Aim Assist 2D Documentation

March 2023

version 1.0.0

Contents

1 Greetings

Thank you for downloading my asset Aim Assist 2D. The asset intends to help developers of the platformer / 2D Shoot 'em up genres 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 2D into your project.

If the asset was useful to you please consider dropping a good review on the Asset Store so others can find it.

If you found something missing or not quite right, do not hesitate to write an email with your feedback.

2 About the asset

The asset consists of C# scripts that calculate the rotation 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 especially for controllers or handheld devides like a phone to help players aim. See the descriptions for the type of aim assists below to get a feeling of how they would serve you to take your game's experience to the next level.

3 Supported input systems

The asset works by calculating the necessary rotations for rotation, or scaling the magnitude of the player input. The outputs are either the changed input values or rotation adjustments in degrees. **The demo scenes use Unity's new Input System but the asset itself is not tied to any input method - the only requirement is that you need access to the code that handles your inputs as Aim Assist is integrated using a few lines of code.**

When importing the asset, Unity should prompt you to install and enable Input System so please do so. If it doesn't and the input is not working, see the official setup guide here.

4 Project dependencies

The asset itself has no dependencies - in its essence it's just plain C# scripts. The demo scenes however need the following dependencies:

- Cinemachine (platformer demo)
- Input System (both)
- TextMeshPro essentials (both)

The demo scenes are implemented for a Gamepad or a keyboard (and mouse for menus) however the recommended input device is the Gamepad. Again, the asset itself works with any device you can use in your project.

5 Supported character control methods

The asset works with any movement solution, be it based on Rigidbody2D, handmade or a third party plugin. That is because the asset calculates player velocity based on the position difference between frames of the player.

6 General structure

As mentioned before, all aim assist types contain calculations that adjust the rotation of the player (or the gun 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.

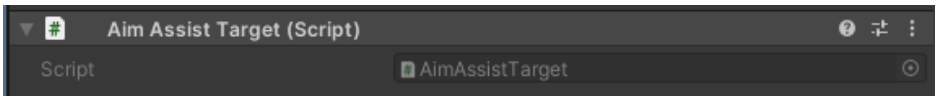


Figure 1: The simple target script to let the system know what to aim for

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

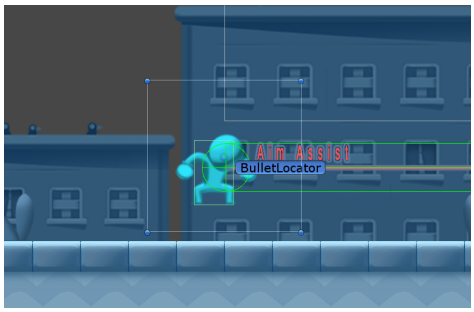
- `TargetSelected` event is called once, when an aim assist finds the given target. It isn't called repeatedly in the assist loop.
- `TargetLost` event is called once, when an aim assist doesn't see the target anymore. After this, `TargetSelected` can be invoked again.

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 in the cache, using the object id of that component. If it finds it, it returns with it. If it doesn't, it calls `GetComponent` on the object, stores the result and returns with that.

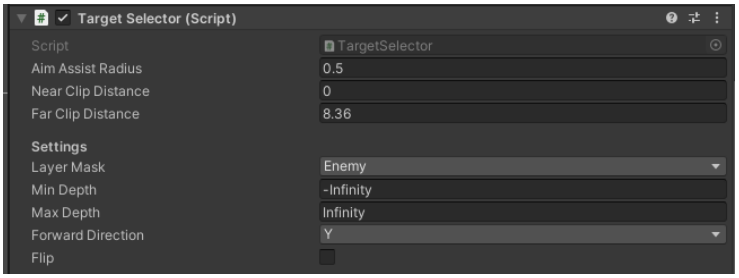
- If an assist script is already stored for the specific object, it will just return that and won't call `GetComponent`.
- If it doesn't find it, it's going to cache null and won't try to look for it again. **When assigning these target scripts to *GameObject*s runtime, make sure to either add it to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

6.2 Target selector

The `TargetSelector` is in charge of finding a target using Unity's built-in physics2D system. It assumes an `AimOrigin`, which is going to be the transform of the GameObject it is put on. As a result, when integrating aim assist, you need to put the components on the object that you'll aim with - it can be the player itself, or a gun that is a child of the player object. It also provides the AimDirection that is used for the aim assist.



