**README**

**Overview**

This project showcases a modular Unity application structured with clean, extensible, and reusable components. The system incorporates various gameplay mechanics such as audio management, player movement, interaction, task handling, and animations. The architecture promotes separation of concerns and extensibility by utilizing interfaces and Unity’s event-driven design.

**Architecture**

**Key Scripts**

1. **AudioManager**
   * **Purpose**: Controls the sound settings for the application.
   * **Key Methods**:
     + ToggleSound(bool isEnabled): Enables or mutes audio across all AudioSource components.
2. **IInteractable and InteractableObject**
   * **Purpose**: Define and implement an interaction system for game objects.
   * **Key Methods**:
     + Register(IinteractionLogger interactor): Subscribes a logger to interaction events.
     + Remove(IinteractionLogger interactor): Unsubscribes a logger.
     + Interact(): Executes the UnityEvent action associated with the object.
     + OnMouseDown(): Detects interaction through mouse input, notifies loggers, and triggers the action.
3. **IinteractionLogger and InteractionLogger**
   * **Purpose**: Logs interactions with objects and executes associated events.
   * **Key Methods**:
     + UpdateLogger(): Invokes Unity events tied to interaction logs.
     + Register() and DeRegister(): Manage logger registration for interactable objects.
4. **PlayerMovement**
   * **Purpose**: Handles 2D player movement using keyboard input.
   * **Key Methods**:
     + Update(): Calculates movement based on Input.GetAxis and applies transformations to the player.
5. **PopupHandler**
   * **Purpose**: Manages animations for opening and closing UI popups.
   * **Key Methods**:
     + Open() and Close(): Triggers animator states using serialized trigger names.
6. **ITask and TaskHandler**
   * **Purpose**: Tracks task completion and manages win conditions.
   * **Key Methods**:
     + UpdateTaskCount(): Increments completed tasks and updates UI.
     + Check(): Verifies win conditions and transitions to the victory scene.
     + ExecuteWinCondition(): Delays scene transition with a coroutine.
7. **IParticleSystemExecute and ParticleSystemExecute**
   * **Purpose**: Manages the execution of particle systems.
   * **Key Methods**:
     + Execute(): Triggers the playback of associated particle systems.

**Design Choices**

**1. Modular and Extensible Design**

* **Interfaces**: Core functionalities like IInteractable, IinteractionLogger, ITask, and IPopup abstract behavior for easy extension and substitution.
* **UnityEvents**: Decouples the logic from direct scripting, making the system flexible and reusable.
* **Separation of Concerns**: Each script focuses on a single responsibility, enhancing maintainability.

**2. Event-Driven Architecture**

* Interaction events and task updates are driven by UnityEvents, enabling dynamic linking of functionality without tightly coupling components.

**3. Animation Handling**

* Popups use Unity’s Animator component with customizable triggers (\_triggerNameOpen and \_triggerNameClose) for extensible and scalable animations.
* This approach ensures that animations are state-controlled and can easily adapt to new UI designs.

**4. Transitions**

* Smooth scene transitions are handled using coroutines, allowing delays (e.g., in ExecuteWinCondition) for polished gameplay feedback.

**5. Player Input and Movement**

* Player movement is kept simple, relying on Unity’s Input.GetAxis for cross-platform compatibility and scalability to new input systems.

**Extensibility**

1. **Adding New Interactable Objects**
   * Implement IInteractable and define custom behavior in the Interact() method.
   * Attach the InteractableObject script to any GameObject and set the desired UnityEvent in the inspector.
2. **Expanding Loggers**
   * Implement IinteractionLogger to create new types of loggers for various purposes (e.g., analytics or event tracking).
   * Register the new logger with InteractableObject instances.
3. **Customizing Animations**
   * Modify the Animator Controller and set custom triggers for popup animations.
   * Adjust serialized trigger names in the PopupHandler script to match the Animator states.
4. **Task Progression and Win Conditions**
   * Expand the TaskHandler script by overriding Check() to add new win conditions or additional rewards.
5. **Audio Management**
   * Add more advanced audio settings (e.g., volume sliders) by extending AudioManager.
6. **Particle Effects**
   * Associate additional particle effects by extending the ParticleSystemExecute array and updating the logic.