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 typically provides a ready-to-use foundation with a predefined gameplay structure. This is useful 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 you with a technical foundation to save time, structure your features, and avoid starting from scratch, all while being reliable, secure, and high-performing. 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 taken in 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 a problem or a limitation, we recommend contacting support instead of leaving a negative review. Constructive feedback helps us identify areas for improvement and steer the systems in the right direction.

The ADS Framework Philosophy

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

Each system in the Framework can be integrated individually and does not require the purchase or use of 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 systems from the Framework, 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 a basic understanding of Unreal Engine and Blueprints, without necessarily having an advanced level of expertise. While some modules are relatively simple to use, they are nonetheless complex; it is therefore normal to allow a few days to familiarize yourself with them in order to understand their logic, settings, and integration.

The provided demos are intentionally lightweight. They use simple, easy-to-follow logic to demonstrate how the systems work without unnecessarily burdening the project. Their purpose is to showcase functionality, possibilities, and ease of use, without providing a polished visual result. This approach keeps project size manageable and focuses attention 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.

However, each system exposes a Blueprint layer designed to facilitate integration into your projects. This approach allows you to benefit from a C++ architecture while maintaining accessibility from the Unreal Engine Editor.

Depending on the system, certain parts can be configured, called, or extended directly in Blueprints. Elements specific to your project can therefore be created in Blueprints, or developed in C++ if you prefer to work with native logic.

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

Compatibility with Unreal Engine 6

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

Epic Games has presented Unreal Engine 6 as a major evolution of the Unreal Engine ecosystem, with a greater emphasis on 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, the ADS systems will continue to be maintained and adapted as much as possible to changes in Unreal Engine. Once the UE6/Verse ecosystem is available, documented, and sufficiently stable, the relevant Blueprints may be gradually 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's evolution.

System Description

ADS Gameplay Ability System is a modular gameplay ability system for Unreal Engine 5. It allows you to manage skills, powers, special actions, passive abilities, or context-sensitive abilities on an actor without having to build a complex architecture from scratch.

The system is designed as a simpler and more straightforward alternative to Unreal Engine's native Gameplay Ability System (GAS). It provides a standalone foundation for creating, activating, locking, unlocking, removing, and organizing gameplay abilities from Blueprint, while building on a stable and high-performance C++ foundation.

Each ability retains its own logic: the plugin provides the general structure, but you define the effects, conditions, animations, inputs, visuals, and behaviors specific to your game.

The plugin can be used on its own, with your own architecture, or combined with other ADS Framework systems if your project requires it.

Installation

Install the plugin in the project, not only in the engine. The project copy can contain project-specific assets and settings.

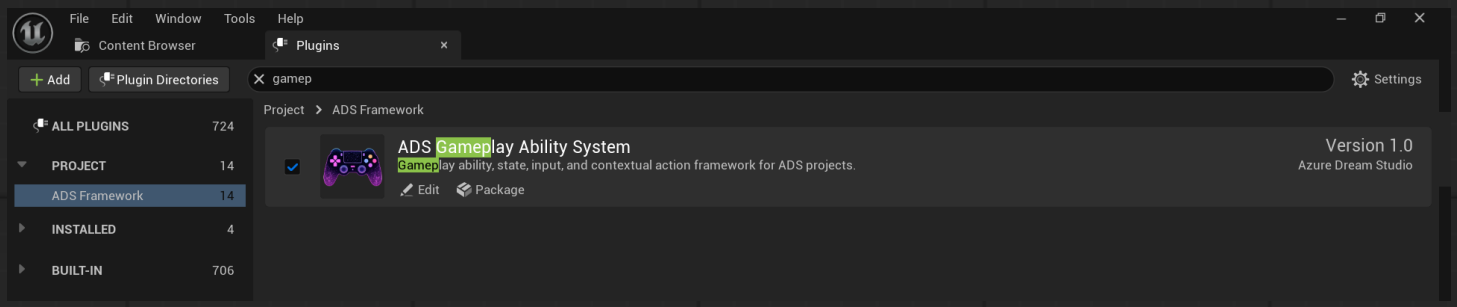
1- After installing the plugin from the Epic Games Launcher, close Unreal Engine. Copy the plugin folder from the engine Plugins folder to the project's Plugins folder.

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

"<Project folder>/Plugins/ "

2- Open Unreal Engine.
(If Unreal Engine prompts you to compile the plugin upon opening, accept.)

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



The plugin can be installed in both the engine and the project. In that case, Unreal Engine will use the project version. After copying it to the project, you may remove the engine version if desired.

How It Works

The ADS Gameplay Ability System is based on a central **Actor Component** that must be added to the character or actor intended to use abilities. This central Actor Component handles all management of the character's abilities.

Each ability must be implemented in a **dedicated Actor Component**. These ability Actor Components do not need to be added manually to the character. When an ability is used for the first time, the central Actor Component automatically adds the corresponding Actor Component if it does not yet exist.

This approach helps keep the character lightweight during loading. Initially, the character contains only the system's central Actor Component. Ability-specific Actor Components are added only when they become necessary.

Abilities can be gained, lost, locked, or unlocked at runtime. This allows you to dynamically adapt available abilities based on progression, context, or the project's gameplay rules. For example, the *Run* ability can be locked when the player enters a house to prevent them from running inside, and then unlocked when they leave.

