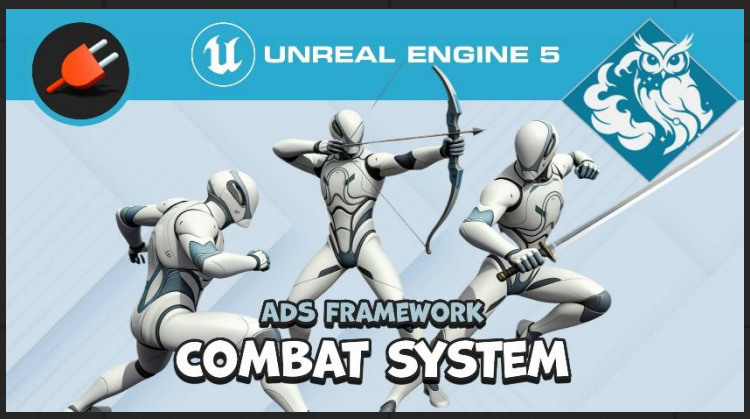


V1.0

Table of Contents

Foreword

IMPORTANT

ADS Framework is a modular framework for Unreal Engine. **It is not a complete game** template.

A template usually provides a ready-to-use foundation with a predefined gameplay structure. This is convenient for getting started quickly, but it can also limit the project to a specific logic, genre, or architecture.

A framework works differently. It provides reusable, configurable, and extensible tools designed to help you build your own game without imposing a specific type of gameplay. ADS Framework therefore does not replace project-specific development : it provides a technical foundation to save time, structure your features, and avoid starting from scratch, while remaining reliable, safe, and performant. Each system is designed to integrate into all types of projects: action, adventure, RPG, sports, etc.

ADS Framework is developed and maintained with a commitment to continuous improvement. Despite the care put into development, documentation, and testing, it remains a technical product that may evolve based on user feedback and future versions of Unreal Engine.

If you encounter an issue or limitation, it is recommended to contact support instead of leaving a negative review. Constructive feedback helps identify areas for improvement and move the systems in the right direction.

The ADS Framework Philosophy

ADS Framework is based on a simple idea: providing independent systems that can be used on their own, combined with each other, or combined with your own systems depending on your project needs.

Each Framework system can be integrated individually and does not require purchasing or using the other systems to function. You can therefore use it with your own architecture, your own Blueprints, your own interfaces, or other assets purchased on Fab.

If you choose to use multiple Framework systems, they share a common philosophy: modularity, clear logic, Blueprint integration, accessible configuration, and separation of responsibilities.

The systems are designed to be accessible to users who already have basic Unreal Engine and Blueprint knowledge, without necessarily requiring an advanced level. Some modules, although relatively simple to use, are still complex; it is therefore normal to allow a few days of familiarization to understand their logic, settings, and integration.

The included demos are intentionally lightweight. They use simple, readable logic to show how the systems work without unnecessarily increasing project size. They demonstrate functionality, possibilities, and ease of use, without providing final visuals. This choice limits project size and keeps the focus on the systems logic.

C++ / Blueprints

ADS Framework systems are primarily developed in C++ to provide a stable, high-performance, structured foundation suitable for production use.

Each system also exposes a Blueprint layer designed to make integration into your projects easier. This approach lets you benefit from a C++ architecture while keeping the system accessible from the Unreal Engine editor.

Depending on the system, some parts can be configured, called, or extended directly in Blueprints. Project-specific elements can therefore be created in Blueprints, or developed in C++ if you prefer working with native logic.

The goal is to provide a balance between robustness, performance, accessibility, and flexibility.

Compatibility with Unreal Engine 6

ADS Framework is developed with a long-term maintenance and evolution mindset.

Epic Games has presented Unreal Engine 6 as a major evolution of the Unreal Engine ecosystem, with a stronger role for Verse and the new Scene Graph gameplay framework. This evolution may gradually change the way certain gameplay logic is developed in future versions of the engine.

With this in mind, ADS systems will continue to be maintained and adapted as much as possible to Unreal Engine evolutions. When the UE6 / Verse ecosystem is available, documented, and stable enough, the relevant Blueprint parts may gradually be adapted or replaced with solutions compatible with this new architecture.

The goal is to preserve the logic of the ADS systems while keeping pace with the engine evolution.

System Description

ADS Combat System is a toolbox designed to help you create your own combat system in Unreal Engine. It is not a complete ready-to-use combat system with predefined rules, damage, statistics, inventory, or behaviors.

The plugin provides a simple, reliable, safe, and high-performance technical foundation to structure your combat logic. It provides a World Subsystem to centralize combat management, an Actor Component to add to combatants, ready-to-use collision classes, configurable animation notifiesa lock targetingsystem, as well as Blueprint classes for combat rules and rewards.

The system is designed to be universal. It can serve as the foundation for real-time, turn-based, hybrid, action-oriented, RPG, boss fight, enemy wave, or any other combat logic specific to your project.

ADS Combat System does not impose any inventory, quest, equipment, or save system. It can work with your own architecture, with other Fab assets, or with other ADS Framework systems if your project needs them.

This is the ADS Framework system that requires the most project-specific work. The plugin provides tools and entry points, but the final combat rules, damage, effects, statistics, rewards, animations, and behaviors must be defined by your project.

Installation

The plugin must be installed in the project. Do not use the version installed in the engine, because it will need to contain project-specific assets and configurations.

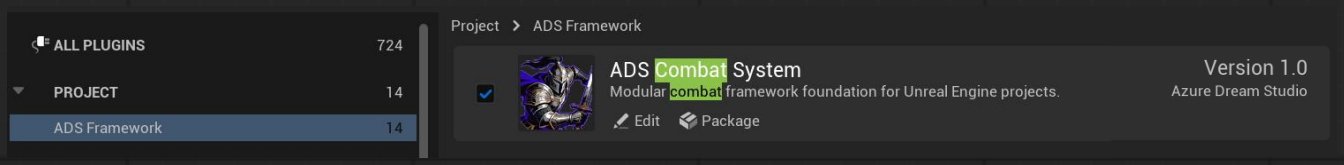
1- After installing the plugin in the engine via the Epic Games Launcher, close Unreal Engine and, from Windows File Explorer, copy the `plugin` folder located in the engine plugins folder to the project `Plugins` folder.

`"<UE installation folder>/Engine/Plugins/<ADS plugin folder>"`

`"<Project folder>/Plugins/ "`

2- Open Unreal Engine.
(If Unreal Engine asks to compile the plugin on startup, accept.)

3- Enable the plugin located in "**PROJECT/ADS Framework**", then restart



The plugin can be installed both in the engine and in the project. If so, since there will be two versions of the same plugin, Unreal Engine will use the one located in the project. After copying the plugin into the project, you can delete it from the engine if you want. In all cases, you should always use the version installed in the project, not the engine version.

How It Works

ADS Combat System is based on a clear separation of responsibilities.

The Combat Subsystem is the central point of the system. It knows the participants registered in combat, manages global combat information, stores the combat rules if provided, and exposes events that allow the project to react.

Each combatant must have a Combat Actor Component. This component is the link between the actor and the central system. It configures the combatant, registers it in combat, defines its role, side, or parameters, and communicates with the subsystem.

The rules specific to a combat are not hard-coded directly in the subsystem. They can be defined in dedicated Blueprint classes. This allows each combat to have its own logic: standard combat, timed combat, enemy waves, boss fights, turn-based combat, or any other project-specific rule.

The ADS collisions are used to detect combat interactions: attack, impact, hitbox, hurtbox, projectile, area of effect, or any other logic defined by the project. They can be activated directly in Blueprint or synchronized with an animation using the animation notifies provided by the plugin.

The animation notifies allow precise windows to be triggered during an animation: hit window, combo window, invulnerability frames, lock targeting, or collision-specific behavior. They are mainly used to synchronize combat logic with animation.

When a collision or combat action is validated, the system sends the useful information to the project. Your Blueprint logic then decides what to do: apply damage, block, parry, knock back, trigger feedback, grant a reward, update a quest, or call another system.

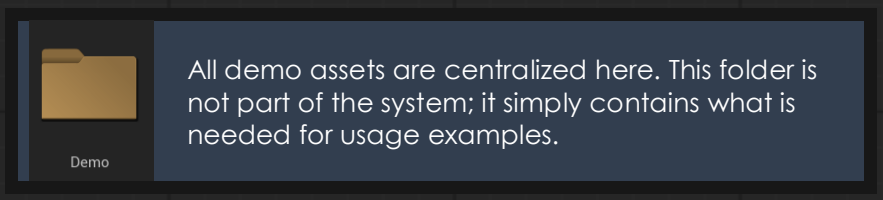
The lock targeting works with target components that can be added to actors. This makes it possible to define several lock points on the same actor, such as the head, torso, hands, or legs.

This architecture allows the plugin to remain universal. The system provides the structure, tools, and events, while your project remains responsible for the final combat rules.

Plugin Contents

This documentation is intended as a user guide. C++ classes will therefore not be detailed in this documentation. They will only be mentioned when they serve as parent classes or when they need to be called from Blueprint. The system is developed in C++, but its main usage is in Blueprints. This documentation therefore focuses on integrating and using the system.

The plugin has its own folder 'Content' for Blueprint assets. It contains only one folder, "Demo".

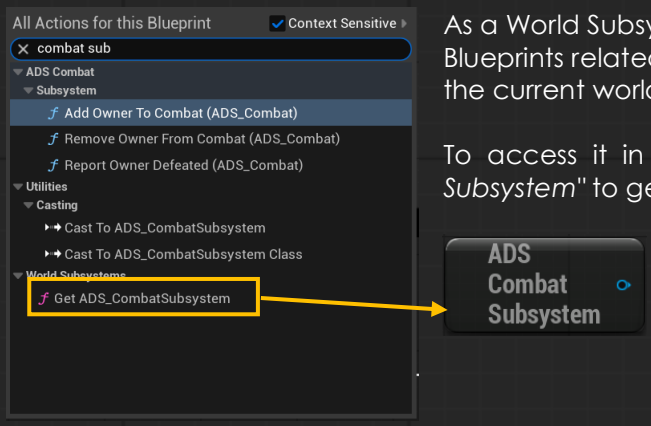


All demo assets are centralized here. This folder is not part of the system; it simply contains what is needed for usage examples.

Detailed Usage

World Subsystem

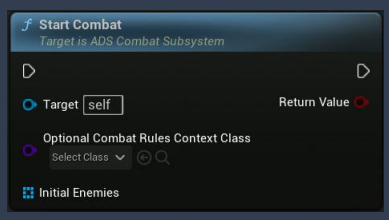
Access



As a World Subsystem, the ADS Combat Subsystem can be accessed from Blueprints related to the game world. It centralizes combat information for the current world.

To access it in a Blueprint graph, right-click and search for "*Combat Subsystem*" to get the reference to the instance used by the current world.

Subsystem Functions



Starts combat with a list of enemies for that combat.

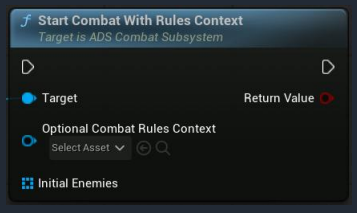
It is not required to start combat to fight. Combat management is optional. Collisions notify the project -> the project reacts. So even without explicit combat explicit, it is possible to attack or be attacked, and even to receive rewards after defeating an enemy.

When combat is explicitly started, it is considered finished when all enemies in the list are defeated, or when all allies are defeated.

Explicitly starting combat is mainly useful in certain cases, such as turn-based combat.

If combat is already running when the function is called, it will not start another combat. You must wait until the current combat is finished before starting another one.

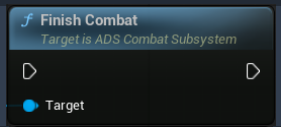
If no valid enemy is provided, combat enters the Pendingstate. It then waits for a valid enemy to be added in order to become active. If no enemy is added quickly, startup is automatically canceled.



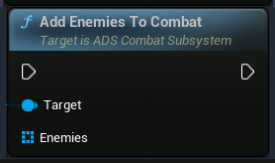
Starts combat with defined rules and the list of enemies for that combat.

Useful when combat must follow specific rules, such as turn-based combat, enemy waves, boss fights, timed combat, scoring combat, etc.

If combat is already running when the function is called, it will not start another combat. You must wait until the current combat is finished before starting another one.

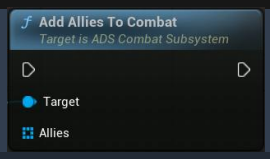


Ends combat.



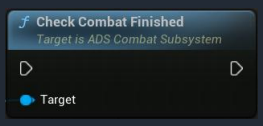
Adds enemies to the active combat.

If no combat is active or pending, the function calls Start Combat with those enemies.



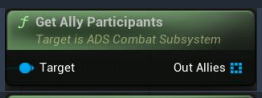
Adds allies to active or pending combat.

If no combat is active or pending, the call is ignored.



