Aim Assist Pro Documentation

March 2023

version 1.2.2

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

The asset consist of C# scripts that calculate rotation / pitch addition on top of your player's input. It is also equipped with demo scenes 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.

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 was developed using Unity's Input System but it's in no way tied to that input format - due to it being C# code in its essence, it should work with any third party input system that is available for Unity.

Test scenes are included using Unity's Input System.

4 Supported character control methods

The asset works with either `RigidBody` or `CharacterController`. After all, all they do is calculate rotations. Test scenes are included for both `RigiBody` and `CharacterController` based solutions.

5 Enabling the input method for test scenes

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.

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.

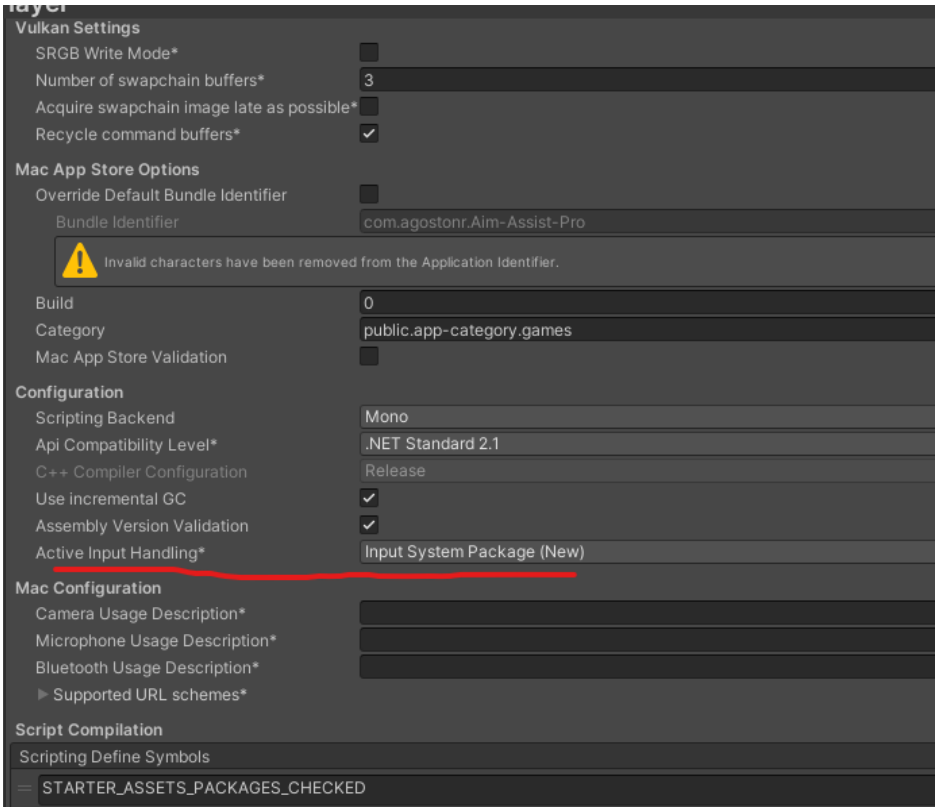


Figure 1: Enable the new input system

6 General structure

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.

6.1 Target

`AimAssistTarget` is a script you need to assign to the *GameObject*s you want to shoot at. If this script is missing from it, then the aim assist is not going to pick up your target.

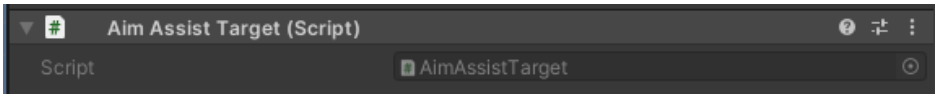


Figure 2: The simple target script to let the system know what to aim for

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.

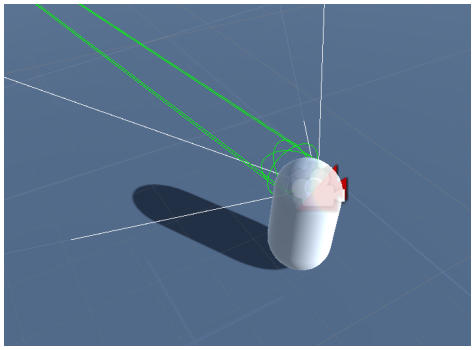
To save performance, caching is enabled to reduce unnecessary `GetComponent` calls that look for the target script. This cache is built on the fly - when the aim assist finds a collider, it looks for the aim assist component.

