GMS (Game Management Service) Version 1.0

Designed and Programmed by JohnVanD

- Foreword
- What is and Why use GMS?
  - Benefits behind data driven processing for games
- Quick Start Guide
  - 1.- Create a game Instance
  - 2.- Create a new GameManagerData scriptable object.
  - 5.- Add GameManagerData to your Addressables
  - 6.- Test your game.
    - Loading GameManagerData:
    - Creating the GameManager:
    - Binding and Initialization of SubManagers:
    - SubManagers Running the Game:
    - Testing Dependencies and Race Conditions:
- Comprehensive Guide
  - Audience
  - Game Samples
    - 1.- Open Addressables Group
    - 2.- Create Addressables Settings button
    - 3.- Drag the Samples Addressable Groups
  - Minimal Game Sample
  - TicTacToe Game Sample
- GMS Workflow
  - Scalability
  - Iterative design
  - Modular
- Terminology
  - Game Designer
  - GameInstance
  - GameManager
  - GameManagerData
  - SubManager or Service
  - Data-Oriented processing
  - Addressables
  - Scriptable Objects

Foreword

First and foremost, thanks for having an interest in GMS

I bring extensive and diverse experience in professional game development, with a strong foundation in the three main pillars: Programming, Art, and Game Design. Throughout my career, across projects at all levels—from beginner to professional—I've observed a common issue: the lack of a central system that provides the modularity we, as game developers, need.

This challenge often manifests as a rigid and fragile structure, much like a Jenga tower, where the removal of a single piece can cause everything to collapse. To address this, I've developed a streamlined asset that tackles this "Jenga" problem by combining simplicity with core robustness. This asset not only streamlines the development process but also enhances the most critical aspect of any game: Game Design.

Which tower do you like the most, the fragile funky one? Or the fun and sturdy?

With a Bachelor of Science in Animation and 10 years of experience in the video game industry, I've worked on titles such as Kerbal Space Program—renowned for its technical achievements—as well as mobile FPS and castle defense games.

My biggest inspirations are the japanese Game developer legends:
Masahiro Sakurai, and Shigeru Miyamoto.

Literature inspirations from american Game designers and teachers include:
Ian Schreiber, Jesse Schell and Raph Koster

What is and Why use GMS?

GMS is a foundational architectural system for any game, offering custom editor tools that empower developers to build games **iteratively**, **modularly**, and with **high performance**—all while maintaining a clear "**big picture**" perspective.

With the tools provided developers are able to see at a glance all their systems and core data related to them. No more guessing around which prefabs are for what system and are instanced by which.

The system is capable of exist as pure c# classes outside the game engine, such as Unity3D, unreal engine, godot or others (proprietary). This is particularly useful for situations where a game is running on a server without a Game Engine which is crucial for performance and cheat safe Multiplayer games.

In the provided examples, GMS utilizes unity's Addressables, Scriptable objects and amazing view side tooling to cut down development time, making your game cheaper to make and iterate upon.

When a team or even a solo dev talks about spaghetti code, or bad old code, they are actually referring to some of these persistent issues:

- Code dependencies
- Hard references
- Code breakage (bugs) everywhere
- Interdependencies and complex communication between game systems or entities.

GMS not only fixes and prevents most of those, it also empowers the Game design aspect of the game to an extremely high level.

GMS consists of 2 basic elements:

1.- **GameInstance**: A SINGLE monobehaviour that acts as an **anchor point** for the view side elements in the game. It acts both as a **Singleton**, and **ServiceProvider** at the same time. Both of which are validated Core Game Programming patterns.

2.- **GameManager**: Where the actual model is processed, it adds or removes SubManagers from GameManagerData objects. These submangers or "services"

have their logic and data binded together for you, and can be dynamically loaded, unloaded, set as base, or even toggled for rapid testing and prototyping.

For more information on Singleton and ServiceProvider those visit: https://gameprogrammingpatterns.com/

Benefits behind data driven processing for games

The code indeed interacts with RAM to read and write values, objects, arrays, and structs. Efficient memory access is crucial for performance, particularly in games where large datasets (like game worlds or character states) are frequently accessed.

Organizing data in memory, such as storing it in contiguous blocks, can significantly enhance performance by reducing cache misses. This approach is a fundamental principle of data-oriented design, where data is structured to align with the CPU's processing patterns for maximum efficiency.

Using plain data structures (often referred to as "Plain Old Data" or POD types) instead of complex objects like MonoBehaviour in Unity can lead to better performance. MonoBehaviour objects in Unity come with overhead due to their integration with Unity's component system and the engine's lifecycle management.

Monobehaviours are awesome, powerful and we love them, but that nicety comes at a performance cost.
That's why they should only be reserved for view side processing, with the least amount of memory allocations or dependencies as possible.

you can generally expect a considerable increase in performance from accessing the memory alone.

Data-oriented processing in the context of games, refers to an approach to software design and optimization that focuses on the efficient organization and manipulation of data to maximize performance, especially in terms of memory access and CPU cache utilization. This approach contrasts with object-oriented design, where the focus is on encapsulating behavior and data together within objects.

Simplifying how data is accessed can save a considerable amount of cpu cycles just to access the data alone, without even optimizing the logic methods being used!

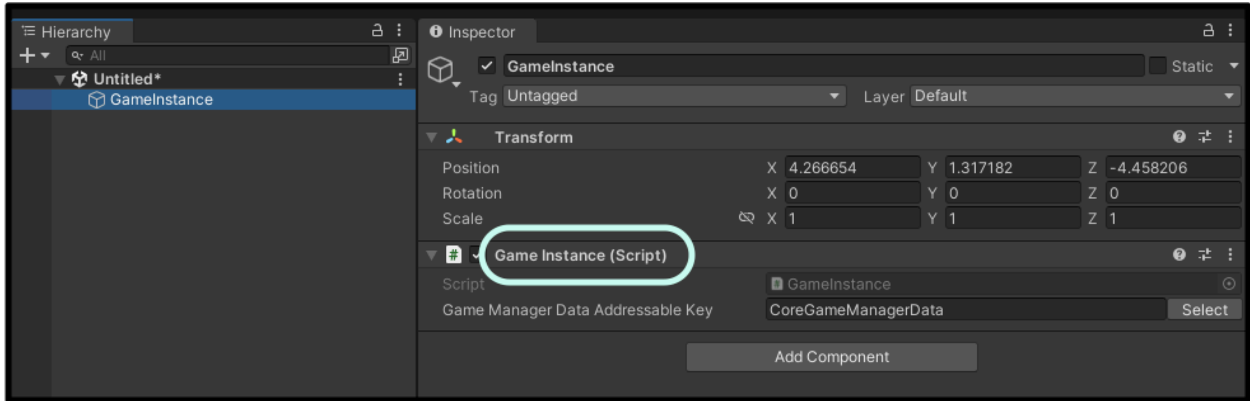
Advanced Key aspects of data-oriented processing in games include:

1. **Data Layout Optimization**: Data is organized in memory to ensure that operations on it are as efficient as possible. For example, related data is often stored contiguously in memory to take advantage of CPU cache lines, which can significantly speed up processing.
2. **Entity-Component-System (ECS) Architecture**: This is a popular architectural pattern in data-oriented design where entities are represented by IDs, components are pure data structures, and systems operate on batches of components. This allows for efficient processing of similar data in bulk, improving performance.
3. **Minimizing Cache Misses**: By organizing data to reduce the likelihood of cache misses (where the CPU has to fetch data from slower main memory rather than the fast cache), data-oriented design can achieve higher performance.
4. **Parallelism and SIMD (Single Instruction, Multiple Data)**: Data-oriented design often facilitates parallel processing, where the same operation is applied to multiple pieces of data simultaneously. SIMD instructions can be used to process multiple data points in a single CPU instruction, which is very effective in game loops.
5. **Separation of Data and Behavior**: Unlike object-oriented design, data and behavior (or logic) are separated. The data is stored in a way that is optimal for performance, while the behavior operates on this data, often in a sequential or parallel manner.

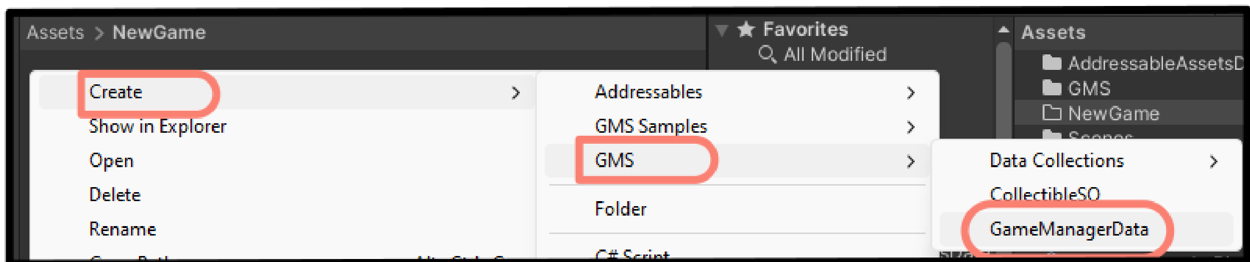
Quick Start Guide

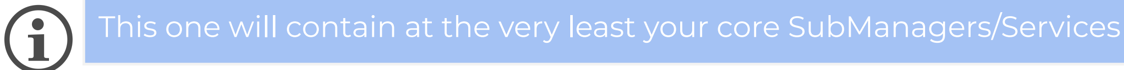
1.- Create a game Instance

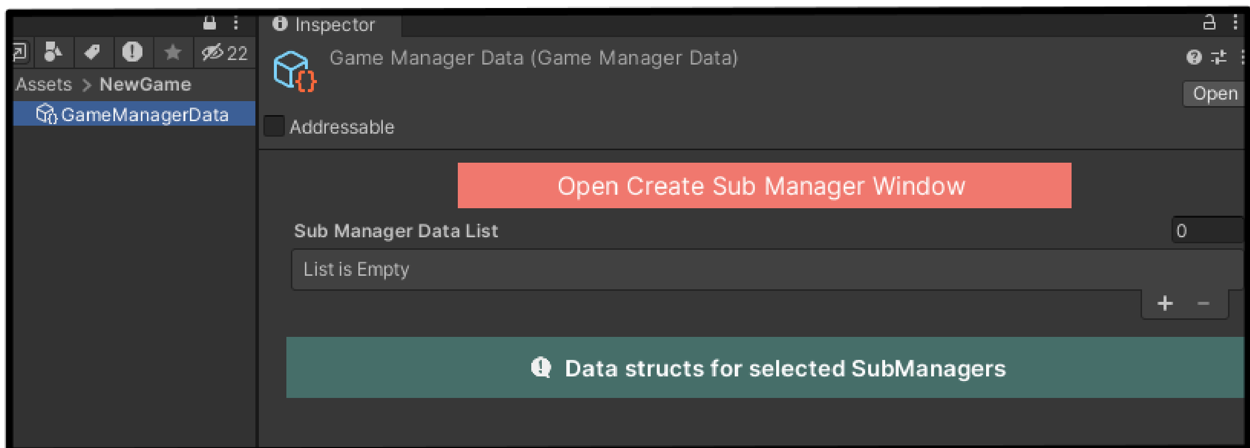
Create a new GameObject, and attach the component GameInstance anchor gameobject



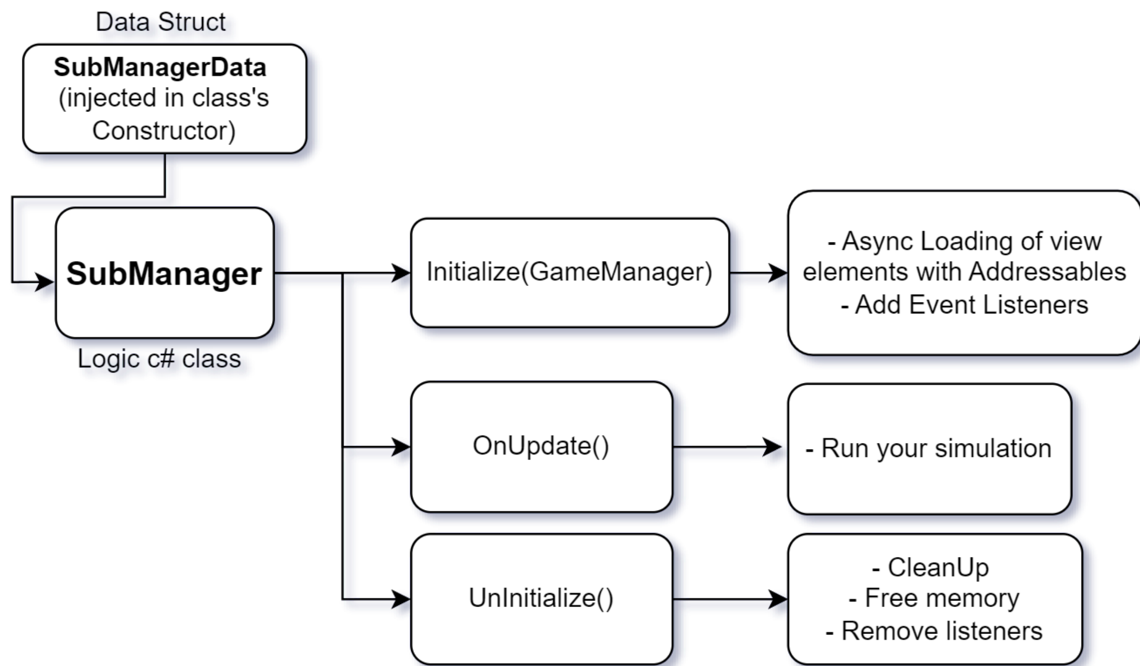
2.- Create a new GameManagerData scriptable object.





