Project Documentation

This documentation details the fundamental components, functionalities, and design principles of a modular project developed using the Unity game engine. The project includes key elements such as player progression, resource management, dynamic production mechanics, and an inventory system. All save/load operations are standardized via the ISaveable interface, allowing components to be easily integrated into the centralized SaveLoadManager through drag-and-drop in the Inspector.

Table of Contents

---

1. Overview

The project presents a modular structure developed with the Unity engine. Key aspects such as player progression, resource management, production mechanics, and the inventory system are designed to work independently yet integrate seamlessly. The ISaveable interface is used for save/load operations so that each component's state is stored and restored in JSON format. This allows data to be easily integrated during development by simply dragging and dropping the components into the SaveLoadManager via the Inspector.

2. Code Structure & Components

ISaveable Interface

- **Purpose:**
  Defines the necessary methods for classes that need to save and restore their state in JSON format.
- **Main Methods:**
  - CaptureState(): Returns the current state as a JSON string.
  - RestoreState(string state): Restores the object's state from the provided JSON string.
  - ClearAll(): Resets the object's state.
- **Property:**
  - UniqueIdentifier: Provides a unique identifier for each saveable object.

Note:
All classes intended to be saved should implement the ISaveable interface to create their data. These components can then be added to the SaveLoadManager via drag-and-drop in the Inspector.

LevelManager (Level Management)

- **Purpose:**
  Manages the player's XP and level.
- **Main Functions:**
  - XP Tracking and Leveling Up:
    XP is added using AddXP(float xp); the CheckLevelUp() method facilitates level-ups when enough XP is accumulated.
    The RecalculateXPRequirement() method computes the XP required for the next level.
  - UI Updating:
    Changes in XP and level are reflected on the screen via the UIManager.
- **Save/Load Support:**
  The state is saved and restored using CaptureState(), RestoreState(string state) (now enhanced with try-catch for error management), and ClearAll().

PlayerManager (Player Management)

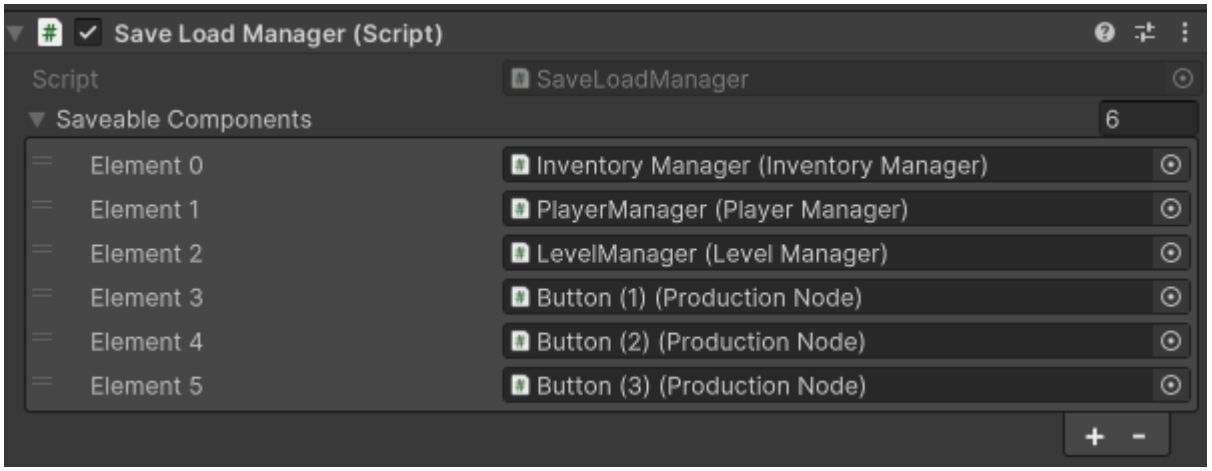
- **Purpose:**
  Maintains and updates the player's name and level, and reflects these changes on the UI.
- **Main Functions:**
  - Player Data:
    The PlayerData class holds the player's name and level.
  - Data Updating:
    Methods such as UpdatePlayerName() and UpdatePlayerLevel(int levelIncrement) modify the player's data.
  - UI Updating:
    Changes are immediately shown on the screen via the UIManager.
- **Save/Load Support:**
  Player data is saved and restored in JSON format, with the RestoreState method now wrapped in try-catch blocks for error handling.

ProductionNode (Production Node)

- **Purpose:**
  Produces items at defined intervals, functioning during runtime as well as in offline mode.
- **Main Functions:**
  - Production Process:
    The UpdateProduction() method handles production at specific time intervals.
    The ComputeOfflineProduction() method calculates the amount produced while the game was closed.
  - Collection Process:
    The CollectProducedItems() method collects produced items and transfers them to the inventory.
- **Save/Load Support:**
  The number of produced items and the last production timestamp are saved in JSON format and restored using try-catch wrapped RestoreState(string state).

