MasterCaster

Technical Design Document

Members:
Damian Dalinger
Lena Ngoc Truong
Philipp Birk
Simon Horatio Gremm
Unik Kelmendi

Last Modified: June 10th, 2025
Do not distribute without permission!

Welcome to the Tech Bible for our project - a game podcast simulator and management game currently in development for PC. This document serves as a central reference for our structure, pipeline, technical direction and programming standards.

Our primary goal is modularity. Every system, asset, and workflow should be designed to scale, adapt, and be reused with minimal friction. Following the guidelines here ensures consistency, cleaner collaboration, and a faster development process.

I'm Damian, the lead programmer of this project. While this document aims to define best practices, it's not written in stone. If you find yourself working against the guide rather than with it — bring it up. We're always open to improvements. Just talk to me, and we'll figure out a better way together.

Let's keep it clean, modular, and fun.

Table of Contents

1. Project Setup

1.1 Project Constants

To maintain a clear and unified structure throughout the project, this section documents all globally used Unity constants such as Sorting Layers, Physics Layers and Tags. Tracking these elements helps avoid duplication, ensures consistent usage across scenes and systems, and makes onboarding easier for new contributors.

1.1.1 Sorting Layers

Sorting Layers determine the visual rendering order of 2D elements. They are especially important for organizing background, gameplay, and UI elements cleanly. Define only what you actually use and document each layer's intended purpose.

Layer Name	Use Case	Notes

1.1.2 Physics Layers

Physics Layers are used for collision detection, raycasting, and performance optimization. Documenting them helps manage interaction rules and makes collision matrices easier to maintain.

Layer number	Layer Name	Use Case
0	Default	The default layer for all scene elements.
1	TransparentFX	Unity uses this layer in the flare system.
2	Ignore Raycast	Physics ray cast APIs ignore this layer by default.
4	Water	Unity's Standard Assets for Unity 2018.4 use this layer.

| 5 | UI | The Unity UI uses this as the default layer for UI elements. |
| | | |

1.1.3 Tags

Tags allow identifying objects in scripts without relying on scene references. Keeping this list tidy avoids misuse and improves readability across systems.

Tag Name	Use Case	Notes

1.2 Tools

- Unity Engine 6000.0.44f1
- GitHub
- GitHub Desktop
- Visual Studio Code 1.99.0
  With these Extensions:
    - C#
    - C# Dev Kit
    - GitHub Actions
    - Unity
    - Unity Code Snippets
- FMOD Studio Version 2.03.07
- Procreate
- Reaper
- (FL Studio)
- Krita

1.3 System Requirements

Note: These requirements are based on the current concept and intended scope of the game. As development progresses and features are added, these values might change.

Minimum Requirements
These specs aim to ensure the game is playable on low-end or older systems, especially useful for testing accessibility and portability.

- OS: Windows 10 (64-bit)
- Processor: Intel Core i3-6100 or equivalent
- Memory: 4 GB RAM
  Graphics: Integrated GPU (e.g., Intel HD 530) or dedicated GPU with 1 GB VRAM
- Storage: 2 GB available space
- DirectX: Version 11
- Display: 1280x720 minimum resolution

Recommended Requirements
Optimized for smooth gameplay and better visuals.

- OS: Windows 10 or 11 (64-bit)
- Processor: Intel Core i5-8400 or equivalent
- Memory: 8 GB RAM
- Graphics: NVIDIA GTX 1050 Ti or AMD equivalent (2 GB VRAM)
- Storage: SSD with 2 GB free space
- DirectX: Version 11 or higher
- Display: 1920x1080 resolution

2. Modular Game Architecture

In modern game development, modular design is essential for creating flexible, maintainable, and scalable systems. In our project we use an architecture which was introduced by Ryan Hipple from Schell Games, which uses ScriptableObjects as foundational building blocks. Scenes should be lists of Prefabs and the Prefabs should contain the individual functionality. By structuring systems around data assets instead of tight object references, we gain three major advantages:

- **Modular:** Systems don't rely on each other - no hard references. Scenes are clean slates (no cross-scene data). Prefabs work independently and hold their own functionality. Components do one job only.

- **Editable:** Focus on data, not code. Designers can adjust gameplay and balance at runtime - without a developer.

- **Debuggable:** Test systems in isolation. Understand every bug before fixing it. Inspect and tweak values live in the Editor.

2.1 Variables

To support modular, flexible, and designer-friendly systems, we use ScriptableObjects to represent shared game data - such as floats, vectors, strings, and booleans. Instead of storing variables directly on MonoBehaviours, values like health, gravity, or enemy speed are defined as assets in the project. Each variable asset is structured to support both temporary and persistent usage. It includes the following fields:

- **RuntimeValue:** The current in-game value, which can be read or modified during gameplay.

- **InitialValue** (optional): The value used to initialize RuntimeValue, when a new game starts.

- **isSaveable** (bool): Determines whether the variable should be persistent across game sessions.

Variable Behaviour Based on isSaveable:

- If isSaveable is false:
  Only RuntimeValue is displayed and used. The variable is considered non-persistent and is not saved between sessions. This mode is ideal for temporary, fixed or runtime-only data.
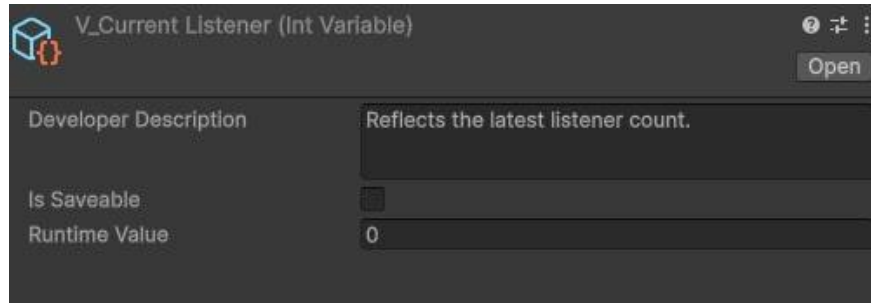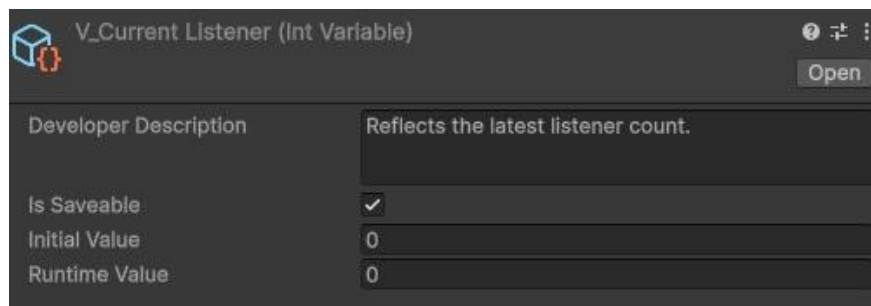