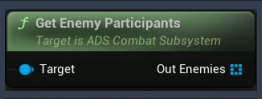
Checks whether the current combat can be ended. If so, the function calls FinishCombat.

If bDecideFinish is enabled in the Combat Rules Context, FinishCombat will not be called. You will need to call it manually.

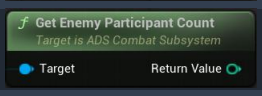


Returns the list of references to allies participating in combat.

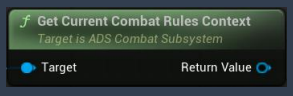
Returns the number of allies participating in combat.



Returns the list of references to enemies participating in combat.

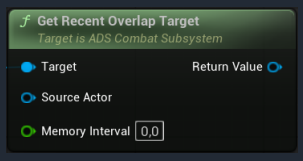


Returns the number of enemies participating in combat.



Returns the reference to the Combat Rules Context instance currently used by the active combat.

If no rules context is active, the function returns None.

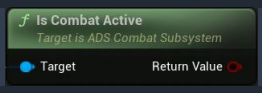


Returns the last target recently hit by the Source Actor, if it is still valid.

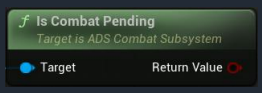
Source Actor is the actor that generated the collision.

Memory Interval defines the duration, in seconds, during which this target remains remembered.

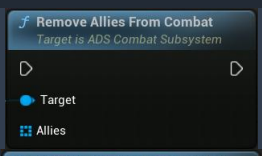
The output returns the found target, or None if no valid target exists.



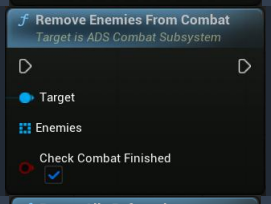
Returns whether combat is currently active.



Returns True when combat is waiting for activation. This is notably the case after Start Combat without a valid enemy, or while a Combat Rules Context prepares the enemies to add.

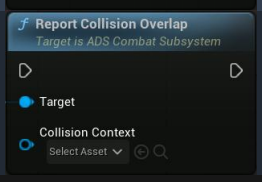


Removes a list of allies from the current combat, then checks whether combat should end.

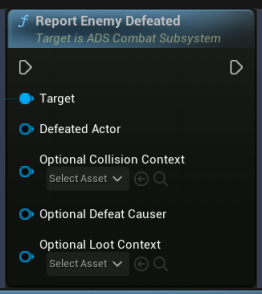


Removes a list of enemies from the current combat. If `Check Combat Finished` is enabled, the system then checks whether combat should end.

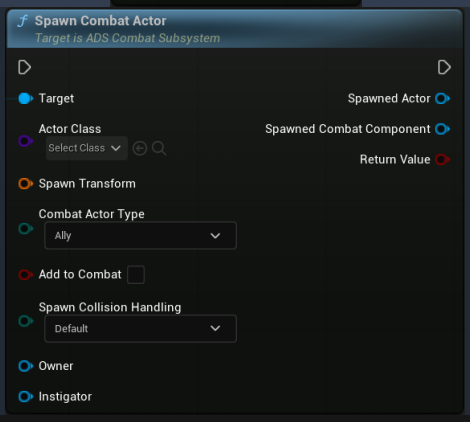
Reports to the system that an ally has been defeated. The function triggers the corresponding Event Dispatcher, removes the ally from combat, then checks whether combat should end.



Reports to the system that an ADS collision has been validated. The function sends the `Collision Context`, stores the last target hit by the source actor, then triggers the corresponding Event Dispatcher.



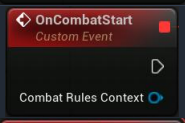
Reports to the system that an enemy has been defeated. The function triggers the corresponding Event Dispatcher, duplicates the provided Loot Context if needed, removes the enemy from combat, then checks whether combat should end.



Spawns an actor at runtime. If the actor has an `ADS Combat Component`, the function can set its combatant type and automatically add it to combat.

Event Dispatchers of the Subsystem

Called when an ADS collision triggers a valid BeginOverlap. It passes a context for the relevant collision.



Called when combat starts. It passes the rules context of the combat that just started.

Called when combat ends. It passes the rules context of the combat that just ended.

Called when an ally is defeated. It passes the reference to the defeated actor, a reference to the actor causing this defeat, and the collision context.

Called when an enemy is defeated. It passes the reference to the defeated actor, a reference to the actor causing this defeat, and the collision context.

Combat Actor Component

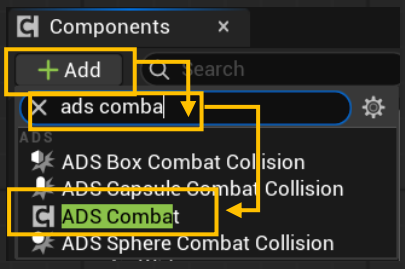
Description

Used to define and configure an actor that can participate in combat: player, enemy, NPC, summon, or any other compatible actor.

It defines the actor role in the system, adds it to or removes it from combat, reports its defeat, and provides actor-specific information to the system.

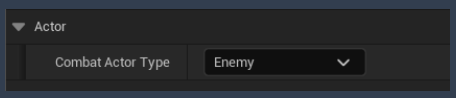
Installing

- Open the Blueprint of the relevant actor
- In the "Components" panel, click "+ Add", then search for and add "ADS Combat".



Configuring

Once the component has been added, select it and configure it in the Details panel.

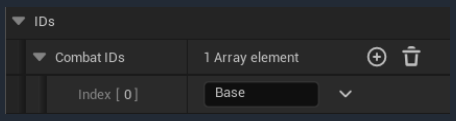


Defines the combatant type used by the system.

Ally : the actor is considered an ally. This is usually also the case for the Player Character.

Enemy : the actor is considered an enemy.

This parameter is used to place the actor in the correct participant list and for collision or targeting filters.



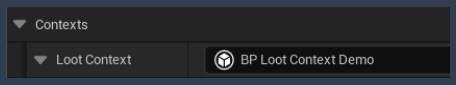
Free identifier list used to characterize this actor for your project logic.

These IDs can be used to identify an enemy type, a group, a creature family, a quest objective, or a specific combat rule.

These IDs can be used by the project, for example by a quest system. Think of these IDs as tags.

Exemple :
Enemy A : IDs -> Troll, IceTroll
Enemy B : IDs -> Troll, FireTroll

A quest may ask the player to kill 6 trolls. Enemies A and B will both be compatible with the quest.
Another quest may ask the player to kill 2 fire trolls; only Enemy B will be compatible with that quest.

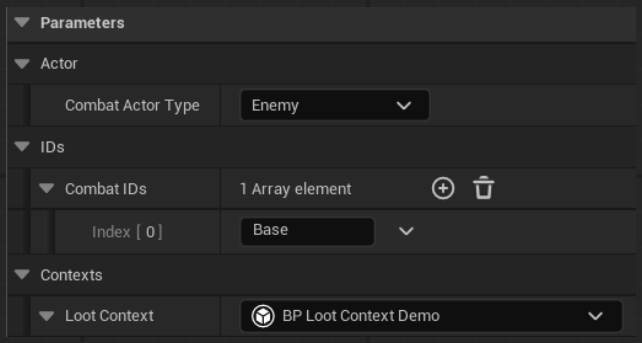


Optional reward context associated with this actor.

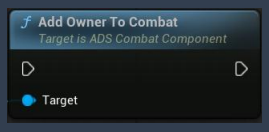
It is mainly used when "Combat Actor Type" = "Enemy". When this actor is defeated, the system passes this context so your project can decide which rewards to grant, for example through your own inventory system.

The context variables are directly exposed in the Details panel just below. This allows you to edit this enemy loot directly in the Detailspanel.

Exemple :



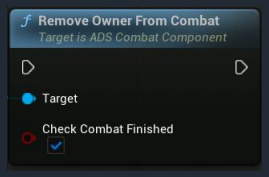
Actor Component Functions



Adds the actor owning this component to combat, according to its Combat Actor Type.

If the actor is Enemy, it is added as an enemy and can start combat if no combat is active or pending.

If the actor is Ally, it is added as an ally only if combat is already active or pending.



Removes the actor owning this component from combat, according to its Combat Actor Type.

If the actor is an enemy, the Check Combat Finished parameter can then check whether combat should end.

If the actor is an ally, the system automatically checks whether combat should end.

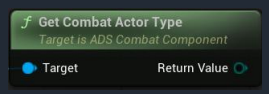
Reports to the system that the actor owning this component has been defeated.

The function uses the Combat Actor Type to automatically call the appropriate logic: enemy defeat or ally defeat. The actor is then removed from combat, and the system checks whether combat should end.

Optional Collision Context passes the collision context related to the defeat.

Optional Defeat Causer specifies the actor responsible for the defeat.

Optional Loot Context provides a specific reward context. If it is not set, the system uses the Loot Context defined in the component, only for an enemy.

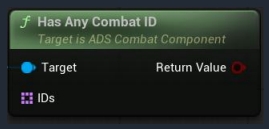


Returns the combatant type defined in the Actor Component.

This value tells whether the actor is considered Ally or Enemy by the system. It is notably used by collisions, targeting, and combat add/remove functions.

Checks whether the actor has a specific Combat ID.

Returns True if the provided ID is present in the Actor Component Combat IDs list. Returns False if the ID is empty or missing.

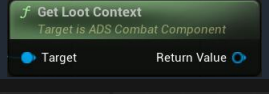


Checks whether the actor has at least one of the provided Combat IDs.

Useful for testing several possible categories. For example, checking whether an enemy is Goblin or Boss.

Checks whether the actor has all the provided Combat IDs.

Useful for testing a precise combination. For example, checking whether an enemy has Goblin, Melee, and QuestTarget.



Returns the Loot Context defined in the Actor Component.

This context can be used when an enemy is defeated to pass reward information to your project logic. If no Loot Context is defined, the function returns None.

Collisions ADS

Description

The plugin provides several collision shapes, such as Box, Sphere, and Capsule, to fit different project needs. These collisions work through overlap and do not replace Unreal Engine physical collisions used for blocking.

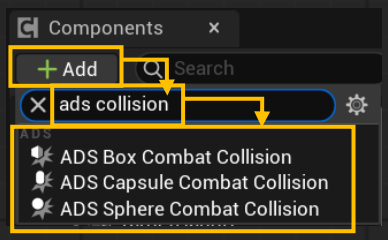
An ADS collision does not automatically decide which damage or effects to apply. When an interaction is validated, it generates a Collision Context containing the important information: source actor, target actor, involved components, Collision ID, Hit Parameter, and optional Hit Result. Your Blueprint logic then decides what to do with this information.

The behavior of an ADS collision is controlled by several filtering layers: ADS channel, ADS responses, hit rules, and target filter. This defines which collisions can interact, how many times a target can be hit, how often, and which actor types should be ignored.

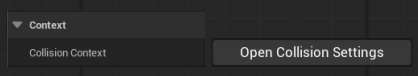
ADS collisions can be activated directly in Blueprint, or automatically during an animation using the Anim Notifies provided by the plugin. They are therefore suitable both for animation-synchronized attacks and for permanent or temporary detection areas.

Adding and Configuring an ADS Collision

These are components that inherit from the native Box Collision, Sphere Collision, and Capsule Collision components. They are added in the same way.



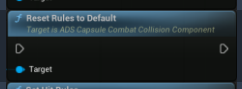
Once added and selected, it can be configured in the Details panel.



Debug Collision Reports	If True, displays diagnostic logs to help understand why an ADS collision is accepted or rejected.
Collision ID	Collision identifier. It must be unique on the actor. Several actors can use the same identifier, but within a single actor each collision must have a unique ID. For example: two actors can have a collision with ID Body, but a single actor must not have two collisions with the Body ID. The plugin does not impose any ID, so you can define any IDs you want.
Auto Active	If True, the collision is automatically activated on BeginPlay. If False, it must be activated manually or through animation notifies.
ADS Channel	Defines the ADS channel of this collision.
Can Overlap Channels	Defines a list of ADS channels that this collision can overlap.
Can Be Overlapped By Channels	Defines a list of ADS channels that can overlap this collision.
Open Collision Settings	Button that opens the Project Settings in the collision section.

	IMPORTANT: The plugin automatically adds ADS channels and ADS presets on startup. You therefore do not need to add them yourself. This button will almost always be unnecessary. However, it is possible that the plugin does not create them the first time. If this happens, restart the project again.
Hit Parameter	Optional hit parameter, for example to define whether the collision is used for a heavy or light attack.
Max Actors	Maximum number of actors the collision can overlap during one activation. -1 = unlimited
Max Hits Per Actor	Number of times the collision can overlap the same actor. <1 = unlimited
Hit Interval Per Actor	Interval between two hits on the same actor.
Ignore Owner	Defines whether the collision ignores its owner.
Ignore Allies	If True, the collision ignores all collisions from actors whose Actor Component defines the "Combat Actor Type" as "Ally".
Ignore Enemies	If True, the collision ignores all collisions from actors whose Actor Component defines the "Combat Actor Type" as "Enemy".
Collision Context	Defines the Collision Context class used by this collision. If set to "None", it will use the default class.

