integrator

This is the component that uses the rotational aim assist manager and actually applies the rotations to the camera control or the player body.

It is important to note that this component can only work when the target transforms are not directly driven by another script which tends to be the case with character control frameworks or templates. In the manual integration section the documentation will detail two examples of this and shows the simple solution to integrate the results through code.
See its settings:

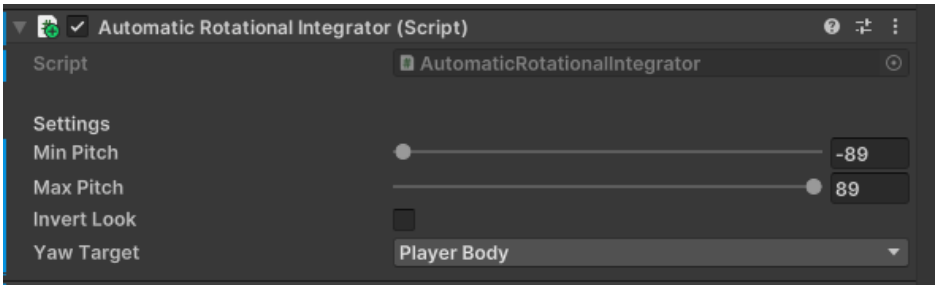


Figure 21: Automatic Integrator Settings

- `Min Pitch` is the lower limit of the pitch that the aim assist is allowed to rotate to
- `Max Pitch` is the upper limit of the pitch that the aim assist is allowed to rotate to
- `Invert Look` will flip the pitch
- `Yaw Target` will select how to handle the yaw component of the given rotation:
  - `Player Body` will rotate the player's gameobject itself (uses its rigidbody, character controller or transform component)
  - `Camera Control` will rotate the camera with whatever transform the camera provider returned.
- `Update` (Cine 3 only) - select how to update the cinemachine camera.
  - `Late Update`: Update in LateUpdate method
  - `On Cine Brain Updated`: The new Cinemachine Brain event that fires whenever Cinemachine Brain is updated.

It is important to note that **POV / Orbital Follow etc will only work if the yaw target is camera control**.
A brief breakdown of the component is as follows:

- Set up and find the necessary components (initialization stage)
- With Cinemachine 3:
  - Find the PanTilt and OrbitalFollow components and if either was present, modify their appropriate values. If not, but there was a virtual camera, use that.
  - If there wasn't, use the camera control overrides from the camera provider, or the player body for the yaw integration.
- With Cinemachine 2:
  - Find the POV or FreeLook components
  - if present use that, if not but there's a virtual camera, use that
  - else revert to fallback which also handles the manual camera rig.
- Without Cinemachine, the fallback option is used, which uses the camera controls if present, or finds the main camera's transform for camera control, or uses the player itself.

It is also worth to point out that if the player body is rotated and it has a rigidbody, then this rotation will happen in FixedUpdate to let Unity interpolate the result for a smooth rotation.

4.9 Input Aim Assist Manager

This component is responsible for integrating the input modifying aim assists. It's a singleton, that also has its belonging aim assists self-register on game startup.

The component being a singleton also means that this component cannot be used with local multiplayer because it only supports a single instance of aim assist of each script. Local multiplayer may still use the scripts, but they will have to be collected and used manually from code.

4.9.1 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.

4.10 Player Physics Info

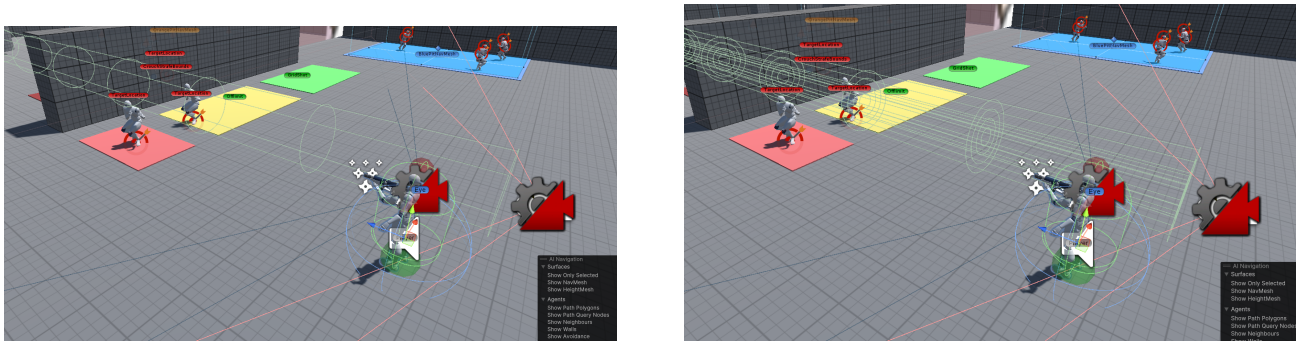
The Player Physics Info abstract class serves the aim assists with the player's velocity to be used in the calculations.

