Cerberus Framework

Overview

The **Cerberus Framework (CF)** is a proprietary module and custom game development framework, built as a personal codebase specifically designed for my own projects. It is not a third-party asset, but rather a tailored solution that serves as a core component of the game's technical architecture.

CF streamlines essential functionalities and operations, utilizing asynchronous functions, structured communication patterns, and dependency injection to enhance performance and scalability. The framework follows the Model-View-Controller (MVC) pattern, ensuring clean separation of concerns and maintainability throughout the development process.

By providing a robust, modular foundation for various game-related tasks, **CF** emphasizes both flexibility and efficiency. While intended for personal use, access to the Git repository can be granted upon request.

Dependencies

CF leverages the power of four core assets to enhance its functionality and performance;

1. UniTask

- UniTask is a highly efficient asynchronous programming library for Unity that enhances performance, particularly in environments where multiple tasks need to be handled concurrently.
- It offers an alternative to Unity's built-in coroutines, providing better control over asynchronous operations with lower overhead.
- Improved performance due to lightweight task handling.
- Easy integration for managing asynchronous processes, allowing CF to handle game events, loading screens with minimal performance impact.

2. VContainer

- VContainer is a dependency injection (DI) framework designed for Unity. It promotes clean architecture principles by decoupling dependencies, making the framework more maintainable and scalable.
- It enhances code structure by ensuring that different components of the game communicate in a loosely coupled manner, adhering to SOLID principles.
- Facilitates a modular design, enabling easier management of components and services.
- Improves scalability by making it easier to integrate new features and functionalities without tightly coupling them with existing systems.

3. MessagePipe

- MessagePipe is a lightweight, fast messaging system used for event management within Unity. It efficiently handles communication between different game objects or systems without creating direct dependencies between them.
- It serves as the backbone of the framework's event-driven architecture, providing a clear and structured way for components to communicate through messages and signals.

- Enables decoupled communication between systems, allowing for clean and organized event management.
- Reduces complexity in managing game events (e.g., UI updates, player actions, or game state changes) by centralizing messaging.

4. Unity's Addressables

Unity's Addressables system is utilized for efficient asset management and dynamic loading of resources during gameplay. This system enables the framework to handle assets in a more flexible manner, allowing developers to load and unload resources as needed. By using Addressables, CF minimizes memory usage and enhances performance, particularly in larger projects where resource management is crucial.

CF Structure

CF is organized into several key components, each contributing to the overall functionality and efficiency of game development. These components include **injection scripts**, **managers and systems**, **scriptable objects**, **core scripts** and **template assets**.

1. Injection Scripts

These scripts handle DI, ensuring that all required services and components are properly instantiated and accessible throughout the game. By using DI, the framework promotes modularity and decoupling, making systems more flexible and easier to manage or replace.

2. Managers and Systems

Managers are global components instantiated during the game's initialization. They remain active throughout the entire gaming experience, providing core services and features.

Systems, in contrast to Managers, are localized and operate within specific gameplay phases. They focus on tasks that are activated and deactivated based on the game's progression. Examples include enemy spawning, level transitions, and in-game events, ensuring smooth and responsive gameplay.

3. Scriptable Object Scripts

Scriptable Objects are used to store data that is shared across different systems or used to configure various gameplay elements. These scripts define the logic and structure for managing reusable data assets within the project.

4. Core Scripts

CF incorporates a comprehensive set of core scripts that establish the foundation for various game systems and functionalities. This includes enumerations and constants that provide clarity and consistency throughout the codebase, as well as event management scripts that facilitate communication between different components. The framework utilizes the

Model-View-Controller (MVC) architecture to effectively separate concerns, enhancing maintainability and scalability by managing game data, user interfaces, and control logic.

Additionally, scene controllers play a critical role in managing transitions between game states and scenes, ensuring a smooth flow of gameplay through effective loading and unloading processes. Utility scripts offer essential helper functions for common tasks, such as math operations, string manipulation, and data handling, enhancing overall efficiency. Finally, UI scripts are dedicated to rendering and managing user interface elements, encompassing core functionalities for buttons, popups, and HUD components to ensure a responsive and interactive user experience.