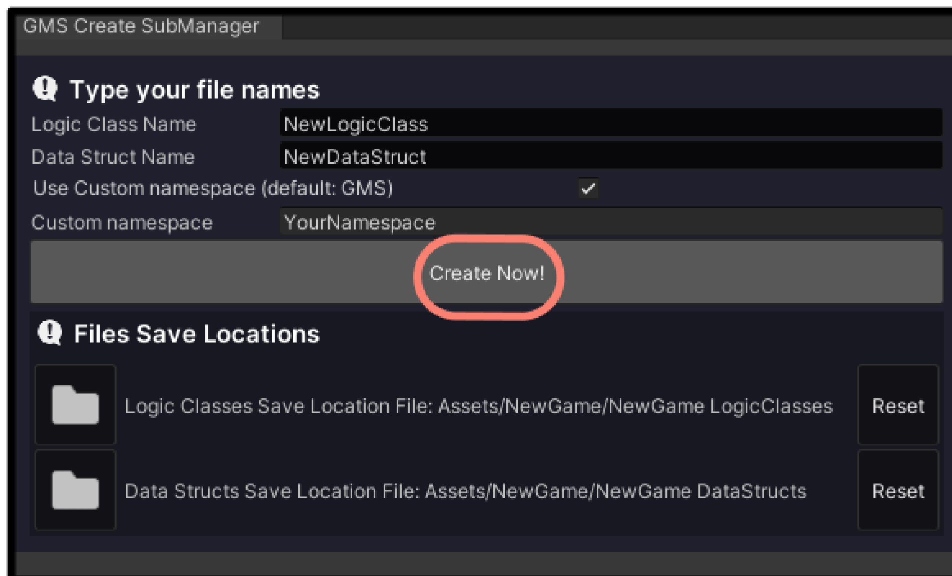


3.- Create your own SubManagers



SubManager Diagram

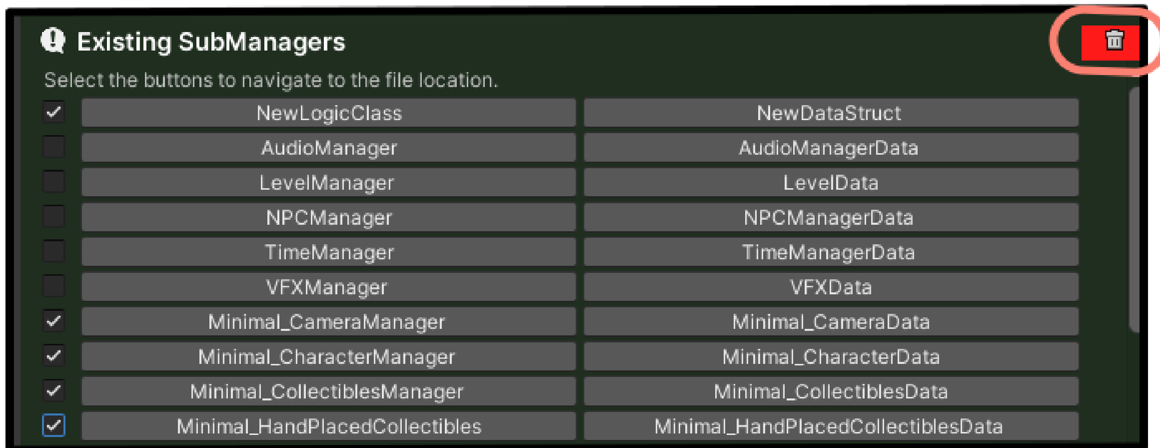
A SubManager can be created faster than ordering a combo meal at your local fast food restaurant.
As professional Game development managers can be handling the following: Camera, NPCs, Level loading, Props, Collectibles, Audio, UI, Loading screens etc...



Here you have the chance to Add new subManagers and their data struct with a simple window provided by GMS.

- **Name your class and struct** in the fields,
- **Select the File save location**, (can always use the default one,set it by hitting the reset button)
- (Optional) select your own namespace, as it defaults to GMS.

If you want to delete both associated files for the sub manager, you can always just select one or multiple and hit the Trash icon.



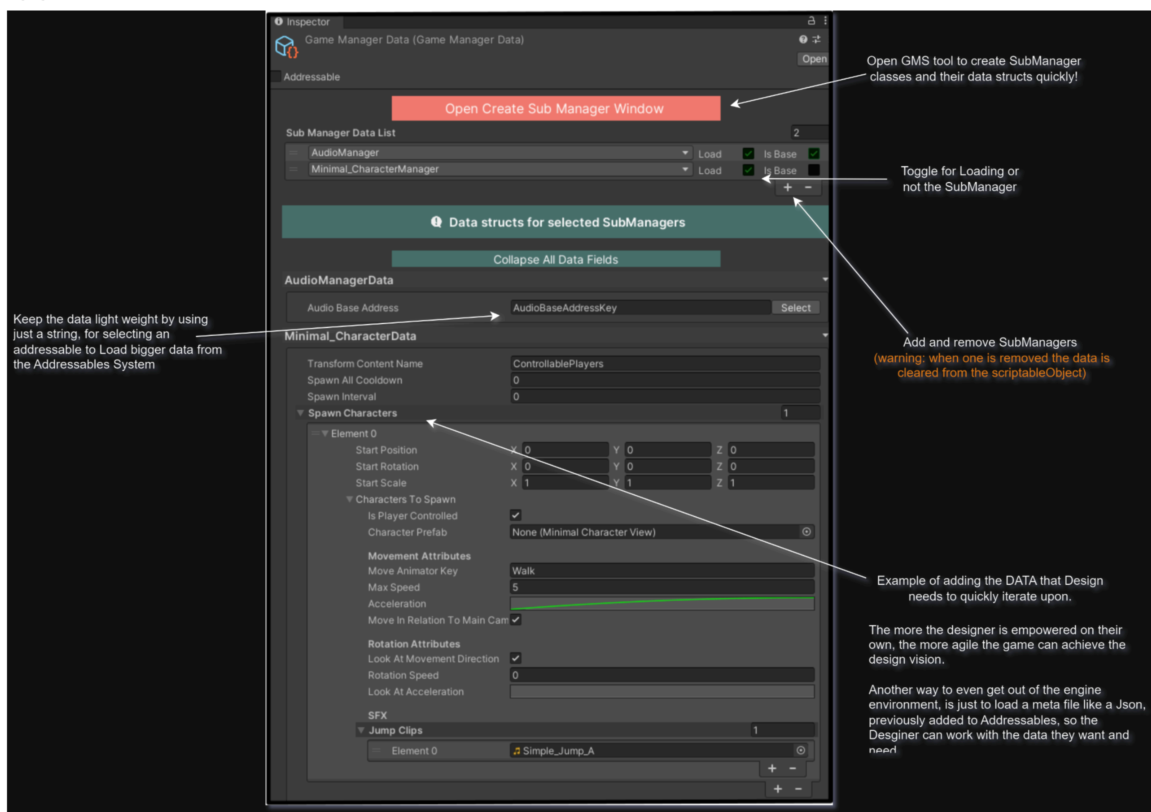
(Optionally) you can also create your logic class and data struct yourself, but chances are it will not work with this system as easily as there is an advanced system of Editor drawing techniques and reflection to make this experience as quick and user friendly as possible.