SaveLoadManager (Save/Load Management)

- **Purpose:**
  Centrally manages the state of all ISaveable components in the game.
- **Main Functions:**
  - Saving:
    The SaveAll() method iterates through each ISaveable component, capturing its state and writing it to a JSON file.
  - Loading:
    The LoadAll() method reads the JSON file and restores the state for each component.
  - Clearing Saves:
    The ClearSaveFile() method clears the saved data.
- **Integration:**
  - The Data intended for persistence is aggregated within a class and subsequently, using the ISaveable interface, is returned in JSON format via the CaptureState() function.
  - All ISaveable components are added to the SaveLoadManager via drag-and-drop in the Inspector.



Inventory & Inventory Management

Inventory (Inventory Model)

- **Purpose**:
  Stores the player's currency and items.
- **Main Functions:**
  - Currency Management:
    Currency is added with AddCurrency(float amount) and spent with SpendCurrency(float amount).
  - Item Management:
    Items are added with AddItem(ItemSO itemData, int qty) and removed with RemoveItem(ItemSO itemData, int qty).
  - Data Reset:
    The ClearAllData() method resets the inventory.

InventoryManager (Inventory Manager)

- **Purpose:**
  Manages and updates the inventory data while triggering UI updates through event mechanisms.
- **Main Functions:**
  - Transactions:
    Handles the addition and removal of currency and items.
  - Event System:
    Notifies other components of changes via the OnInventoryChanged and OnInventoryReset events.
- **Save/Load Support:**
  The inventory state is stored and restored in JSON format, with the RestoreState method protected by try-catch blocks.

InventoryUI (Inventory UI)

- **Purpose:**
  Provides a visual representation of the inventory on the user interface.
- **Main Functions:**
  - Slot Creation:
    Dynamically creates UI slots for each item in the inventory.
  - UI Updating:
    The RefreshUI() method reflects changes (such as currency and item counts) on the screen.
  - Slot Resetting:
    The ResetSlots() method clears and rebuilds the current slots.

UIManager (User Interface Management)

- **Purpose:**
  Manages and updates general UI elements such as the player name, XP, level, and currency.
- **Main Functions:**
  - Updating Player Name:
    The ApplyNameChange() method updates the player's name using the input field's value.
  - Visual Updates:
    Methods like SetPlayerName(), SetXpText(), SetLevelText(), and SetCurrencyText() update UI components in real time.

---

3. Testing & Mobile Compatibility

- **Testing:**
  Developers have added test keys on the main screen to manipulate values like XP, level, currency, and inventory items. These changes are immediately visible on the screen and are automatically saved when the game is exited, then reloaded upon re-entry.
- **Mobile Compatibility:**
  The canvas is configured to be scalable on all mobile devices. The current UI layout is designed for testing purposes only and can be adjusted for final user experience in a production environment.

4. Design Principles

- **Single Responsibility Principle:**
  Each class focuses on a specific functionality (e.g., level management, player management, inventory management, UI updating), enhancing code readability and maintainability.
- **Open/Closed Principle:**
  New saveable classes can be added without modifying existing code—simply by implementing the ISaveable interface.
- **Liskov Substitution Principle:**
  Any component implementing the ISaveable interface can be used interchangeably in the SaveLoadManager.
- **Interface Segregation Principle:**
  The ISaveable interface includes only the essential methods required for persistence, avoiding unnecessary methods.
- **Dependency Inversion Principle:**
  Necessary components (e.g., UIManager, InventoryManager) are injected via the Inspector, reducing tight coupling and increasing testability.

---

5. Usage & Extensibility

- **Modular Structure:**
  To add a new component for saving, ensure that the class implements the ISaveable interface and then add the component to the SaveLoadManager via drag-and-drop in the Inspector.
- **Production Mechanics:**
  The ProductionNode calculates item production dynamically during both runtime and offline modes. This mechanism can be easily extended for different production speeds or types.
- **Testability:**
  Test keys on the main screen allow dynamic manipulation of values such as XP, level, currency, and inventory items. Changes are saved upon exiting the game and reloaded when re-entering.
- **Mobile Compatibility:**
  The scalable canvas adjusts to various screen sizes. The current UI layout is for testing purposes and can be customized to optimize final user experience.

6. Conclusion

The project presents a robust architecture developed with Unity using modular design principles.

- **ISaveable Interface:**
  Standardizes save/load operations across all components, making the system extensible and easy to maintain.
- **SaveLoadManager:**
  Centrally stores and restores the state of components added via drag-and-drop in the Inspector, saving data in a JSON file.
- **Testing & Mobile Compatibility:**
  Integrated test keys and a scalable canvas enhance flexibility during development and optimize user experience.

This documentation outlines the overall structure of the project, the functionalities of its components, and the applied design principles in detail. If further information or modifications are needed, please let me know!

7. URL:

Test Video:

https://youtu.be/bpMEEzqT4wk

Github:

https://github.com/YAMTAR-98/InventorySaveAndAutoGenerateSystem.git

Portfolio:

https://muhammednafikirman.info/