- If isSaveable is true:
  Both InitialValue and RuntimeValue are visible in the Inspector. When a new game starts, RuntimeValue is initialized using InitialValue. During gameplay, RuntimeValue may be modified by game systems or events during gameplay. These changes are saved at the end of each in-game day and restored in future sessions.

Each variable asset also provides a set of core methods, such as SetValue, ApplyChange, Multiply, or Divide, depending on the data type. This allows for basic value manipulation directly via code or UnityEvents. Additionally, each variable includes a Developer Description field, allowing you to annotate its purpose in the Inspector. This helps clarify usage for other developers or designers.

Persistent data - including all saveable variables and serializable runtime sets - is stored in a .json file on disk: AppData → LocalLow → DefaultCompany → ProjectCeros → save.json
Designers and developers do not need to manage it manually, but can inspect it for debugging purposes.

This `FloatVariable` can now be referenced in any MonoBehaviour, giving all objects a shared, editable value. However, sometimes you don't want to always use the

shared value. To add flexibility, we use a wrapper called `FloatReference`, which lets you switch between a constant value or a linked `FloatVariable`:

```csharp
[Serializable]
0 references
public class FloatReference{
    1 reference
    public bool UseConstant = true;
    1 reference
    public float ConstantValue;
    1 reference
    public FloatVariable Variable;

    0 references
    public float Value{
        get {return UseConstant ? ConstantValue : Variable.Value; }
    }
}
```

This is especially useful during testing or prototyping phases, where designers may want to quickly try out different values without creating or modifying separate assets. By switching to constant mode, variables can be adjusted in place and seen in action immediately - ideal for fast iteration.

Example Use Case: Let's say we have a simple enemy that needs a health value

```csharp
public class Enemy : MonoBehaviour
{
    0 references
    public FloatReference Health;
}
```

In the Inspector, you can choose to enter a constant value directly (e.g. 100) or reference a FloatVariable asset (e.g. "EnemyDefaultHealth")
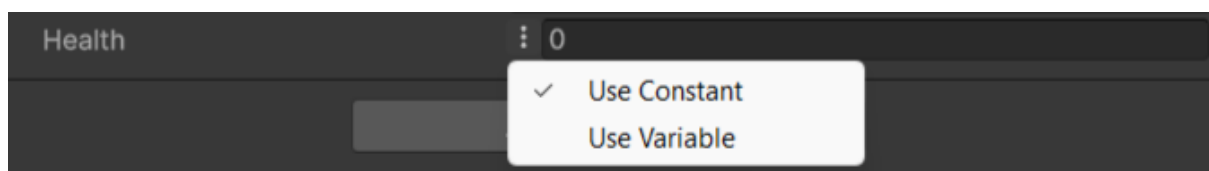
When using a FloatVariable, you can set isSaveable to true and assign an InitialValue of 100. At the start of a new game, the RuntimeValue will be initialized to 100. During gameplay, damage or healing will modify the RuntimeValue and those changes will be saved and restored when the game is continued.

Our system currently supports the following variable types, which can be found via Create → Variables.
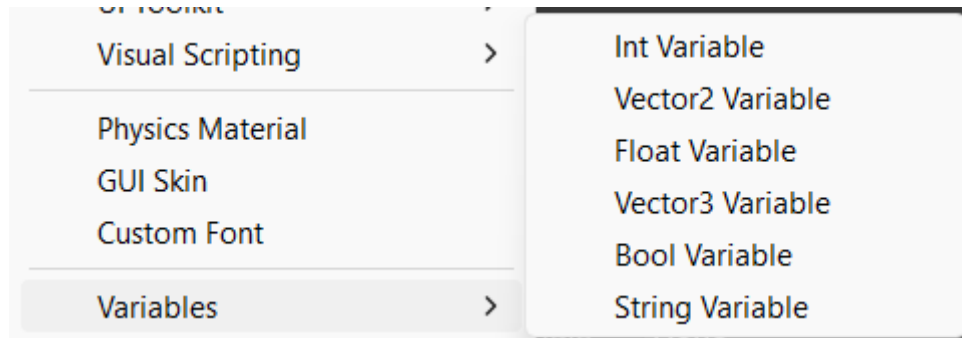
2.2 Event System

The Event System enables clean, decoupled communication between gameplay systems using Unity's ScriptableObject architecture. Instead of hardcoding references between components, we raise GameEvents, which notify all registered GameEventListeners - even across unrelated scenes or systems.

Let's walk through a typical example: triggering a "Player Died" event. We begin by creating a new GameEvent with Create → GameEvent. This asset acts as a signal - it holds no logic by itself, but when raised, it will notify any registered listeners.

You can raise the event from code like so:

```
PlayerDied.Raise();
```

Or manually in the Unity Editor using the built-in "Raise" button during Play Mode:
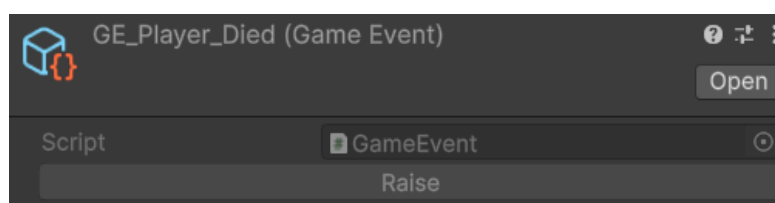
To react to the event, attach a GameEventListener component to any GameObject and hook up a UnityEvent to define what should happen (e.g. play a sound).
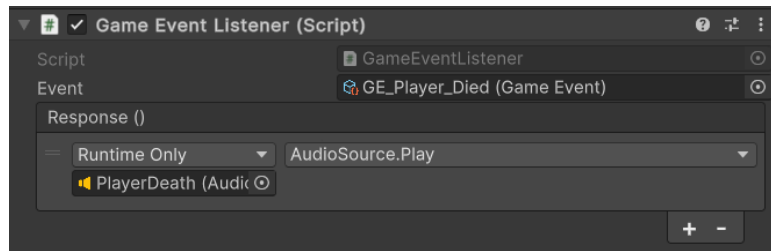
Now, when the event is raised - either from a script or via the Raise button - all listeners will execute their UnityEvent actions. This makes it easy for designers to define behaviors without writing code.