You are now expected to add your own code to your submangers in order to get your game started.

Inspecting the provided game samples in the Game Samples folder, you might be able to see examples including:

- Load Assets from Addressables using Async operations.
- Dispose of resources adequately
- Use of included utility ViewContent, for easily handling view-side object pooling.
- Dynamically add and remove SubManagers to the main GameManager class. All in the Model.
- Implement your **unit-tests** and **Automated testing**, since all your game should be able to run in the sim with GMS, you are able to simulate more than one SubManager/Service from your game and validate it works as expected.

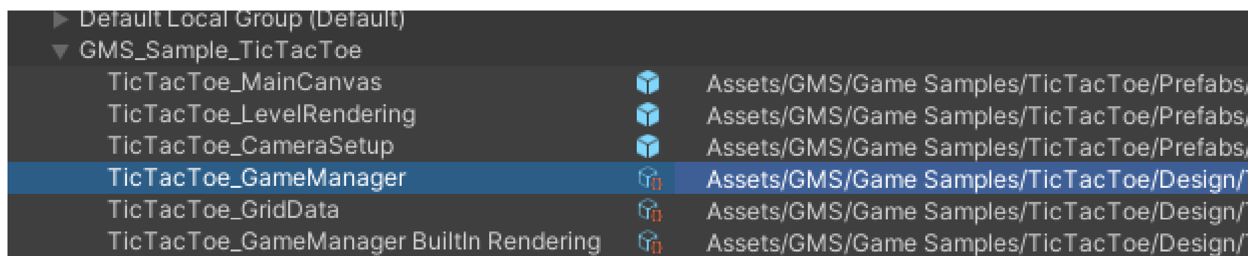
4.- Add your SubManagers and their Data
Example GameManagerData holding any amount of SubManagers, with loading and base toggles.



5.- Add GameManagerData to your Addressables

If your project does not contain Addressables, follow this guide to set them up.
https://docs.unity3d.com/Packages/com.unity.addressables@0.8/manual/AddressableAssetsGettingStarted.html

Now when Addressables are ready, simply drag your file into the Addressables Tab, as easy as that!



6.- Test your game.

If the setup is correct the steps run as follows

Loading GameManagerData:

- The GameInstance will first load the GameManagerData from the Addressables system using the provided key. This step ensures that the correct configuration and data for your game are loaded dynamically, allowing for flexible and efficient game management.

Creating the GameManager:

- The GameInstance then creates a GameManager using the loaded data. During this process, all SubManagers (or services) specified in the GameManagerData are instantiated and added to the GameManager. This step binds the core logic and data together, setting up the foundational structure of your game.

Binding and Initialization of SubManagers:

- Each SubManager is then bound with its corresponding data, linking the logic to the specific parameters and configurations defined in your GameManagerData. Following this binding, the SubManagers are initialized in a sequential step, ensuring that all necessary components are ready to manage their respective systems in the game.

SubManagers Running the Game:

- After initialization, the SubManagers take over the management of their designated systems, effectively running the rest of the game. Each SubManager has access to an Update method, allowing it to continuously manage and update its assigned tasks as the game progresses.

Testing Dependencies and Race Conditions:

- To ensure the robustness of your system, toggle SubManagers on and off during gameplay. This process will help you identify any hard dependencies between them that could lead to errors or failures. Additionally, observe whether any race conditions arise when SubManagers are toggled or initialized in different orders. GameManager ensures to create all sub Managers first before initializing them, so this already prevents most racing conditions when trying to fetch another Sub Manager.

```
gameManager.TryGetSubManager(out AudioManager _audio);
```

Single line example of how to get another SubManager

All of this takes care of the usual Awake, Start, Update methods that regular Monobehaviours have that most systems use.

Comprehensive Guide

Audience

This guide is intended for intermediate to advanced users with a solid understanding of game development and experience working with Unity. It assumes familiarity with the following concepts and tools:

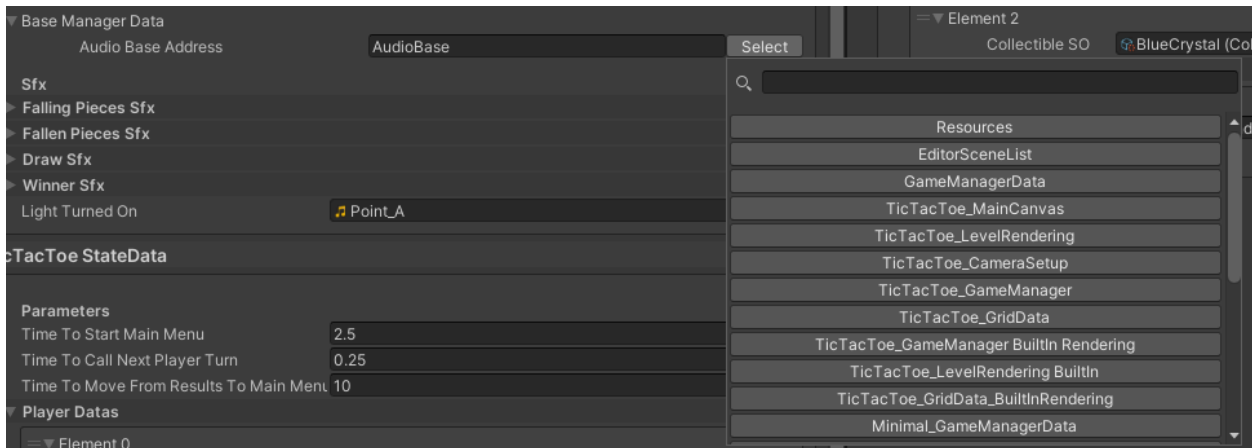
- **C# Programming:** Proficient in writing and debugging C# code, including object-oriented programming principles.
- **Unity Editor:** Comfortable navigating the Unity Editor, including creating and managing GameObjects, using components, and setting up scenes.
- **Scriptable Objects and MonoBehaviours:** Understanding of how to create and use Scriptable Objects and MonoBehaviour classes in Unity for game data and logic.
- **Addressables:** Basic experience with Unity's Addressable Asset System for dynamic asset management and loading.
- **Game Architecture Patterns:** Familiarity with common game programming patterns such as Singletons, Service Providers, and Entity-Component-System (ECS) architecture.