ADS Collision Functions

	Activates the collision with defined hit and filter rules.
	Deactivates the collision.
	Resets the hit registry for this collision to its default values, meaning the values defined in the Details panel.
	Resets this collision rules to their default values, meaning the values defined in the Details panel.
	Sets the hit rules of this collision using the data received by the input structure.
	Sets the filtering rules of this collision using the data received by the input structure.

Collision Context

Description

A Collision Context is a data object used to pass ADS collision information to your Blueprint logic.

It is not an actor to place in the level. The ADS collision can use the base context provided by the plugin, or a custom Blueprint child.

When an ADS collision is validated, the system creates a runtime context instance, fills it with the collision data, then passes it to the Combat Subsystem and Event Dispatchers.

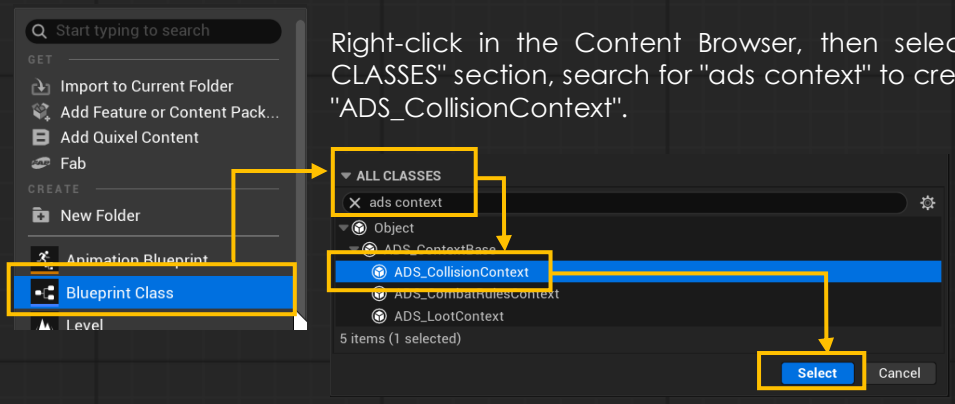
Creating a Blueprint child of ADS_CollisionContext lets you add variables or project-specific logic. For example, a custom context can prepare additional data in Set Data before being passed to Blueprints.

The default context will be enough in 90 to 95% of cases. But when it is not enough, you can extend it through a Blueprint child. For example, the default class can already pass:

Collision ID : Sword / Hit Parameter : HeavyAttack

But if you want it to directly pass: Damage Type = Fire / Guard Break = True / Poise Damage = 30 / Camera Shake = Strong
In that case, creating a Blueprint child will be necessary, and you will need to assign that child to the collision.

Creating a Custom Collision Context

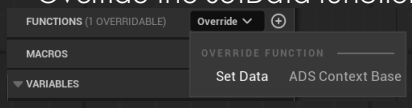


Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_CollisionContext".

Adding Data to a Custom Context

- Open the relevant asset

- Override the SetData function



- Develop the data transmission logic. SetData is called by the system when the context is created at runtime. The transmitted data will therefore be the data defined here.

Rules Context

Description

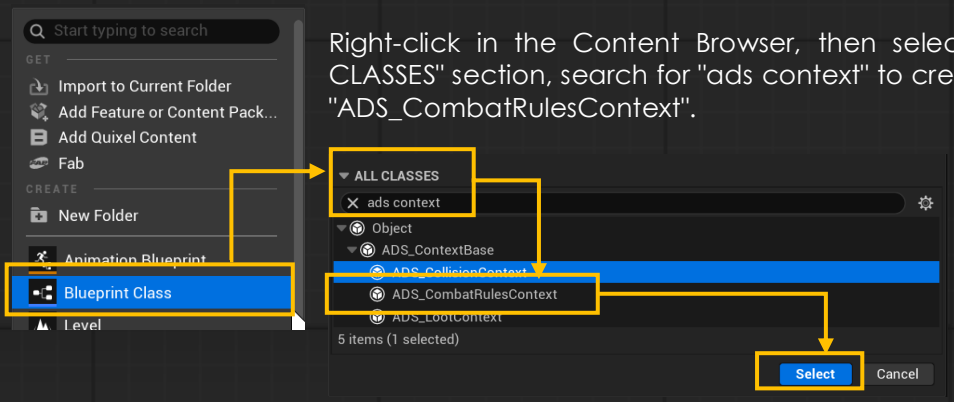
A Rules Context lets you define specific rules for a combat.

It is optional. If no Rules Context is used, combat follows the default system behavior: participants are added to combat, defeats are reported, then combat can end automatically when there are no valid enemies or no valid allies left.

A Rules Context becomes useful when combat must follow a specific logic: turn-based combat, enemy waves, boss fight, timed combat, scoring, custom victory conditions, etc.

It allows the project to centralize the rules specific to that combat without coding them directly in the Combat Subsystem.

Creating a Custom Rules Context

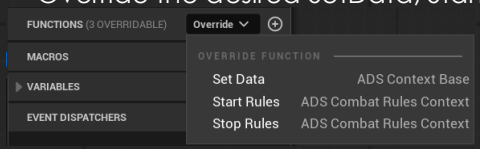


Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_CombatRulesContext".

Creating the Logic of a Custom Rules Context

- Open the relevant asset

- Override the desired SetData, StartRules, and StopRules functions



- Develop the relevant logic

Note that Set Data, Start Rules, and Stop Rules have different responsibilities.

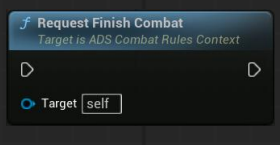
Set Data is used to receive and prepare the data sent to the rules context.

Start Rules is called when the combat rules start. It can be used to initialize combat logic: timer, score, objectives, enemy waves, player state, etc.

Stop Rules is called when combat ends. It is used to clean up or stop the logic set up by the rules: timer, delegates, temporary states, tracking variables, etc.

You do not have to override all three functions. For example, if a rule simply needs to regenerate the player health at the start of combat, Start Rules may be enough.
If combat must end according to a custom rule, such as a timed combat or a specific victory condition, enable the Decide Combat Finish option.

With Decide Combat Finish enabled, the Combat Subsystem does not automatically end combat when there are no valid enemies or allies left. The Rules Context must then call Request Finish Combat when its own end conditions are met.



Loot Context

Description

A Loot Context is an optional context used when an enemy is defeated.
It passes reward-related information for that enemy to your Blueprint logic. The system does not automatically grant rewards: it passes the context, then your project decides what to do with it.

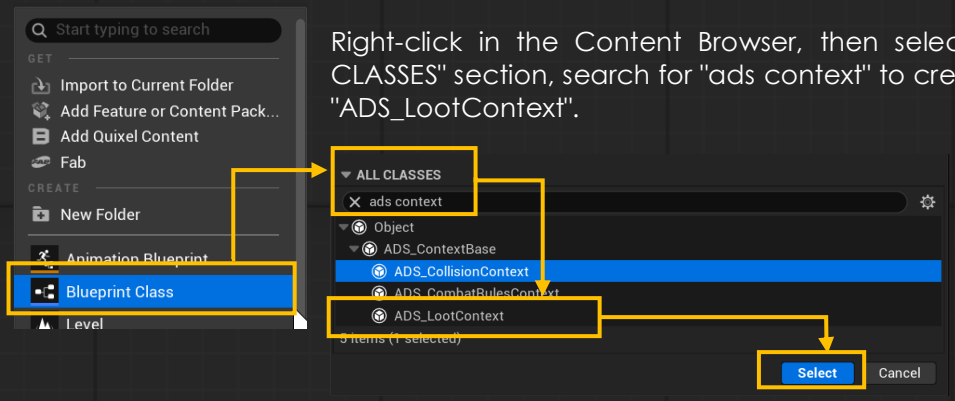
The Loot Context can be defined directly in an enemy ADS Combat Component, or provided when calling Report Owner Defeated / Report Enemy Defeated.

At runtime, the system duplicates this context before passing it to the OnEnemyDefeated Event Dispatcher. This gives each defeat its own context instance.

If no Loot Context is defined, no specific reward data is passed.

In most projects, a single Blueprint child of ADS_LootContext may be enough. Variables added to this custom context are exposed in the ADS Combat Component Details panel, allowing loot to be defined directly on each enemy.

Creating a Custom Loot Context



Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_LootContext".

Once the context has been created, open the asset and create the variables required by the project. These variables will automatically be exposed in the Actor Component Details panel so each enemy actor can define them for itself.

Target Lock Component

Description

The Target Lock Component allows a player-controlled actor to lock onto a target and automatically orient the camera toward it.

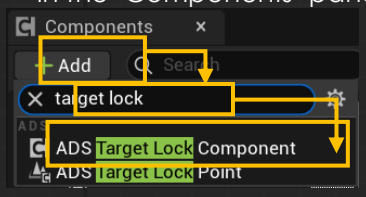
It must be added to the actor that uses lock targeting, usually the Player Character. Actors that can be locked onto must have at least one Target Lock Point Component, which defines a precise point to aim at: head, torso, hand, leg, etc.

The component searches for valid targets in a cone in front of the camera, based on a defined distance and angle. Once a target is locked, it continuously updates the player control rotation to keep the target framed.

Depending on the chosen settings, the system can simply orient the camera toward the target, or adjust framing to keep both the player and the target visible. It can also block camera input while locked, restore the previous rotation when locking ends, and automatically stop the lock if the target becomes invalid or too far away.

Installation

- Open the Player Character
- In the "Components" panel, add the Target Lock Component.

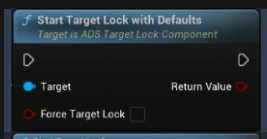
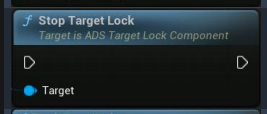
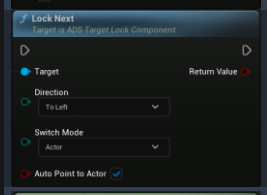
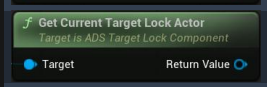
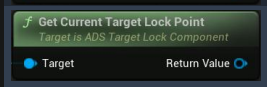


Configuring the Target Lock Component

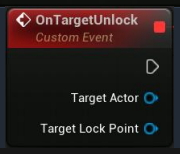
It represents the Lock Targeting system, so configuring this component configures the system for this playable character.

Default Distance	Maximum distance used to search for a Target Lock Point.
Default Auto Unlock Distance	Maximum allowed distance from the locked target. If the target exceeds this distance, locking stops automatically.
Default Angle	Search cone angle, in degrees, relative to the camera forward axis.
Default Target Component Class	Optional filter to target only actors that have a specific component. The aimed position is always provided by a Target Lock Point Component.
Default Interp Speed	Rotation interpolation speed while locked.
Default Lock Camera Input	If enabled, the player camera input is ignored while locked.
Default Restore Previous Control Rotation	If enabled, the previous control rotation is restored when locking stops.
Default Framing Mode	Defines the framing mode used while locked: orient toward the target only, or frame player + target.
Default Framing Bias	Aim point between the player and the target. 0 aims at the player, 1 aims at the Target Lock Point.
Default Minimum Framing Distance	Minimum distance used to calculate framing when the player and target are very close.
Default Framing Height Offset	Vertical offset applied to the point the camera should aim at.
Default Min Camera Distance	Minimum Spring Arm length in Frame Player And Target mode.
Default Max Camera Distance	Maximum Spring Arm length in Frame Player And Target mode.
Default Distance Scale	Multiplier applied to the player-target distance to calculate the desired camera distance.
Default Camera Distance Interp Speed	Interpolation speed used when the Spring Arm length is adjusted.
Default Min Pitch	Minimum vertical angle allowed during framing, to avoid an overly vertical camera.
Default Max Pitch	Maximum vertical angle allowed during framing, to avoid an overly vertical camera.
Default Restore Camera Distance On Stop	If enabled, the initial Spring Arm length is restored when locking stops.

Target Lock Component Functions

	Starts target locking using the default settings defined on this component. Returns True if a valid target is found and locked.
	Searches for a Target Lock Point in front of the camera using the provided distance, angle, and filters, then starts locking if a valid target is found.
	Starts locking onto a specific actor. The target actor must have at least one ADS Target Lock Point. Returns False if the actor is invalid, is the owner, or if no lock point is found.
	Stops the active lock. Depending on the settings, control rotation and/or camera distance can be restored.
	If locking is active, stops it. Otherwise, attempts to start locking with the provided settings.
	Switches target or Target Lock Point in the requested direction. Actor mode switches to another actor; Lock Point mode switches to another point on the current actor.
	Returns True if locking is active and if the target actor and current Target Lock Point are still valid.
	Returns the currently locked actor. Returns None if no actor is locked.
	Returns the Target Lock Point currently used. Returns None if no lock point is active.