5. Template Assets

The framework also comes with pre-built template assets, such as UI elements (buttons, popups), pool and sound dictionaries, scenes etc. which can be reused or modified based on the needs of the project.

How CF Operates

CF manages the game's initialization and progression through a structured flow of scenes, ensuring both the framework and game managers are properly set up before gameplay begins. The process starts with two key scenes: the **Preloader Scene** and the **Loading Scene**.

1. Preloader Scene

The **Preloader Scene** is the first scene loaded when the game starts, and its primary role is to initialize the core managers of CF and the other managers created by the user. This scene is controlled entirely by CF and is inaccessible to the user, as its operations are internal and focused on preparing the framework.

The preloader's **injection scope script** begins by initializing all the **Managers** asynchronously. The **Scene Controller** in the preloader waits for these initialization operations to complete. Once the managers are fully initialized and ready, the scene transitions to the next step in the flow.

2. Loading Scene

After the preloader finishes, control is handed over to the **Loading Scene**, which marks the beginning of game-specific setup. At this point, the **Loading Manager** is responsible for transitioning to this scene, and its operations depend on the needs of the game.

In the **Loading Scene**, the game may start loading **object pools**, **online data**, or other assets required for gameplay. The tasks performed in this scene vary depending on the project, making it highly customizable.

The key difference between the **Preloader Scene** and the **Loading Scene** is that the preloader is focused solely on preparing CF, while the loading scene sets up everything required for the game itself. This division ensures that CF is fully operational before any game-specific elements are initialized, providing a clear separation between framework setup and game preparation.

3. Main Scene

Once the **Loading Scene** completes its operations—loading assets, object pools, or online data—the control of the game is handed over to the user. The **Loading Manager** then transitions the game to the **Main Scene**, which serves as the central hub for game interactions.

The **Main Scene** is the first scene where the player can interact with the game. In the template provided by the framework, the main scene includes several buttons for common actions, such as:

- **New Game**: Starts a new game instance, bringing the player into the gameplay experience.
- **CF Demo Scene**: Provides access to the demo scene where the framework's features are showcased.
- **Settings Popup**: Allows the player to adjust game settings, such as audio or controls.

4. CF Demo Scene

CF **Demo Scene** is a special scene within the template, designed to demonstrate all the reusable template assets, systems, and UI elements that the framework offers, giving the user an opportunity to see its capabilities.

By transitioning from the **Loading Scene** to the **Main Scene**, and offering CF **Demo Scene** for exploration, CF not only prepares the game environment but also provides a structured way to demonstrate its modular assets and capabilities to developers.

Overview of Scripts

CF includes a variety of core scripts that are essential for its operation, providing constants, events, and MVC architecture.

1. Constants Scripts

These scripts define various constant strings that are used throughout the framework by different managers. They provide a centralized location for frequently used values, ensuring consistency and ease of maintenance.

Animation Constants: Specific constants related to animation names and parameters, allowing for streamlined animation management across different components.

Event Constants: These constants represent specific events within the framework, ensuring that managers can respond correctly to various state changes.

2. Event Scripts

The event scripts are defined as structs and are used in conjunction with the **MessagePipe** for communication between different components of the framework. Examples of event scripts include:

- **ManagerStateChanged**: Triggered when the state of a manager changes, allowing other systems to react appropriately.
- **ApplicationFocusChanged**: Indicates when the application gains or loses focus, useful for managing game pauses or resuming.
- **PopupOpened** and **PopupClosed**: These events manage the opening and closing of UI popups, enabling smooth user interactions.

3. MVC Scripts

The MVC scripts are designed to implement the Model-View-Controller pattern, providing a structured approach to manage the game's architecture. They serve as controllers that

manage the relationship between the **View** (UI components) and **Data** (game state and logic). Abstract interfaces define how these components interact, promoting separation of concerns and maintainability.

4. System and Session Scripts

System Scripts encapsulate specific functionalities that handle localized tasks within the game, such as enemy behavior, level transitions, and gameplay events.

