Game Framework Component Manager

The Game Framework Component Manager is a Game Instance Subsystem in the Modular Gameplay plugin that provides functionality designed to be used with Game Feature Plugins.

The **Game Framework Component Manager** is a **Game Instance Subsystem** in the **Modular Gameplay plugin** that provides functionality designed to be used with **Game Feature Plugins**. The functions implemented in this subsystem can be used by **Game Feature Actions** to support extensibility. Game Feature Actions are used by general gameplay code to coordinate communication between different gameplay objects. The manager implements two basic systems, **Extension Handlers**, and **Initialization States**.

Extension Handler System

The Extension Handler system allows the modification of game objects when game features are activated. There are two parts to this system: **Actors** work as **Receivers** that register to be extended, and **Extension Handlers** are delegates that are fired in response to events. These events include handling new receivers, removal of existing receivers, and arbitrary events that are called by gameplay code.

Receivers and Extension Handlers

To correctly register as a Receiver, an Actor should call the `AddGameFrameworkComponentReceiver` function from the `PreInitializeComponents` method, and the `RemoveGameFrameworkComponentReceiver` function from the `EndPlay` method. This ensures that it registers as a receiver as part of normal component initialization and unregisters when the Actor is removed or disabled.

Receivers can call the `SendGameFrameworkComponentExtensionEvent` function to send an arbitrary event. Unlike the Initialization State System described below, these extension events are not stateful and will only modify handlers that are currently active. To correctly register an Extension Handler, classes like `GameFeatureAction_AddComponents` can either call `AddExtensionHandler` to register a manual delegate, or call `AddComponentRequest` to call a wrapper function which will automatically add the desired component.

In both cases, the handles returned by the add functions need to be stored like an array because the delegates only stay registered as long as there are live shared pointer references to the returned handle struct.

Lyra Example

For an example of how to use this system, you can look at the implementation in the Lyra Sample Game. The `ALyraCharacter` class is used for all characters in the game and inherits from the `AModularCharacter` class that handles registering as a receiver. Additionally, you can observe the **LyraHUD** Actor which calls this function manually to enable UI extensions. Game feature plugins in Lyra like `ShooterCore` use the engine-defined `UGameFeatureAction_AddComponents` action to add components to spawned Actors. Lyra uses some game-specific actions like `UGameFeatureAction_AddInputBinding` to handle some game-specific cases.

For the game-specific `UGameFeatureAction_AddInputBinding` action, the `HandlePawnExtension` function is registered as a manual extension handler and responds to several different extension events. Events like `NAME_ExtensionRemoved` and `NAME_ExtensionAdded` are called when the extension handler is first added or removed for all relevant Actors. It responds to a game-specific `NAME_BindInputsNow` event that is emitted by the **LyraHeroComponent**, when it is time for binding feature-specific input events.

Initialization State System

The Initialization State system (**Init State**) provides functions for tracking the initialization and general life cycle of different features (usually implemented by components) attached to Actors in the game world. This system is not meant to be a generic gameplay state machine because the states are globally defined for an entire game, and are arranged linearly from creation to full initialization.

Synchronizing the initialization of components on an Actor is a complex process, especially when network replication is involved. This system provides registration and notification functions that make that coordination simpler. The low-level functions are implemented by the Game Framework Component Manager, and there is an optional native GameFrameworkInitStateInterface that can be inherited by components (or other gameplay objects) that implement a specified feature.

Actor Features

An Actor that has been registered with this system will have multiple **Actor Features**, which are defined as unique **Names**. These names are defined by the game and can either correspond to native class names or functional features. The subsystem keeps track of the **Init State** and Implementer object (often a component) for all features registered for an Actor. For objects that implement the `GameFrameworkInitStateInterface`, the feature name is returned by the `GetFeatureName` interface function and used for all other operations.

Init States

Init States are implemented as Gameplay Tags and must be registered with the subsystem by calling `RegisterInitState` during GameInstance initialization. These states are registered in order and shared by all Actors in a game. For instance, a game could support a simple 2 state system with `InitState.Spawning` and `InitState.Ready` or a more complex system like the Lyra example below.

Reporting and Querying States

All features registered with this system need to report to the Game Framework Component Manager whenever they change the init state because the manager stores this state for later querying. The manager does not enforce restrictions on changing the state and is designed to be flexible.

`GameFrameworkInitStateInterface` provides the framework for a simple C++ state machine that can be quickly implemented by overriding a few functions:

Function	Override Description
`CanChangeInitState`	This function should be overridden to return true if the requested state transition is allowed. This is where you would implement checks to see if the required data is available.
`HandleChangeInitState`	This function should be overridden to perform any object-specific changes that should occur on a specific state transition.
`CheckDefaultInitialization`	Can be overridden to attempt to follow the default initialization path for the feature. If the `ContinueInitStateChain` function is called with an array of init states, it will call `CanChangeInitState` and `HandleChangeInitState` to get as far as possible in the state chain. This function should be called from places like `OnRep` functions that may progress initialization.