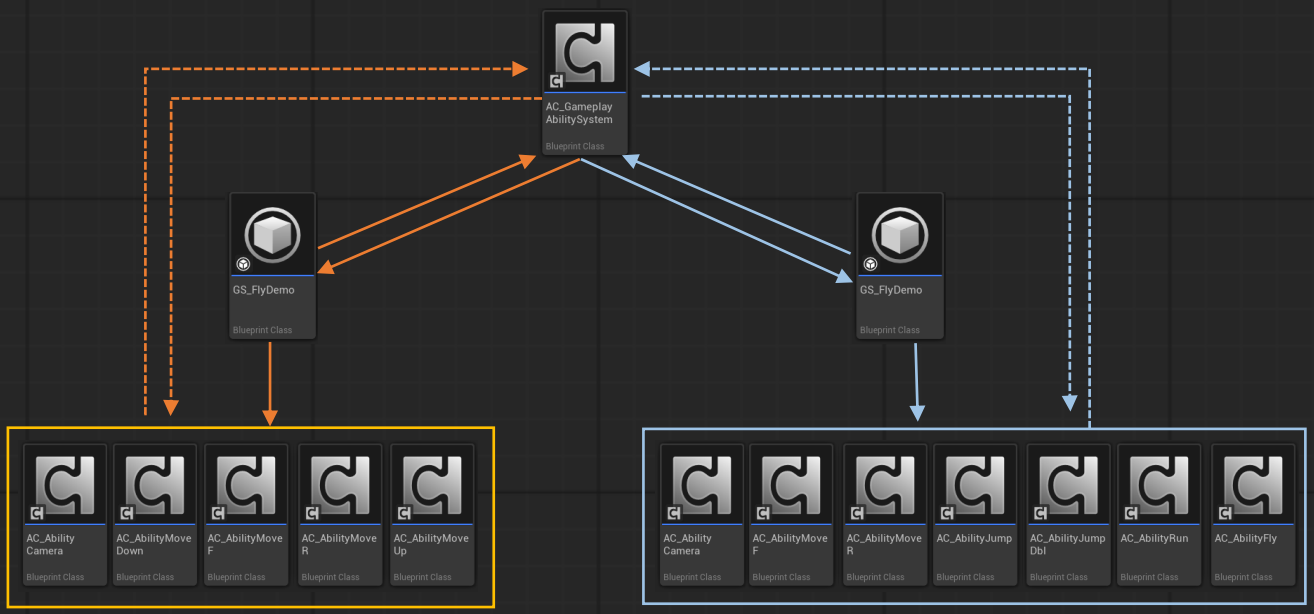
The system also uses states. Each state is represented by a dedicated asset that describes its behavior. States receive player inputs and redirect them directly to the Actor Component of the relevant ability.

This mechanism allows the same input to produce different results depending on the character's current state. For example, in a *Travel* state, pushing the joystick up may cause the character to move forward. In a *Ladder* state, that same input may allow the character to climb a ladder.

The system also allows you to manage context-sensitive actions using a ready-to-use collision box. This box automatically adds the Input Mapping Context corresponding to an action when a character enters its area, and then automatically removes it when the character leaves.

For example, in the *Travel* state, the `A` key can be used to jump. If the character enters the collision box placed in front of a door, the Input Mapping Context dedicated to that door is added. As long as the character remains in front of the door, the `A` key input is then received by the door actor instead of being processed by the *Travel* state. This allows the same input to be used for a different action depending on the context. When the character exits the collision box, the door's Input Mapping Context is automatically removed, and the `A` key reverts to its normal behavior in the Travel state.

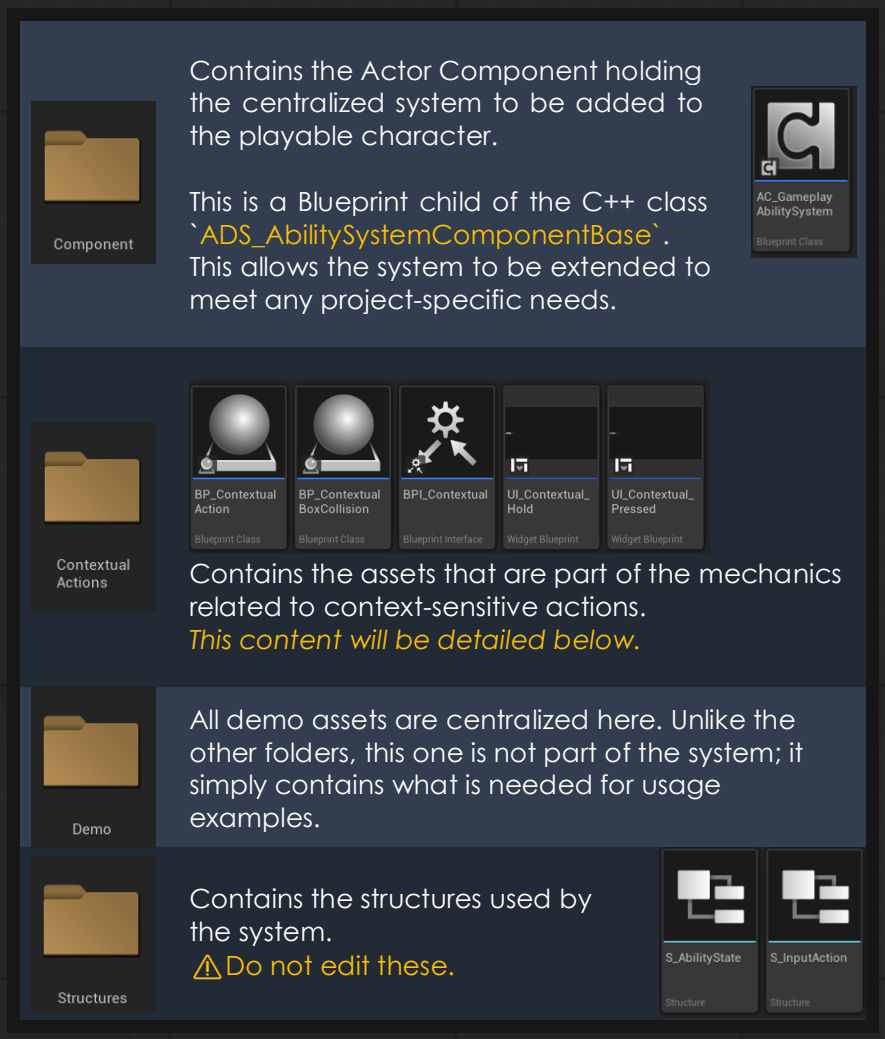
This architecture, in addition to being easy to use, helps ensure the project's stability. Since each ability is isolated within its own Actor Component, it is possible to add, modify, or remove an ability without compromising the project's stability. Modifying or removing an ability will not compromise the project's stability if no other ability's component contains a hard reference to it, making the architecture more modular, more maintainable, and safer to evolve.