- 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.

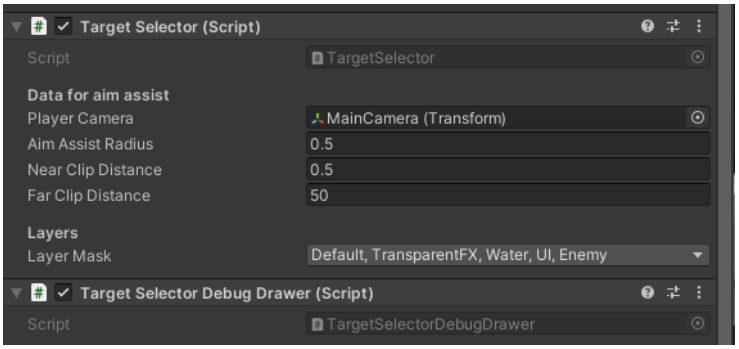
6.2 Target selector

The `TargetSelector` is in charge of finding a target using Unity's built-in physics system. It requires the transform of the player camera. If it isn't set, it throws a `MissingComponentException`.

- `Player Camera` is the transform of the player's camera. Needed to know where to shoot the spherecasts and raycasts from. This will be supplied to the aim assist scripts for aim adjustment calculations.
- `Aim Assist Radius` is the radius or spread of the aim assist in metres. The narrower this is, the closer the player have to aim to the target for the aim assist to actually take effect. Make it too large though, the spherecast will bump into obstacles unless the target is standing in the middle of nowhere.



(a) Aim assist radius



(b) Selector settings

Figure 3: Target selector and its settings

- `Near clip distance` - if the player is closer to the target than this, then the aim assist won't take effect. Useful when the aim assist would get annoying if the player moves close to enemies.
- `Far clip distance` - how far the aim assist perceives targets.
- `LayerMask` - which layers are actually taken into consideration when using the aim assist. **If you only add the enemy layer then obstacles and cover will be ignored, tracking the enemy through walls**.

The `TargetSelector` uses a SphereCast first, with the radius set by the developer in the Inspector, through *Aim assist radius*. If this doesn't find anything, then a corner might be blocking the target you're looking at - so it shoots a raycast too. If that doesn't find anything either, then we have no target to assist against. If there are multiple targets, the selected one is at the discretion of Unity's spherecast and raycast.

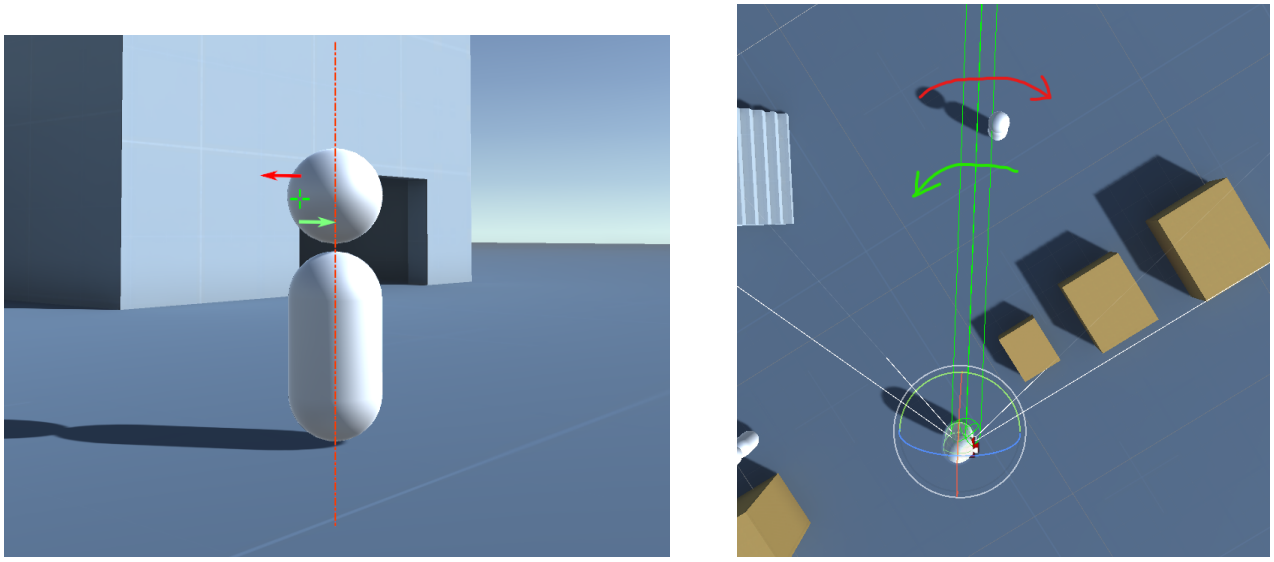
Once there's a target, If we weren't already looking at it, the `TargetSelected` event is fired. If we found no target but used to see one (one frame ago) then the `TargetLost` event is fired.