(a) Aim assist radius



(b) Selector settings

Figure 2: Target selector and its settings

- `Aim Assist Radius` is the radius or spread of the aim assist in metres. The narrower this is, the closer the player has to aim to the target for the aim assist to actually take effect. Make it too large though, the circlecast will bump into obstacles unless the target is standing in the middle of nowhere.
- `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**.
- `Min Depth` is the minimum Z depth to consider when looking for targets. By default it's negative infinity.
- `Max Depth` is the maximum Z depth to consider when looking for targets. By default it's infinity.
- `Forward Direction` is the axis of the transform in which direction we cast the circle to look for targets. It can be -X, X, -Y, Y. Depending on your game you need to set this to desginate your aim assist in the direction where you'll be shooting at - in the platformer demo this is X (shoot to the right), in the spaceship demo it's Y (shoot up).
- `Flip` flips your aim assist direction - from -X to X, from -Y to Y and vice versa. Useful when you flip your sprite instead of rotating your gun.

When the player turns back the target selector should work with either flipping the gun's direction or rotating it 180 degrees. That is because when calculating the signed angles for the assist, the code takes the aim origin's Z direction into account to determine the accurate sign of the rotation. That being said, in the platformer demo the **flip is handled by just rotating the aim origin.** In the platformer demo, if you flip your aim then the aim input is inverted. This is a shortcoming of the demo itself, not the aim assist.

The `TargetSelector` uses a CircleCast 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 circlecast 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, all of them rely on this selector to provide them with a target. This eliminates all of them shooting circle 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 movement and jumps to make it easier to keep track of the enemy. It returns the rotation adjustment in degrees that you add to your rotations along the Z axis.

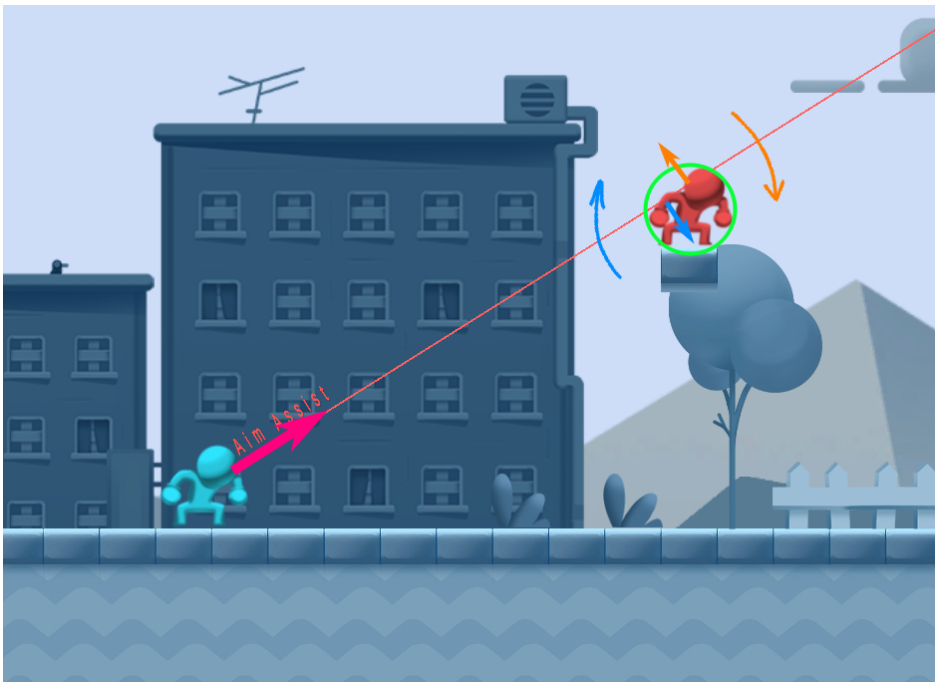


Figure 3: Magnetism explained

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 **blue** arrow on figure **3**. It basically means that the player is moving in the target's direction and is not yet facing the target, thus moving

towards its center. The figure also shows an arrow of the same color in the opposite direction - that will be the compensation of the rotation. It is divided by the smoothness.

- Moving **away** from the target is the movement represented by the **orange** arrow on figure **3**. It means that the player is moving the opposite direction, away from the target's center. An arrow of the same color shows where the compensation will rotate the aim to.

So movement compensation means turning the aim the opposite direction of the player's move direction (compared to the target). Magnetism does this but the compensation is 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, dropping the aim to the target.

See the settings detailed below.