During development, it's important to verify that events are actually firing and being received. That's where the Event Log Window comes in.
Open it via: Window → Event Log
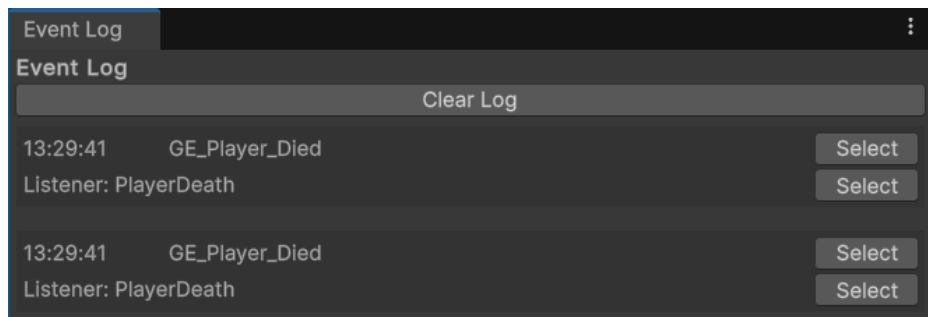
2.3 Runtime Sets

Runtime Sets are ScriptableObject-based lists that keep track of active objects of a specific type during gameplay. They're useful for tracking dynamic collections like enemies, interactables, or pickups – especially when the scene changes frequently or you want global access without tight coupling.

Persistent vs. Non-Persistent Runtime Sets
There are two types of Runtime Sets in the system:

- Non-Serializable Runtime Sets
  These are designed for temporary, in-session tracking (e.g., active enemies, visible interactables). Their content exists only during gameplay and resets every time the game is started. These types of sets do not display any persistence options in the Inspector - the isSaveable toggle is not shown.
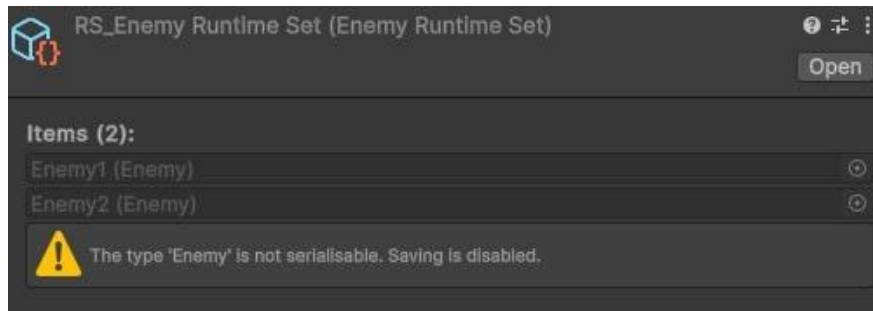
- Serializable Runtime Sets
  These support saving and restoring content between sessions (e.g., an item database, discovered lore entries, unlocked features). In this case, the Inspector will include an isSaveable boolean.

Persistent data - including all saveable variables and serializable runtime sets - is stored in a .json file on disk: AppData → LocalLow → DefaultCompany → ProjectCeros → save.json
Designers and developers do not need to manage it manually, but can inspect it for debugging purposes.

Let's say we want a list of all active enemies. Create a class inheriting from RuntimeSet<T>, like so:

```csharp
[CreateAssetMenu(menuName = "Runtime Sets/EnemyRuntimeSet")]
1 reference
public class EnemyRuntimeSet : RuntimeSet<Enemy> { }
```

Right-click in the Project window → Create → Runtime Sets → EnemyRuntimeSet

This creates the actual asset that holds the list during runtime. In your tracked MonoBehaviour (e.g., Enemy.cs), reference the EnemyRuntimeSet and add register/unregister methods:

```
public class Enemy : MonoBehaviour
{
    2 references
    [SerializeField] private EnemyRuntimeSet _runtimeSet;

    0 references
    private void OnEnable()
    {
        _runtimeSet.Add(this);
    }

    0 references
    private void OnDisable()
    {
        _runtimeSet.Remove(this);
    }
}
```

Now you can access the list of active enemies anywhere in your game logic (UI, managers, etc.).

2.4 Audio System

The Audio System uses ScriptableObject assets (FMODSoundEvent) to reference FMOD events and wrap audio playback functionality in a designer-friendly way. These assets allow for consistent access to audio events across the project without manually handling FMOD instances. To create an event, right-click in the Project window → Create → Audio → FMOD Sound Event.
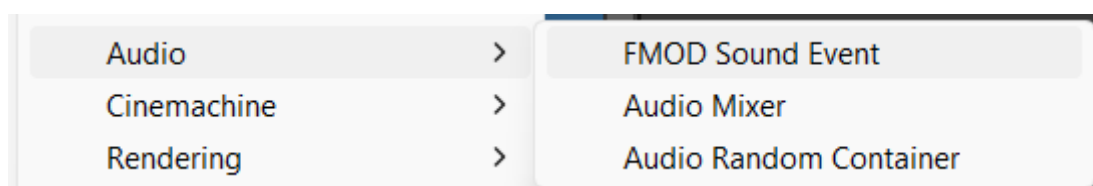
Available Methods:
Each FMODSoundEvent exposes the following methods:
  ● **Play()**
    Plays the sound as a 2D one-shot.

  ● **PlayLoop(Transform target)**
    Starts a looped sound on a specific GameObject. Internally managed by an FMODSoundAutoHandler.

- **StopLoop(Transform target)**
  Stops the looped sound on the given target.