Since you may use multiple aim assists, most of them rely on this selector to provide them with a target. This eliminates all of them shooting sphere and raycasts over and over again, hindering performance.

`TargetSelector` **also provides two events,** `OnTargetSelected` **and** `OnTargetLost` that are fired once a new target is found, or the target is lost. It does not fire off every frame, only once per new target.

6.3 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.
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.



(a) Movement to and away from target

(b) Turn compensation for movement

Figure 4: Magnetism

- 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.

See the settings detailed below.

- `Player control type` select your player's controller setup: either `CharacterController` or `Rigidbody`.
- `Player Controller` or `Player Body` Drag the reference of the `CharacterController` or `Rigidbody` here.
- `Master switch` enable or disable the aim assist.
- `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).

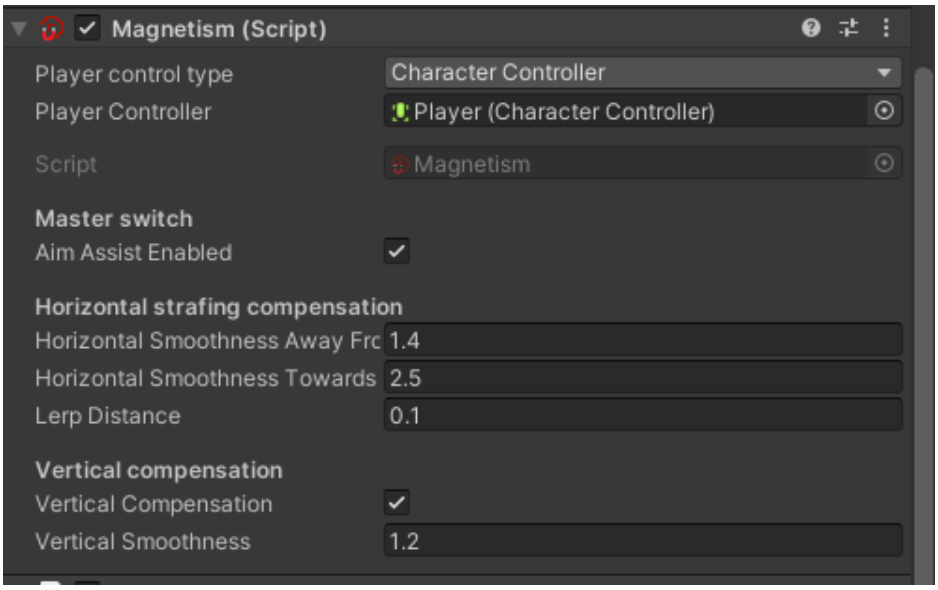


Figure 5: Magnetism settings

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 should be less than the aim assist radius - it it is set to more, it is clamped anyway. 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.

6.4 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 center of the target, e.g. its `transform.position`. Set up your scripts to the target's hitbox accordingly. It returns the adjustments that you add to your rotation in degrees.