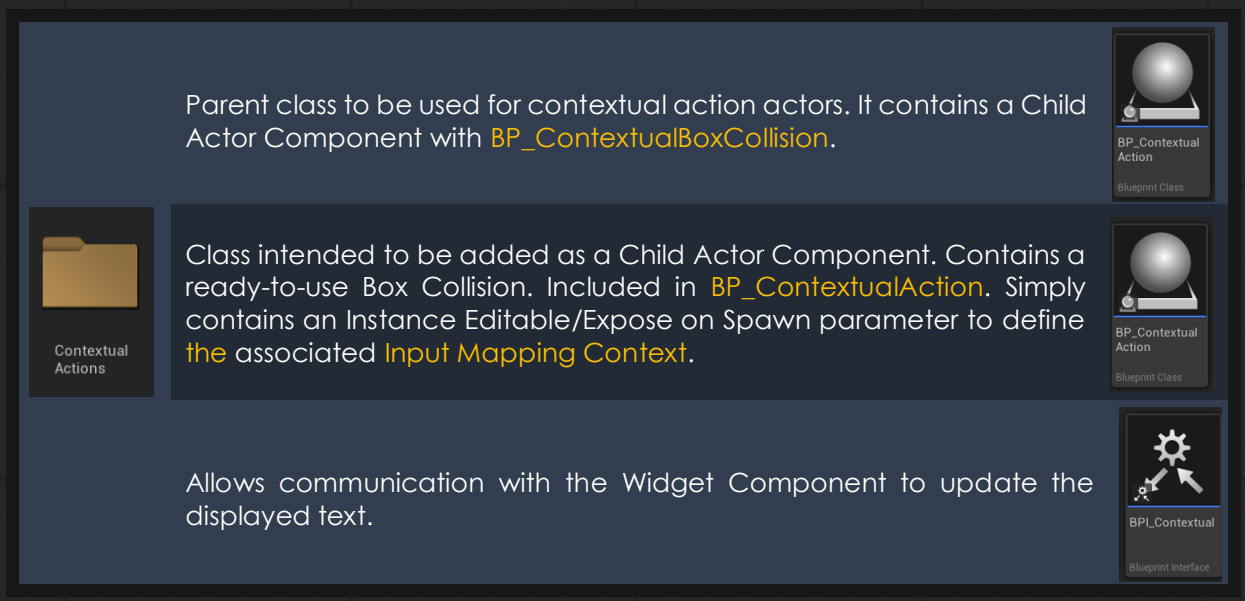


Plugin Contents

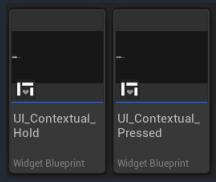
This documentation is intended as a user guide. C++ classes will therefore not be detailed in this documentation. They will only be mentioned when they serve as parents or when they need to be called from Blueprints. The system is developed in C++, but it is primarily used in Blueprints. This documentation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

The plugin has its own "Content" folder, organized as follows:





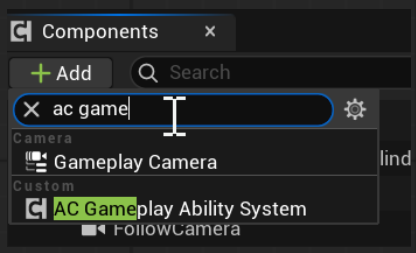
User interfaces intended for use in a Widget Component to indicate which key to press or hold to perform the relevant contextual action.
The **Widget Component** is included in **BP_ContextualAction**. The BP child of the latter will therefore only need to define which UI class to use.



Detailed Usage

System Actor Component

Adding the System to the Character

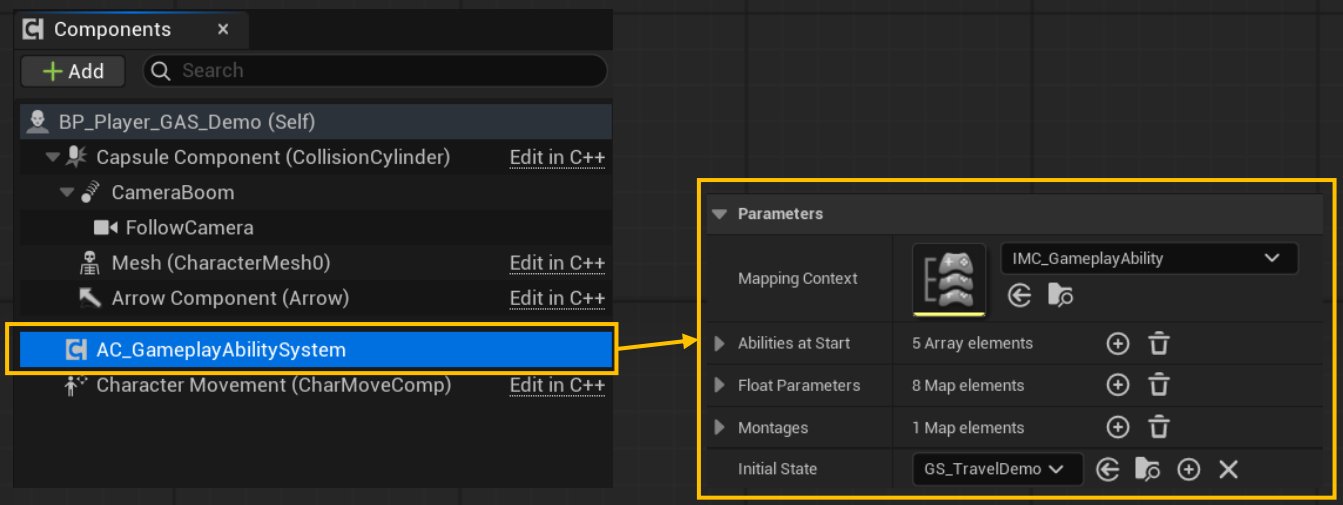


In your playable character Blueprint, add AC_GameplayAbilitySystem:

In the "Components" panel, click "+ Add," then search for "AC *gameplay ability system*" and click on it.

Configuring the Actor Component

Select the component in the "Components" panel, then adjust the settings in its "Details" panel.



This allows you to define the Input Mapping Context (IMC) that will be used for all "Abilities." Contextual abilities are not affected because they will have their own IMC.

Mapping Context

There's no need to create multiple IMCs for different situations, as **"Gameplay** State" classes are designed specifically for this purpose. For example, the "*Move* forward" and "*Ladder* up" abilities can use the same **Input Action**, since the **"Gameplay** State" will determine how the action is used and will route the input to the ability's Actor Component.

Allows you to create a list of abilities that will be automatically Obtained and unlocked when the character enters the game.

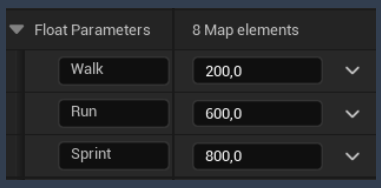
Abilities at Start

(The concepts of "Obtained," "lost," "locked," and "unlocked" will be detailed later in the documentation)

Float Parameters

Allows you to define a centralized list of Float parameters that can be used by ability Actor Components. Each entry is defined by a parameter name associated with its value.

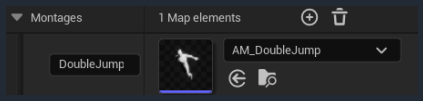
Example:



Montages

Allows you to define a centralized list of Montage animations that can be used by ability components. Each entry is defined by a montage name associated with a montage.

