Gameplay Ability System Overview

A breakdown of the Gameplay Ability System and how each of its component classes contribute to abilities.

The **Gameplay Ability System** is a framework for building abilities and interactions that Actors can own and trigger. This system is designed mainly for RPGs, action-adventure games, MOBAs, and other types of games where characters have abilities that need to coordinate mechanics, visual effects, animations, sounds, and data-driven elements, although it can be adapted to a wide variety of projects. The Gameplay Ability System also supports replication for multiplayer games, and can save developers a lot of time scaling up their designs to support multiplayer.

With this system, you can create abilities as simple as a single attack, or as complex as a spell that triggers many status effects depending on data from the user and the targets. This page provides an overview of the Ability System and how its components work together.

What is a Gameplay Ability?

A **Gameplay Ability** is an in-game action that an Actor can own and trigger repeatedly. Common examples include spells, special attacks, or effects triggered by items. This concept is very common in video games, so much so it is often taken for granted, though the processes involved in running an ability are often complex and require specific timing. For

example, while coding an attack activation is fairly simple in itself, over the course of a long-term project the complexity of building abilities can explode as you add resource costs, buff or debuff effects to add or remove from players, combo systems, and other details. As such, there are three major considerations involved in how Unreal Engine's Gameplay Ability System is designed.

Tracking an Ability's Owner

Abilities and their effects must maintain a concept of ownership. When an effect runs a calculation, it needs to know who its owner is so that it can use their attributes, and when an ability does something that would score a point for a player, it needs to know what player owned it so that it can give credit correctly.

> The "Owner" for a Gameplay Ability is not the same as Ownership in network replication terms.

Tracking an Ability's State

An ability must be able to track a cycle of states:

- When the ability is activated.
- When the ability's execution is currently in-progress.
- When the ability is fully completed and no longer active.

> The Lyra sample project also tracks when an ability is granted.

As an example, an Actor in the middle of using an ability typically wouldn't be able to activate the same ability again until it is done. However, abilities could have special rules that make it possible to **cancel** an ability, ending its execution early, and then start another one.

Coordinating an Ability's Execution

An ability must be able to interact with multiple different systems during its execution, with specific timing. These interactions can be anything you can do in Blueprint, including:

- Activating animation montages.
- Taking temporary control of a character's movement.
- Triggering visual effects.
- Performing overlap or collision events.
- Changing characters' stats, either temporarily or permanently.
- Increasing or decreasing in-game resources.
- Allowing or blocking the activation of other abilities.
- Handling cooldowns to restrict ability usage.
- Getting interrupted by in-game events.
- Canceling other abilities in-progress.
- Making major state changes to a character, such as activating a new movement mode.
- Responding to input in the middle of other interactions.
- Updating UI elements to show in-game status for abilities.

Depending on how an ability works, it could perform any of these interactions at many different points in time while it is active, including in the middle of animations, and some effects may need to persist after the ability itself completes.

Components of the Gameplay Ability System

The Gameplay Ability System is designed to address all of these use-cases by modeling abilities as self-contained entities responsible for their own execution. This system consists of several components:

- An owning Actor with an **Ability System Component**, which maintains the following:
  - A list of all the abilities the Actor owns and handles activation.
  - The set of Gameplay Tags (usually representing the status) granted to an Actor.
  - A list of Gameplay Attributes that act as enhanced float properties, allowing predictive changes, replication, and temporary buffs/debuffs.
  - A list of modifications to the Actor’s attributes, that can change how calculations such as damage are performed (buffs/debuffs).

    - A list of Gameplay Cues (audio/visual effects) currently active on the Actor.
- **Gameplay Abilities Blueprints** that represent individual abilities, and coordinate their in-game execution. They can:
    - Be Blueprint or C++.
    - Can do anything a Blueprint can do, as well as use **Gameplay Ability Tasks** to perform async functions.
- An **Attribute Set** attached to the Ability System Component that contains the **Gameplay Attributes** that drive calculations and represent resources.
- **Gameplay Effects** that handle changes to Actors as a result of using Abilities.
    - Used to modify an Actor's attributes and Gameplay Tags in a way that is compatible with network prediction.
    - Contains a list of Gameplay Effect Components that dictate behavior.
    - **Gameplay Effect Calculations** that provide modular, reusable methods for calculating effects.
    - **Gameplay Cues** that are associated with Gameplay Effects and provide a data-driven method for handling audio and visual effects.