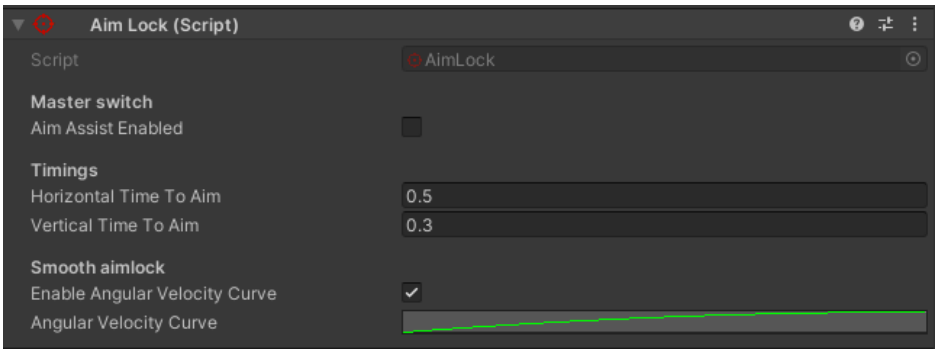


Figure 6: Aimlock settings

- `Master switch` enables or disables the aim assist.
- `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.
- `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.

6.5 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` - its behaviour is present always and is not tied to a target.

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

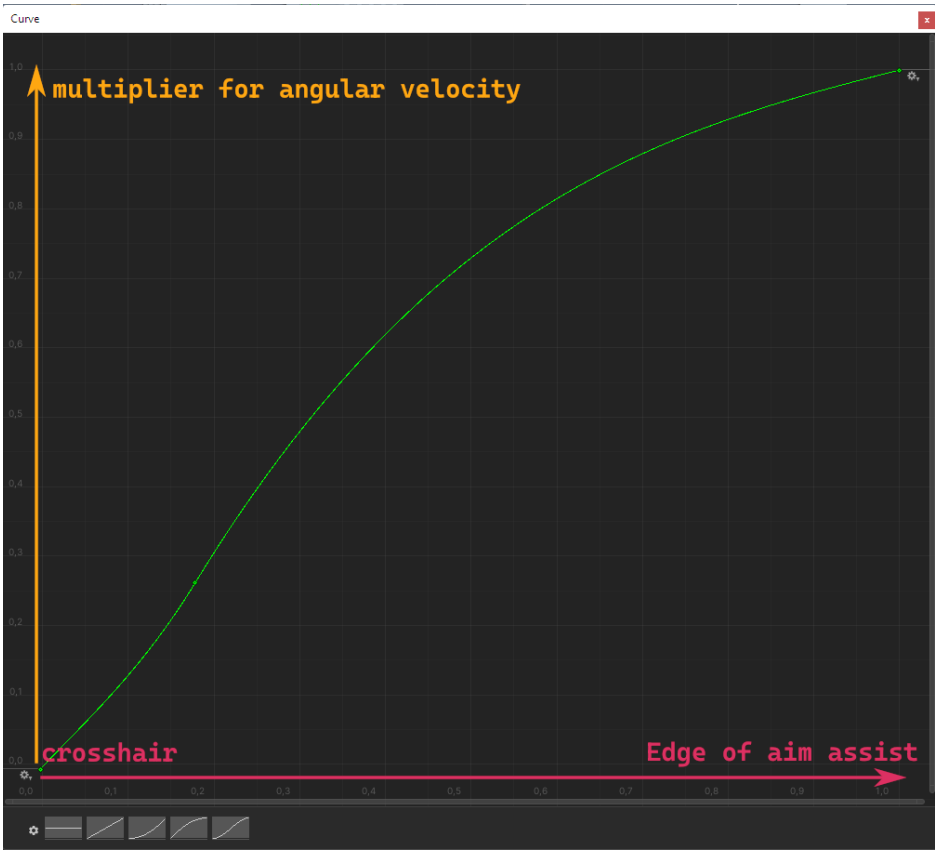


Figure 7: Aimlock angular velocity curve

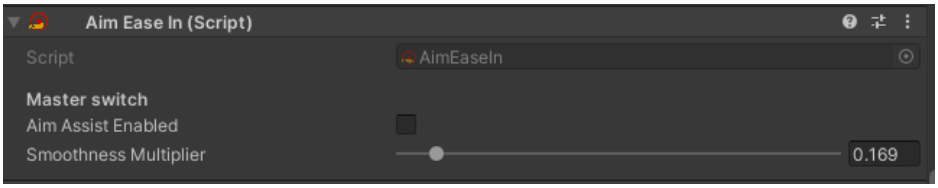


Figure 8: Aim ease in settings

6.6 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. This aim assist works akin to Aimlock. Just run your input delta through this script before using it in your look rotations. 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.