This guide is not recommended for beginners who are just getting started with Unity or C# programming, as it dives into advanced concepts and practices that require a foundational knowledge of game development.

Take a look at the provided examples so you can learn more
All game samples contain scenes supporting both URP and Built-In Rendering pipelines.

GMS has included an attribute drawer for easily selecting Addressable keys.
Simply add the attribute "AddressableSelector" to your strings

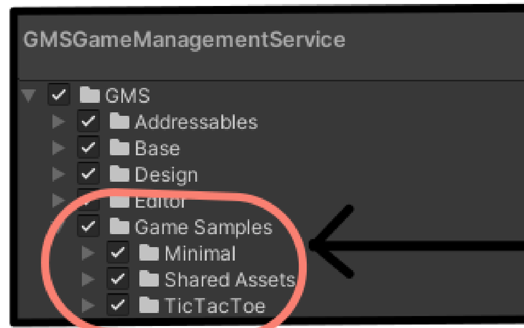
```
[AddressableSelector]
public string AudioBaseAddress;
```



String with Addressable selector, in order to easily select the addressable key, and avoid typos or string errors.

Game Samples

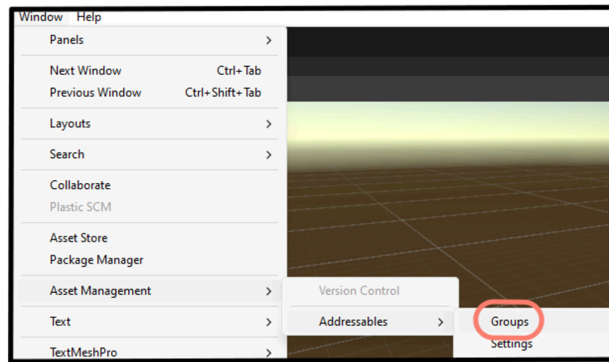
When importing the asset, ensure to include the folders containing the Game Samples
They all are under the namespace GMS.Samples to avoid getting in your way.



Since GMS heavily focuses on engineering best practices and fully utilizes unity's built in features like Addressables, Ensure to add the Samples Addressables so their Assets can be loaded using that system.

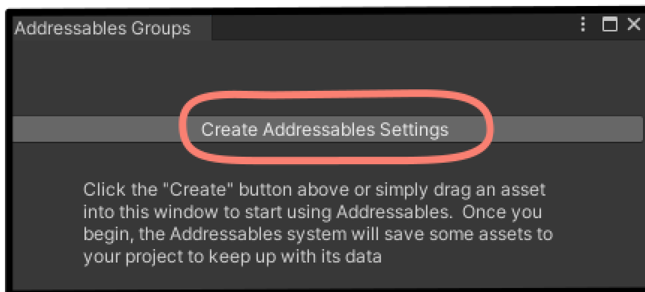
In case your project doesn't have Addressables Setup yet, just follow this 2 steps:

1.- Open Addressables Group

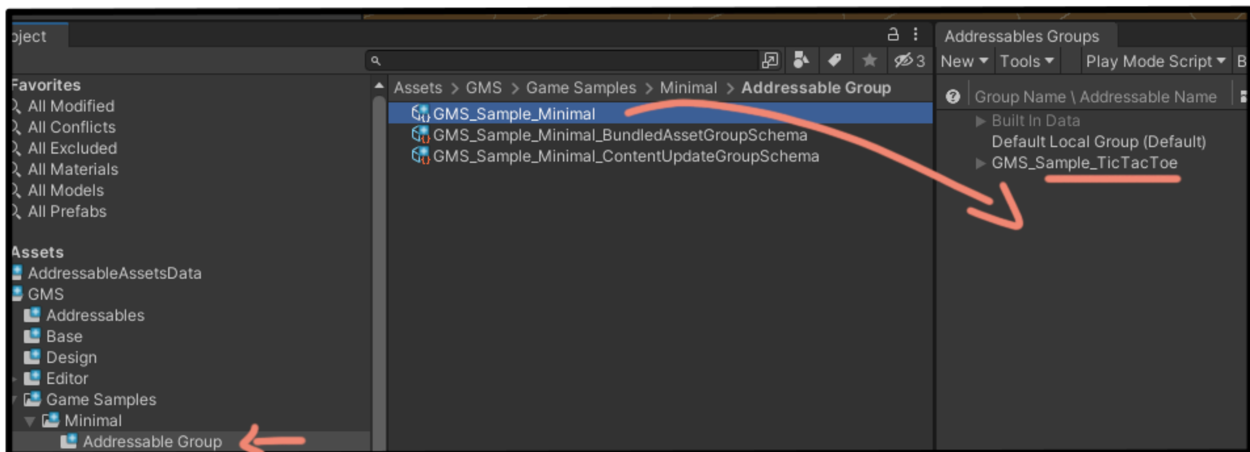


Window/Asset Management/Addressables/Groups

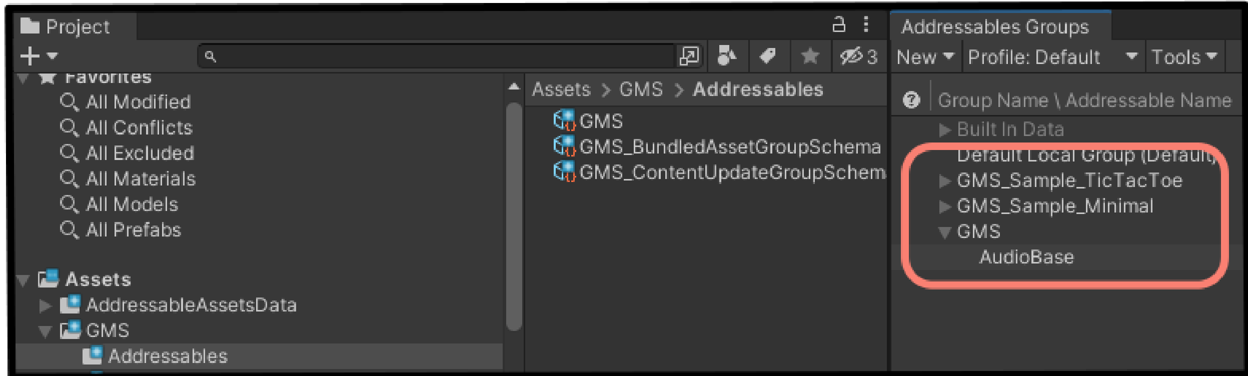
2.- Create Addressables Settings button



3.- Drag the Samples Addressable Groups



Simply drag n drop the Addressable files into the AddressableGroups Tab

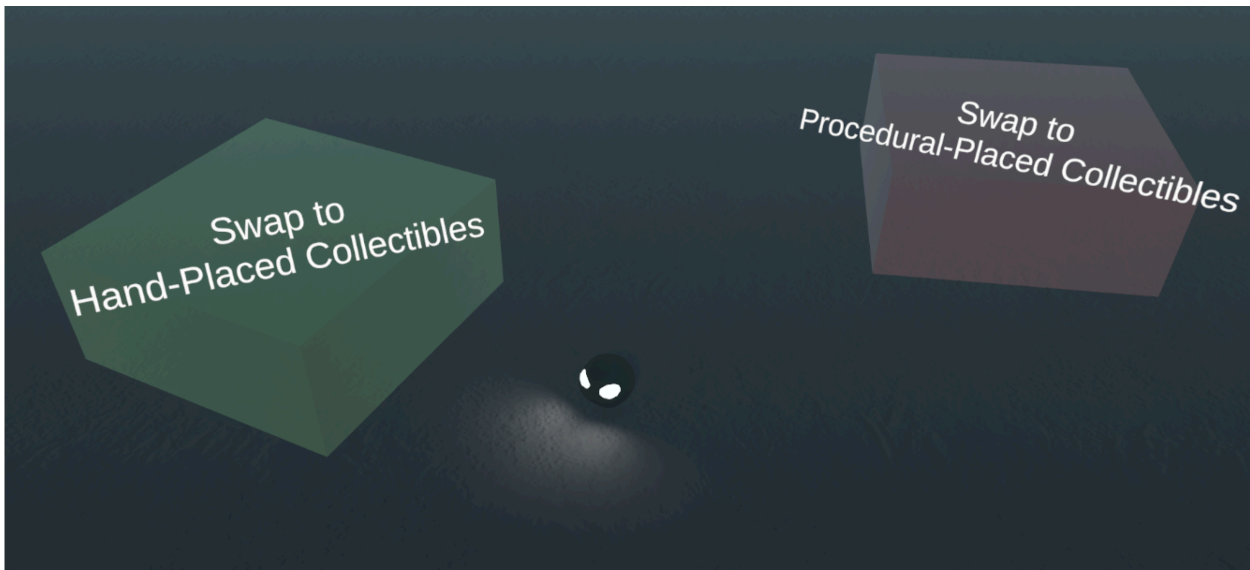


Ensure all 3 Addressable files from the Addressable folders are dragged into your project's Addressable Groups

Minimal Game Sample

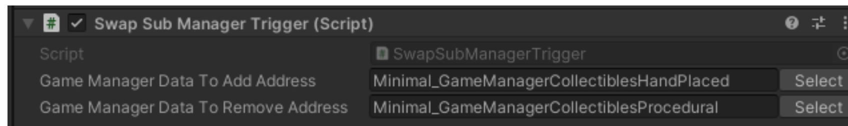
This is a small game where there is a single npc being spawned that is player controllable,
And some collectible gems spawn, pickups are communicated using "The observer pattern" or game event as a simple action invoke system.

Fire up scene "MinimalExample" in GMS\Game Samples\Minimal\Scenes\URP - Universal Render Pipeline and give it a try.



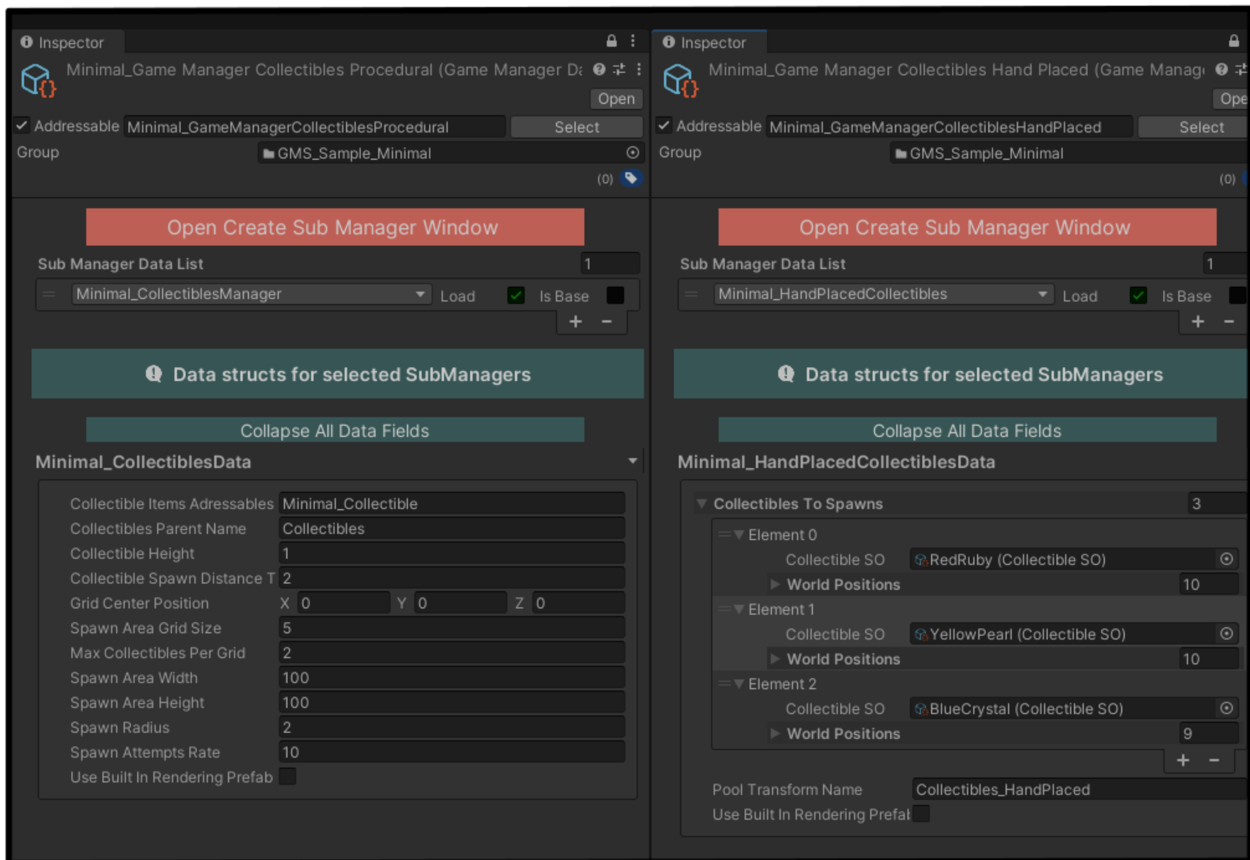
There is an example Scene on how the collectibles are spawned, and in game 2 sub managers can be dynamically swapped when stepping into 2 trigger colliders.

