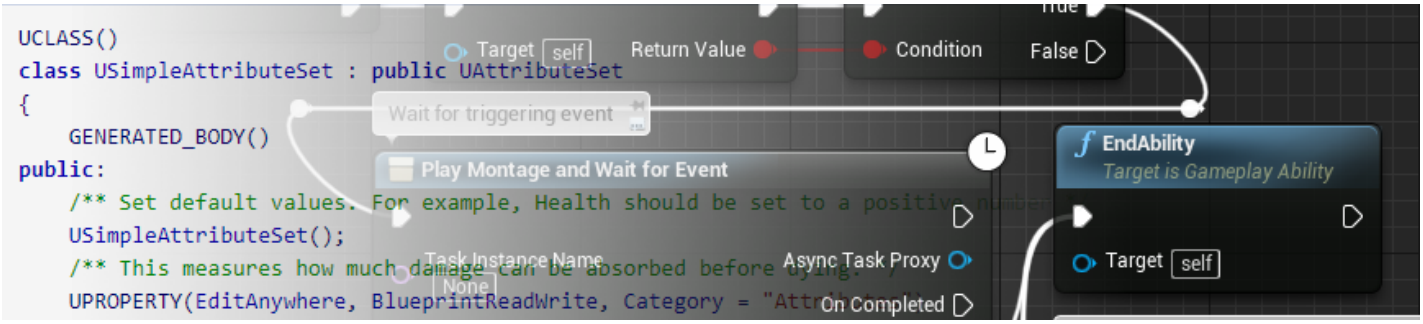
Gameplay Ability

Overview of the Gameplay Ability class.



A **Gameplay Ability**, derived from the `UGameplayAbility` class, defines what an in-game ability does, what (if anything) it costs to use, when or under what conditions it can be used, and so on. Because Gameplay Abilities are capable of existing as instanced objects running asynchronously, you can run specialized, multi-stage tasks involving character animation, particle and sound effects, and even branching based on player input or character interactions that occur during execution. Gameplay Abilities can replicate themselves across the network, run on client or server machines (including client-side prediction support), and even sync variables and make Remote Procedure Calls (RPCs). Gameplay Abilities also provide flexibility in terms of how the Engine implements them during a game session, such as extensible functionality to implement cooldown and usage costs, player input, animation with Anim Montages, and reacting to the Ability itself being granted to an Actor.

Granting and Revoking Abilities

Before an Actor can use an Ability, its Ability System Component must be granted that Ability. The following Ability System Component functions can grant access to an Ability:

- `GiveAbility`: Specifies the Ability to add with an `FGameplayAbilitySpec`, and returns an `FGameplayAbilitySpecHandle`. Only the server can give or revoke Abilities.
- `GiveAbilityAndActivateOnce`: Specifies the Ability to add with an `FGameplayAbilitySpec`, and returns an `FGameplayAbilitySpecHandle`. Because only the server can give Abilties, the Ability must be instanced and able to run on the server. After attempting to run the Ability, a `FGameplayAbilitySpecHandle` will be returned. If the Ability did not meet the required criteria, or if it could not execute, the return value will be invalid and the Ability System Component will not be granted the Ability.

Similar to giving abilities, only the server can remove abilities. The following functions can revoke access to an Ability from an Ability System Component, using the `FGameplayAbilitySpecHandle` that was returned when the Ability was granted:

- `ClearAbility`: Removes the specified Ability from the Ability System Component.
- `SetRemoveAbilityOnEnd`: Removes the specified Ability from the Ability System Component when that ability is finished executing. If the Ability is not executing, it will be removed immediately. If the Ability is executing, its input will be cleared immediately so that the player cannot reactivate or interact with it any further.
- `ClearAllAbilities`: Removes all Abilities from the Ability System Component. This function is the only one that does not require an `FGameplayAbilitySpecHandle`.

Basic Usage

A Gameplay Ability's basic execution lifecycle, after being granted to an Actor's Ability System Component, looks like this:

1. `CanActivateAbility` lets the caller know whether or not an Ability is available for execution without attempting to execute it. For example, your user interface may need to gray out and deactivate icons that the player can't use, or play a sound or particle effect on the character to show that a certain Ability is available.
2. `CallActivateAbility` executes the game code associated with the Ability, but does not check to see if the Ability should be available. This function is usually called in cases

where some logic is needed between the `CanActivateAbility` check and the execution of the Ability.