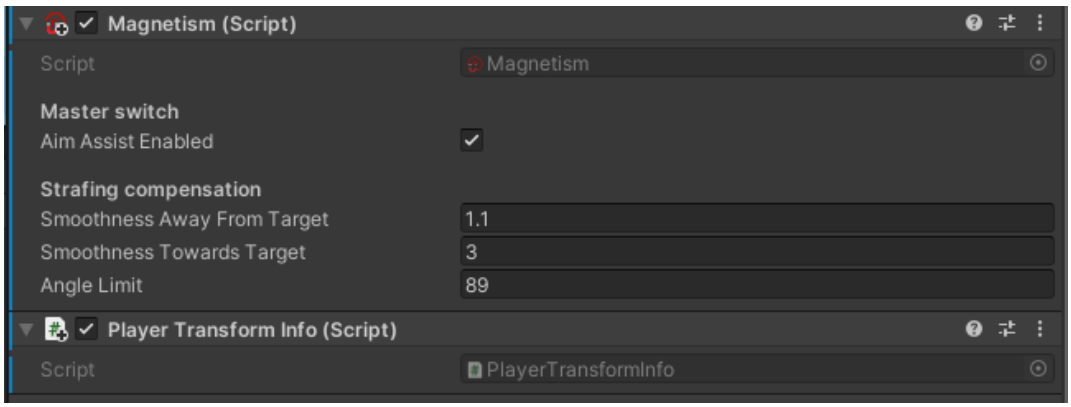


Figure 4: Magnetism settings

- `Aim Assist Enabled` is a master switch that enables or disables Magnetism. If it's disabled, it always returns a zero rotation addition without doing any calculations. As a result, there's nothing that needs to be changed at the place of the integration.
- `Smoothness away from the target` is the divisor for the rotation calculated when the player is moving away from the target and the aim assist rotates the aim back to the target. Increasing this will make the aim assist less prominent.
- `Smoothness towards the target` is the divisor for the rotation calculated when the player is moving towards the target and the aim assist rotates the aim away to give the player more time to hit the shot (easing the aim to the target).
- `Angle limit` is the angle limit to both negative and positive directions for the assist. If the adjusted rotation would exceed this, then it will just return zero to clamp the aim assist to prevent overextension.
- `Player Transform Info` below serves to provide up to date information on the player's velocity, which is calculated from the delta position of the player. As a result, it is not dependent on a component like a Rigidbody2D.

6.4 Aimlock

A smooth aimlock that tracks the target for you with the set angular velocity. 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 target's hitbox accordingly. It returns the adjustments that you add to your rotation in degrees.

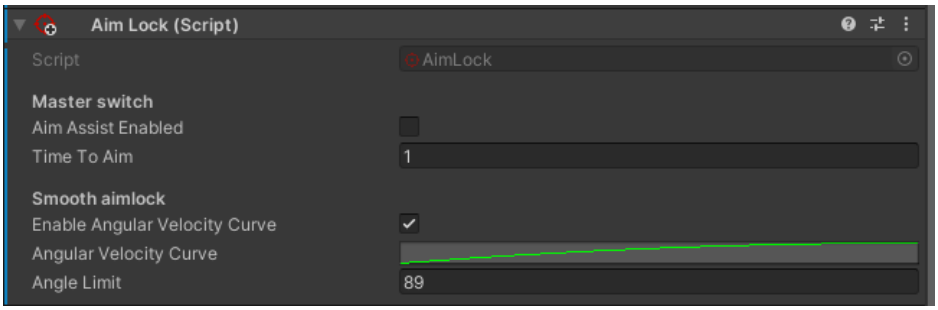


Figure 5: Aimlock settings

- `Aim Assist Enabled` is a master switch that enables or disables Magnetism. If it's disabled, it always returns a zero rotation addition without doing any calculations. As a result, there's nothing that needs to be changed at the place of the integration.
- `Time to aim` is the time it should take from picking up the target with the circlecast's edge to turning the aim directly at the center of the target. The angular velocity is calculated based on this value so **decreasing** it will yield faster aiming while **increasing** it will yield slower, lazier aim.
- `Enable angular velocity curve` will enable or disable the additional smoothing via the velocity curve.
- `Angular velocity curve` is an additional multiplier that slows down the aim as it gets closer to the center of the target. Enabling this makes the aim more natural and less snappy or robotic. As shown in figure 6, the X axis of the curve represents how close the aim is to the center of the target. 0 means we are looking directly at the target, while 1 means the circlecast barely picked up the target. The Y axis represents the multiplier that will be applied to the angular velocity. **It is advised to keep the curve on both X and Y axes between 0 and 1**.

