THE PERFECT PROJECT - MANUAL

PROJECT ARCHITECTURE

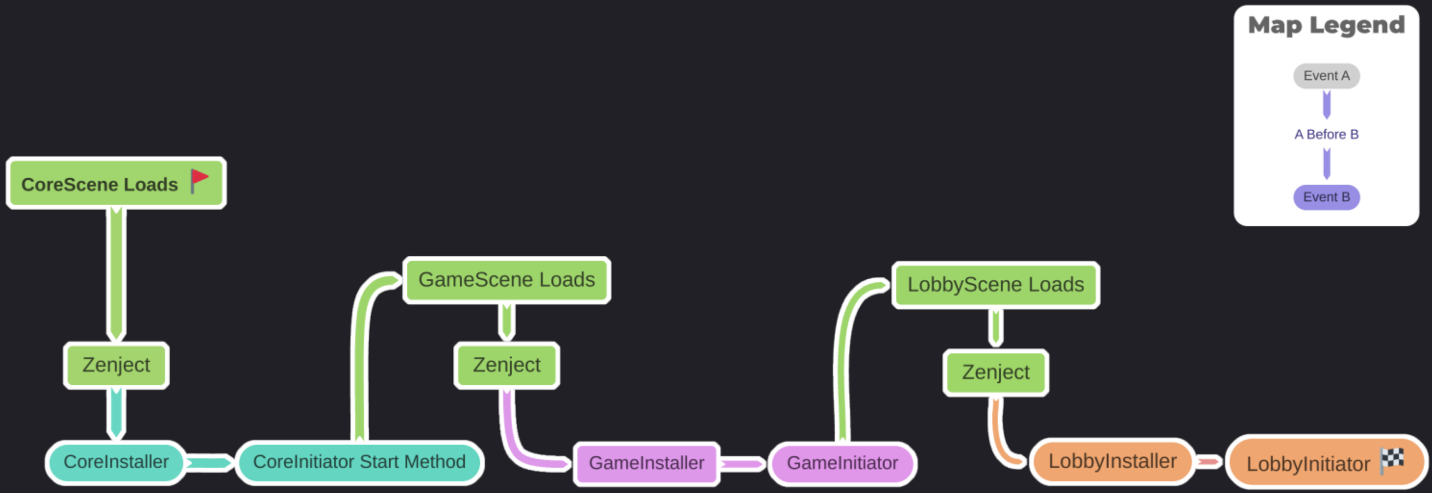
SINGLE ENTRY POINT (SEP)

This project follows the Single Entry Point (SEP) methodology.

1. The first scene to load is always the *CoreScene*, which is almost empty. The starting scene is configured in the *DefaultSceneSelector* script.

2. For the Bindings step, we use *Zenject*, which provides a conveniet Dependency Injection (DI) system. Firstly, Zenject connects our bindings automatically once the scene loads. This is done in the *Installers* scripts. Each scene has its corresponding *Installer*.
   For example: The *CoreScene* has its *CoreInstaller*, and the *LobbyScene* has its *LobbyInstaller*.

3. Next comes the Initialization step. Each scene has its corresponding Initiator which is responsible for initializing the Enter & Exit points.
   For example: The *CoreScene* has its *CoreInitiator.*
   Note: The *CoreInitiator* has the ONLY Start method in our game!



DIVISION OF RESPONSIBILITIES

The scripts in the project can be divided into several categories, each with a distinct responsibility.

1. Installer
    - Binds our instances to their references (as mentioned above).
      We ALWAYS bind to interfaces when possible! But why?
        a. Some classes must be interfaces because we need to bind them to different instances during our game. So, to maintain alignment across the project, we use interfaces even where it's not strictly necessary.
        b. Interfaces clearly reveal the public methods and the purpose of the class.
        c. Can inject Mock instances in Unit Tests.
        d. It's generally a good practice to follow the *Dependency inversion pattern, which guides us* to rely on abstractions, because you never know when you'll need them.

2. Initiator
    - Provides the single Entry & Exit points to a scene.

3. Commands
    - The conductors of our game's features. A Command is the only class allowed to reference other features (Controllers).
    - We don't raise any events in this project, instead we execute Commands.
    - A Command is a simple class with an Execute method, which invokes a desired logic in a specific order, by communicating between multiple scripts.
    - Note: Circular Dependencies may occur when a class executes a Command, and one of the Command's effects is updating that class. To avoid circular dependencies in Commands, each Command resolves its own references from the *DIContainer*.