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

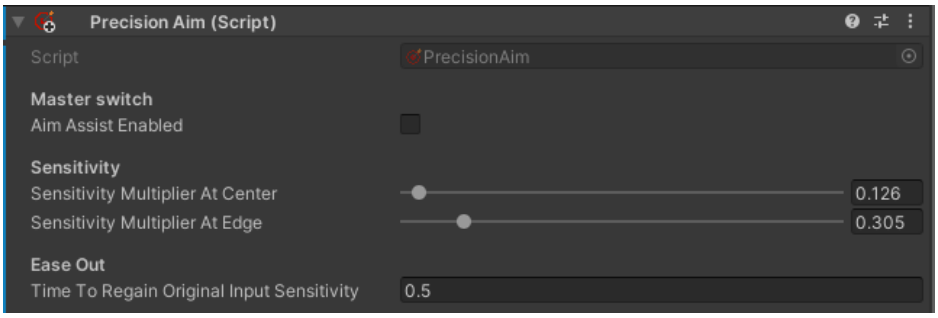


Figure 9: Precision aim settings

- `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.
- `Time to regain original input sensitivity` 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.

6.7 Auto aim

Auto aim is similar to `AimLock` in a sense that it'll move your crosshair towards the center of the target that is found by `TargetSelector`. 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.

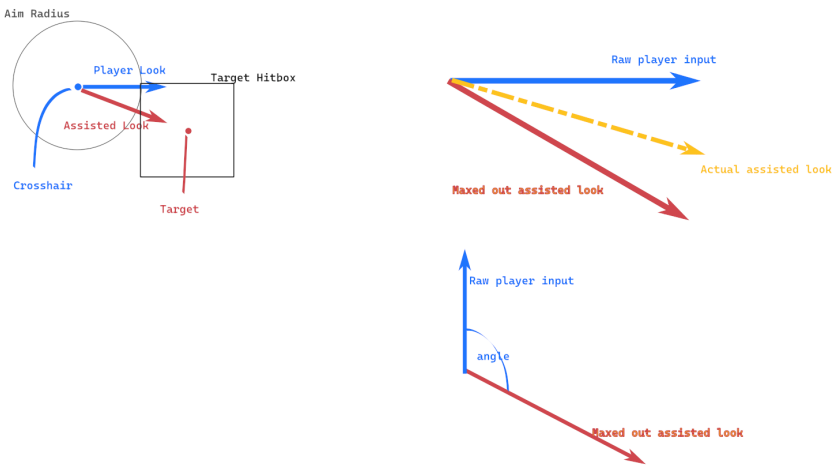


Figure 10: Auto aim explained

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.

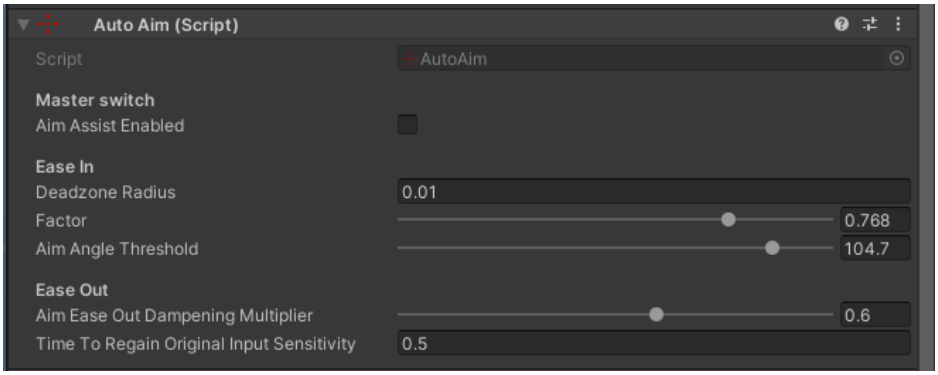


Figure 11: Auto aim settings

- `Aim assist enabled` a master switch to enable or disable aim assist.
- `Deadzone radius` the radius of 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.
- `Factor` defines the mix factor between player input and aim assist input. Low values favour the player's input so the aim assist barely intervenes. High values favour the aim assist input, making it more agressive in aiming towards the target. The crosshair won't get stuck looking at the target though, because
- `Aim angle threshold` defines an angle between the player's input, and the input needed to aim towards the target. If the player looks away (angles outside of the aim towards target) then the aim assist won't take effect. Small values define a narrow angle, so that aim assist only takes place when the player almost perfectly aims towards the target. High values define a more lenient angle

so that vaguely looking towards the target maintains aim assist active. The inputs are capped so the crosshair won't get stuck looking at the target.