Example:

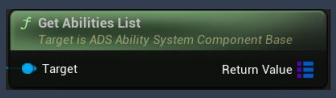


Initial State

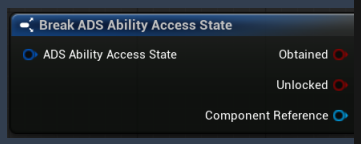
Allows you to define the character's initial gameplay state when it enters the game.

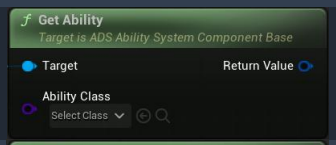
Example:

Actor Component Functions

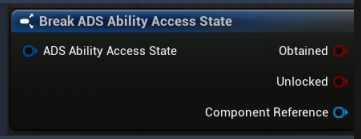


Returns a list of abilities in a Map where the key is the class of the relevant Actor Component, and the value is a structure containing whether it is Obtained, whether it is unlocked, and a reference to the instance used by the system.



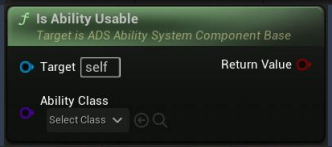


Takes an ability class as input and returns its acquisition status, unlock status, and a reference to the instance used by the system.



Receives an ability class and returns the instance used by the system.

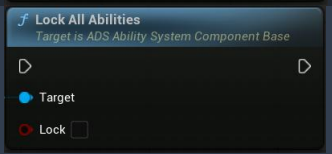
Takes an ability class as input and outputs via the execution pins depending on whether the ability in question is usable or not, along with a reference to the instance used by the system.
A usable ability is one that has been obtained and unlocked.



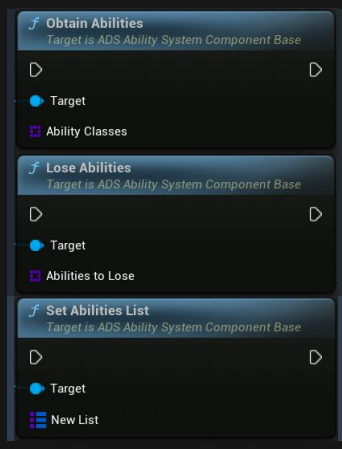
Takes an ability class and returns whether it is usable or not.
A usable ability is one that has been Obtained and unlocked.

Allows you to lock or unlock a list of abilities.

Tip: You can manually build this list without having to create a variable for it. Simply draw a cable from the "Abilities" input and call a "Make Array" to create an array on the fly.

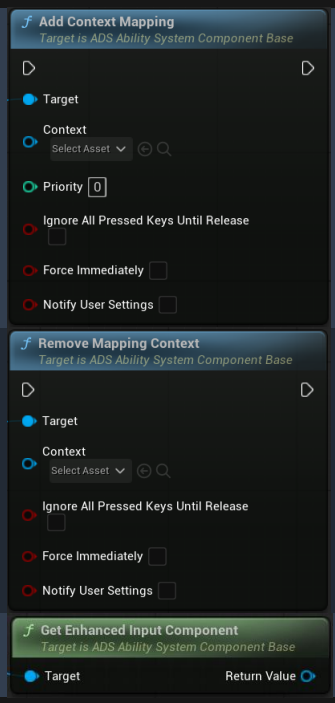


Locks or unlocks all abilities.



Allows you to acquire or lose a list of abilities. A newly obtained ability is also automatically unlocked so it can be used immediately. When an ability is obtained, its Actor Component is automatically added to the character. If the ability is already obtained, the system does nothing: it will not obtain it a second time or unlock it again.

Allows you to define the list of abilities along with their status. Specifically, this function should be used by a save system to restore the list upon loading exactly as it was saved.



Allows you to dynamically add an Input Mapping Context to the player via Unreal Engine's Enhanced Input system. At startup, the system uses this function to add the default Mapping Context defined in the Actor Component's settings. It can also be used at runtime to add other Mapping Contexts, for example as part of contextual actions.

Allows you to dynamically remove an Input Mapping Context previously added to the player.

Returns a reference to the player's Enhanced Input Component

Returns the value of a float parameter. The function takes the parameter name and a default value. If the specified parameter is not defined in the list, bFound returns False along with the default value. Otherwise, the function returns True along with the relevant value.

Allows you to set the current speed. It takes a parameter name that must be found in *Float Parameters*. This function is not used to set current speed = 200, but rather current speed = walking speed (for example).

Returns the current speed. For example, if the SetCurrentSpeed(Walk) function was previously called, GetCurrentSpeed will return the value corresponding to "Walk" in *Float Parameters*.

Sets the MaxCrouchSpeed value of the character's Character Movement Component. The benefit of this function is that it allows you to modify this value using a single node, without having to retrieve the reference and perform a set operation.

Sets the active Gameplay State. The function takes a Gameplay State class and sets it as active. The captured inputs will therefore be those defined by this Gameplay State.

Returns the instance of the active Gameplay State used by the system.

Tells the system that no Gameplay State is active. No further inputs will be intercepted.

Event Dispatchers

Called when an ability is locked or unlocked. It passes the class of the relevant ability and its new lock status.

Called when an ability has just been Obtained. The class of the relevant ability is passed.

Called when an ability has just been lost. The class of the relevant ability is passed.

Called when the active Gameplay State changes. It returns the class of the newly active Gameplay State.

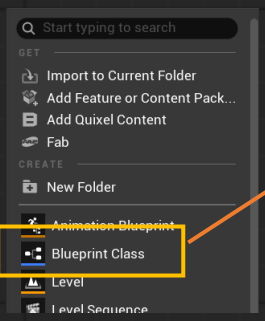
Ability Actor Component

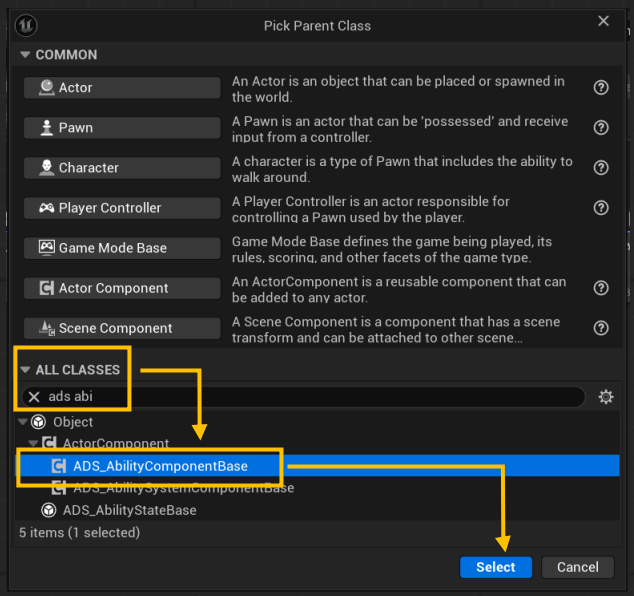
Creating an ability

1- Right-click in the Content Browser, then click "Blueprint Class"

2- In the creation window, in the "**ALL CLASSES**" section, search for "*ADS Ability*" -> Select "**ADS_AbilityComponentBase**" and then click "**Select**"

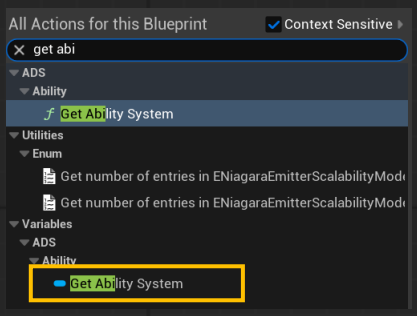
3- Name the newly created asset.