- **SetParameter(Transform target, string parameterName, float value)**
  Updates a real-time parameter for a currently playing loop (e.g., intensity, pitch).

All looped sounds are managed automatically and cleaned up on object destruction. This modular approach ensures that designers can trigger or stop audio via UnityEvents or visual scripting, without writing custom FMOD logic.

2.5 Scene Architecture

At the core of the system is a single entry point - the Bootstrapper Scene - which stays loaded throughout the entire application lifecycle. It contains all global managers such as the Audio or SaveManager. This scene is always loaded first and never unloaded. From there, the rest of the game follows a modular flow: scenes are loaded additively when needed and unloaded when no longer required. Each contains only what is necessary for its purpose, keeping systems decoupled.

Scene loading is handled via GameEvents, enabling systems to interact without direct references. The following helper components handle scene transitions:

- **LoadSceneAdditiveOnEvent** → Loads scenes when triggered

- **UnloadSceneOnEvent** → Unloads scenes when triggered

- **ToggleSceneOnEvent** → Alternates between load/unload on each trigger

These components are reusable and designed to work with Unity's inspector-friendly SceneAsset drag & drop.

Scene Overview
The following table lists all scenes used in the project along with their primary responsibilities. Please keep the list alphabetically sorted by scene name.

Scene Name	Purpose & Content
Bedroom	Contains the bedroom environment and interactable objects.
Bootstrapper	Holds global singleton systems such as AudioManager, SaveManager, etc.

GameSession	Manages the gameplay session; includes systems like DayManager, NewsManager.
HUD	Displays the in-game heads-up display.
MainMenu	Instantiates the New Game and Continue Game prefabs to handle the game start logic.
NewspaperUI	Displays the newspaper interface.
PauseMenu	Handles game pausing and resume logic.
PodcastCreationUI	Provides the podcast creation user interface.

To simulate a complete game environment while editing an individual scene, use the @EditorSceneInitializer prefab. Add it to the scene root, and it will automatically load required scenes (e.g., Bootstrapper) to mirror a full runtime context - without needing to start from the Bootstrapper manually.

Main Menu Logic
The MainMenu Scene is responsible for starting or continuing a game session via two prefabs:

New Game Prefab
Triggers the following sequence:

1. Display the loading screen

2. Reset all saveable variables and runtime sets & delete the existing save file

3. Load all required gameplay scenes

4. Instantiate and initialize session-level managers

5. Unload the Main Menu scene

6. Raise the GE_OnNewGameStarted event

7. Hide the loading screen

8. Self-destructs after completing its task

Continue Game Prefab
The Continue Game prefab follows a similar flow, but instead of resetting data, it restores all saveable data from the save.json file.

These prefabs encapsulate game state transitions cleanly and can be expanded with additional setup logic in the future.

2.6 Loading Screen Manager

The LoadingScreenManager is responsible for displaying a fullscreen UI during scene transitions to visually indicate loading or waiting states. It is instantiated once by the Bootstrapper Scene and persists throughout the application's lifecycle.

On game start, the Bootstrapper instantiates the manager. It provides two public methods:

- Show(): Instantiates and activates the loading screen prefab.

- HideAndDestroy(): Removes the loading screen after loading is completed.

The instantiated screen uses DontDestroyOnLoad() to persist during scene loads. This system is used, for example, by the New Game and Continue Game prefabs to block input and display progress during scene setup and save data restoration.

2.7 Input System

The project uses Unity's new Input System to manage all player and UI input in a modular, scalable, and device-independent way. Instead of hardcoding specific keys or buttons, all input is defined as named actions (e.g. Pause, Navigate, Submit) within a central InputActionAsset. These actions can be triggered by any number of bindings (keyboard, gamepad, mouse, etc.) and are referenced in scripts via InputActionReference.

Input Action Maps in this project are not permanently active. Instead, maps are explicitly enabled or disabled depending on the current game state, UI context or scene requirements. To maintain consistency and flexibility across the codebase, the legacy input APIs (Input.GetKey, Input.GetButton, etc.) must not be used.

Input Action Maps:

Pause: Game pausing via Escape.

Input Action	Binding	Description
Pause	Escape	Opens and closes the pause

		menu.

UI: Used for all menu navigation.

Input Action	Binding	Description
Navigate	ArrowUp, ArrowLeft, ArrowRight, ArrowDown	Moving between selectable UI elements.
Cancel	Escape	Back-navigation in UI.
Submit	Enter	Confirms a selection.
Point	Mouseposition, Penposition	Tracks the pointer position. Required for hover.
Click	LMB, Pentip	Select UI elements.
MiddleClick	MMB	Not used/Required by Eventsystem.
RightClick	RMB	Not used/Required by Eventsystem.
ScrollWheel	MouseScroll	Captures mouse scroll wheel input for scrolling in UI elements like lists or menus.

2.8 Other Considerations

Traditional enum types in Unity come with several limitations. They must be modified in code, making them harder to iterate on and less designer-friendly. Additionally, it can be difficult to remove or reorder them. To overcome these limitations, this project avoids enums in favor of a ScriptableObject-based approach. Each element that would typically be represented by an enum is instead created as a distinct ScriptableObject asset.

3. Style Guide

Consistency is key. All structure, assets, and code should feel like they were created by a single person - regardless of how many contributors there are. A clear, shared style guide removes ambiguity, speeds up collaboration, and reduces errors.

The entire project should be kept in English – this includes folder names, asset labels, comments, variables, and documentation.

Friends don't let friends write messy code. If you see someone ignoring the guide or working without one, help them get back on track.

3.1 Project Structure

A clean and consistent folder structure is essential for working efficiently in Unity. In this project, we use a context-based structure. Unity's search and filtering features make it easy to find assets by type, so the folder hierarchy can focus on content and logic instead.