The sections below summarize these classes in greater detail.

Tracking Ownership

You need to attach an Ability System Component to an Actor for it to use Gameplay Abilities. This component is responsible for adding and removing abilities to an Actor, keeping track of what abilities an Actor owns, and activating them. It is also the main representation of the owning Actor in the context of the ability system, providing a system for tracking attributes, ongoing effects, **Gameplay Tags**, and **Gameplay Events**, as well as the interface for accessing the owning Actor directly.

In multiplayer games, the Ability System Component is also responsible for replicating information to clients, communicating player actions to the server, and verifying that clients are authorized to change the Ability System Component's state. The Ability System Component's parent Actor must be owned by a locally controlled player for remote activation to take place, meaning you can only execute abilities on Actors you control.

Handling Abilities and Their Execution

A Gameplay Ability is a Blueprint object that is responsible for executing all of an ability's events, including playing animations, triggering effects, fetching attributes from its owner, and displaying visual effects.

Controlling Activation

You can **Activate** Gameplay Abilities through four main methods:

- You can activate an Ability explicitly through Blueprint or C++ code using a **Gameplay Ability Handle**. This is provided by the Ability System Component when an Ability is granted.
- Using Gameplay Events. This fires all abilities with a matching Ability Trigger. If you need to abstract your input and decision mechanisms, this method is preferable, as it provides the greatest degree of flexibility.
- Using Gameplay Effects with matching tags. This fires all abilities with a matching Ability Trigger. This is the preferred method for triggering abilities off of Gameplay Effects. A typical use case would be a Sleep debuff, which triggers an ability that plays a disabled animation and inhibits other game actions.

  > Gameplay Abilities can represent a wide array of in-game actions, and are not limited to powers or spells that players explicitly use. Hit reactions, or the above example of a Sleep animation, are good examples.

- Using **Input Codes**. These are added to the Ability System Component, and when called they will trigger all Abilities that match. This functions similarly to Gameplay Events.

When you **Activate** a Gameplay Ability, the system recognizes that ability as being in-progress. It then fires off any code attached to the Activate event, moving through each function and Gameplay Task until you call the **EndAbility** function to signal the ability is finished executing. You can attach further code to the **OnRemove** event if you need to do any extra cleanup. You can also **Cancel** an ability to stop it mid-execution. Most functions in GameplayAbility have a counterpart in the AbilitySystemComponent, so you can choose different functionality per-GameplayAbility, or per-AbilitySystemComponent class.

Gameplay Abilities use Gameplay Tags to limit execution. All abilities have a list of tags that they add to their owning Actor when they activate, as well as lists of tags that block activation or automatically cancel that ability. While you can manually cancel, block, or allow

abilities' execution with your own code, this provides a method that is systemically consistent.

Controlling Execution

Gameplay Abilities support a variety of common use-cases, like cooldowns and assigning resource costs, and there is a pre-made library of Gameplay Ability Tasks that handle animations and other common Unreal Engine systems.

Whereas standard Blueprint function nodes finish executing immediately, Gameplay Ability Tasks keep track of whether they're inactive, in-progress, or finished, and they can be programmed to fire off other events during their execution. They can also keep track of whether their parent Gameplay Ability has been canceled and clean up accordingly. It is common for games to implement custom gameplay logic by extending Gameplay Ability Tasks.

Gameplay Abilities can also respond to Gameplay Events, which are generic event listeners that wait to receive a Gameplay Tag and an **Event Data** struct from the owning Actor.

Attribute Sets and Attributes

The Gameplay Ability System interacts with Actors mainly through **Attribute Sets**, which contain **Gameplay Attributes**. These are enhanced floating point values that can be temporarily buffed / debuffed, and are used in calculations or modified by Gameplay Abilities. Attributes can be used for any purpose you want, but common use-cases include tracking a character's health or hit points, as well as values for a character's core statistics (such as Strength and Intelligence). While you can use basic variables to represent these values, Gameplay Attributes provide several advantages:

- Attribute Sets provide a consistent, reusable group of attributes that you can build systems around.
- Gameplay Abilities can access Gameplay Attributes through reflection, making it possible to create simple calculations and effects directly in the Blueprint editor.
- Gameplay Attributes track their default value and current value separately, making it easier to create temporary modifications (buffs and debuffs) and persistent effects. Gameplay Attributes, like normal UProperties, can replicate their value to all clients, and are safe for local UI visualizations such as enemy health bars.