- `Aim ease out dampening multiplier` is a multiplier that slows down look sensitivity when looking away from the target. This gives more time for the player to aim and prevent overshoot.
- `Time to regain original input sensitivity` is the time in seconds that takes for the aim assist to scale the look sensitivity back to original from the dampened value.

7 Integrating the aim assist scripts to your code

To see how to integrate the aim assist scripts to your code, follow the steps below.

7.1 Integrating Aim assist target

1. Make sure your GameObject has a collider
2. Make sure you select the appropriate layer so the `TargetSelector` takes this into consideration
3. Assign the `AimAssistTarget` script to the object.

If you also want the notifications when targets are found, subscribe to the events on the script. `TargetSelector` handles the invocation. These events can be useful when e.g. you have abilities in your game that you assign to certain NPCs or other players, it gives you a more generous hitbox and you don't have to perfectly look at your target.

7.2 Integrating Target Selector

`TargetSelector` itself is added by the aim assist scripts because it's a required component. You can add it manually if you would like to though.

1. Drag the player's camera to the `Player camera` field
2. Set your aim assist radius
3. Set your near clip distance
4. Set your far clip distance
5. Set your layer masks. Again, don't forget that if you don't set covers, only layers with the Enemy then the aim assist will track your targets through walls.

There's no code integration for `TargetSelector` - target selection works on its own and aim assist scripts will then use the target that is provided by it.

7.3 Integrating Magnetism

1. Add `Magnetism` script to your player character. This also adds a `TargetSelector` as it's a required component.
2. Drag your camera to the `Player camera` reference on the target selector
3. Set up your smoothness values as you see fit.

After that, `Magnetism` has to be used in your C# scripts. See an excerpt below of a demo script that shows an integration of `Magnetism`.

```
public class FPSController_CC : MonoBehaviour {
    private Magnetism _magnetism;
    ...

    private void Awake() {
        _magnetism = GetComponent<Magnetism>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
        AssistAim();
    }

    private void AssistAim() {
        var magnetismResult = _magnetism.AssistAim(_input.move);
        // this example is for Character Controller
        transform.Rotate(magnetismResult.TurnAddition);

        _cinemachineTargetPitch += magnetismResult.PitchAdditionInDegrees;
        _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
            0f, 0f);
    }
}
```

Get a reference to `Magnetism` and call it in the same loop you handle your camera rotation, after you handled it. You rotate the camera the same for the aim assist as you do for player input. This example is based on Unity's Character Controller Playground example.

`MagnetismInput` contains all information necessary for `Magnetism` to work. It needs the player's movement input axis and the current time delta. Note that the result `Magnetism` gives is an addition to either the rotation or the pitch, it isn't a set direction your camera needs to face as that would eliminate player input. `TurnAddition` is the rotation addition in degrees along Y axis, a *Vector3* that is added for convenience over the turn addition in degrees.

As per Unity's exmple, you need to clamp your pitch to prevent the camera "falling over".

For a `Rigidbody` based approach the example is as follows:

```
public class FPSController_RB : MonoBehaviour {
    private Magnetism _magnetism;
    ...

```

```
    private void Awake() {
        _magnetism = GetComponent<Magnetism>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
        AssistAim();
    }

    private void AssistAim() {
        var aimAssist = _magnetism.AssistAim(_input.move);

        var turnAddition = Quaternion.Euler(aimAssist.TurnAddition);
        // don't forget to keep your rotation and use Magnetism as an addition.
        // MoveRotation sets a target to look to, as opposed to Rotate before.
        _rigidbody.MoveRotation(_rigidbody.rotation * turnAddition);

        _cinemachineTargetPitch += aimAssist.PitchAdditionInDegrees;
        _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
            0f, 0f);
    }
}
```

Using `Magnetism` for a Rigidbody based controller is very similar, the only difference is that you need to pay attention not to eliminate your old rotation as MoveRotation sets a rotation to look to. Adding *Quaternion*s and *vector*s happen by multiplying them.