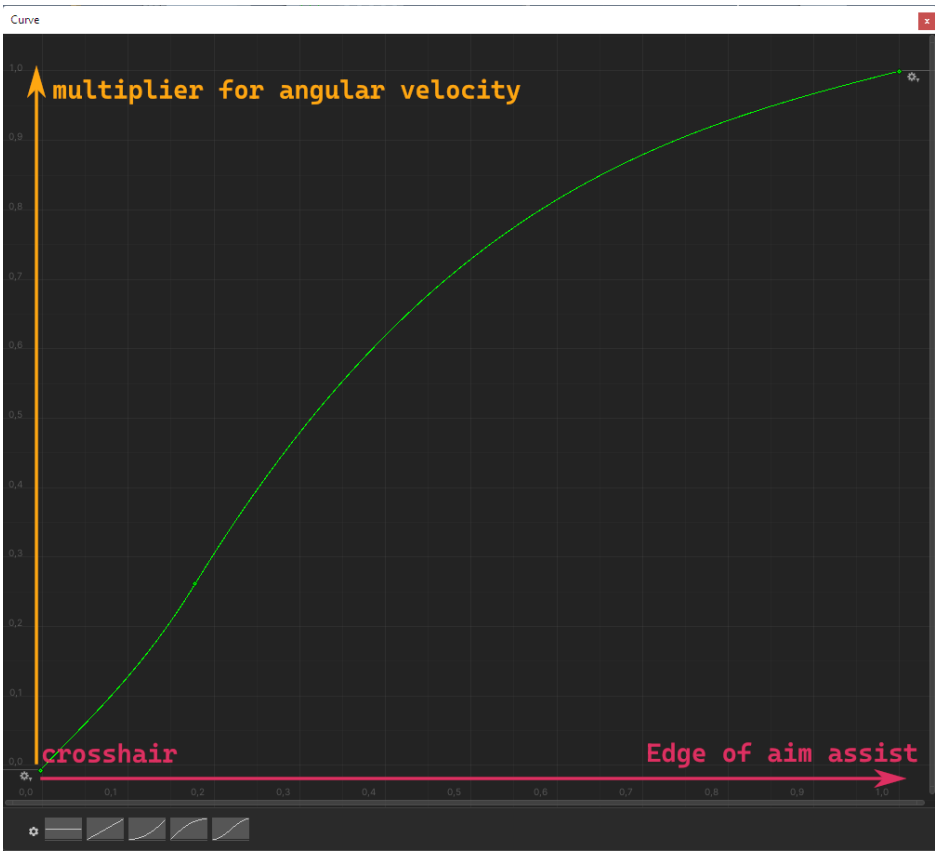


Figure 6: Aimlock angular velocity curve

6.5 Precision aim

Simply scales down the input delta using two designated multipliers - one for the edge of the circlecast, one for when the player is looking directly at the target. The interpolation between these is linear. 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 slowed aim to unassisted aim. Precision aim is particularly useful with flickshots that are otherwise complicated to do on anything that's not a mouse.

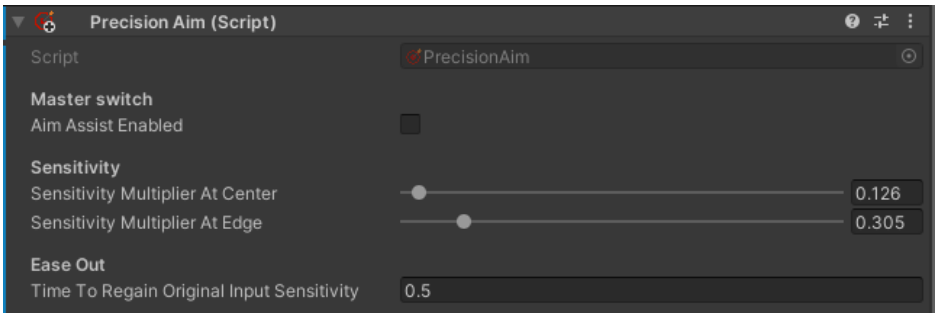


Figure 7: Precision aim settings

- `Aim Assist Enabled` is a master switch that enables or disables Magnetism. If it's disabled, it always returns a zero rotation addition without doing any calculations. As a result, there's nothing that needs to be changed at the place of the integration.
- `Sensitivity multiplier at center` is the multiplier that will be applied to the aim input when the player is directly looking at the target. Making this slow means more effort needed to look away from the target - so the aim will be more precise on the target.
- `Sensitivity multiplier at edge` is the multiplier that will be applied to the aim input when the aim assist barely touches the target - so it's on the edge of the aim assist.
- `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.
- `Time to regain original input sensitivity` The time (in seconds) it takes for the input sensitivity to get back to the unassisted value after the target was lost. This is immediately stopped when a new target has been found, in which case the aim assist will take over.