Developing an Ability

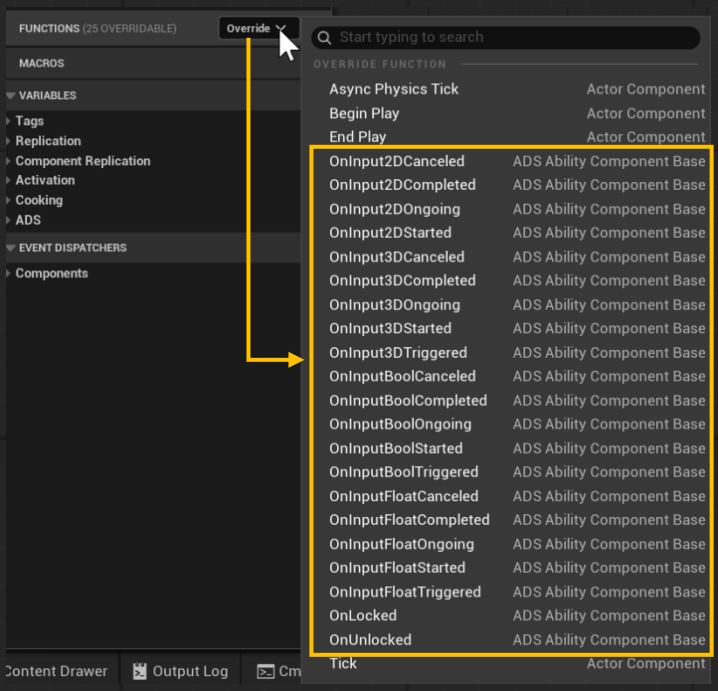
All ability Actor Components have a reference to the instance of the system's Actor Component that will manage its instance.



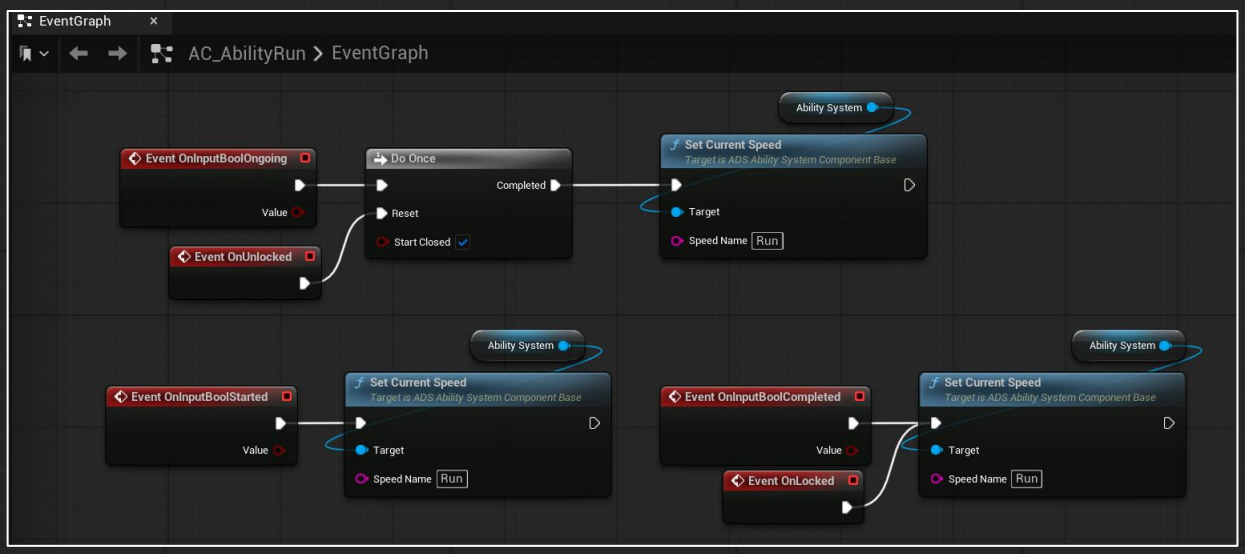
This allows the ability to access Float parameters and animations centralized in the system as needed.

The Gameplay State (detailed further below) redirects inputs to the Actor Component of the relevant ability. You must therefore retrieve this input transmitted by the Gameplay State to react accordingly.

To do this, you must override the relevant functions.



Example:



The ability's Actor Component is automatically added to the character upon its first use

Ability Locking and Acquisition

Each ability can be:

- Obtained
- Lost
- Locked
- Unlocked

⚠ **For an ability to trigger (and thus be usable), it must be Obtained + Unlocked.**

Obtained	Means that the ability has been Obtained. For example, when an NPC teaches the Player the "Super Jump" technique, the "Super Jump" ability becomes Obtained.
Lost	Means that the ability is no longer Obtained.
Locked	Means that the ability cannot be activated, even if it has been Obtained. For example, entering a house locks the "Run" ability to prevent the player from running inside. Even if the ability has been Obtained, it cannot be activated. Upon leaving the house, the "Run" ability can be unlocked.
Unlocked	Means that the ability can be triggered if it has been Obtained. For example, leaving a house unlocks the "Run" ability, but if it hasn't been Obtained yet, it cannot be triggered even though it is unlocked. Once the ability is Obtained, this same mechanism will allow the "Run" ability to be triggered.

Gameplay State

Definition of a Gameplay State

A Gameplay State defines the player's gameplay state:

For example:

- **Travel**: Standard movement mode, on the ground.
  The character can move, run, jump, and crouch.

- **Ladder**: Mode for moving up or down a ladder.
  The character can move up or down.

Each Gameplay State defines which actions can be listened to and redirected in the active gameplay context. For example, in the "Ladder" state, the character cannot crouch -> Therefore, the "IA_Crouch" action input will not be intercepted in this state.