Target Lock Component Event Dispatchers

	Called automatically when locking starts or switches to a new actor or Target Lock Point. Passes the new target, the new lock point, as well as the previous target and previous point if locking changed.
	Called automatically when locking stops or when the system leaves a target / Target Lock Point to lock onto another. Passes the actor and point that have just been left.

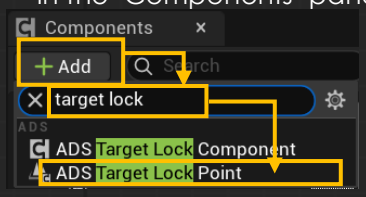
Target Lock Point Component

Description

The Target Lock Point Component is a Scene Component to add to actors that can be locked by the Target Lock Component. It defines a precise point the system can aim at: head, torso, hand, leg, etc. A single actor can have several lock points.

Placing a Target Lock Point Component

- Open the Blueprint of the actor that can be locked
- In the "Components" panel, add the Target Lock Point.



- Position the component where desired and configure it

Configuring a Target Lock Point Component

Can Be Target Locked	Defines whether this point can be used by the lock targeting system. If this option is disabled, the point is ignored.
Lock Point ID	Optional lock point identifier. It can be used to identify the aimed point in project logic or UI, for example Head, Torso, LeftHand, or RightLeg.
Lock Priority	Lock point priority. The higher the value, the more this point is favored when an actor has several valid points. If priorities are equal, the closest point is used.

Anim Notify

Description

The combat system Anim Notifies synchronize combat logic with animations. Anim Notifies execute a one-shot action on a precise animation frame, while Anim Notify States act over a time window between their start and end. They can notably activate ADS collisions, handle combos, i-frames, lock targeting, Blueprint events, spawns, or camera behaviors.

One-shot Anim Notifies:

	Description	Parameters
ADS Activate Collision	Activates the ADS collisions of the actor whose Collision ID matches, starting from the notify frame. The collision remains active until it is deactivated.	Collision ID : identifier of the collision to activate. Hit Rules : hit rules applied during activation. Target Filter : target filter applied during activation. Auto Reset Registry : resets the hit registry before activation.
ADS Deactivate Collision	Deactivates the ADS collisions of the actor whose Collision ID matches.	Collision ID : identifier of the collision to deactivate.
ADS Trigger Combat Event	Calls a Blueprint function or event without parameters on the expected Anim Blueprint and/or on a target Actor Component.	Event Name : name of the function or event to call. Expected Anim BP Class : expected Anim Instance class Target Component Class: target component that should receive the event.
ADS Spawn Combat Actor	Spawns an actor on the notify frame, based on the mesh, a socket, or the component transform.	Actor Class : actor class to spawn. Spawn Socket Name : socket used as the spawn base. Spawn Location Offset : location offset.

		Spawn Rotation Offset : rotation offset. Spawn Scale Multiplier : scale multiplier.
ADS Trigger Level Sequence	Plays a Level Sequence on the notify frame. Can temporarily lock controls during the sequence.	Level Sequence : sequence to play. Play Rate : playback speed. Lock Controls During Sequence : ignores movement and camera inputs during the sequence. Restore Controls On Finished : restores controls at the end of the sequence.
ADS Look At	Immediately rotates the actor toward the best found target: locked target, recently hit target, or nearby valid target.	Search Radius : search radius. Memory Interval : duration for remembering recently hit targets. Use Target Lock If Active : uses the currently locked target if possible. Fallback To Combat Actor : accepts a compatible combat actor if no lock point is found. Yaw Only : limits rotation to Yaw. Target Component Class : optional filter by component.

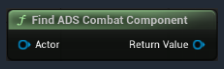
Anim Notify States :

	Description	Parameters
ADS Hit Window	Activates ADS collisions for the full duration of the notify state, then deactivates them at the end and restores the default rules.	Collision ID : identifier of the collision to activate. Hit Rules : temporary hit rules. Target Filter : temporary target filter. Auto Reset Registry : resets the hit registry at the start of the window.
ADS IFrames	Creates an invulnerability window by preventing an ADS collision from receiving interactions for the duration of the state.	Collision ID : identifier of the relevant collision. Restore Previous State : restores the previous collision state at the end of the window.
ADS Combo Window	Opens a combo window. Success can depend on input, a valid hit, or a combination of both. Can trigger an event or montage on success, and apply behavior on failure.	Input Action : expected input. Combo Success Requirement : success condition. Required Collision ID : Collision ID required to validate a hit. Required Hit Parameter : required Hit Parameter. Hit Memory Interval : accepts a recent hit before the window starts. Consume First Valid Hit : keeps the first valid hit. Require Same Source Actor : requires the hit source to be the animated actor. Debug Combo Window : displays diagnostic logs. On Success Event Name : event called on success. On Fail Event Name : event called on failure. Failure Behavior : behavior on failure. Failure Section Name : montage section on failure. Success Montage : montage played on success. Success Montage Section : success montage section. Success Play Rate : success montage playback speed. Expected Anim BP Class : target Anim BP. Target Component Class : target component.

ADS Control Lock	Temporarily locks player controls for the duration of the notify state.	Lock Movement : blocks movement inputs. Lock Rotation : blocks rotation inputs. Lock Camera : blocks camera inputs.
ADS Look At Window	Gradually rotates the actor toward a valid target for the full duration of the notify state.	Search Radius : search radius. Memory Interval : duration for remembering recently hit targets. Interp Speed : rotation interpolation speed. Use Target Lock If Active uses the target locked if possible. Fallback To Combat Actor : accepts a compatible combat actor if no lock point is found. Yaw Only : limits rotation to Yaw. Target Component Class : optional filter by component.
ADS Target Lock	Starts lock targeting for the duration of the notify state through the actor Target Lock Component, then stops it at the end.	Distance : search distance. Angle : search angle. Interp Speed : rotation speed toward the target. Lock Camera Input : blocks camera input during lock. Restore Previous Control Rotation : restores the previous rotation at the end. Force Target Lock : replaces an already active lock if necessary. Target Component Class : optional filter by component.
ADS Spawn Actor Loop	Spawns actors repeatedly for the duration of the notify state, based on an interval and a maximum count.	Actor Class : actor class to spawn. Spawn Socket Name : socket used as the spawn base. Spawn Location Offset : location offset. Spawn Rotation Offset : rotation offset. Spawn Scale Multiplier : scale multiplier. Spawn Interval : delay between two spawns. Spawn On Begin : immediate spawn at the beginning. Max Spawn Count : maximum number of spawns (-1 = unlimited).

Blueprint Functions Library

The plugin includes a C++ Blueprint Functions Library containing generic functions accessible from anywhere.

	Attempts to cast an ADS context to the expected context class. Returns the cast instance if the context matches that class, along with a boolean indicating whether the cast succeeded.
	Returns the reference to the ADS Combat Component of a given actor.
	Searches for an ADS collision in the specified actor with the specified ID. If the collision is found, it returns a reference to it; otherwise, it returns an invalid reference.
	Sets a successful combat window for the specified mesh component with the specified Window Name.

What the System Does Not Do

ADS Combat System provides a technical foundation for creating your own combat system, but it does not automatically define the final rules of your game or a ready-to-use combat system, because it is not a template.

The system does not provide complete ready-to-use combat. It does not decide for you how damage is calculated, how health is managed, how statistics work, which effects are applied, which animations are played, or which rewards are granted.

The plugin also does not replace an inventory, quest, equipment, save, or AI system. It can communicate with these systems, but it does not impose them or generate them automatically.

ADS collisions can detect combat interactions, but they do not automatically apply damage. When a collision is validated, the system passes a Collision Context to your Blueprint logic. Your project then decides what to do with that information.

Combat Rules Contexts allow you to structure specific combat rules, but they do not provide every possible rule for a game. You must define the exact logic yourself: turn-based combat, enemy waves, victory conditions, score, boss fight, rewards, etc.

The lock targeting system provides the tools needed to lock onto a target and orient the camera, but it does not automatically define the associated game design: target switching, restrictions, final camera behavior, or UI integration.

Finally, the demo assets are only usage examples. They do not constitute a complete art direction or a finalized combat system for a released game.

Troubleshooting

A feature may sometimes not react as expected. In most cases, the issue comes from a missing parameter, an actor not registered in combat, a collision not activated, or a context not transmitted.

Combat Does Not Start

Make sure at least one valid enemy is added to combat.

Start Combat can be called without enemies, but combat will then remain in Pending state until a valid enemy is added. If no enemy is added quickly, startup is automatically canceled.

Also make sure no other combat is already active or pending. If combat is already running, Start Combat will not start a new combat.

An Actor Is Not Added to Combat

Make sure the actor has an ADS Combat Component.

Also check its Combat Actor Type:

- Enemy adds the actor as an enemy.
- Ally adds the actor as an ally.

Adding an enemy can automatically start combat. Adding an ally does not automatically start combat: combat must already be active or pending.

Combat Does Not End

Make sure there are no valid enemies or no valid allies left in the participant lists.

Also check the Combat Rules Context. If Decide Combat Finish is enabled, the system does not automatically end combat. In this case, your rules logic must call Request Finish Combat.

An ADS Collision Detects Nothing

Make sure the ADS collision is enabled.

If Auto Activate is disabled, the collision must be activated manually in Blueprint or through an Anim Notify.

Also check:

- the Collision ID;
- the ADS Channel;
- the Hit Other Channels;
- the Can Be Hit By Channels;
- the Ignore Owner, Ignore Allies, and Ignore Enemies filters;
- the Max Hits, Max Hit Per Actor, and Hit Interval Per Actor limits.

The Ignore Allies and Ignore Enemies filters require the relevant actors to have an ADS Combat Component.

A Collision Is Detected but No Effect Occurs

An ADS collision does not automatically cause damage or effects.

When a collision is validated, the system passes a Collision Context. Your Blueprint logic must then use that context to apply damage, trigger feedback, play a sound, launch a VFX, update a quest, or call another system.

Make sure the On Collision Overlap Event Dispatcher is used, or that your logic correctly receives the Collision Context.

Rewards Are Not Granted

Make sure the enemy has a Loot Context in its ADS Combat Component, or that a Loot Context is provided when calling Report Enemy Defeated.

The system does not automatically grant rewards. It passes the loot context to your Blueprint logic, which must then communicate with your inventory, progression, or reward system.

Lock Targeting Cannot Find a Target

Make sure the actor using lock targeting has a Target Lock Component.

Also make sure targets have at least one Target Lock Point Component.

If no target is found, check the maximum distance, search angle, target point class used, and the position of Target Lock Point Components on targetable actors.

An Anim Notify Does Not Produce the Expected Effect

Make sure the Anim Notify is placed at the correct time in the animation.

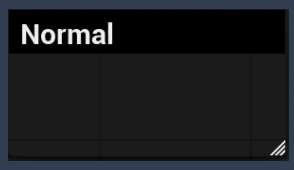
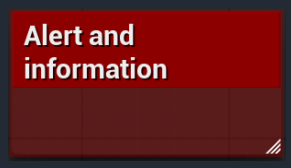
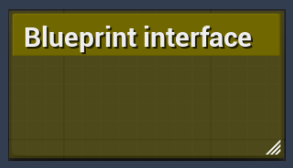
For collision-related notifies, make sure the Collision ID matches an ADS collision present on the relevant actor.

For collision or combo windows, make sure the Notify State duration covers the portion of the animation during which the action should be active.

Conventions

Comments

This Framework uses color coding for Blueprint comments:

	Standard comment
	For warning and informational messages. Most of the time, these will be comments in demo Blueprints warning that this is only an example and that in real projects you will need to do better.
	Blueprint Interface event

UI, Menu, and Widget

Most Unreal Engine users prefix all Blueprint Widgets with "WBP_" without distinguishing their role. This Framework uses a different rule:

- "**UI_**": For user interfaces with no direct interaction with the player.
  Exemple : *UI_Health*

- "**MENU_**": For user interfaces the player can directly interact with.
  Exemple : *MENU_Save*

- "**W_**": Widget used as part of a UI or menu.
  Exemple : *W_SaveSlot*

Contact & Support

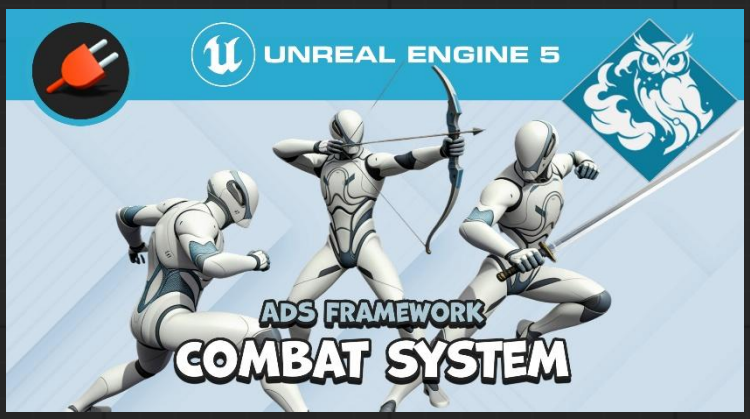
Azure Dream Studio is an independent organization run by Benjamin, who handles development, documentation, training, and support.