4. MVC
   A software architecture pattern used in applications and gaming, separating our feature's classes into 3 groups:
    - **M**odel: A simple class that contains only data, such as fields, properties, and small conditional methods. We chose not to include any Events/Actions in our models, for this we have Commands! (see above)
      Note: We don't like to use the word 'model' in gaming because it can be confused with 3D models, so instead we use 'Data'.
    - **V**iew: A class that represents visual elements/components of an object, such as Sprite/ParticleSystem/3D Model/UI. It will ALWAYS inherit from *MonoBehaviour* and sit on a *GameObject.*
      A View can only reference smaller, inner Views. It never invokes anything on its own, it only receives orders or passing callbacks to a Controller.
    - **C**ontroller: The "brain" of a feature in our game. It queries the Model, updates the View accordingly, and invokes Commands for complex flows. The Controller is the only class allowed to reference its Model & View. For example: The *ArrowController* controls the *ArrowView*.

5. Services
    - A Service has no View and doesn't affect other scripts. Its sole responsibility is to provide a common functionality to other scripts.
    - Services can be accessed from Controllers/Commands/Services.
    - If data needs to be accessed from multiple places, it should not be stored in a standalone DataService, which is like a public Model. For example: *LevelDataService*.

6. Factory
    - Encapsulates the logic for creating a specific object. Foe example: *LevelFactory.*

7. Utils
    - A static class containing common, reusable functionality..

8. Extensions
    - A static class that provides extension methods.

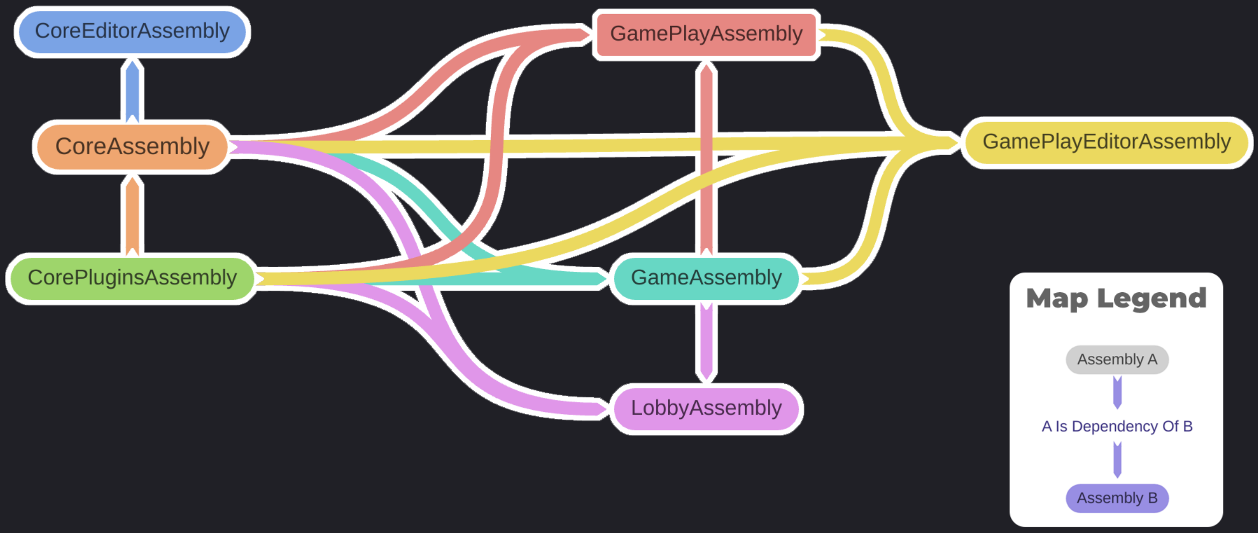
9. GameState
    - Represents a phase of our game, with Enter & Exit methods, which are used in a StateMachine pattern. Our game can only be in a single GameState at any given moment.
    - We have 2 GameStates: *LobbyState*, and *GamePlayState*.

PROJECT STRUCTURE

This project follows the Domains Project Structure for folders organization.

1. Domains in this project:
   - Core: Holds all scripts/assets that are not dedicated to this project and can be used in other games. As mentioned above, the *CoreScene* is the first scene of our game.
   - Game: Holds all scripts/assets that are dedicated to this specific game and used across multiple scenes/states. The *GameScene* is loaded on top of the *CoreScene*.
   - Lobby/GamePlay: Domains that represent a *GameState*. Loaded on top of the *CoreScene*.