- `Automatic Player Physics Info` is the reference implementation that attemps to find velocity providers like a rigidbody on the player during startup, and returns the Unity interpolated value of these.
- `Legacy Player Physics Info` is the previous, manual solution that expects the developer to drag and drop the used component to it

If there's a different solution needed for your project, extend this base class with your code.

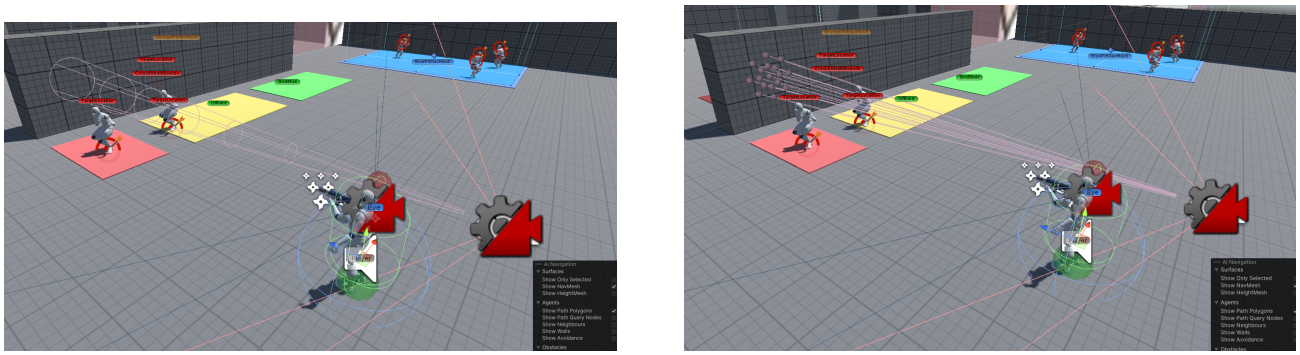
4.11 Debug Drawer

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 detailed and accurate.



(a) Sphere Selector Overview (b) Sphere Selector Detailed

Figure 22: Sphere selector debug drawer



(a) Cone Selector Overview (b) Cone Selector Detailed

Figure 23: Sphere selector debug drawer

In Play mode, different colors can be used if a target was found or not, which further simplifies debugging.