Note: Work-in-progress content should go into the **_Dev** folder, organized by contributor name.

```
Assets
│
├─── _Dev
│    └─── YourName
│         └─── (Experimental Assets, Test Scenes)
│
├─── _ProjectCeros
│    ├─── Characters
│    │    └─── Player
│    │         ├─── Art
│    │         ├─── Audio
│    │         ├─── Data
│    │         ├─── Prefabs
│    │         └─── Scripts
│    │
│    ├─── Environments
│    │    └─── Bedroom
│    │
│    ├─── Gameplay
│    │    ├─── Newssystem
│    │    ├─── PodcastCreation
│    │    └─── …
│    │
│    ├─── Resources
│    │    └─── Fonts
```

```
            ┌── Scenes
            │   ├── Bedroom
            │   ├── NewspaperUI
            │   └── …
            │
            ├── Scripts
            │   ├── Architecture
            │   ├── Core
            │   ├── Systems
            │   └── UI
            └── Shared
                ├── Events
                ├── Input
                ├── Prefabs
                ├── RuntimeSets
                └── Variables
├── Plugins
```
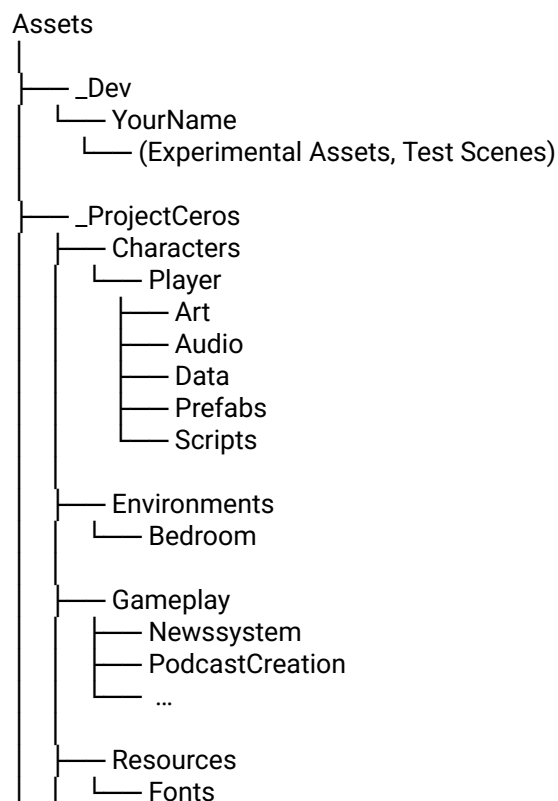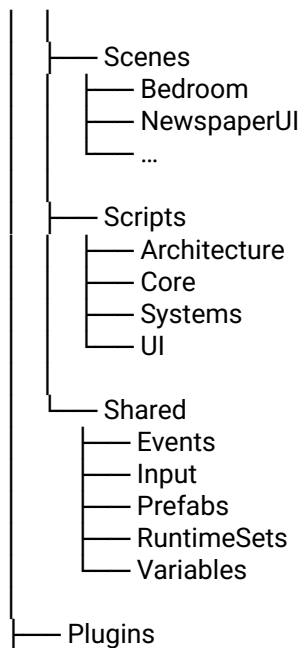
3.1.1 Folder Naming Rules

A consistent naming convention for folders is crucial to ensure cross-platform compatibility, minimize errors in automated tools, and improve project clarity. The following rules apply to all folders within the Unity project:

1. PascalCase Only (e.g. DesertEagle, RocketPistol, ASeriesOfWords)
2. No Spaces
3. No Unicode or Special Characters (only use a-z, A-Z, 0-9, e.g. 'Zoë should be Zoe)
4. No Empty Folders

3.1.2 Developer Folder

During development, team members often need a space to experiment, prototype, or iterate without affecting the core content. For this reason, Developer Folders (also known as sandbox folders) are provided. These folders allow contributors to store work-in-progress assets that are not yet ready for integration into the main project. This prevents unstable or temporary content from unintentionally entering production workflows.

Every asset should have a purpose, otherwise it does not belong in a project. If an asset is an experimental test and shouldn't be used by the project, it should be put in a Developer folder.

Once an asset is ready for production use, the creator simply moves it into the appropriate project folder. This process is referred to as "promoting" the asset - from a prototype to an official part of the game.

3.1.3 Scene Hierarchy Structure

Every scene should maintain a clean and organized hierarchy to ensure readability and fast navigation. While the exact structure depends on the scene's purpose, the root of the hierarchy should only contain empty GameObjects used as folder-like containers, such as UI, Environment, or Management.

- All empty objects should be located at 0,0,0 with default rotation and scale.

- For empty objects that are only containers for scripts, use "@" as prefix (e.g. @Cheats)

- When you're instantiating an object at runtime, make sure to put it in _Dynamic - do not pollute the root of your hierarchy or you will find it difficult to navigate through it.

3.2 Scripts

To ensure clarity, scalability, and consistency across the project, all contributors are expected to follow the established C# coding conventions, largely based on Microsoft's official style guide and extended with Unity-specific best practices.

3.2.1 File and Class Structure

Source files should contain only one public type, although multiple internal classes are allowed and they should be given the name of the public class in the file. Class members should be alphabetized, and grouped into sections by regions (organizing fields by their function or inspector layout is preferred over strict alphabetical order):

1. Constant Fields
2. Static Fields
3. Fields
4. Constructors
5. Properties
6. Events / Delegates
7. LifeCycle Methods (Awake, OnEnable, OnDisable, OnDestroy)
8. Public Methods
9. Private Methods
10. Nested types

Within each of these groups order by access:
1. public
2. internal
3. protected
4. private

3.2.2 Script Naming

Scripts are named based on their role and function using PascalCase. No prefixes are used, as the script folder hierarchy (e.g. Scripts/UI/, Scripts/Systems/) provides the necessary context. Each script file should match the class name exactly.

Examples:
- PlayerController
- InventorySystem
- HealthBar

- DialogueTrigger

3.2.3 Namespaces

All scripts must reside inside the ProjectCeros namespace (or deeper logical subdivision). This prevents conflicts with third-party packages.

3.2.4 Documentation & Comments

At the top of each script there should be a summary and a remarks section. Fill in the date, your name and your change.

```
1    /// <summary>
2    /// Place for the summary.
3    /// </summary>
4
5    /// <remarks>
6    /// 09/04/2025 by Your Name: Script Creation.
7    /// </remarks>
8
9    using UnityEngine;
```

Every function should have a comment explaining its purpose. Comments should be used to describe intention, algorithmic overview, and/or logical flow. It would be ideal if from reading the comments alone someone other than the author could understand a function's intended behavior and general operation. All comments should be above the code, begin with an uppercase letter, have a space between the comment delimiter (//) and the comment text and end with a period.

```
// Sample comment above a variable.
    0 references
    private int _myInt = 5;
```

3.2.5 Spacing

Do use a single space after a comma between function arguments.
Example: Console.In.Read(myChar, 0, 1);
  ● Do not use a space after the parenthesis and function arguments.
  ● Do not use spaces between a function name and parenthesis.
  ● Do not use spaces directly inside parentheses or brackets.