2. Assembly Definitions
   - Each domain has its own Assembly Definition, forcing references separation between domains. For example: The Core domain has CoreAssembly.
   - On the way, this reduces our scripts compilation times since the compiler need to recompile only Assemblies that changed.



DESIGN PATTERNS AND BEST PRACTICES

TECHNIQUES

There are many techniques which became a standard in the industry that can be found in this project.

1. UpdateSubscriptionService (UpdateManaget)
    - This project uses the UpdateManager pattern to improve performance and to allow any script to subscribe to Update/FixedUpdate/LateUpdate methods.

2. Cameras Separation
    - We use 2 Cameras: One for rendering the world, and another for rendering the UI. This separation has become a standard in game development to ensure Post-Processing or any other adjustment do not affect the UI.

3. Object Pooling
    - Objects that are created frequently during gameplay are recycled to reduce memory allocation.
    - For example: *ScoreGainedFXPool* recycles *ScoreGainedFXView.*

4. Asset Loading
    - This project use's Unity's Addressable System to load assets dynamically.
    - Note: All assets are currently part of the projects, they but can easily be moved to an external server and loaded remotely.
    - Well-known problem: In games that have multiple live versions concurrently (like mobile games), if we build prefabs with scripts attached and those scripts are modified in newer versions, older versions may fail to load them.
    - Solution in this project: Build bundles multiple times, once from each version. Each live version will load its corresponding bundles and won't have issues.
      Alternative solution: Only build primitive assets (e.g. Images, 3D models, audio files, or prefabs with no scripts attached to them).

5. Unit Tests
    - This project covers only its core logic services with Unit Tests. This can be extended to more classes.
    - As part of the CI/CD pipeline, the *PreBuildUnitTestsValidator* class fails the build if any unit test fails.

6. Animations
    - This project uses the Legacy Animation to perform simple animations, which is significantly more performant — especially for UI objects. Note: If we had a complex animated object (e.g. humanoid character), it would use an Animator component instead.

7. Prefabs
    - Most objects are prefabs to avoid scene conflicts when multiple developers are working in parallel.

8. Lights Baking
    - The environment's lighting is pre-baked to improve runtime performance. At the cost of a ~170 MB increase in build size.

9. Draw Calls Batching
    - The environment uses a single material for all objects, which allows them to be rendered in a single draw call, improving performance.

10. Shaders
    - We use shaders instead of animations when possible, for example, the horizontally scrolling mountains in the lobb, because shader calculations run in parallel on the GPU and are much faster than CPU-based animations.

11. Logs
    - The entire codebase is monitored with logs to make debugging easier.

12. Tweens
    - The DoTween package is used to create simple tween animations via code.
      e.g. spinning the Arrow in the *ArrowMovementController.*

13. Persistent Data Encryption
    - To save data cross games, we use Unity's PlayerPrefs.
    - To make hackers' lives harder, we encrypt the data (See *PlayerPrefsDataPersistence).*

14. Editor Tools
    - The Editor Script *LevelTrackViewEditor* is a great example of how to improve runtime performance by caching objects in the *LevelTrackView*. Ahead of time.

PROJECT GUIDELINES

CODING STANDARDS

1. This project largely follows the Microsoft's Coding Conventions.

ASYNC OPERATIONS

1. This project uses Awaitables for async operations.
2. Because async operations don't catch exceptions by default, each time we start an async operation we must surround it by a try/catch block.
3. We pass *CancellationTokens* to every async operation in the project, which allows us to cancel operations when needed.
4. Each level in our game has its own *CancellationToken* held in the *LevelCancellationTokenService*, that is canceled once the Level is disposed.

CLEAN CODE PRINCIPLES

This project adopts clean code principles—but only those we've found to be truly effective.

1. **Write Self-Documenting Code**
    - Code should be easy to read and tell its story without comments—using meaningful names and reusable functions.
      Comments are used only for warnings where no clear alternative exists.
2. Single Responsibility Principle (SRP)
    - Each class, method, or module should have only one reason to change.
3. **Small, Focused Functions and Classes**
4. **Avoid Deep** if **Nesting**
    - Keep branching logic shallow when possible.
5. **Avoid** switch **Statements**
    - Replace them with proper abstractions.
6. **DRY (Don't Repeat Yourself)**
    - Avoid code duplication, but only when unnecessary.
7. **Immutable** Get **Methods**
    - Get methods should never modify internal state or cause side effects.
8. **No "Magic Numbers"**
    - All constants variables are stored as consts.