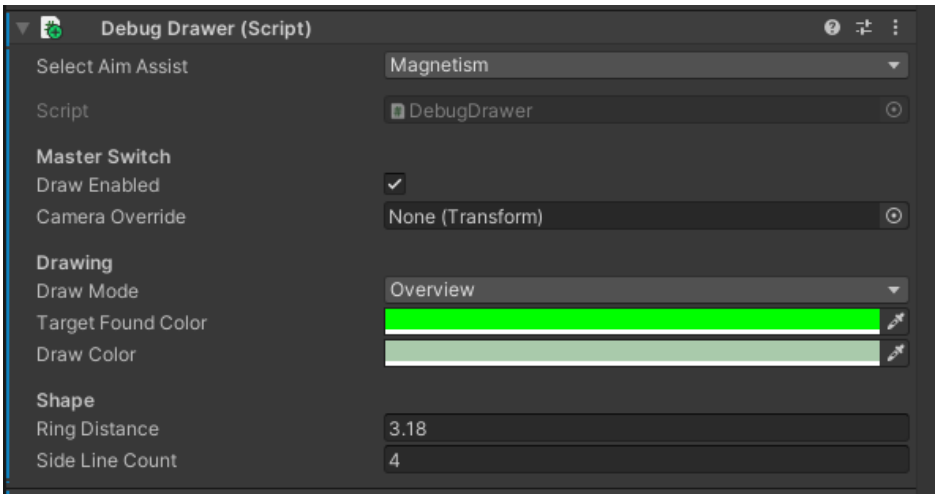


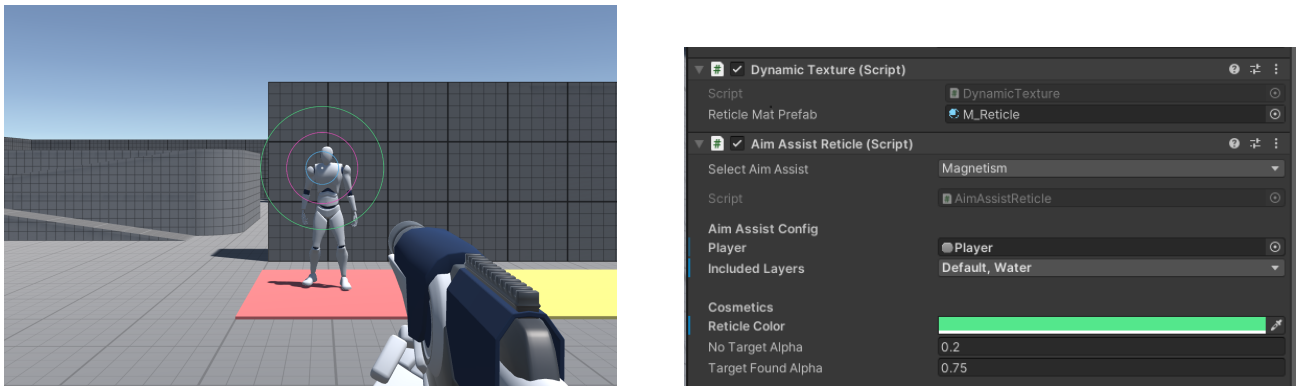
Figure 24: Debug Drawer settings

- `Select Aim Assist` is a custom editor that is present only when the given game object has multiple AimAssistBase subclasses. It allows the developer to select which aim assist's target selector is presented.
- `DrawEnabled` enables or disables drawing
- `Camera Override` manually set the camera used to draw the debug lines from. If not set, the given aim assist's PlayerCamera is used.
- `DrawMode` is the draw mode discussed above
- `Target Found Color` is the gizmo color when a target in sight is found
- `Draw Color` is the color when no target is found
- `Ring Distance` is the spacing between the rings
- `Side Line Count` is how many lines make up the gizmo's side

4.12 Reticle

The previous version's projector has been replaced with a procedurally calculated ring that shows the current aim assist radius both for spheres and cones.
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.
- `Included Layers` are the layers that this reticle's physics will interact with when it shoots its sphere cast for the sphere projection



(a) Reticle in action

(b) Reticle settings

Figure 25: Reticle

- `Reticle Color` is the color of the reticle
- `No Target Alpha` is the transparency of the reticle when it hasn't found a target yet
- `Target Found Alpha` is the transparency of the reticle when it has found a target.

4.13 Aim Assisted Bullet Manager

This component is a singleton, that references Magic Bullet, if there is one in the scene. MagicBullet self-registers into this singleton, so there is nothing else that needs to be done to make this aware of the script.

To make use of MagicBullet, simply call `GetMagicBulletCameraForward`, that sets a parameter modifier to be the modified 'camera forward' that points forward if there is no target found, or right at the target position (reminder: can be defined) if there is a target found. It returns whether MagicBullet took effect at all.

Call this in your code like so:

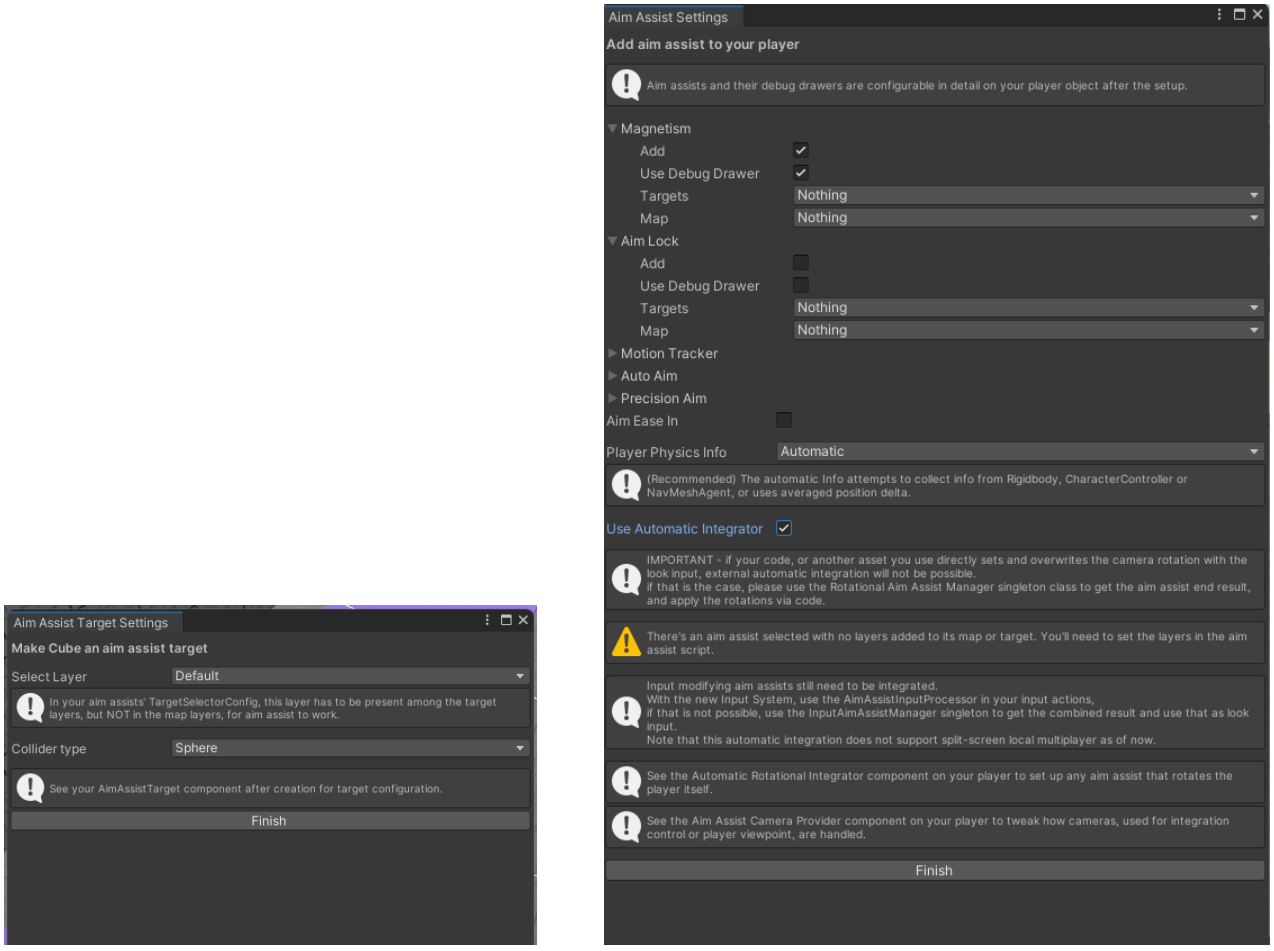
```
1 |     AimAssistedBulletManager.Instance.GetMagicBulletCameraForward(out var bulletDirection);
```