In this framework, each Gameplay State corresponds to a dedicated asset that is a Blueprint child of the C++ class "ADS Ability State Base," and thus has its own graph. This allows you to say:

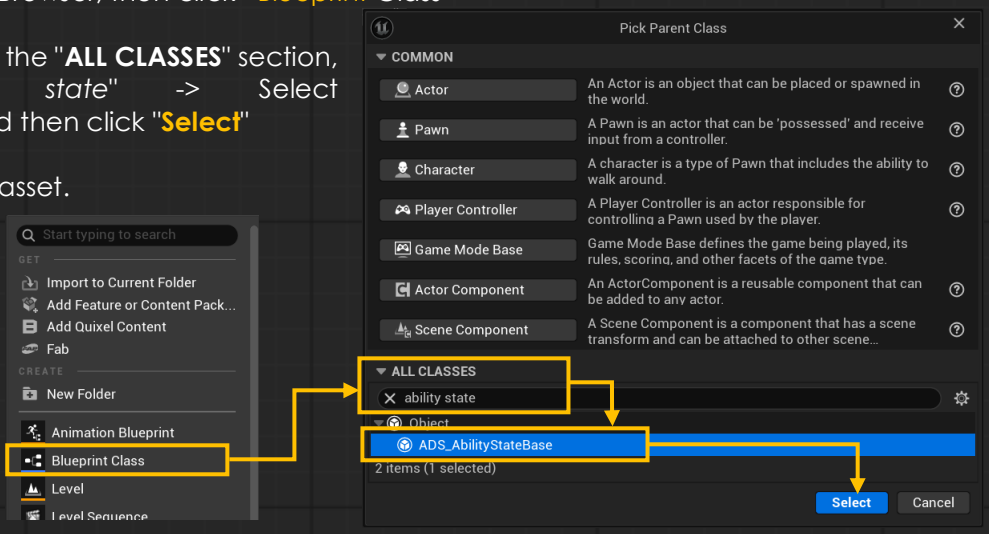
"In this state, I can do this and that."

Creating a Gameplay State

1- Right-click in the Content Browser, then click "Blueprint Class"

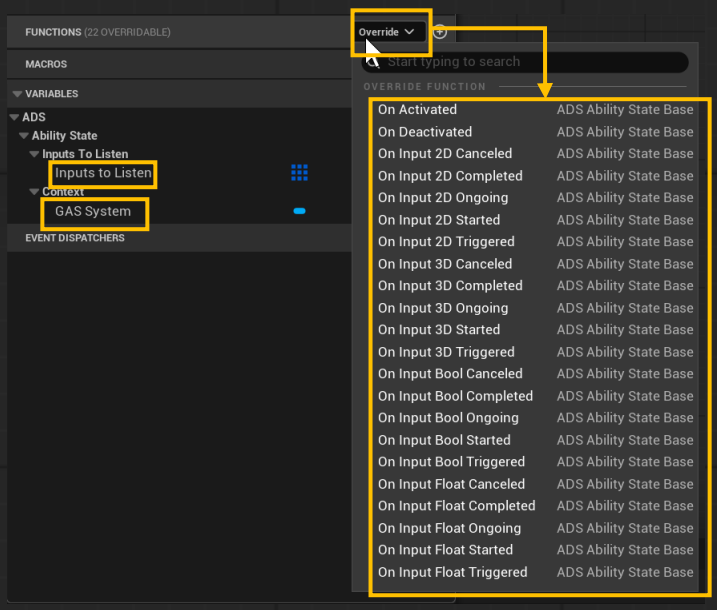
2- In the creation window, in the "**ALL CLASSES**" section, search for "*Ability state*" -> Select "**ADS_AbilityStateBase**" and then click "**Select**"

3- Name the newly created asset.

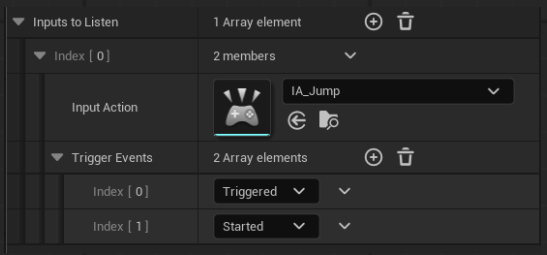


Developing a Gameplay State

A Gameplay State asset has a dedicated graph. It contains a reference to the instance of the system Actor Component that uses this Gameplay State instance, as well as several functions to override.



1- The first step is to define the list of inputs that will be intercepted by the Gameplay State. To do this, edit the "Inputs to Listen" variable. For each Input Action used, define which triggers will be listened to.



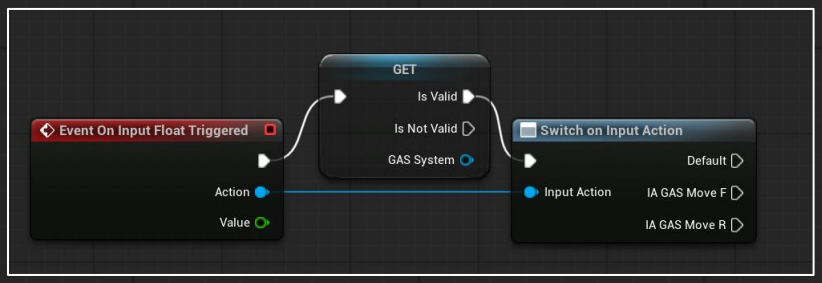
Example:
Here, we are only listening for the "Started" and "Triggered" triggers of "IA_Jump"

2- Next, you need to override the desired functions.

"On Activated" is triggered when the Gameplay State becomes active.
"On Deactivated" is triggered when the Gameplay State is no longer active.

For the others, this applies to inputs.