3.2.6 Variables

All non-boolean variable names must be clear, unambiguous, and descriptive nouns and use PascalCase unless marked as private which use camelCase. All variable names must not be redundant with their context as all variable references in the class will always have context.

Examples: Consider a Class called PlayerCharacter.
> Bad
> - PlayerScore
> - PlayerKills
> - MyTargetPlayer
> - MyCharacterName
> - CharacterSkills
> - ChosenCharacterSkin

All of these variables are named redundantly. It is implied that the variable is representative of the PlayerCharacter it belongs to because it is PlayerCharacter that is defining these variables.
> Good
> - Score
> - Kills
> - TargetPlayer
> - Name
> - Skills
> - Skin

Variable Access Level

In C#, variables have a concept of access level. Public means any code outside the class can access the variable. Protected means only the class and any child classes can access this variable internally. Private means only this class and no child classes can access this variable. Variables should only be made public if necessary. Prefer to use the attribute [SerializeField] instead of making a variable public.

Local Variables

Local variables should use camelCase. Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment, or when the precise type is not important.

```csharp
var var1 = "This is clearly a string.";
var var2 = 27;
var var3 = Convert.ToInt32(Console.ReadLine());
// Also used in for loops
for (var i = 0; i < bountyHunterFleets.Length; ++i) {};
```

Do not use var when the type is not apparent from the right side of the assignment.

Private Variables

Private variables should have a prefix with an underscore _myVariable and use camelCase. Unless it is known that a variable should only be accessed within the class it is defined and never a child class, do not mark variables as private. Until variables are able to be marked protected, reserve private for when you absolutely know you want to restrict child class usage.

Tooltips

All Serializable variables should have a description in their [Tooltip] fields that explains how changing this value affects the behavior of the script.

Variable Slider And Value Ranges

All Serializable variables should make use of sliders and value ranges if there is ever a value that a variable should not be set to.

Booleans

All booleans should be named in PascalCase but prefixed with a verb.
Example: Use isDead and hasItem, not Dead and Item.

All booleans should be named as descriptive adjectives when possible if representing general information. Try to not use verbs such as isRunning. Verbs tend to lead to complex states.

Do not use booleans to represent complex and/or dependent states. This makes state adding and removing complex and no longer easily readable. Use Scriptable Objects instead.

Arrays

Arrays follow the same naming rules as above, but should be named as a plural noun.
Example: Use Targets, Hats, and EnemyPlayers, not TargetList, HatArray.

Interfaces

Interfaces are led with a capital I then followed with PascalCase.
Example: public interface ICanEat { }

3.2.7 Functions, Events, and Event Dispatchers

The naming of functions, events, and event dispatchers is critically important. Based on the name alone, certain assumptions can be made about functions. For example:

- Is it a pure function?
- Is it fetching state information?
- Is it a handler?
- What is its purpose?

All functions and events perform some form of action, whether it's getting info, calculating data, or causing something to explode. Therefore, all functions should start with verbs. They should be worded in the present tense whenever possible. They should also have some context as to what they are doing.

Good examples:

- Fire - Good example if in a Character / Weapon class, as it has context. Bad if in a Barrel / Grass / any ambiguous class.
- Jump - Good example if in a Character class, otherwise, needs context.
- Explode
- ReceiveMessage
- SortPlayerArray
- GetArmOffset
- IsEnemy

Bad examples:

- Dead - Is Dead? Will deaden?
- Rock
- ProcessData - Ambiguous, these words mean nothing.
- PlayerState - Nouns are ambiguous.
- Color - Verb with no context, or ambiguous noun.

Functions Returning Bool Should Ask Questions
When writing a function that does not change the state of or modify any object and is purely for getting information, state, or computing a yes/no value, it should ask a question.

Good examples:

- IsDead
- IsOnFire
- IsVisible
- HasWeapon
- WasCharging - "Was" is past-tense of "be". Use "was" when referring to 'previous frame' or 'previous state'.

Bad examples:

- Fire - Is on fire? Will it fire? Do fire?
- OnFire - Can be confused with event dispatcher for firing.
- Dead - Is dead? Will deaden?
- Visibility - Is visible? Set visibility? A description of flying conditions?

Event Handlers and Dispatchers Should Start With On

Any function that handles an event or dispatches an event should start with On and continue to follow the verb rule. This also applies to the ScriptableObject GameEvents.

Good examples:

- OnDeath - Common collocation in games
- OnPickup
- OnReceiveMessage
- OnClick
- OnLeave

Bad examples:

- OnData
- OnTarget

3.2.8 Compiling

All scripts should compile with zero warnings and zero errors. You should fix script warnings and errors immediately as they can quickly cascade into very scary unexpected behavior. Do not submit broken scripts to source control.

3.3 Asset Naming Conventions

Consistent asset naming is essential for maintainability, scalability, and team-wide clarity. A strict naming convention ensures assets are easy to search, organize, and parse, reducing friction across disciplines such as design, art, and engineering. Assets follow a **Prefix_BaseAssetName_Variant_Suffix** format, with PascalCase used throughout.

- **Prefix:** A short acronym of the asset type (view table below, e.g., SK_, T_, M_).

- **BaseAssetName:** A recognizable, logical group name for related assets (e.g., Bob for all assets related to the character Bob).

- **Variant:** A descriptive name or number representing variations (e.g., Evil or 01).

- **Suffix:** Additional context (view table below, e.g., texture type, _D for diffuse).