With this setup you have your aim assist in place and will compensate for strafing against a target.

7.4 Integrating Aimlock

`Aimlock`'s integration works quite akin to that of `Magnetism`'s, but the script needs no additional input parameters to work.

1. Add the `AimLock` script to your player object. This will add a `TargetSelector` too if not present already.
2. Drag the camera's reference to the target selector's camera slot.
3. Set your angular velocities and the velocity curve. It is highly recommended to use the angular velocity curve.
4. Set your adaptive velocity if you want to prevent a snappy aim at a distance.

Integrating `Aimlock` in your scripts is very much akin to integrating `Magnetism` as discussed earlier.

```
public class FPSController_CC : MonoBehaviour {
    private AimLock _aimLock;
    ...

    private void Awake() {
        _aimLock = GetComponent<AimLock>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
```

```
12          AssistAim();
13      }
14
15      private void AssistAim() {
16          var aimLockResult = _aimLock.AssistAim();
17
18          transform.Rotate(aimLockResult.TurnAddition);
19
20          _cinemachineTargetPitch += aimLockResult.PitchAdditionInDegrees;
21          _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
22          CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
23              0f, 0f);
24      }
25  }
```

`Aimlock`, akin to `Magnetism`, returns a struct of degrees that contain the additional rotation for your player and camera. Call this in the loop you handle your camera rotations, after you did so and the adjustments to your player rotation and camera pitch, as if you'd add rotations based on player input.

Rigidbody setup is very similar. Make sure you keep your original rotation while using Rigidbody's `MoveRotation`.

```
1   public class FPSController_RB : MonoBehaviour {
2       private AimLock _aimLock;
3       ...
4
5       private void Awake() {
6           _aimLock = GetComponent<AimLock>();
7       }
8       ...
9
10      private void LateUpdate() {
11          HandleCameraRotation();
12          AssistAim();
13      }
14
15      private void AssistAim() {
16          var aimAssist = _aimLock.SnapAim();
17
18          var turnAddition = Quaternion.Euler(aimAssist.TurnAddition);
19          _rigidbody.MoveRotation(_rigidbody.rotation * turnAddition);
20
21          _cinemachineTargetPitch += aimAssist.PitchAdditionInDegrees;
22          _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
23          CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
24              0f, 0f);
25      }
26  }
```

7.5 Integrating Aim ease in

`Aim ease in` is the simplest to use and integrate. It doesn't even require a `TargetSelector` to work as its effect is always present, not only when looking at targets.

1. Add `AimEaseIn` to your player object
2. Set the smoothness (multiplier) that downscales your less dominant look axis.

```
public class FPSController_CC : MonoBehaviour {
    private AimEaseIn _aimEaseIn;
    ...

    private void Awake() {
        _aimEaseIn = GetComponent<AimEaseIn>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
    }

    private void HandleCameraRotation() {
        if (_input.look.sqrMagnitude < _threshold) {
            return;
        }

        // this single line integrates aim ease in.
        var look = _aimEaseIn.AssistAim(_input.look);
        _cinemachineTargetPitch += look.y * RotationSpeed * Time.deltaTime;
        _rotationVelocity = look.x * RotationSpeed * Time.deltaTime;

        _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
            0.0f, 0.0f);
        transform.Rotate(Vector3.up * _rotationVelocity);
    }
}
```

The single line that is marked integrates `Aim ease in`. All you have to do is run your look input delta through the script before using it to handle player input for rotations.

7.6 Integrating Precision aim

`Precision Aim` is another simple aim assist that slows down your input by a given curve.

1. Add `PrecisionAim` to your player object. This will add a `TargetSelector` too if not present already.
2. Drag your camera to target selector's camera slot.
3. Set the smoothness curve. See guidelines above.