For an Actor to use Gameplay Attributes, you must add an Attribute Set to its Ability System Component. After that, the Ability System Component can automatically access the attributes you assigned to the Attribute Set.

Handling Gameplay Effects

The Gameplay Ability System uses Gameplay Effects to apply changes to Actors targeted by Gameplay Abilities. These can be one-shot effects, such as applying damage, or persistent effects, such as ongoing poison damage, buffs, and debuffs. In the case of persistent effects, the Gameplay Effect attaches itself to the target Actor until it is removed, and they can be pre-set to have a limited lifetime before they expire and clean themselves up, undoing any changes to the target Actor's Gameplay Attributes.

Instant Gameplay Effects modify the Base value of Attributes. Any Gameplay Effect that has a Duration will affect the Current value of Attributes and thus be undone when the Gameplay Effect ends.

Gameplay Effects use Gameplay Effect Calculations to handle calculations based on Gameplay Attributes. While you can create simple calculations directly in the Blueprint editor, you can also program custom Effect Calculations that have more complex logic and affect multiple attributes at a time. These are able to process information from both the owning Actor of the Gameplay Ability and the target Actor, so you can concentrate common calculations into one reusable piece of code.

Handling Cosmetic Effects

Gameplay Cues are Actors and UObjects responsible for running visual and sound effects, and are the preferred method for replicating cosmetic feedback in a multiplayer game. When you create a Gameplay Cue, you run the logic for the effects you want to play inside its Event Graph. Gameplay Cues can be associated with a series of Gameplay Tags, and any Gameplay Effect matching those tags will automatically apply them.

For example, if you add the tag Ability.Magic.Fire.Weak to a Gameplay Cue, any Gameplay Effect that has Ability.Magic.Fire.Weak will automatically spawn that Gameplay Cue and run it.

This makes it quick and easy to create a universal library of visual effects without manually triggering them from code.

> Alternatively, you can trigger Cues without a Gameplay Effect association. For an example of this implementation, you can look at the Lyra sample game's weapon firing feedback.

Gameplay Cues do not use reliable replication, so there is a chance that some clients do not receive the Cue or display its feedback. This can cause desyncs if you tie in gameplay code to these. Therefore, Gameplay Cues should be used for cosmetic feedback only. For gameplay-relevant feedback that needs to be replicated to all clients, you should rely on Ability Tasks to handle replication instead. The **Play Montage** Ability Task is a good example of this.

Supporting Network Multiplayer

Gameplay Abilities provide some built-in functionality for supporting networked multiplayer games, but have some limitations and guidelines you should be aware of. Many of these guidelines reflect general network multiplayer best practices.

Ability System Components and Replication

To save bandwidth and prevent cheating, Ability System Components do not replicate their full state to all clients. Specifically, they do not replicate Abilities and Gameplay Effects to all clients, only Gameplay Attributes and Tags that they affect.

Replicating Abilities and Using Prediction

Most abilities in networked games should run on the server and replicate to clients, so there is normally lag in ability activation. This is not desirable in most fast-paced multiplayer games. To mask this lag, you can activate an ability locally, then tell the server you have activated it so that it can catch up.

There is a chance that the server rejects the ability activation, which means it must undo the changes the ability made locally. You can handle these cases using **locally predicted** abilities.

To help support this, some Gameplay Effects support rollback if the ability that granted them gets rejected by the server. These include most non-instant GEs, but notably excludes things like damage and other instantaneous attribute / tag changes. It is this rollback feature that causes us to prefer Gameplay Effects for applying all of your Gameplay Tags, Gameplay Cues, and Attribute changes.

Using Abilities to Interact With Server-Owned Objects

Gameplay Abilities can handle interactions with bots, NPCs, and other server-owned Actors and objects. You have to do this through a locally owned Actor – usually the player's Pawn – and a replicated ability or another non-GAS mechanism that calls a server function. This replicates the interaction to the server, which then has authority to perform changes with the NPCs.

Further Reading

For a working model of the Gameplay Ability System in action, examine the Lyra sample game project, which implements several abilities and weapons.