If you have a request or comment about this system, you can contact me by email or via Discord. I will reply as soon as possible. Please keep the time difference in mind. I do my best to reply as quickly as possible, but depending on your time zone, several hours may pass between the moment you send your message and the moment I can read it.

Please use only French or English for your messages. Requests sent in any other language cannot be processed.

azure.dream.studio@gmail.com
Please use the following format in the email subject line:
"`[ADS {System Name}]{Your subject}`" as the email subject.
Example: "`[ADS Gameplay Ability System] Need information`"

https://discord.gg/Vy3TvsjKpQ



V1.0

Table of Contents

Foreword

IMPORTANT

ADS Framework is a modular framework for Unreal Engine. **It is not a complete game** template.

A template usually provides a ready-to-use foundation with a predefined gameplay structure. This is convenient for getting started quickly, but it can also limit the project to a specific logic, genre, or architecture.

A framework works differently. It provides reusable, configurable, and extensible tools designed to help you build your own game without imposing a specific type of gameplay. ADS Framework therefore does not replace project-specific development : it provides a technical foundation to save time, structure your features, and avoid starting from scratch, while remaining reliable, safe, and performant. Each system is designed to integrate into all types of projects: action, adventure, RPG, sports, etc.

ADS Framework is developed and maintained with a commitment to continuous improvement. Despite the care put into development, documentation, and testing, it remains a technical product that may evolve based on user feedback and future versions of Unreal Engine.

If you encounter an issue or limitation, it is recommended to contact support instead of leaving a negative review. Constructive feedback helps identify areas for improvement and move the systems in the right direction.

The ADS Framework Philosophy

ADS Framework is based on a simple idea: providing independent systems that can be used on their own, combined with each other, or combined with your own systems depending on your project needs.

Each Framework system can be integrated individually and does not require purchasing or using the other systems to function. You can therefore use it with your own architecture, your own Blueprints, your own interfaces, or other assets purchased on Fab.

If you choose to use multiple Framework systems, they share a common philosophy: modularity, clear logic, Blueprint integration, accessible configuration, and separation of responsibilities.

The systems are designed to be accessible to users who already have basic Unreal Engine and Blueprint knowledge, without necessarily requiring an advanced level. Some modules, although relatively simple to use, are still complex; it is therefore normal to allow a few days of familiarization to understand their logic, settings, and integration.

The included demos are intentionally lightweight. They use simple, readable logic to show how the systems work without unnecessarily increasing project size. They demonstrate functionality, possibilities, and ease of use, without providing final visuals. This choice limits project size and keeps the focus on the systems logic.

C++ / Blueprints

ADS Framework systems are primarily developed in C++ to provide a stable, high-performance, structured foundation suitable for production use.

Each system also exposes a Blueprint layer designed to make integration into your projects easier. This approach lets you benefit from a C++ architecture while keeping the system accessible from the Unreal Engine editor.

Depending on the system, some parts can be configured, called, or extended directly in Blueprints. Project-specific elements can therefore be created in Blueprints, or developed in C++ if you prefer working with native logic.

The goal is to provide a balance between robustness, performance, accessibility, and flexibility.

Compatibility with Unreal Engine 6

ADS Framework is developed with a long-term maintenance and evolution mindset.

Epic Games has presented Unreal Engine 6 as a major evolution of the Unreal Engine ecosystem, with a stronger role for Verse and the new Scene Graph gameplay framework. This evolution may gradually change the way certain gameplay logic is developed in future versions of the engine.

With this in mind, ADS systems will continue to be maintained and adapted as much as possible to Unreal Engine evolutions. When the UE6 / Verse ecosystem is available, documented, and stable enough, the relevant Blueprint parts may gradually be adapted or replaced with solutions compatible with this new architecture.

The goal is to preserve the logic of the ADS systems while keeping pace with the engine evolution.

System Description

ADS Combat System is a toolbox designed to help you create your own combat system in Unreal Engine. It is not a complete ready-to-use combat system with predefined rules, damage, statistics, inventory, or behaviors.

The plugin provides a simple, reliable, safe, and high-performance technical foundation to structure your combat logic. It provides a World Subsystem to centralize combat management, an Actor Component to add to combatants, ready-to-use collision classes, configurable animation notifiesa lock targetingsystem, as well as Blueprint classes for combat rules and rewards.

The system is designed to be universal. It can serve as the foundation for real-time, turn-based, hybrid, action-oriented, RPG, boss fight, enemy wave, or any other combat logic specific to your project.

ADS Combat System does not impose any inventory, quest, equipment, or save system. It can work with your own architecture, with other Fab assets, or with other ADS Framework systems if your project needs them.

This is the ADS Framework system that requires the most project-specific work. The plugin provides tools and entry points, but the final combat rules, damage, effects, statistics, rewards, animations, and behaviors must be defined by your project.

Installation

The plugin must be installed in the project. Do not use the version installed in the engine, because it will need to contain project-specific assets and configurations.

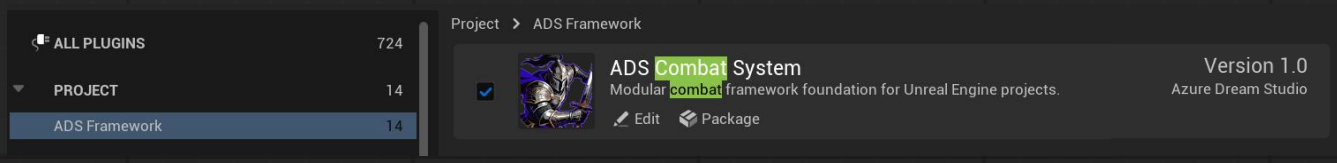
4- After installing the plugin in the engine via the Epic Games Launcher, close Unreal Engine and, from Windows File Explorer, copy the `plugin` folder located in the engine plugins folder to the project `Plugins` folder.

`"<UE installation folder>/Engine/Plugins/<ADS plugin folder>"`

`"<Project folder>/Plugins/ "`

5- Open Unreal Engine.
(If Unreal Engine asks to compile the plugin on startup, accept.)

6- Enable the plugin located in "**PROJECT/ADS Framework**", then restart



The plugin can be installed both in the engine and in the project. If so, since there will be two versions of the same plugin, Unreal Engine will use the one located in the project. After copying the plugin into the project, you can delete it from the engine if you want. In all cases, you should always use the version installed in the project, not the engine version.

How It Works

ADS Combat System is based on a clear separation of responsibilities.

The Combat Subsystem is the central point of the system. It knows the participants registered in combat, manages global combat information, stores the combat rules if provided, and exposes events that allow the project to react.

Each combatant must have a Combat Actor Component. This component is the link between the actor and the central system. It configures the combatant, registers it in combat, defines its role, side, or parameters, and communicates with the subsystem.

The rules specific to a combat are not hard-coded directly in the subsystem. They can be defined in dedicated Blueprint classes. This allows each combat to have its own logic: standard combat, timed combat, enemy waves, boss fights, turn-based combat, or any other project-specific rule.

The ADS collisions are used to detect combat interactions: attack, impact, hitbox, hurtbox, projectile, area of effect, or any other logic defined by the project. They can be activated directly in Blueprint or synchronized with an animation using the animation notifies provided by the plugin.

The animation notifies allow precise windows to be triggered during an animation: hit window, combo window, invulnerability frames, lock targeting, or collision-specific behavior. They are mainly used to synchronize combat logic with animation.

When a collision or combat action is validated, the system sends the useful information to the project. Your Blueprint logic then decides what to do: apply damage, block, parry, knock back, trigger feedback, grant a reward, update a quest, or call another system.

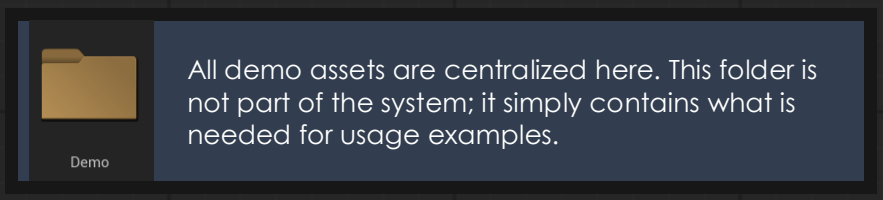
The lock targeting works with target components that can be added to actors. This makes it possible to define several lock points on the same actor, such as the head, torso, hands, or legs.

This architecture allows the plugin to remain universal. The system provides the structure, tools, and events, while your project remains responsible for the final combat rules.

Plugin Contents

This documentation is intended as a user guide. C++ classes will therefore not be detailed in this documentation. They will only be mentioned when they serve as parent classes or when they need to be called from Blueprint. The system is developed in C++, but its main usage is in Blueprints. This documentation therefore focuses on integrating and using the system.

The plugin has its own folder 'Content' for Blueprint assets. It contains only one folder, "Demo".

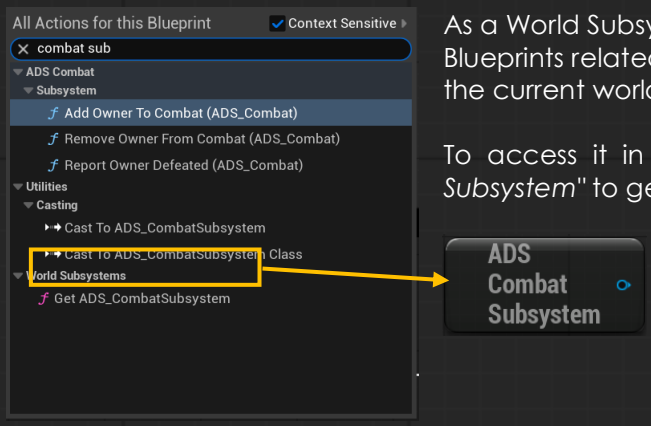


All demo assets are centralized here. This folder is not part of the system; it simply contains what is needed for usage examples.

Detailed Usage

World Subsystem

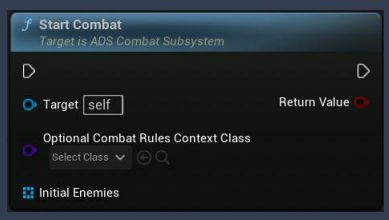
Access



As a World Subsystem, the ADS Combat Subsystem can be accessed from Blueprints related to the game world. It centralizes combat information for the current world.

To access it in a Blueprint graph, right-click and search for "*Combat Subsystem*" to get the reference to the instance used by the current world.

Subsystem Functions



Starts combat with a list of enemies for that combat.

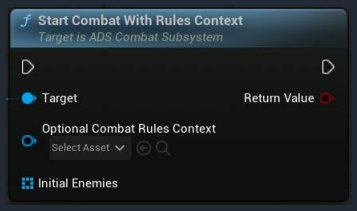
It is not required to start combat to fight. Combat management is optional. Collisions notify the project -> the project reacts. So even without explicit combat explicit, it is possible to attack or be attacked, and even to receive rewards after defeating an enemy.

When combat is explicitly started, it is considered finished when all enemies in the list are defeated, or when all allies are defeated.

Explicitly starting combat is mainly useful in certain cases, such as turn-based combat.

If combat is already running when the function is called, it will not start another combat. You must wait until the current combat is finished before starting another one.

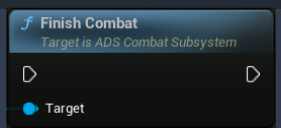
If no valid enemy is provided, combat enters the Pendingstate. It then waits for a valid enemy to be added in order to become active. If no enemy is added quickly, startup is automatically canceled.



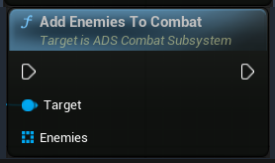
Starts combat with defined rules and the list of enemies for that combat.

Useful when combat must follow specific rules, such as turn-based combat, enemy waves, boss fights, timed combat, scoring combat, etc.

If combat is already running when the function is called, it will not start another combat. You must wait until the current combat is finished before starting another one.

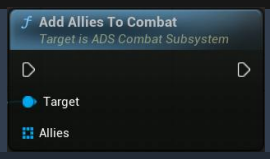


Ends combat.



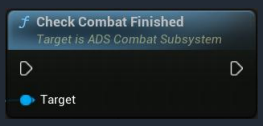
Adds enemies to the active combat.

If no combat is active or pending, the function calls Start Combat with those enemies.



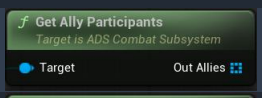
Adds allies to active or pending combat.

If no combat is active or pending, the call is ignored.



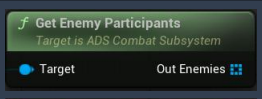
Checks whether the current combat can be ended. If so, the function calls FinishCombat.

If bDecideFinish is enabled in the Combat Rules Context, FinishCombat will not be called. You will need to call it manually.