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 collider2D
2. Make sure you select the appropriate layers 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. That being said, `TargetSelector`, along with the aim assists **have to go on the GameObject that's responsible for aiming, that is, the object that will be rotated for aiming.** It could be the player itself (see the SpaceShip demo) or a gun that is a child object of the player.

1. Set your aim assist radius
2. Set your near clip distance
3. Set your far clip distance

4. Set your layer masks. Again, don't forget that if you don't set layers that represent covers (the map), 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 gameobject that will rotate when aiming. This might be the player itself, or a gun that is a child of the player. This also adds a `TargetSelector` as well as a `PlayerTransformInfo` as they are required components.
2. Set up your values values as you see fit.

After that, `Magnetism` has to be used in your C# scripts, similarly to how you handle rotations based on look input. See an excerpt below of a demo script that shows an integration of `Magnetism`. `WeaponControls` script goes to the same GameObject that is your Gun/Aim Origin.

```
public class WeaponControls : MonoBehaviour {
    private Magnetism magnetism;
    ...

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

    private void Update() {
        ... // other input handling

        var rawInput = Input.GetAxis("Aim");

        if (Mathf.Abs(rawInput) > 0.1) {
            // this handles the actual aim input, which is part of your game
            Aim(rawInput);
        }

        // after handling the aim, use the aim assist
        UseMagnetism();
    }

    private void UseMagnetism() {
        // get the aim assist result
        var result = magnetism.AssistAim();

        // rotate the transform of your gun / player.
        transform.Rotate(result.RotationAddition);
    }
}
```

Get a reference to `Magnetism` and call it in the same loop you handle your look input, after you handled it. You rotate the player/gun the same for the aim assist as you do for player input.

With this setup you have your aim assist in place and will compensate for moving or jumping against a target. Note that the integration could have been part of the magnetism script itself but for the most accurate results the aim assist should happen in the same loop as the player input handling, right after it.

7.4 Integrating Aimlock

`Aimlock`'s integration works quite akin to that of `Magnetism`'s.

1. Add the `AimLock` script to your object that will be rotated when aiming. This can be the player itself, or a gun that is a child of the player. This will add a `TargetSelector` too if not present already.

2. Set up the parameters of AimLock as you see fit.

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

```
public class WeaponControls : MonoBehaviour {
    private AimLock aimLock;
    ...

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

    private void Update() {
        ... // other input handling

        var rawInput = Input.GetAxis("Aim");

        if (Mathf.Abs(rawInput) > 0.1)
        {
            // this handles the actual aim input, which is part of your game
            Aim(rawInput);
        }

        // after handling the aim, use the aim assist
        UseAimLock();
    }

     private void UseAimLock() {
        // invoke the aim assist
        var result = aimLock.AssistAim();

        // rotate your transform based on the results of aimlock.
        transform.Rotate(result.RotationAddition);
    }

}
```

Akin to `Magnetism` it returns a struct of degrees that contain the additional rotation for your aim. Call this in the loop you handle your look rotations, after you did so, and add the adjustments to your rotation along Z axis, as if you'd add rotations based on player input.

7.5 Integrating Precision aim

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

1. Add `PrecisionAim` to your aiming object, as before, to either the player itself, or a gun of the player. This will add a `TargetSelector` too if not present already.

Integrating `Precision aim` in your script happens by creating a `PrecisionAimInput` struct that contains 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 WeaponControlller : MonoBehaviour {
    private PrecisionAim precisionAim;
    ...

    private void Awake() {
        precisionAim = GetComponent<PrecisionAim>();
    }
    ...
    private void Update() {
        ... // other input handling
        var rawInput = Input.GetAxis("Aim");

        // run your look input through precision aim before use
        var assistedInput = precisionAim.AssistAim(rawInput);

        if (Mathf.Abs(assistedInput) > 0.05f)
        {
            // use the assisted input as if it was the raw user input
            Aim(assistedInput);
        }
    }
}
```

All you need to do is run your look input delta through it before using that input delta for handling your look rotations.