Session Script acts as the overall manager for gameplay sessions. It initializes various systems, starts the game, and manages win/fail conditions. The session script ensures that gameplay flows smoothly and that all systems are coordinated effectively.

5. UI Scripts

The **UI Scripts** are essential for creating and managing the various components of the user interface within the game. They facilitate interactions and enhance the overall user experience by providing a range of reusable elements.

Component Scripts define the fundamental UI components that players interact with. Examples include CFButton, CFText, SafeArea and more.

Popup Scripts scripts are designed to create various modal windows that enhance user interaction by providing essential functionality without disrupting the gameplay. Examples include settings popup, pause popup, win and fail popups.

FlyTween Scripts are responsible for animating UI assets, specifically for creating smooth transitions and movements. These scripts manage tweenable UI assets, taking a start and finish point along with a specified curve to create dynamic movements. This enhances the visual appeal of text elements by allowing them to move in a smooth, engaging manner, improving the overall user experience.

6. Utility Functions

The **Utility Functions** serve as a collection of extension scripts and helper utilities that streamline and enhance the functionality of CF. These scripts are designed to simplify common tasks and improve overall performance.

- Extension scripts extend existing functionalities, providing additional methods and features that can be used across the framework. They enhance code readability and maintainability by offering reusable solutions.
- The **LockBin** utility is a specialized script that manages binary locking mechanisms, ensuring safe and efficient data handling. It helps prevent race conditions and ensures that data access is controlled and synchronized properly.
- The **JSON Serializer** utility provides a convenient way to serialize and deserialize objects to and from JSON format. This functionality is crucial for saving and loading game data, enabling smooth transitions and state management within the game.
- Performance utilities focus on optimizing performance by offering tools for profiling and monitoring. They help identify bottlenecks in the code, allowing developers to make informed decisions about optimizations and enhancements, ensuring a smooth gameplay experience.

Managers Overview

The **Managers** in CF are responsible for overseeing and coordinating different functionalities within the game. Each manager plays a critical role in ensuring smooth operations and enhancing the overall user experience.

1. AddressableManager

This manager is responsible for retrieving assets from Unity's Addressables system. It facilitates efficient asset management by allowing for dynamic loading and unloading of resources as needed during gameplay.

2. RemoteAssetManager

The **RemoteAssetManager** works with the Addressables system's remote functionality, enabling the retrieval of assets from a remote catalog. This is particularly useful for games that require frequent updates or downloadable content, as it allows for seamless integration of external resources.

3. DataManager

The **DataManager** handles saving and loading JSON serialized storage defined by the user. It provides a straightforward way to manage game data, ensuring that player progress and settings can be easily saved and retrieved.

4. DeviceConfigurationManager

This manager detects the platform on which the game is running and allows users to define the necessary changes for each platform. This ensures that the game operates optimally across different devices and configurations.

5. InventoryManager

The **InventoryManager** manages the submission, commitment, or rollback of inventory changes. While inventory items are defined by the user, the template includes predefined items such as star and coin, providing a starting point for customization.

6. LoadingManager

The **LoadingManager** is responsible for switching between scenes, ensuring smooth transitions and loading operations. It plays a key role in maintaining a seamless gameplay experience.

7. PoolManager

This manager handles the pooling of defined pool items, optimizing resource management and reducing instantiation overhead. By reusing objects, the **PoolManager** helps improve performance and efficiency during gameplay.

8. SoundManager

The **SoundManager** is dedicated to handling background music and sound effects, allowing for dynamic audio control throughout the game. It ensures that the audio experience is engaging and responsive to gameplay events.

9. VibrationManager

This manager is responsible for managing device vibrations, enhancing the tactile feedback of the game. It allows developers to implement vibrations for various actions, contributing to a more immersive experience.

10. UI Interaction Managers

Several managers facilitate user interface interactions:

- **PopupManager**: Manages the display and behavior of popups within the game.
- **TextTweenManager**: Handles the tweening of text elements, creating smooth animations for UI text.
- **FlyTweenManager**: Responsible for animating UI elements along defined paths, enhancing visual interactions.