This showcases how Submanagers can be easily loaded and unloaded cleanly, which can happen when loading a cave, a building interior, a cabin interior, a hub scene, in between loading screens for your game.



The triggers contain Swap SubManagerTrigger

This Swaps the GameManagerDatas linked to those through addressables.
What a best feature to be able to test different mechanics in game with easy data manipulation.



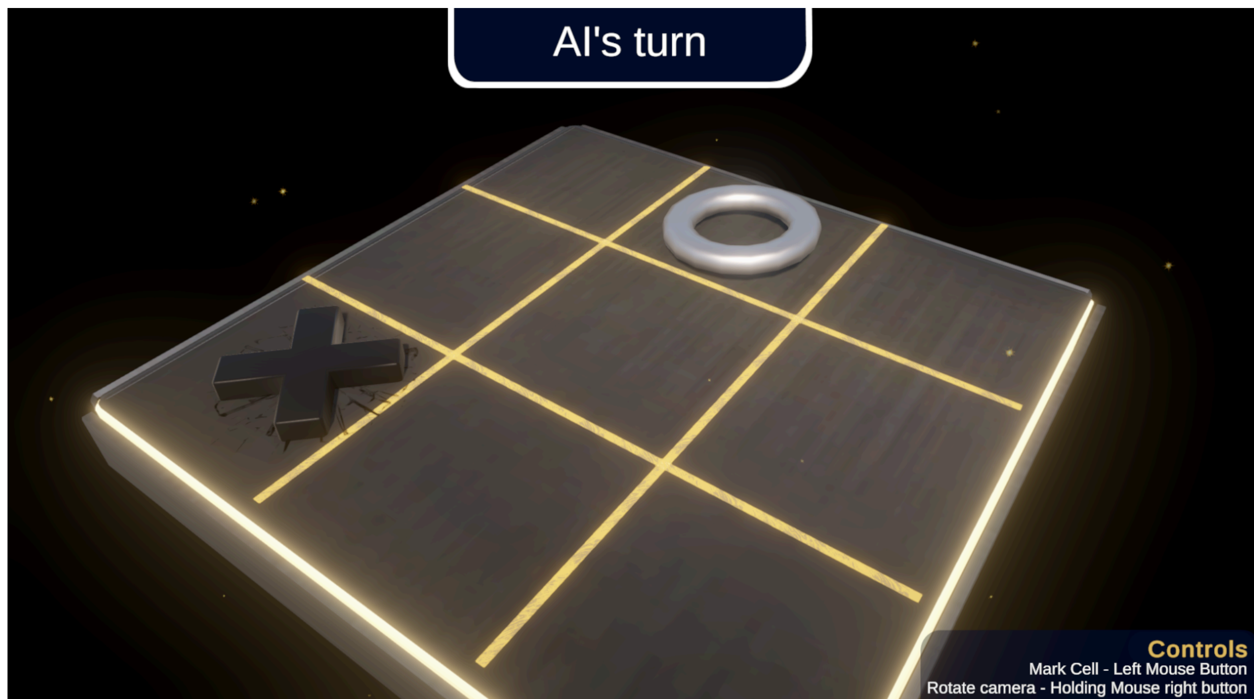
SubManagers data are editable by the Game Designer.

Empowering game design pays off in dividends, allowing them to test different sub managers that do similar stuff, and validate their design through gameplay and feedback.

TicTacToe Game Sample

This game sample implements the classic TicTacToe and all the logic since it's simplicity allows to, can run in some unit tests that validates matches can be won by a player or end up in a draw.

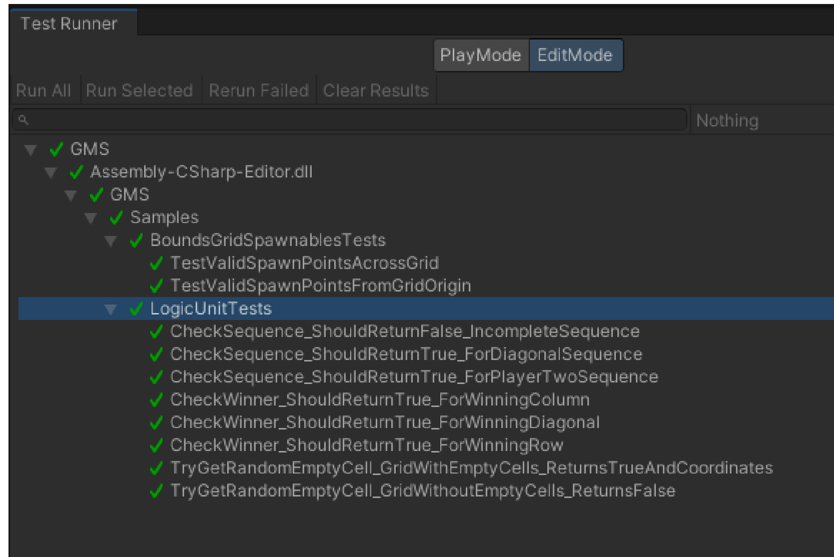
Fire up scene "MinimalExample" in GMS\Game Samples\Minimal\Scenes\URP - Universal Render Pipeline and give it a try.



The view is handled with ViewContent, a utility provided by GMS, and provided graphics, meshes and prefabs.