Integrating `Precision aim` in your script happens by creating a `PrecisionAimInput` struct that contains the current `deltaTime` and the player's look input, then using the resulting look input to handle your look controls. You basically run your look input through it and use the result as if you'd use the original look input.

```
public class FPSController_CC : MonoBehaviour {
    private PrecisionAim _precisionAim;
    ...

    private void Awake() {
        _precisionAim = GetComponent<PrecisionAim>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
```

```
    }

    private void HandleCameraRotation() {
        if (_input.look.sqrMagnitude < _threshold) {
            return;
        }

        // this single line integrates precision aim.
        var look = _precisionAim.AssistAim(_input.look);
        _cinemachineTargetPitch += look.y * RotationSpeed * Time.deltaTime;
        _rotationVelocity = look.x * RotationSpeed * Time.deltaTime;

        _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
            0.0f, 0.0f);
        transform.Rotate(Vector3.up * _rotationVelocity);
    }
}
```

The line marked with a comment integrates `PrecisionAim`. All you do is run your look input delta through it before using that input delta for handling your rotation and pitch.

7.7 Integrating AutoAim

`AutoAim` is another simple aim assist that slows down your input by a given curve.

1. Add `AutoAim` to your player object. This will add a `TargetSelector` too if not present already.
2. Drag your camera to the target selector's camera slot.
3. Set the smoothness curve. See guidelines above.

Integrating `AutoAim` in your script happens by creating a `AutoAimInput` struct that contains the current `deltaTime` and the player's look input, then using the resulting look input to handle your look controls. You basically run your look input through it and use the result as if you'd use the original look input.

```
public class FPSController_CC : MonoBehaviour {
    private AutoAim _autoAim;
    ...

    private void Awake() {
        _autoAim = GetComponent<AutoAim>();
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
    }

    private void HandleCameraRotation() {
        if (_input.look.sqrMagnitude < _threshold) {
            return;
        }

        // this single line integrates auto aim.
        var look = _autoAim.AssistAim(_input.look);
        _cinemachineTargetPitch += look.y * RotationSpeed * Time.deltaTime;
        _rotationVelocity = look.x * RotationSpeed * Time.deltaTime;

        _cinemachineTargetPitch = ClampAngle(_cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(_cinemachineTargetPitch,
```

```
            0.0f, 0.0f);
        transform.Rotate(Vector3.up * _rotationVelocity);
    }
}
```

The line marked with a comment integrates `AutoAim`. All you do is run your look input delta through it before using that input delta for handling your rotation and pitch.

7.8 Integrating multiple aim assists that work by scaling look input

As of version 1.2.0, support has been added to integrate look input scaling aim assists (AimEaseIn, AutoAim, PrecisionAim) easier. The class responsible is `LookInputBasedAimAssistChainer`.
What it does is, it enables you to run your look input delta through these aim assists using a simple builder pattern.

```
public class FPSController_CC : MonoBehaviour {
    private AutoAim _autoAim;
    private AimEaseIn _aimEaseIn;
    private PrecisionAim _precisionAim;

    private readonly LookInputBasedAimAssistChainer lookAimAssistChainer =
        new LookInputBasedAimAssistChainer();
    ...

    private void Awake() {
        // get component to the aim assist scripts
        ...
    }
    ...

    private void LateUpdate() {
        HandleCameraRotation();
    }

    private void CameraRotation() {
        if (input.look.sqrMagnitude < _threshold) {
            return;
        }

        // chain your look input modifying aim assists using a builder pattern.
        var look = lookAimAssistChainer
                .WithLookInputDelta(input.look)
                .UsingAimEaseIn(aimEaseIn)
                .UsingPrecisionAim(precisionAim)
                .UsingAutoAim(autoAim)
                .GetModifiedLookInputDelta();

        cinemachineTargetPitch += look.y * RotationSpeed * Time.deltaTime;
        rotationVelocity = look.x * RotationSpeed * Time.deltaTime;

        cinemachineTargetPitch = ClampAngle(cinemachineTargetPitch, BottomClamp, TopClamp);
        CinemachineCameraTarget.transform.localRotation = Quaternion.Euler(cinemachineTargetPitch, 0.0f, 0.0f
        transform.Rotate(Vector3.up * rotationVelocity);
    }
}
```

Not all three of these aim assists have to be present for the chain to work and the order of invocations will not affect the outcome as it is fixed inside the chainer. If no aim assist script is assigned then it will return the original look input. If that isn't supplied either, it return a `Vector2.zero`.