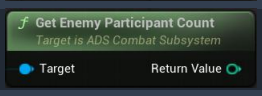


Returns the list of references to allies participating in combat.

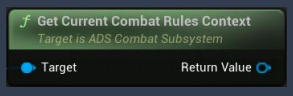
Returns the number of allies participating in combat.



Returns the list of references to enemies participating in combat.

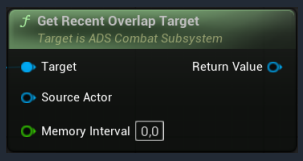


Returns the number of enemies participating in combat.



Returns the reference to the Combat Rules Context instance currently used by the active combat.

If no rules context is active, the function returns None.

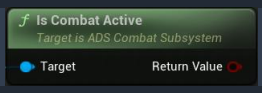


Returns the last target recently hit by the Source Actor, if it is still valid.

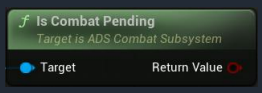
Source Actor is the actor that generated the collision.

Memory Interval defines the duration, in seconds, during which this target remains remembered.

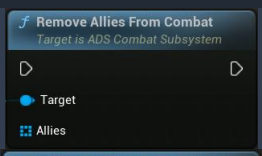
The output returns the found target, or None if no valid target exists.



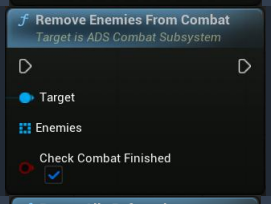
Returns whether combat is currently active.



Returns True when combat is waiting for activation. This is notably the case after Start Combat without a valid enemy, or while a Combat Rules Context prepares the enemies to add.

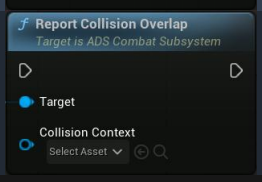


Removes a list of allies from the current combat, then checks whether combat should end.

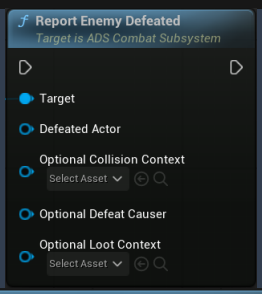


Removes a list of enemies from the current combat. If `Check Combat Finished` is enabled, the system then checks whether combat should end.

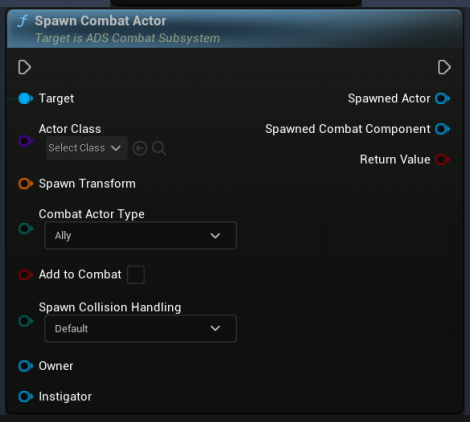
Reports to the system that an ally has been defeated. The function triggers the corresponding Event Dispatcher, removes the ally from combat, then checks whether combat should end.



Reports to the system that an ADS collision has been validated. The function sends the `Collision Context`, stores the last target hit by the source actor, then triggers the corresponding Event Dispatcher.



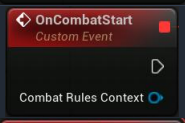
Reports to the system that an enemy has been defeated. The function triggers the corresponding Event Dispatcher, duplicates the provided Loot Context if needed, removes the enemy from combat, then checks whether combat should end.



Spawns an actor at runtime. If the actor has an `ADS Combat Component`, the function can set its combatant type and automatically add it to combat.

Event Dispatchers of the Subsystem

Called when an ADS collision triggers a valid BeginOverlap. It passes a context for the relevant collision.



Called when combat starts. It passes the rules context of the combat that just started.

Called when combat ends. It passes the rules context of the combat that just ended.

Called when an ally is defeated. It passes the reference to the defeated actor, a reference to the actor causing this defeat, and the collision context.

Called when an enemy is defeated. It passes the reference to the defeated actor, a reference to the actor causing this defeat, and the collision context.

Combat Actor Component

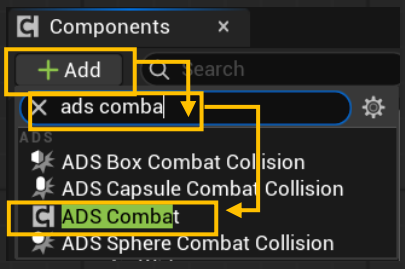
Description

Used to define and configure an actor that can participate in combat: player, enemy, NPC, summon, or any other compatible actor.

It defines the actor role in the system, adds it to or removes it from combat, reports its defeat, and provides actor-specific information to the system.

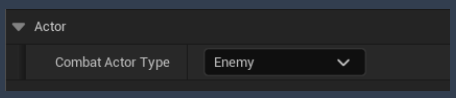
Installing

- Open the Blueprint of the relevant actor
- In the "Components" panel, click "+ Add", then search for and add "ADS Combat".



Configuring

Once the component has been added, select it and configure it in the Details panel.

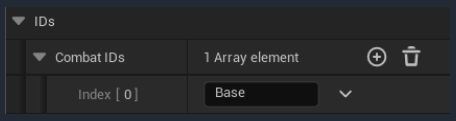


Defines the combatant type used by the system.

Ally : the actor is considered an ally. This is usually also the case for the Player Character.

Enemy : the actor is considered an enemy.

This parameter is used to place the actor in the correct participant list and for collision or targeting filters.



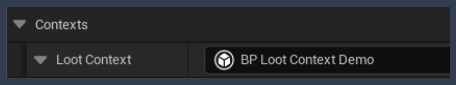
Free identifier list used to characterize this actor for your project logic.

These IDs can be used to identify an enemy type, a group, a creature family, a quest objective, or a specific combat rule.

These IDs can be used by the project, for example by a quest system. Think of these IDs as tags.

Exemple :
Enemy A : IDs -> Troll, IceTroll
Enemy B : IDs -> Troll, FireTroll

A quest may ask the player to kill 6 trolls. Enemies A and B will both be compatible with the quest.
Another quest may ask the player to kill 2 fire trolls; only Enemy B will be compatible with that quest.

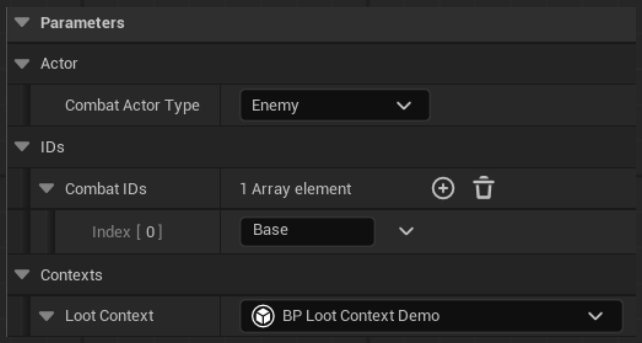


Optional reward context associated with this actor.

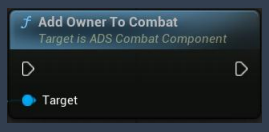
It is mainly used when "Combat Actor Type" = "Enemy". When this actor is defeated, the system passes this context so your project can decide which rewards to grant, for example through your own inventory system.

The context variables are directly exposed in the Details panel just below. This allows you to edit this enemy loot directly in the Detailspanel.

Exemple :



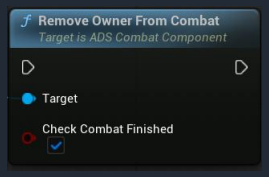
Actor Component Functions



Adds the actor owning this component to combat, according to its Combat Actor Type.

If the actor is Enemy, it is added as an enemy and can start combat if no combat is active or pending.

If the actor is Ally, it is added as an ally only if combat is already active or pending.



Removes the actor owning this component from combat, according to its Combat Actor Type.

If the actor is an enemy, the Check Combat Finished parameter can then check whether combat should end.

If the actor is an ally, the system automatically checks whether combat should end.

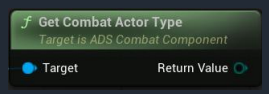
Reports to the system that the actor owning this component has been defeated.

The function uses the Combat Actor Type to automatically call the appropriate logic: enemy defeat or ally defeat. The actor is then removed from combat, and the system checks whether combat should end.

Optional Collision Context passes the collision context related to the defeat.

Optional Defeat Causer specifies the actor responsible for the defeat.

Optional Loot Context provides a specific reward context. If it is not set, the system uses the Loot Context defined in the component, only for an enemy.

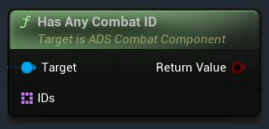


Returns the combatant type defined in the Actor Component.

This value tells whether the actor is considered Ally or Enemy by the system. It is notably used by collisions, targeting, and combat add/remove functions.

Checks whether the actor has a specific Combat ID.

Returns True if the provided ID is present in the Actor Component Combat IDs list. Returns False if the ID is empty or missing.

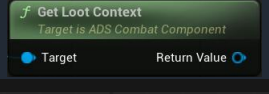


Checks whether the actor has at least one of the provided Combat IDs.

Useful for testing several possible categories. For example, checking whether an enemy is Goblin or Boss.

Checks whether the actor has all the provided Combat IDs.

Useful for testing a precise combination. For example, checking whether an enemy has Goblin, Melee, and QuestTarget.



Returns the Loot Context defined in the Actor Component.

This context can be used when an enemy is defeated to pass reward information to your project logic. If no Loot Context is defined, the function returns None.

Collisions ADS

Description

The plugin provides several collision shapes, such as Box, Sphere, and Capsule, to fit different project needs. These collisions work through overlap and do not replace Unreal Engine physical collisions used for blocking.

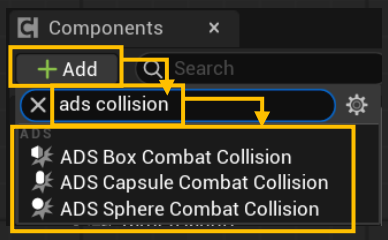
An ADS collision does not automatically decide which damage or effects to apply. When an interaction is validated, it generates a Collision Context containing the important information: source actor, target actor, involved components, Collision ID, Hit Parameter, and optional Hit Result. Your Blueprint logic then decides what to do with this information.

The behavior of an ADS collision is controlled by several filtering layers: ADS channel, ADS responses, hit rules, and target filter. This defines which collisions can interact, how many times a target can be hit, how often, and which actor types should be ignored.

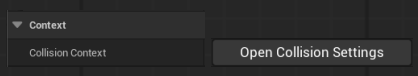
ADS collisions can be activated directly in Blueprint, or automatically during an animation using the Anim Notifies provided by the plugin. They are therefore suitable both for animation-synchronized attacks and for permanent or temporary detection areas.

Adding and Configuring an ADS Collision

These are components that inherit from the native Box Collision, Sphere Collision, and Capsule Collision components. They are added in the same way.



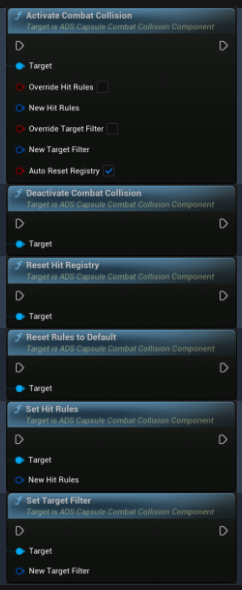
Once added and selected, it can be configured in the Details panel.



Debug Collision Reports	If True, displays diagnostic logs to help understand why an ADS collision is accepted or rejected.
Collision ID	Collision identifier. It must be unique on the actor. Several actors can use the same identifier, but within a single actor each collision must have a unique ID. For example: two actors can have a collision with ID Body, but a single actor must not have two collisions with the Body ID. The plugin does not impose any ID, so you can define any IDs you want.
Auto Active	If True, the collision is automatically activated on BeginPlay. If False, it must be activated manually or through animation notifies.
ADS Channel	Defines the ADS channel of this collision.
Can Overlap Channels	Defines a list of ADS channels that this collision can overlap.
Can Be Overlapped By Channels	Defines a list of ADS channels that can overlap this collision.
Open Collision Settings	Button that opens the Project Settings in the collision section.

	IMPORTANT: The plugin automatically adds ADS channels and ADS presets on startup. You therefore do not need to add them yourself. This button will almost always be unnecessary.
Hit Parameter	Optional hit parameter, for example to define whether the collision is used for a heavy or light attack.
Max Actors	Maximum number of actors the collision can overlap during one activation. -1 = unlimited
Max Hits Per Actor	Number of times the collision can overlap the same actor. <1 = unlimited
Hit Interval Per Actor	Interval between two hits on the same actor.
Ignore Owner	Defines whether the collision ignores its owner.
Ignore Allies	If True, the collision ignores all collisions from actors whose Actor Component defines the "Combat Actor Type" as "Ally".
Ignore Enemies	If True, the collision ignores all collisions from actors whose Actor Component defines the "Combat Actor Type" as "Enemy".
Collision Context	Defines the Collision Context class used by this collision. If set to "None", it will use the default class.