- The main code that users need to override with their Ability's custom functionality is either the C++ function called `ActivateAbility`, or the Blueprint Event called Activate Ability.
- Gameplay Abilities do not carry out their primary work in a "tick" function like Actors and Components do. Instead, they launch Ability Tasks during activation which do most of the work asynchronously, and then handle the output of those Tasks by hooking into Delegates (in C++) or connecting nodes to output execution pins (in Blueprints).
- The `CommitAbility` function, if called from within Activate, will apply the cost of executing the Ability, such as by subtracting resources from Gameplay Attributes (such as "magic points", "stamina", or whatever fits your game's systems) and applying cooldowns.
- `CancelAbility` provides a mechanism to cancel the Ability, although the Ability's `CanBeCanceled` function can reject the request. Unlike `CommitAbility`, this function is available for callers outside of the Ability itself. A successful cancelation will broadcast to On Gameplay Ability Cancelled before going through the standard code path for ending the Ability, giving the Ability a chance to run special cleanup code or otherwise behave differently when canceled than it would if it had ended on its own terms.

3. `TryActivateAbility` is the typical way to execute Abilities. This function calls `CanActivateAbility` to determine whether or not the Ability can run immediately, followed by `CallActivateAbility` if it can.
4. `EndAbility` (in C++) or the End Ability node (in Blueprints) shuts the Ability down when it is finished executing. If the Ability was canceled, the `UGameplayAbility` class will handle this automatically as part of the cancelation process, but in all other cases, the developer must call the C++ function or add the node into the Ability's Blueprint graph. Failing to end the ability properly will result in the Gameplay Ability System believing that the Ability is still running, and can have effects such as preventing future use of the Ability or any Ability that it blocks. For example, if your game has a "Drink Health Potion" Gameplay Ability that doesn't end properly, the character using that Ability will be unable to take any action that drinking a health potion would prevent, such as drinking another potion, sprinting, climbing ladders, and so on. This Ability blockage will continue indefinitely, since the Gameplay Ability System will think that the character is stil busy drinking the potion.

For information about setting this up in a Unreal Engine project, check out Gameplay Abilities in Action RPG.

Tags

Gameplay Tags can help to determine how Gameplay Abilities interact with each other. Each Ability possesses a set of Tags that identify and categorize it in ways that can affect its behavior, as well as Gameplay Tag Containers and Gameplay Tag Queries to support these interactions with other Abilities.

Gameplay Tag Variable(s)	Purpose
Cancel Abilities With Tag	Cancels any already-executing Ability with Tags matching the list provided while this Ability is executing.
Block Abilities With Tag	Prevents execution of any other Ability with a matching Tag while this Ability is executing.
Activation Owned Tags	While this Ability is executing, the owner of the Ability will be granted this set of Tags.
Activation Required Tags	The Ability can only be activated if the activating Actor or Component has all of these Tags.
Activation Blocked Tags	The Ability can only be activated if the activating Actor or Component does not have any of these Tags.
Target Required Tags	The Ability can only be activated if the targeted Actor or Component has all of these Tags.
Target Blocked Tags	The Ability can only be activated if the targeted Actor or Component does not have any of these Tags.

Replication

Gameplay Abilities support replication of internal state and Gameplay Events, or to turn replication off and save network bandwidth and CPU cycles. The Ability's **Gameplay Ability Replication Policy** can be set to "Yes" or "No", and this will control whether the Ability replicates instances of itself, makes state updates, or sends Gameplay Events across the network. This is not the same as replicating the activation or ending of the ability, which can occur regardless of that setting.

For multiplayer games with Abilities that do replicate, there are a few options for how replication is handled, known as the **Gameplay Net Execution Policy**:

1. **Local Predicted:** This option provides a good balance of responsiveness and accuracy. The Ability will run on the local client immediately upon the client issuing the command, but the server will have the final word, and can override the client in terms of what the Ability's actual impact was. The client is effectively asking permission from the server to execute the Ability, but also proceeding locally as if the server is expected to agree with the client's view of the outcome. Because the client is locally predicting the behavior of the Ability, it will feel perfectly smooth and lag-free as long as the server does not contradict the client's prediction.
2. **Local Only:** The client simply runs the Ability locally. There is no replication to the server, although the Ability will run on the server if the client using it is the host (playing on the physical server machine), or if this is a single-player game. This does not apply to dedicated-server games, as there is never a client playing on the server machine. Anything the client affects with this Ability will still follow normal replication protocol, including potentially receiving corrections from the server.
3. **Server Initiated:** Abilities that initiate on the server will propagate to clients. These are often more accurate, from the client's perspective, to what is actually happening on the server, but the client using the Ability will observe a delay due to the lack of local prediction. While this delay should be very short, some types of Abilities, especially rapidly-performed actions in pressured situations, will not feel as smooth as they would in Local Predicted mode.
4. **Server Only:** "Server Only" Abilities will run on the server, and will not replicate to clients. Any data outside of the Gameplay Ability will replicate as normal. In this way, the Ability can still have effects that clients observe, even though the Ability itself will only run on the server.