Examples:
- Bob character assets: SK_Bob, M_Bob, T_Bob_Evil_D
- Rock props: SM_Rock_01, SM_Rock_02
- UI Sprites: SPR_UI_Textbox_01, SPR_UI_Icon

Asset Type	Prefix	Suffix	Notes
Levels			
Level (Persistent)		_P	
Level (Audio)		_Audio	
Level (Lighting)		_Lighting	
Level (Geometry)		_Geo	
Level (Gameplay)		_Gameplay	
Models			
Static Mesh	SM_		
Skeletal Mesh	SK_		
Skeleton	SKEL_		
Rig	RIG_		
Materials			

Material	M_		
Material Instance	MI_		
Physical Material	PM_		
Material Shader Graph	MSG_		
Prefabs			
Prefab			
Prefab Instance	I		
Scriptable Object	SO_		
Game Event	GE_		
RuntimeSet	RS_		
Variable	V_		
Artificial Intelligence			
AI / NPC	AI_	_NPC	
Behavior Tree	BT_		
Blackboard	BB_		
Decorator	BTDecorator_		
Service	BTService_		
Task	BTTask_		
Environment Query	EQS_		
EnvQueryContext	EQS_	Context	
Textures			
Texture	T_		
Texture (Base Color)	T_	_BC	Diffuse / Albedo
Texture (Metallic / Smoothness)	T_	_MS	
Texture (Normal)	T_	_N	
Texture (Alpha)	T_	_A	
Texture (Height)	T_	_H	
Texture (Ambient Occlusion)	T_	_AO	
Texture (Emissive)	T_	_E	
Texture (Mask)	T_	_M	

Texture Cube	TC_		
Media Texture	MT_		
Render Target	RT_		
Cube Render Target	RTC_		
Texture Light Profile	TLP_		
2D			
Sprites	SPR_		
Sprite Atlas	SA_		
Font	Font_		
Animations			
Animation Clip	A_		
Animation Controller	AC_		
Avatar Mask	AM_		
Morph Target	MT_		
Audio			
Audio Clip	A_		
Audio Mixer	MIX_		
Dialogue Voice	DV_		
FMOD Sound Event	SFX_		
Audio Class			No prefix/suffix. Should be put in a folder called AudioClasses
Miscellaneous			
Probe (Reflection)	RP_		
Probe (Light)	LP_		
Volume	V_		
Trigger Area		_Trigger	
Visual Effects	VFX_		
Particle System	PS_		
Light	L_		
Camera (Cinemachine)	CM_		Virtual Camera

Universal Render Pipeline Asset	URP_		
Post Process Volume Profile	PP_		

4. Collaboration

Working in a team means coordinating changes, avoiding conflicts, and keeping everyone in sync. This chapter defines best practices for collaborative workflows in Unity and GitHub, so that team members can work efficiently, even when multiple people are touching the same project.

4.1 Asset Ownership & Editing Reservations

Ownership means a specific team member is primarily responsible for a certain asset or feature. They maintain the logic, structure, and implementation. Ownership brings accountability and makes communication easier. Ownership is defined in the notion asset lists of the individual department.

- Ownership does not forbid others from editing.

- It ensures that no major change is made without talking to the responsible person.

In addition to long-term ownership, we track who is actively editing what. A live table in Notion (in the programming tab) allows developers to list assets they are currently working on. This prevents overlap and conflict. Update the table before and after editing shared assets. But remember, if you want to modify someone else's asset, ask the owner first.

Editing Reservations			
Team Member	Asset(s)	Editing Since	Notes
Damian	Scenes/MainMenu, Scripts/UI	15. Apr 2025	Refactoring menu logic

4.2 GitHub Workflow

We use GitHub Desktop for source control and GitHub for remote hosting and pull requests.

4.2.1 Feature Branch Workflow

Each new task, bug fix, or feature starts its life in a dedicated branch. This branch exists solely for that purpose and is isolated from others until it's complete. This keeps the main branch clean and avoids the blocking of each other.

1. Create a branch from main

2. Work in your branch, develop the feature

3. Commit and push frequently

4. Rebase regularly to avoid conflict (at least before opening a pull request)

5. Open a Pull Request to main on GitHub

6. Once approved: Pull the branch and delete it on GitHub

4.2.2 Pull Requests

Pull requests help us ensure code quality and catch issues before they land in main.

Checklist, before creating a pull request:

1. The feature is complete and tested

2. Your branch rebased onto the latest main

3. No unnecessary debug logs, test assets, or WIP leftovers in the branch

4. Your commits and your code follow the style guide

What to write in a pull request:

- A short summary of what the branch contains.

- Optionally, instructions on how to test the feature.

- Assign the lead programmer as the reviewer and yourself as the assignee

4.2.3 Commit Message Style

We use a structured style for commit messages to ensure readability and clarity across the team.

Subject Format: {type}({scope}): {subject}

Subject Rules

- Max 60 characters

- Written in **present tense, imperative voice**

- **No period** at the end

- **First letter is not capitalized**

Allowed Types - {type}

{types}	Description
feat	feature
fix	bug fix
docs	documentation

style	formatting, lint stuff
refactor	code restructure without changing external behaviour
test	adding missing tests
chore	maintenance
init	initial commit
rearrange	files moved, added, deleted, etc.
update	update code (versions, library compatibility)

Scope - {scope}
Where the change was (i.e. the file, the component, the package).

You can add a body if needed (e.g., to explain something tricky or note a testing instruction), but it's not required.

Examples:
- feat(ui): add pause menu with resume and quit

- fix(quest): prevent quest step skipping

- style(core): format GameManager.cs

4.2.4 Branch Naming Conventions

Keep branch names short and descriptive. Use a prefix like feature/, bugfix/, hotfix/, release/, improvement/ or experiment/ and separate words with hyphens for readability.
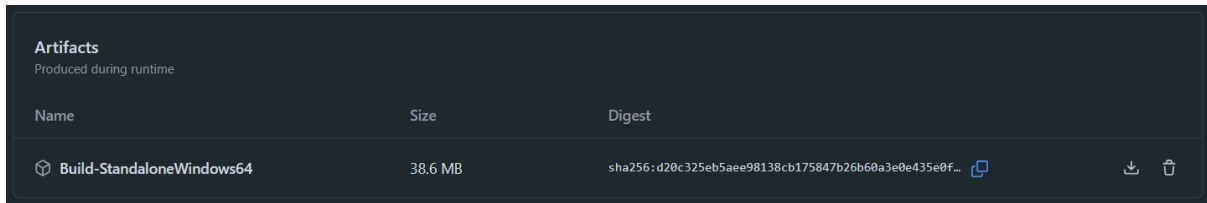
Examples:
- feature/add-user-authentication
- bugfix/fix-header-css
- hotfix/security-vulnerability-patch
- release/v1.3.0
- improvement/refactor-database-layer
- experiment/implement-new-ui

4.3 GitHub Actions (Test Build Automation)

We use GitHub Actions for the creation of automated testing builds.