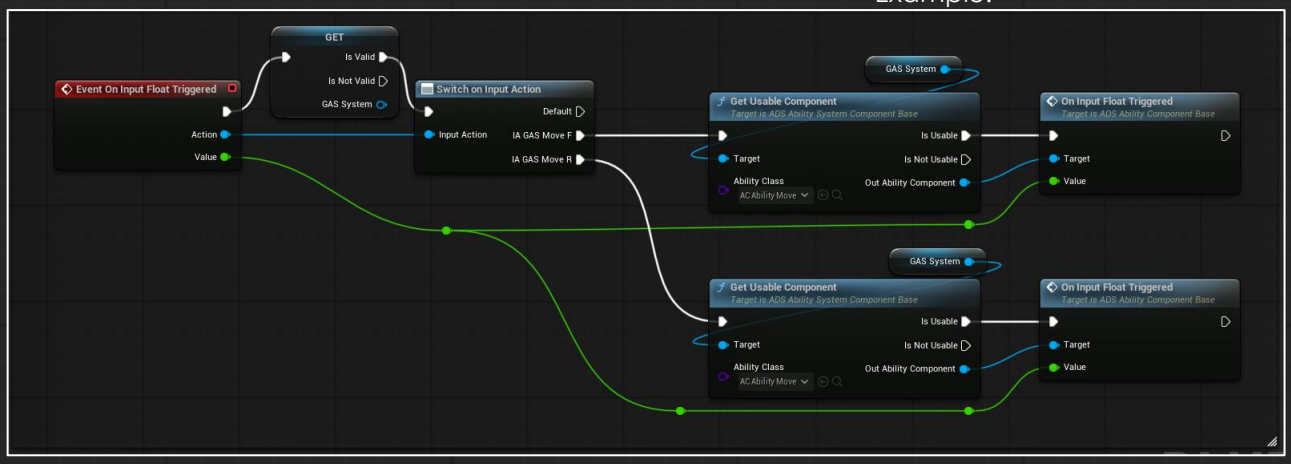
For inputs, you need to intercept them like this:



Using the Switch on Input Action function, you can handle each case individually.

"**Switch on Input Action**" is a node provided by this plugin. It is used like any other switch node, by defining the list in its "Details" panel.

Example:



Inputs

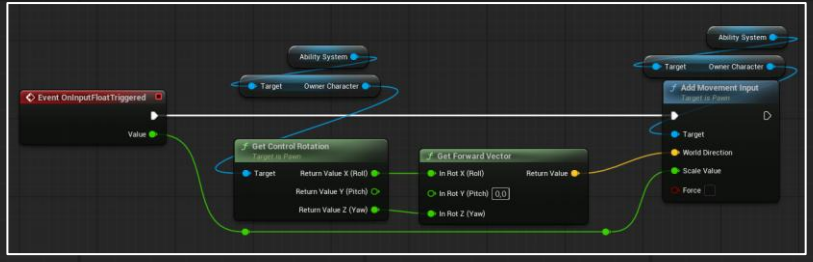
The Gameplay State intercepts the player's inputs. It then forwards them to the relevant ability.

Gameplay State	→	Ability
Receives input	Reroutes to the ability	Receives the final input

Complete example using the "Move F" ability:

1- The Gameplay State

2- The Ability component



Reminder: Even after doing all this, the ability won't necessarily be triggered because it must be Obtained and unlocked to be triggered. And the associated Input Action must be defined in the Input Mapping Context used by the system.

Contextual Action

Definition of a Contextual Action

Contextual actions are actions that are triggered in a specific context.

- Open a door: be in front of the door
- Clearing a tree from the path: stand in front of the tree
- Activate a switch: be in front of a switch
- Etc…

These are interactions that require an input which, outside of that context, serves a different purpose.

> For example, IA_Jump is used to jump, but that same Input Action can be used in front of a door to open the door.

To put it more simply: normally, the (A) button is used to jump, but in front of a door, it's used to open the door instead of jumping. The interaction with the door is therefore a contextual action.

How It Works

With the ADS Gameplay Ability System, contextual actions are handled differently from standard abilities. Since you must be in front of an element to interact with it, a collision with that element is required. When entering the collision area, the interaction's Input Mapping Context is added with a very high priority. Thus, if the input action for this interaction is also present in the system's Input Mapping Context, only the input related to this interaction will be taken into account by Enhanced Input. When leaving the collision area, the interaction's Input Mapping Context will be removed, thereby eliminating any input conflict created upon entering the collision.

In principle, therefore, each contextual action is associated with an Input Mapping Context that is different from the one used by the system. The provided collision class automatically handles the management of Input Mapping Contexts, and the actor involved in the collision can directly trigger the input action defined for it.

Creating a Contextual Action

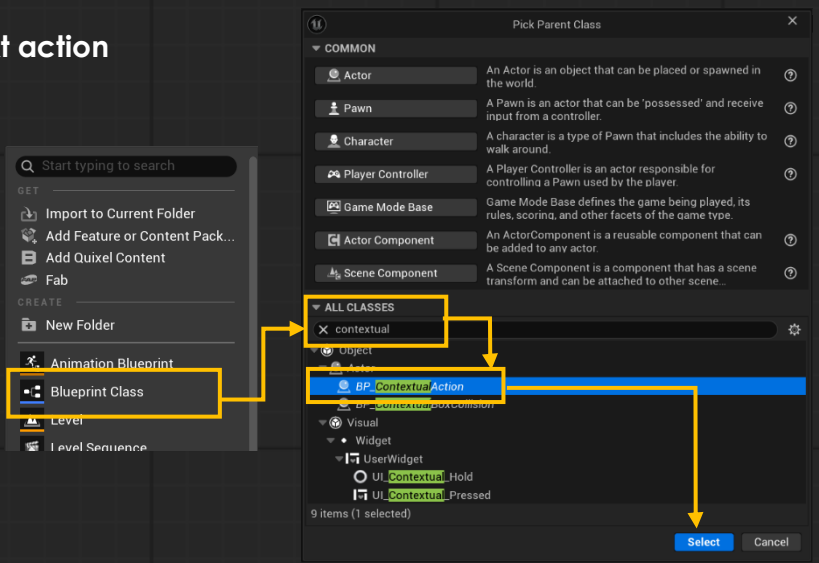
A contextual action involves several assets, but there is so little to say about each one that showing a complete example directly is clearer. Especially since, in the end, only two assets will need to be created (perhaps three):

- The actor
- The Input Mapping Context for this action. This can be defined using an existing Input Action or one created specifically for this purpose.