Instancing Policy

When a Gameplay Ability executes, a new Object (of the Ability's type) will usually spawn to track the Ability in progress. Since Abilities can execute very frequently in some cases, such as battles between a hundred or more players and AI characters in Battle Royale, MOBA, MMO, or RTS games, there may be cases where rapid creation of Ability Objects can negatively impact performance. To address this, Abilities have a choice of three different instantiantion policies, offering tradeoffs between efficiency and functionality. The three supported instancing types supported are:

- **Instanced per Execution:** A copy of the Ability's Object will spawn each time the Ability runs. The advantage to this is that you can use Blueprint graphs and member variables freely, and everything will be initialized to default values at the beginning of the execution. While this is the simplest instancing policy to implement, it is ideal for Abilities that run infrequently due to the larger overhead involved. For example, an "Ultimate" in a MOBA would be a reasonable use case, as there tends to be a long cooldown (usually 60-90 seconds) between executions, and there are only a few characters (usually about ten) who can use these Abilities at all. A basic attack Ability used by the computer-controlled "minions" would be a poor candidate, as there may be hundreds of them at a time, and each can issue basic attacks fairly frequently, causing rapid creation (and possibly replication) of new Objects.
- **Instanced per Actor:** Each Actor will spawn one instance of this Ability when the Ability is first executed, and future executions will reuse it. This creates the requirement to clean up member variables between executions of the Ability, but also makes it possible to save information across multiple executions. Per-Actor is ideal for replication, as the Ability has a replicated Object that can handle variable changes and RPCs, but does not waste network bandwidth and CPU time spawning a new Object every time it runs. In larger-scale situations, this policy performs well, since large numbers of Actors using the Ability (for example, in a big battle) will only spawn Objects on their first Ability use.
- **Non-Instanced:** This is the most efficient instancing policy in all categories. The Ability will not spawn any Object when it runs, and will instead use the Class Default Object. However, this efficiency introduces several restrictions. First, this policy uniquely requires that the Ability is written entirely in C++, as Blueprint Graphs require an Object instance. You can create Blueprint classes of a non-instanced Ability, but only to change the default values of exposed Properties. Further, the Ability must not change member variables or bind Delegates during its execution of the Ability. The Ability also cannot replicate variables or handle RPCs. This should be used only for Abilities that require no

internal variable storage (although setting Attributes on the user of the Ability is possible) and don't need to replicate any data. It is especially well-suited to Abilities that run frequently and are used by many characters, such as the basic attack used by units in a large-scale RTS or MOBA title.

> Changing the Instancing Policy will change how the Gameplay Ability behaves. For example, calls to OnAvatarSet, OnGiveAbility, ShouldAbilityRespondToEvent, OnRemoveAbility, and CanActivateAbility can happen without activating an Ability. If using InstancedPerActor, these calls occur on the instanced Ability (since we instance it immediately upon giving the Ability). However, Non-Instanced and InstancedPerExecution will receive these calls on their Class Default Object instead since they have no instances at the execution point.

Triggering with Gameplay Events

Gameplay Events are data structures that can be passed around to trigger Gameplay Abilities directly, sending a data payload for context, without going through the normal channels. The usual way to do this is by calling Send Gameplay Event To Actor and providing an Actor that implements the `IAbilitySystemInterface` interface and the contextual information that Gameplay Events require, but it is also possible to call Handle Gameplay Event directly on an Ability System Component. Because this isn't the normal path to calling Gameplay Abilities, context information that the Ability may need will be passed in through the `FGameplayEventData` data structure. This structure is generic and will not be extended for any specific Gameplay Event or Ability, but should be sufficient for any use case. The polymorphic `ContextHandle` field will provide support for additional information as needed.

> When a Gameplay Ability is triggered by a Gameplay Event, it will not run through the Activate Ability code path, but will instead use Activate Ability From Event, which provides the additional context data as a parameter. Be sure to handle this code path if you want your Ability to respond to Gameplay Events.