ADS Collision Functions

	Activates the collision with defined hit and filter rules.
	Deactivates the collision.
	Resets the hit registry for this collision to its default values, meaning the values defined in the Details panel.
	Resets this collision rules to their default values, meaning the values defined in the Details panel.
	Sets the hit rules of this collision using the data received by the input structure.
	Sets the filtering rules of this collision using the data received by the input structure.

Collision Context

Description

A Collision Context is a data object used to pass ADS collision information to your Blueprint logic.

It is not an actor to place in the level. The ADS collision can use the base context provided by the plugin, or a custom Blueprint child.

When an ADS collision is validated, the system creates a runtime context instance, fills it with the collision data, then passes it to the Combat Subsystem and Event Dispatchers.

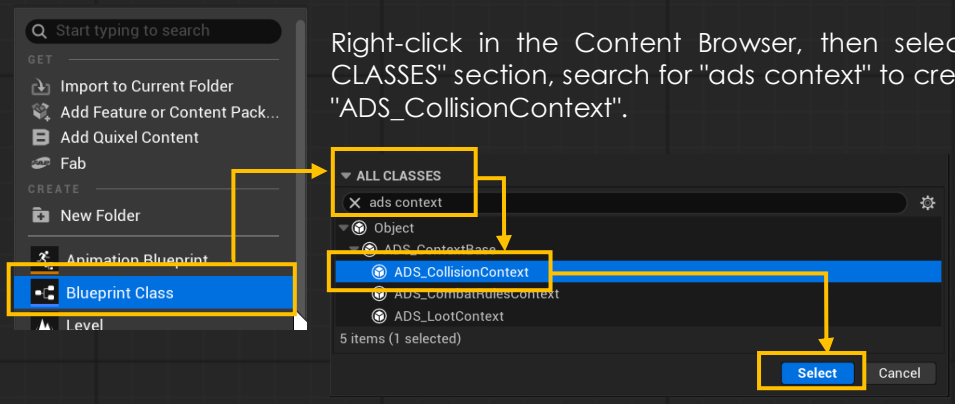
Creating a Blueprint child of ADS_CollisionContext lets you add variables or project-specific logic. For example, a custom context can prepare additional data in Set Data before being passed to Blueprints.

The default context will be enough in 90 to 95% of cases. But when it is not enough, you can extend it through a Blueprint child. For example, the default class can already pass:

Collision ID : Sword / Hit Parameter : HeavyAttack

But if you want it to directly pass: Damage Type = Fire / Guard Break = True / Poise Damage = 30 / Camera Shake = Strong
In that case, creating a Blueprint child will be necessary, and you will need to assign that child to the collision.

Creating a Custom Collision Context

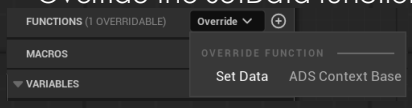


Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_CollisionContext".

Adding Data to a Custom Context

- Open the relevant asset

- Override the SetData function



- Develop the data transmission logic. SetData is called by the system when the context is created at runtime. The transmitted data will therefore be the data defined here.

Rules Context

Description

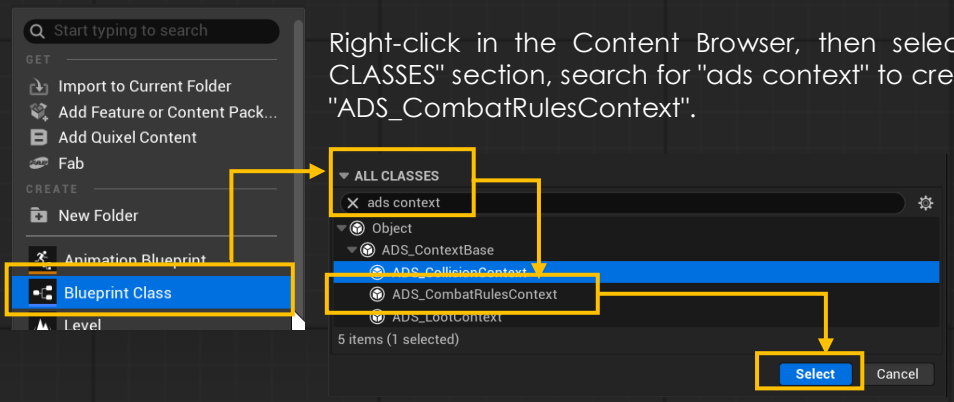
A Rules Context lets you define specific rules for a combat.

It is optional. If no Rules Context is used, combat follows the default system behavior: participants are added to combat, defeats are reported, then combat can end automatically when there are no valid enemies or no valid allies left.

A Rules Context becomes useful when combat must follow a specific logic: turn-based combat, enemy waves, boss fight, timed combat, scoring, custom victory conditions, etc.

It allows the project to centralize the rules specific to that combat without coding them directly in the Combat Subsystem.

Creating a Custom Rules Context

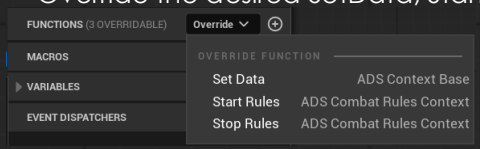


Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_CombatRulesContext".

Creating the Logic of a Custom Rules Context

- Open the relevant asset

- Override the desired SetData, StartRules, and StopRules functions



- Develop the relevant logic

Note that Set Data, Start Rules, and Stop Rules have different responsibilities.

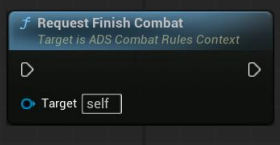
Set Data is used to receive and prepare the data sent to the rules context.

Start Rules is called when the combat rules start. It can be used to initialize combat logic: timer, score, objectives, enemy waves, player state, etc.

Stop Rules is called when combat ends. It is used to clean up or stop the logic set up by the rules: timer, delegates, temporary states, tracking variables, etc.

You do not have to override all three functions. For example, if a rule simply needs to regenerate the player health at the start of combat, Start Rules may be enough.
If combat must end according to a custom rule, such as a timed combat or a specific victory condition, enable the Decide Combat Finish option.

With Decide Combat Finish enabled, the Combat Subsystem does not automatically end combat when there are no valid enemies or allies left. The Rules Context must then call Request Finish Combat when its own end conditions are met.



Loot Context

Description

A Loot Context is an optional context used when an enemy is defeated.
It passes reward-related information for that enemy to your Blueprint logic. The system does not automatically grant rewards: it passes the context, then your project decides what to do with it.

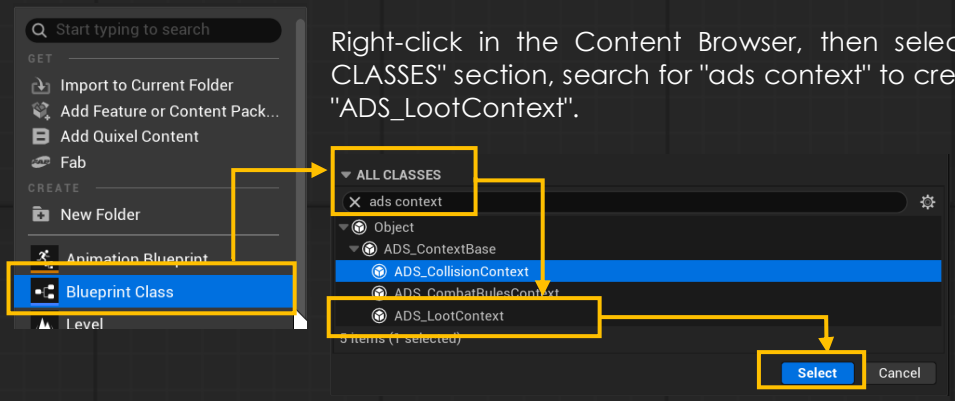
The Loot Context can be defined directly in an enemy ADS Combat Component, or provided when calling Report Owner Defeated / Report Enemy Defeated.

At runtime, the system duplicates this context before passing it to the OnEnemyDefeated Event Dispatcher. This gives each defeat its own context instance.

If no Loot Context is defined, no specific reward data is passed.

In most projects, a single Blueprint child of ADS_LootContext may be enough. Variables added to this custom context are exposed in the ADS Combat Component Details panel, allowing loot to be defined directly on each enemy.

Creating a Custom Loot Context



Right-click in the Content Browser, then select "Blueprint Class". In the "ALL CLASSES" section, search for "ads context" to create a Blueprint that inherits from "ADS_LootContext".

Once the context has been created, open the asset and create the variables required by the project. These variables will automatically be exposed in the Actor Component Details panel so each enemy actor can define them for itself.

Target Lock Component

Description

The Target Lock Component allows a player-controlled actor to lock onto a target and automatically orient the camera toward it.

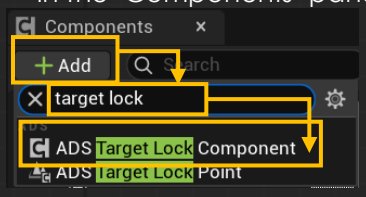
It must be added to the actor that uses lock targeting, usually the Player Character. Actors that can be locked onto must have at least one Target Lock Point Component, which defines a precise point to aim at: head, torso, hand, leg, etc.

The component searches for valid targets in a cone in front of the camera, based on a defined distance and angle. Once a target is locked, it continuously updates the player control rotation to keep the target framed.

Depending on the chosen settings, the system can simply orient the camera toward the target, or adjust framing to keep both the player and the target visible. It can also block camera input while locked, restore the previous rotation when locking ends, and automatically stop the lock if the target becomes invalid or too far away.

Installation

- Open the Player Character
- In the "Components" panel, add the Target Lock Component.

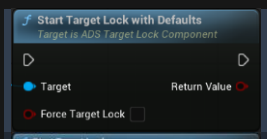
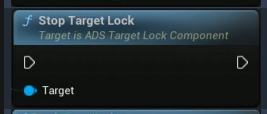
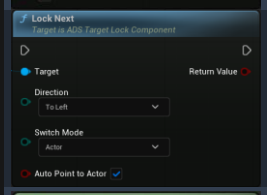
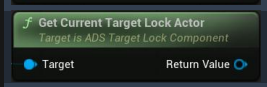
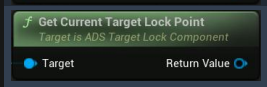


Configuring the Target Lock Component

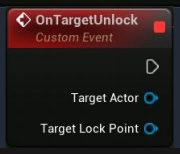
It represents the Lock Targeting system, so configuring this component configures the system for this playable character.

Default Distance	Maximum distance used to search for a Target Lock Point.
Default Auto Unlock Distance	Maximum allowed distance from the locked target. If the target exceeds this distance, locking stops automatically.
Default Angle	Search cone angle, in degrees, relative to the camera forward axis.
Default Target Component Class	Optional filter to target only actors that have a specific component. The aimed position is always provided by a Target Lock Point Component.
Default Interp Speed	Rotation interpolation speed while locked.
Default Lock Camera Input	If enabled, the player camera input is ignored while locked.
Default Restore Previous Control Rotation	If enabled, the previous control rotation is restored when locking stops.
Default Framing Mode	Defines the framing mode used while locked: orient toward the target only, or frame player + target.
Default Framing Bias	Aim point between the player and the target. 0 aims at the player, 1 aims at the Target Lock Point.
Default Minimum Framing Distance	Minimum distance used to calculate framing when the player and target are very close.
Default Framing Height Offset	Vertical offset applied to the point the camera should aim at.
Default Min Camera Distance	Minimum Spring Arm length in Frame Player And Target mode.
Default Max Camera Distance	Maximum Spring Arm length in Frame Player And Target mode.
Default Distance Scale	Multiplier applied to the player-target distance to calculate the desired camera distance.
Default Camera Distance Interp Speed	Interpolation speed used when the Spring Arm length is adjusted.
Default Min Pitch	Minimum vertical angle allowed during framing, to avoid an overly vertical camera.
Default Max Pitch	Maximum vertical angle allowed during framing, to avoid an overly vertical camera.
Default Restore Camera Distance On Stop	If enabled, the initial Spring Arm length is restored when locking stops.

Target Lock Component Functions

	Starts target locking using the default settings defined on this component. Returns True if a valid target is found and locked.
	Searches for a Target Lock Point in front of the camera using the provided distance, angle, and filters, then starts locking if a valid target is found.
	Starts locking onto a specific actor. The target actor must have at least one ADS Target Lock Point. Returns False if the actor is invalid, is the owner, or if no lock point is found.
	Stops the active lock. Depending on the settings, control rotation and/or camera distance can be restored.
	If locking is active, stops it. Otherwise, attempts to start locking with the provided settings.
	Switches target or Target Lock Point in the requested direction. Actor mode switches to another actor; Lock Point mode switches to another point on the current actor.
	Returns True if locking is active and if the target actor and current Target Lock Point are still valid.
	Returns the currently locked actor. Returns None if no actor is locked.
	Returns the Target Lock Point currently used. Returns None if no lock point is active.