and instead of using the camera forward as raycast direction, use this bullet direction.

4.14 Setup with Context Menu

To make setting up a target and the player with aim assist simple, new context menus have been added to the package in version 2.0.0.
The target setup was detailed at the target part. The aim assist dialog has the following highlights:

- Check the aim assists to add, and check if you also want a target selector added to them
- Select the player physics info type to be added. the Default is Automatic which handles a lot of scenarios.
- Check whether you want to use the automatic integrator. The window will remind you that in some cases this may not be an option.



(a) Target Setup

(b) Aim Assisted Player Setup

Figure 26: Setup dialogs for the target and the player

- Select the layers for your targets and map. The window will let you know if your player is on the same layer as either of these, or if one of the layers are missing from your setup.
- The window also lets you know about the other components discussed here.

The configurer won't add components repeatedly. If a component is present already, it won't re-add it.

5 Integration

5.1 Automatic Integration - new in v2.0.0

Since version 2.0.0 automatic integration options are available for all aim assists, depending on your setup.

- Input Modifying aim assists can be integrated with Unity's Input System using `AimAssistInputProcessor`. Remember to use AimLock's `InvertPitch` if your pitch is inverted and the aim assist goes the

wrong direction.

- Rotational aim assists can be integrated using the Automatic Rotational Integrator, if your character's rotation isn't overwritten every frame by a character control script.

5.2 Manual Integration examples

5.2.1 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`:

```
public sealed partial class AgentInput {
    partial void ProcessStandaloneInput(bool IsInputPoll)
    {
        ...
        // original:
        // Vector2 mouseDelta = mouse.delta.ReadValue() * 0.075f;

        // modified:
        Vector2 mouseDelta = InputAimAssistManager.Instance.AssistAim(mouse.delta.ReadValue() * 0.075f);

        ... // rest of the method and class
    }
}
```

In this example, as InputProcessors and exposed methods wasn'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 InputAimAssistManager's aggregator code. In version 2.0.0 using this singleton this is a lot more straightforward because no direct references to the aim assist scripts are needed.

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

    private void Awake()
    {
        _kcc = GetComponent<KCC>();
        _aimAssist = GetComponent<RotationalAimAssistManager>();
    }

    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.");
```

```
24              return;
25          }
26
27          UseAimAssist();
28      }
29
30      private void UseAimAssist()
31      {
32          Rotation delta = _aimAssist.AssistAim();
33
34          if (delta == Rotation.Identity)
35          {
36              return;
37          }
38
39          _kcc.AddLookRotation(delta.Pitch, delta.Yaw);
40      }
41  }
```

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.

5.2.2 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:

```
1   public class ECM2RotationalIntegrator : MonoBehaviour
2   {
3       // or use your own character based on the Character base class
4       private ThirdPersonCharacter _tpsCharacter;
5       private FirstPersonCharacter _fpsCharacter;
6       private RotationalAimAssistManager _manager;
7
8       private void Awake()
9       {
10          _tpsCharacter = GetComponent<ThirdPersonCharacter>();
11          _manager = GetComponent<RotationalAimAssistManager>();
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 = _manager.AssistAim();
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.