- When a pull request into main is made, a Windows build is automatically created.

- The build artifact is downloadable from the GitHub Actions tab.

This setup ensures that we always have a working build and can catch errors early. Always review the build results before merging your branch.

5. Importing Assets

5.1 Sound Assets

All audio content in this project is managed through FMOD Studio. It serves as the main hub for importing raw audio files, building modular audio Events, and organizing them into Banks. FMOD integrates tightly with Unity and supports a data-driven approach to audio, aligned with our modular architecture (see Chapter 2.4 Audio System for details). Use parameter ranges from 0 to 1.

FMOD Naming Conventions
Consistent naming is crucial to maintain clarity, searchability, and scalability across the project.

- Use snake_case for all FMOD event names, bank names, and parameter names

- Avoid spaces, special characters, or capital letters

- Prefix looping audio with lp_ (e.g. lp_river_flow, lp_menu_music)

- Choose descriptive and specific names (e.g. sfx_enemy_hit instead of hit1)

5.2 Art Assets

Creating and managing high-quality 2D assets is crucial for maintaining visual consistency, scalability, and performance throughout the project. This section outlines the pipeline and technical standards for creating, exporting, and importing art assets into the Unity engine.

5.2.1 Asset Creation & Export (Procreate Workflow)

All 2D art is created using Procreate, a flexible drawing tool ideal for hand-drawn styles. The visual baseline for the project is 1920x1080 (Full HD) - all assets are designed and exported at actual in-game resolution. This section outlines the correct setup for creating and exporting game-ready sprites from Procreate.

Before starting any asset, set up your canvas properly to ensure compatibility and optimal quality. Choose your canvas size based on the size the asset will appear in-game and round it up to the next power of two (POT).

Examples:

Estimated In-Game Size	POT Canvas Size
100×100px	128×128
300×300px	512x512
500×250px	512x256

This ensures the exported PNG fits nicely into a texture atlas and doesn't waste VRAM. Additionally, set the DPI to 72 (DPI has no effect in-game, but keeping it low helps avoid unnecessarily large files and memory usage) and choose sRGB IEC61966-2.1 as the color profile.

Export:
After finalizing the artwork, export it as PNG (RGBA). If the asset has a lot of empty transparent space (e.g. tall tree with lots of void), consider splitting it into top/middle/bottom or chunks and reassembling it in Unity. This helps with texture packing and prevents wasted VRAM.

If a graphic (e.g. an icon) is used in multiple sizes across the UI, prefer externally downscaled versions of the asset rather than dynamically scaling inside Unity.

Animated sequences (e.g. walk cycles) should be exported as sprite sheets with even spacing and consistent frame sizes. This allows Unity's Sprite Editor to correctly slice and animate them.

5.2.2 Unity Import Settings

To keep the import process clean and consistent, always use predefined Sprite Import Presets for the 2D assets. These presets ensure correct visual quality, performance, and compression settings without having to tweak every sprite manually. You can select them in the upper right corner of the import settings.
Currently we have three main presets depending on the asset type:

- UI: for pixel-perfect UI elements and icons

- Animation: for animations

- Sprite: for everything else

Apply the appropriate preset as soon as you import the sprite into Unity. For animated assets or sprite sheets, make sure to open the Sprite Editor and slice the image after applying the preset. Adjust the Max Size to the closest Power of Two value that fits the actual image size (e.g. use 64 for a 32x32 sprite).

5.2.3 Sprite Atlas & Showcase Scene

All sprite assets should be grouped into Sprite Atlases for optimized performance and draw call reduction. Group sprites based on usage context, such as:

- UI → Buttons, icons, HUD elements

- Environment → Background tiles, props, terrain

- Characters → Player, NPCs, enemies

- Animation Sheets → Full animation sets can be grouped as long as they belong to the same character or object

Here you should also use the presets for the sprite atlas settings:

- AtlasUI: for UI elements

- AtlasSprite: for everything else

Keep the max texture size as low as possible, after you fill the atlas with sprites. Do not exceed 4096×4096 for the atlas texture size. After that you can drag all imported sprites into the Showcase scene. Display the assets next to each other and use the 2D mode. Keep everything clean and organized in both the scene hierarchy and the project folder structure. Use the same folder structure in both.

This scene is not part of the final game, but acts as a visual QA step to verify:

- Transparency handling
- Animation playback
- Visual consistency
- Accurate sizes

Regularly clean up unused or outdated assets to keep the project clean and avoid confusion during production.

6. Appendix

This section lists all external resources used or referenced during the creation of this technical design document. They served as inspiration, guidance, or direct references for best practices in Unity workflows, Git conventions, and general game architecture principles.

- Overview of asset sizes and graphics performance
  https://www.youtube.com/watch?v=MrPoCGHM80E

- Git commit style guide
  https://gist.github.com/ericavonb/3c79e5035567c8ef3267

- 5 types of Git workflows that will help you deliver better code
  https://buddy.works/blog/5-types-of-git-workflows

- A Simplified Convention for Naming Branches and Commits in Git
  https://dev.to/varbsan/a-simplified-convention-for-naming-branches-and-commits-in-git-il4

- Naming conventions for git branches
  https://www.geeksforgeeks.org/how-to-naming-conventions-for-git-branches/

- Unity Style Guide
  https://github.com/justinwasilenko/Unity-Style-Guide

- 10 Unity Style Tips
  https://docs-style-guide.unity.com/getting-started/style-tips/

- Research about the technical design document
  https://dlorenzolaguno17.github.io/TDD/

- Three ways to architect your game with ScriptableObjects
  https://unity.com/how-to/architect-game-code-scriptable-objects

- Game Architecture with Scriptable Objects
  https://www.youtube.com/watch?v=raQ3iHhE_Kk