The controller is handled by a single SubManager input, and in some cases there are Events that help communicate in a decoupled way, using the Observer Pattern. See GlobalAcions.cs

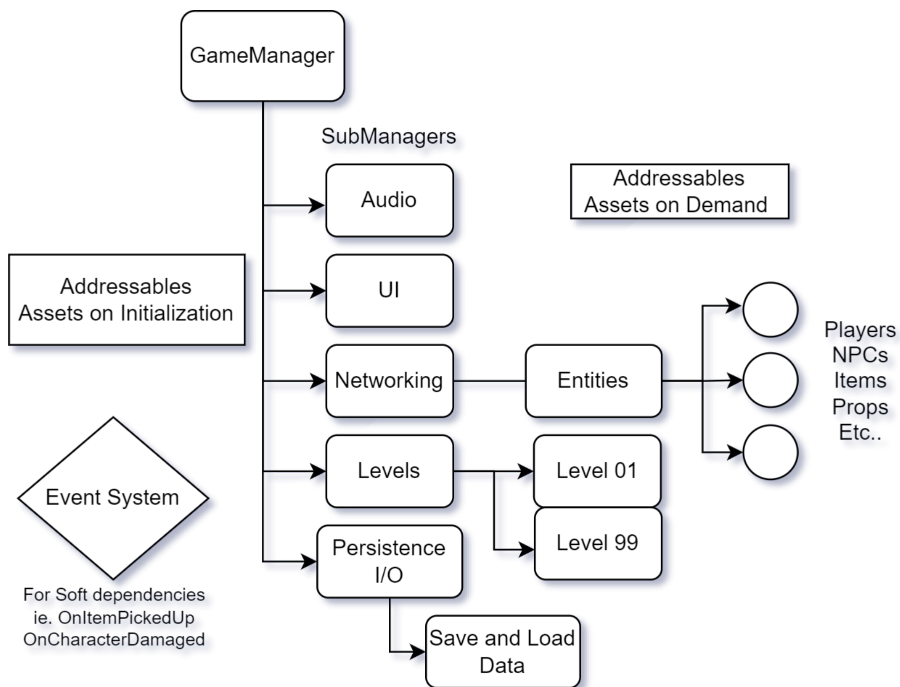
If you open the TestRunner Tab in unity, you'll find provided Unit tests that validate the core gameplay.



GMS Workflow

Scalability

GMS sub manager examples include a data struct using a single Addressable key.
Or in some cases it exposes direct variables and prefab inputs.
However depending on the game scale, GMS workflow encourages to keep it from small and to big as needed.



For instance looking at the diagram above, the initial Data structs may only contain a single Addressable key, for a scriptable object or data file that has more Addressable keys for other medium or smaller asset collections.

Loading Data on demand rather than in huge chunks, and loading only what's needed for the current and or near future game state to run.

Iterative design

Having the ability to toggle off the loading field of SubManagers, allows the team to avoid loading big systems when a particular part of the game is being iterated upon, reviewed, debugged, etc...

Try answering this questions if you have a game already and not using GMS or a similar system:
1.- Are you able to turn off one of your systems in your current game to see if you even miss it, or if something breaks?
2.- Are you able to have alternative databases and your designer can easily test them on their own? “Data bases such as the entire NPC or ITEM data files”
3.-Are you able to create an automated test with 2 systems without relying in the view side?
4.- Are you able to create a new mechanic or system without having to hook it up to several other systems in your game to even get it instanced in view?
5.- Are you able to make a new system that uses more than 75% of an existing system, and would take a lot of time to develop so that instead it just gets discarded, as everything is so entangled that it becomes too expensive to change?

If you answer YES to all of them chances are your game already has a system that allows a high iterative design, if not GMS is a solution that provides examples where it answers YES to all of them.

Modular

Being able to...

- Automate tests from data driven classes
- Empowering the Designer or Design team, to manually configure the game submanagers and data configurations.
- Being able to efficiently test a mechanic without having to load 90% of the stuff we are not testing.

- Being able to transport the game core logic to any other engine

...ultimately save time, money(making the game cheaper to produce), and enjoy a happier efficient game development workflow.

That's what GMS stands for.

Terminology

Game Designer

The sole dev, or team member specialized in designing the game.
Game design technically structures game mechanic in a cohesive way in a safe enclosed system.

The Game designer is not "the ideas" member.

Design is about taking decisions and creating experiences!

GameInstance

The anchor point or gameobject that serves as the main parent for the view side objects that get Instantiated through the unity objects Instantiate method.
https://docs.unity3d.com/ScriptReference/Object.Instantiate.html
We require such entity to exists so we can have a clean inspector when working in the editor, with readable names as we prefer NPC_Batsy_3 under the npc/flying/, over "InstanceNumber 00023827918392183" that would not even have a parent assigned.

GameManager

This class holds the created SubManagers, and runs their lifecycle: the initialization, uninitialization, and update method calls on them.

GameManagerData

A scriptable object that contains a list of submanagers logic classes and their data structs, so they can be binded and created.

SubManager or Service

I constantly use in code the term sub manager instead of service, but I call the asset Game Management Service, why is that? Because in terms of marketing a service sounds better to me and less technical than a sub-manager. But in reality in code it

makes more sense to think of sub managers as smaller parts or under the main Game Manager. Almost as part of a blueprint or recipe if you may.

A sub manager and service is just a core class that handles or “manages” one of the main systems of a game, such as: loading screens, NPC Spawning, UI canvases and windows, Missions systems, Physics systems, etc…

Although this system preaches the use of Composition over inheritance, meaning the main classes don’t inherit from any class, but can use and create other classes internally and reuse code, without inheriting from them. And in the case of having to allow the use of the same method, the interfaces come in to fix that need.

Data-Oriented processing

in the context of games, refers to an approach to software design and optimization that focuses on the efficient organization and manipulation of data to maximize performance, especially in terms of memory access and CPU cache utilization. This approach contrasts with object-oriented design, where the focus is on encapsulating behavior and data together within objects.

Addressables

Addressables in Unity is a powerful system that allows developers to manage and load assets dynamically at runtime. Instead of embedding all assets directly into the game build, Addressables enable you to store them separately and load them only when needed, which can significantly optimize memory usage and reduce load times. This system is particularly useful in large games or projects with many assets, as it allows for efficient content management, easy updates, and even remote content delivery. By using Addressables, you can streamline your asset workflow and improve the performance of your game.

Scriptable Objects

Scriptable Objects in Unity are a flexible and powerful way to store data independently of game objects. Unlike MonoBehaviours, Scriptable Objects do not need to be attached to game objects, making them ideal for defining and organizing game data such as configurations, settings, and shared resources. They can be easily created, edited, and reused across different scenes and projects, helping to keep your codebase clean and modular. By leveraging Scriptable Objects, you can improve data management, reduce code duplication, and simplify the process of making changes to your game’s behavior and content.