Example using the "Open a Door" context action

1- Create the actor

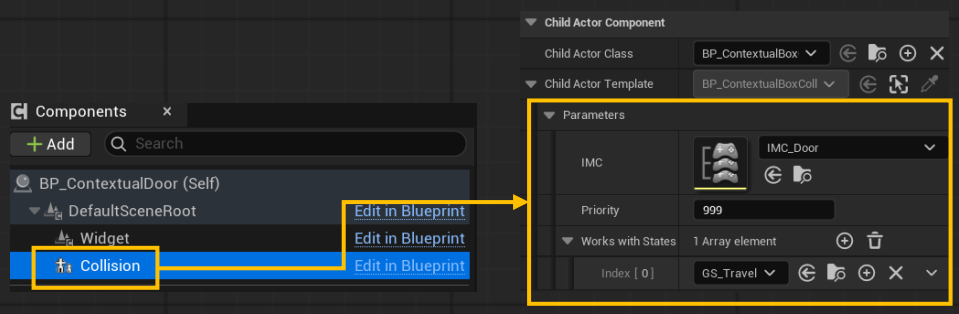
- In the Content Browser, right-click, then select Blueprint Class
- In "ALL CLASSES," search for "contextual"
- Select "BP_ContextualAction"
- Confirm by clicking Select



2- Configure the actor

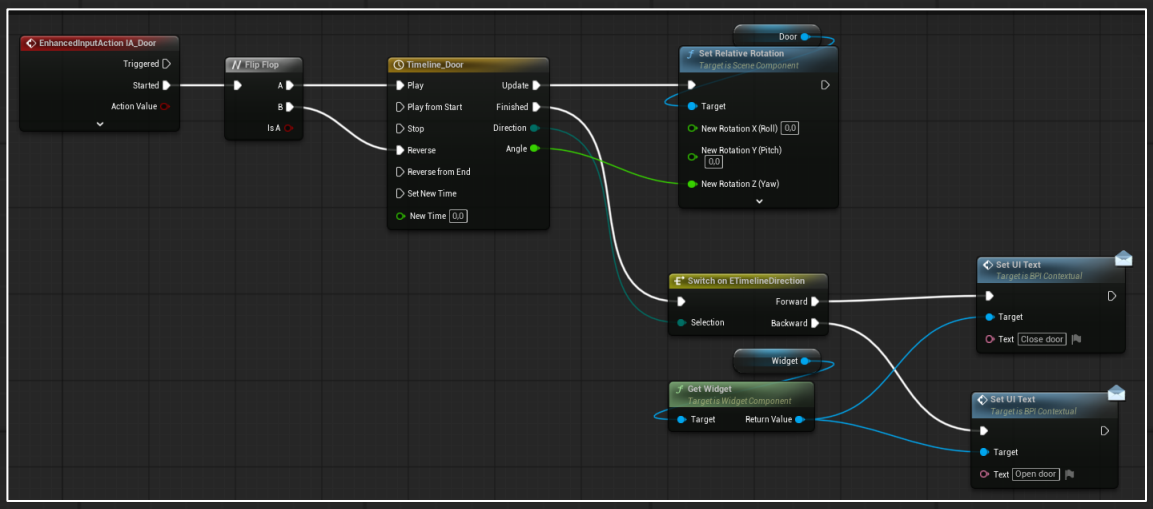
- Open the newly created BP

- Select the collision component and, in its "Details" panel, assign an Input Mapping Context, the priority (it is recommended to leave the default setting), and the states with which this contextual action will work.



- Define the meshes, etc.

- Configure the desired input action, which is defined in the assigned IMC



The system does not require one IMC per contextual action. Multiple contextual actions can share the same IMC. These IMCs must simply be different from the one assigned to the system Actor Component.

By default, only one collision Child Actor Component is present. You can add more if needed. Use BP_ContextualBoxCollision as the actor class, then configure its IMC, priority, and compatible Gameplay States.

What the system does not do

The ADS Gameplay Ability System provides a comprehensive framework for organizing, managing, and triggering abilities, but it does not automatically generate the specific gameplay for your project.

The system does not provide a complete, release-ready character or an exhaustive list of finalized abilities. The demo abilities serve solely as usage examples. Your own abilities must be created according to your game's needs, within dedicated Actor Components.

The plugin does not define game design rules for you. It does not decide which abilities should exist, when they should be obtained, how much they should cost, what effects they should produce, what animations they should play, or what conditions should trigger them. These rules remain specific to your project.

Nor does the system replace your game's other gameplay systems. It can communicate with combat, inventory, equipment, save, or interface systems, but it does not impose them or generate them automatically.

The contextual actions provided by the plugin allow you to technically handle situations such as interacting with a door, an object, or an element of the environment. However, they do not cover all possible interactions in a full game. You will need to create your own actors, inputs, interfaces, and logic based on the desired outcome.

Finally, the plugin does not provide a complete artistic direction. The demo assets are intentionally simple and are intended solely to help you understand how the system works. The final animations, visual effects, sounds, icons, interfaces, and feedback must be adapted to your project.

Troubleshooting

You may sometimes encounter a situation where something isn't working, and you don't understand why. This kind of situation is common in development, and the reason is most often a simple oversight. Here is a list of things to check.

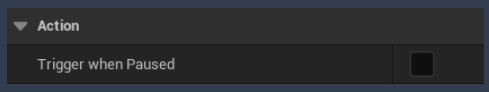
Ability

For an ability to trigger, it must meet several conditions. Check each one individually if you encounter a problem

The system Actor Component must have an Input Mapping Context assigned

The associated input action must be defined in the Input Mapping Context assigned to the system Actor Component

(Depending on the conditions, verify that the input action works while the game is paused)



A Gameplay State must be active

The active Gameplay State must be listening for the relevant input

The relevant input must be captured in the active Gameplay State

Must be Obtained and unlocked

The ability must handle the events for the appropriate input (Triggered, Started, Ongoing, etc.)

Contextual Action

For a contextual action to trigger, it must meet several conditions. Check each one individually if you encounter a problem

The actor you are interacting with must have all its Child Actor Components for collision defined using the BP_ContextualBoxCollision class

All of the actor's Child Actor Components must be correctly configured.
In particular, the IMC and the list of compatible Gameplay States

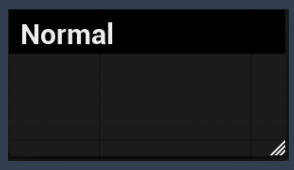
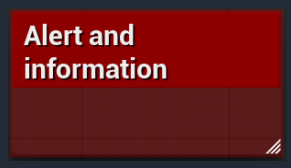
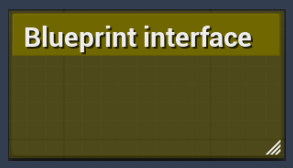
(Reminder: the IMC must be different from the system's)

The associated Input Action must be captured, and its logic must trigger the expected behavior.

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 just an example and that in your actual projects, you'll need to do better
	Blueprint Interface event

UI, Menu, and Widget

Most Unreal Engine users prefix all Widget Blueprints with "WBP_" without distinguishing their roles. This framework uses a different rule:

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

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

- "**W_**": A widget used as part of a UI or menu.
  Example: *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 regarding this system, you can contact me by email or via Discord. I will respond as soon as possible. Please be aware of the time difference, however. I do my best to respond promptly, but depending on your time zone, it may take several hours between when you send your message and when I am able to 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 line.
Example: "`[ADS Gameplay Ability System] Need information`"

https://discord.gg/Vy3TvsjKpQ