Additionally the subsystem and interface provide registration and query functions:

Function	Description
`RegisterInitStateFeature`	Registers with the system but does not set a state, this is useful to call from component `OnRegister`.
`UnregisterInitStateFeature`	This should generally be called from `EndPlay` to unregister from the system and unbind notification delegates.

Function	Description
`HasReachedInitState`	This can be called to see if the feature has reached either the specified state or a later state in the initialization order.
`HaveAllFeaturesReachedInitState`	This is called on the manager to see if all features have reached a certain state. This is useful for coordinating extensions because you can set a central feature to wait for all other features to be ready before transitioning to the next state.

Registering For State Changes

The most useful part of this system is the ability to register for init state changes and call delegates after they reach certain states. The register functions like `RegisterAndCallForActorInitState` call the specified delegate when a feature reaches a certain state, and immediately call the delegate if it has already reached that state.

`RegisterAndCallForClassInitState` can be called with a class name to listen for any feature anywhere reaching that state, which is useful for listening to global initialization. These functions can be called from either C++ code or Blueprints, and the versions on the interface fill in the feature name for you. The delegate execution logic was designed to handle multiple state transitions happening in a row and all relevant delegates will be called.

For ease of use, `BindOnActorInitStateChanged` and `OnActorInitStateChanged` can be used on the interface to quickly listen for changes made to other features on the same Actor. This can then be used to call functions like `CheckDefaultInitialization` that may advance the init state of the feature.

Lyra Example

For an example of how to use this system, look at the implementation in the 5.1 or later version of Lyra Sample Game. The 5.0 release of Lyra predates the Initialization State system and has multiple race conditions this system was designed to help address. Here are the states used by the Lyra sample, as registered in `ULyraGameInstance::Init`

Init State	Description
`InitState.Spawned`	The feature has finished spawning and initial replication, called from `BeginPlay`.
`InitState.DataAvailable`	All data needed by the feature has been replicated or loaded, including dependencies on other actors that may also need to be replicated.
`InitState.DataInitialized`	After all of the data becomes available, it is used to complete other initialization actions like adding gameplay abilities.
`InitState.GameplayReady`	The object has finished all initialization and is ready to be interacted with in normal gameplay.

The two main components that use this system are the `ULyraPawnExtensionComponent` which coordinates the overall initialization and the `ULyraHeroComponent` that handles initialization of player-controlled systems like camera and input.
The initialization of both components depends on replicated data from multiple sources, and they call the `RegisterInitStateFeature` function from the `OnRegister` method to let the component manager know they exist. Both components later call the `CheckDefaultInitialization` function from the `BeginPlay` method after initial replication is done.

The full initialization state machine is needed for these two components because they also depend on data replicated by other Actors like `LyraPlayerState` that can be slow to download. The list below displays the overall timeline of initialization for a Lyra character:

1. When a Character is initially spawned on the Client and Server, it attaches and registers all components, including the two init state components and others like the LyraAbilitySystemComponent.
2. When `BeginPlay` is called on the Character, it tries to call `BeginPlay` on all components. On the server this happens right away, but on the client it will not call `BeginPlay` until all the replicated properties have sent their initial data. This happens at different times for each component depending on how much data they need to replicate.
3. When `BeginPlay` is called on either the Hero Component or Lyra Pawn Component, those components call `BindOnActorInitStateChanged` to listen for init state changes and then call `CheckDefaultInitialization` to attempt to follow the 4 state initialization

chain. At this point both components will reach `InitState.Spawned` and will try to continue initializing.

4. When the Hero Component tries to transition to `InitState.DataAvailable`, it checks to see if the player state and input component are ready. If that data isn't available, the state machine stalls until something calls `CheckDefaultInitialization`. If the required data is available, it transitions to `DataAvailable` but cannot transition to `DataInitialized` yet.
5. When the Pawn Extension Component calls `CheckDefaultInitialization`, if possible, it first tells the other components (like Hero Component) to move their initialization state machine forward. Then when trying to move its own state forward to `InitState.DataAvailable` it checks to see if the `PawnData` and Controller are fully available. The Pawn Extension Component calls `CheckDefaultInitialization` from various `OnRep` functions to try and move the state machine forward after important cross-actor references finish replicating. Another option is to call the initialization functions from a native tick function.
6. When the Pawn Extension Component tries to move forward to `InitState.DataInitialized`, it will not do so until all other features (like the Hero Component) have reached `DataAvailable`. When it actually transitions, this activates the `OnActorInitStateChanged` function on the Hero Component and anything else listening.
7. Once that happens, the extension component moves to `InitState.DataInitialized`, which then causes the Hero Component to also move to `DataInitialized`. During this transition, gameplay abilities are created and bound to player input.
8. Both the Hero Component and Pawn Extension Component then transition to `InitState.GameplayReady`, which activates Blueprint callbacks in classes like **W_Nameplate** that registered to wait for this state to be reached.

The Lyra character initialization flow is complex, but many networked games require similarly complicated initialization flows. The init state system is designed to make it easier to set up complex systems and avoid race conditions or random delay loops.