Target Lock Component Event Dispatchers

	Called automatically when locking starts or switches to a new actor or Target Lock Point. Passes the new target, the new lock point, as well as the previous target and previous point if locking changed.
	Called automatically when locking stops or when the system leaves a target / Target Lock Point to lock onto another. Passes the actor and point that have just been left.

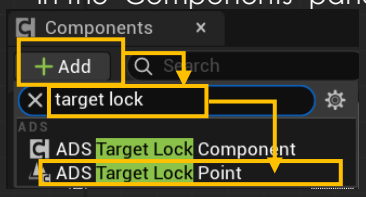
Target Lock Point Component

Description

The Target Lock Point Component is a Scene Component to add to actors that can be locked by the Target Lock Component. It defines a precise point the system can aim at: head, torso, hand, leg, etc. A single actor can have several lock points.

Placing a Target Lock Point Component

- Open the Blueprint of the actor that can be locked
- In the "Components" panel, add the Target Lock Point.



- Position the component where desired and configure it

Configuring a Target Lock Point Component

Can Be Target Locked	Defines whether this point can be used by the lock targeting system. If this option is disabled, the point is ignored.
Lock Point ID	Optional lock point identifier. It can be used to identify the aimed point in project logic or UI, for example Head, Torso, LeftHand, or RightLeg.
Lock Priority	Lock point priority. The higher the value, the more this point is favored when an actor has several valid points. If priorities are equal, the closest point is used.

Anim Notify

Description

The combat system Anim Notifies synchronize combat logic with animations. Anim Notifies execute a one-shot action on a precise animation frame, while Anim Notify States act over a time window between their start and end. They can notably activate ADS collisions, handle combos, i-frames, lock targeting, Blueprint events, spawns, or camera behaviors.

One-shot Anim Notifies:

	Description	Parameters
ADS Activate Collision	Activates the ADS collisions of the actor whose Collision ID matches, starting from the notify frame. The collision remains active until it is deactivated.	Collision ID : identifier of the collision to activate. Hit Rules : hit rules applied during activation. Target Filter : target filter applied during activation. Auto Reset Registry : resets the hit registry before activation.
ADS Deactivate Collision	Deactivates the ADS collisions of the actor whose Collision ID matches.	Collision ID : identifier of the collision to deactivate.
ADS Trigger Combat Event	Calls a Blueprint function or event without parameters on the expected Anim Blueprint and/or on a target Actor Component.	Event Name : name of the function or event to call. Expected Anim BP Class : expected Anim Instance class Target Component Class: target component that should receive the event.
ADS Spawn Combat Actor	Spawns an actor on the notify frame, based on the mesh, a socket, or the component transform.	Actor Class : actor class to spawn. Spawn Socket Name : socket used as the spawn base. Spawn Location Offset : location offset.

		Spawn Rotation Offset : rotation offset. Spawn Scale Multiplier : scale multiplier.
ADS Trigger Level Sequence	Plays a Level Sequence on the notify frame. Can temporarily lock controls during the sequence.	Level Sequence : sequence to play. Play Rate : playback speed. Lock Controls During Sequence : ignores movement and camera inputs during the sequence. Restore Controls On Finished : restores controls at the end of the sequence.
ADS Look At	Immediately rotates the actor toward the best found target: locked target, recently hit target, or nearby valid target.	Search Radius : search radius. Memory Interval : duration for remembering recently hit targets. Use Target Lock If Active : uses the currently locked target if possible. Fallback To Combat Actor : accepts a compatible combat actor if no lock point is found. Yaw Only : limits rotation to Yaw. Target Component Class : optional filter by component.

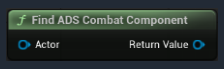
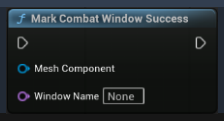
Anim Notify States :

	Description	Parameters
ADS Hit Window	Activates ADS collisions for the full duration of the notify state, then deactivates them at the end and restores the default rules.	Collision ID : identifier of the collision to activate. Hit Rules : temporary hit rules. Target Filter : temporary target filter. Auto Reset Registry : resets the hit registry at the start of the window.
ADS IFrames	Creates an invulnerability window by preventing an ADS collision from receiving interactions for the duration of the state.	Collision ID : identifier of the relevant collision. Restore Previous State : restores the previous collision state at the end of the window.
ADS Combo Window	Opens a combo window. Success can depend on input, a valid hit, or a combination of both. Can trigger an event or montage on success, and apply behavior on failure.	Input Action : expected input. Combo Success Requirement : success condition. Required Collision ID : Collision ID required to validate a hit. Required Hit Parameter : required Hit Parameter. Hit Memory Interval : accepts a recent hit before the window starts. Consume First Valid Hit : keeps the first valid hit. Require Same Source Actor : requires the hit source to be the animated actor. Debug Combo Window : displays diagnostic logs. On Success Event Name : event called on success. On Fail Event Name : event called on failure. Failure Behavior : behavior on failure. Failure Section Name : montage section on failure. Success Montage : montage played on success. Success Montage Section : success montage section. Success Play Rate : success montage playback speed. Expected Anim BP Class : target Anim BP. Target Component Class : target component.

ADS Control Lock	Temporarily locks player controls for the duration of the notify state.	Lock Movement : blocks movement inputs. Lock Rotation : blocks rotation inputs. Lock Camera : blocks camera inputs.
ADS Look At Window	Gradually rotates the actor toward a valid target for the full duration of the notify state.	Search Radius : search radius. Memory Interval : duration for remembering recently hit targets. Interp Speed : rotation interpolation speed. Use Target Lock If Active uses the target locked if possible. Fallback To Combat Actor : accepts a compatible combat actor if no lock point is found. Yaw Only : limits rotation to Yaw. Target Component Class : optional filter by component.
ADS Target Lock	Starts lock targeting for the duration of the notify state through the actor Target Lock Component, then stops it at the end.	Distance : search distance. Angle : search angle. Interp Speed : rotation speed toward the target. Lock Camera Input : blocks camera input during lock. Restore Previous Control Rotation : restores the previous rotation at the end. Force Target Lock : replaces an already active lock if necessary. Target Component Class : optional filter by component.
ADS Spawn Actor Loop	Spawns actors repeatedly for the duration of the notify state, based on an interval and a maximum count.	Actor Class : actor class to spawn. Spawn Socket Name : socket used as the spawn base. Spawn Location Offset : location offset. Spawn Rotation Offset : rotation offset. Spawn Scale Multiplier : scale multiplier. Spawn Interval : delay between two spawns. Spawn On Begin : immediate spawn at the beginning. Max Spawn Count : maximum number of spawns (-1 = unlimited).

Blueprint Functions Library

The plugin includes a C++ Blueprint Functions Library containing generic functions accessible from anywhere.

	Attempts to cast an ADS context to the expected context class. Returns the cast instance if the context matches that class, along with a boolean indicating whether the cast succeeded.
	Returns the reference to the ADS Combat Component of a given actor.
	Searches for an ADS collision in the specified actor with the specified ID. If the collision is found, it returns a reference to it; otherwise, it returns an invalid reference.
	Sets a successful combat window for the specified mesh component with the specified Window Name.

What the System Does Not Do

ADS Combat System provides a technical foundation for creating your own combat system, but it does not automatically define the final rules of your game or a ready-to-use combat system, because it is not a template.

The system does not provide complete ready-to-use combat. It does not decide for you how damage is calculated, how health is managed, how statistics work, which effects are applied, which animations are played, or which rewards are granted.

The plugin also does not replace an inventory, quest, equipment, save, or AI system. It can communicate with these systems, but it does not impose them or generate them automatically.

ADS collisions can detect combat interactions, but they do not automatically apply damage. When a collision is validated, the system passes a Collision Context to your Blueprint logic. Your project then decides what to do with that information.

Combat Rules Contexts allow you to structure specific combat rules, but they do not provide every possible rule for a game. You must define the exact logic yourself: turn-based combat, enemy waves, victory conditions, score, boss fight, rewards, etc.

The lock targeting system provides the tools needed to lock onto a target and orient the camera, but it does not automatically define the associated game design: target switching, restrictions, final camera behavior, or UI integration.

Finally, the demo assets are only usage examples. They do not constitute a complete art direction or a finalized combat system for a released game.

Troubleshooting

A feature may sometimes not react as expected. In most cases, the issue comes from a missing parameter, an actor not registered in combat, a collision not activated, or a context not transmitted.

Combat Does Not Start

Make sure at least one valid enemy is added to combat.

Start Combat can be called without enemies, but combat will then remain in Pending state until a valid enemy is added. If no enemy is added quickly, startup is automatically canceled.

Also make sure no other combat is already active or pending. If combat is already running, Start Combat will not start a new combat.

An Actor Is Not Added to Combat

Make sure the actor has an ADS Combat Component.

Also check its Combat Actor Type:

- Enemy adds the actor as an enemy.
- Ally adds the actor as an ally.

Adding an enemy can automatically start combat. Adding an ally does not automatically start combat: combat must already be active or pending.

Combat Does Not End

Make sure there are no valid enemies or no valid allies left in the participant lists.

Also check the Combat Rules Context. If Decide Combat Finish is enabled, the system does not automatically end combat. In this case, your rules logic must call Request Finish Combat.

An ADS Collision Detects Nothing

Make sure the ADS collision is enabled.

If Auto Activate is disabled, the collision must be activated manually in Blueprint or through an Anim Notify.

Also check:

- the Collision ID;
- the ADS Channel;
- the Hit Other Channels;
- the Can Be Hit By Channels;
- the Ignore Owner, Ignore Allies, and Ignore Enemies filters;
- the Max Hits, Max Hit Per Actor, and Hit Interval Per Actor limits.

The Ignore Allies and Ignore Enemies filters require the relevant actors to have an ADS Combat Component.

A Collision Is Detected but No Effect Occurs

An ADS collision does not automatically cause damage or effects.

When a collision is validated, the system passes a Collision Context. Your Blueprint logic must then use that context to apply damage, trigger feedback, play a sound, launch a VFX, update a quest, or call another system.

Make sure the On Collision Overlap Event Dispatcher is used, or that your logic correctly receives the Collision Context.

Rewards Are Not Granted

Make sure the enemy has a Loot Context in its ADS Combat Component, or that a Loot Context is provided when calling Report Enemy Defeated.

The system does not automatically grant rewards. It passes the loot context to your Blueprint logic, which must then communicate with your inventory, progression, or reward system.

Lock Targeting Cannot Find a Target

Make sure the actor using lock targeting has a Target Lock Component.

Also make sure targets have at least one Target Lock Point Component.

If no target is found, check the maximum distance, search angle, target point class used, and the position of Target Lock Point Components on targetable actors.

An Anim Notify Does Not Produce the Expected Effect

Make sure the Anim Notify is placed at the correct time in the animation.

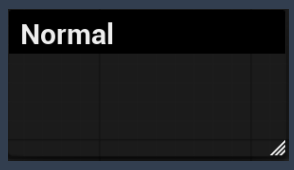
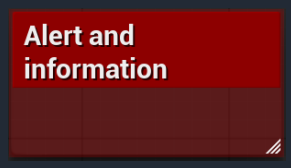
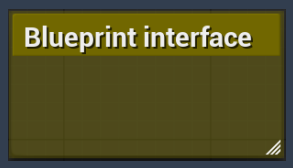
For collision-related notifies, make sure the Collision ID matches an ADS collision present on the relevant actor.

For collision or combo windows, make sure the Notify State duration covers the portion of the animation during which the action should be active.

Conventions

Comments

This Framework uses color coding for Blueprint comments:

	Standard comment
	For warning and informational messages. Most of the time, these will be comments in demo Blueprints warning that this is only an example and that in real projects you will need to do better.
	Blueprint Interface event

UI, Menu, and Widget

Most Unreal Engine users prefix all Blueprint Widgets with "WBP_" without distinguishing their role. This Framework uses a different rule:

- "**UI_**": For user interfaces with no direct interaction with the player.
  Exemple : *UI_Health*

- "**MENU_**": For user interfaces the player can directly interact with.
  Exemple : *MENU_Save*

- "**W_**": Widget used as part of a UI or menu.
  Exemple : *W_SaveSlot*

Contact & Support

Azure Dream Studio is an independent organization run by Benjamin, who handles development, documentation, training, and support.

If you have a request or comment about this system, you can contact me by email or via Discord. I will reply as soon as possible. Please keep the time difference in mind. I do my best to reply as quickly as possible, but depending on your time zone, several hours may pass between the moment you send your message and the moment I can read it.

Please use only French or English for your messages. Requests sent in any other language cannot be processed.

azure.dream.studio@gmail.com
Please use the following format in the email subject line:
"`[ADS {System Name}]{Your subject}`" as the email subject.
Example: "`[ADS Gameplay Ability System] Need information`"

https://discord.gg/Vy3TvsjKpQ