Command System Documentation

Overview

The **CommandSystem** is an extensible framework designed to streamline the implementation of **Command Patterns** in Unity projects. It focuses on providing a modular architecture for managing commands, executing them with conditions, undoing actions, and handling complex sequences with ease.

This system is built with flexibility and scalability in mind, enabling developers to implement a wide variety of gameplay and editor functionalities, such as:

- **Undo/Redo systems**
- **Event-driven mechanics**
- **Sequenced animations**

Key Features

1. Modular Command System

- **Command Abstraction**: Each action (or command) is encapsulated into a modular and reusable unit.
- **Execution and Undo**: Commands support execution and undo logic, making them ideal for creating reversible game actions or editor tools.
- **State Management**: Commands track their lifecycle using the CommandState enum, allowing for precise control and debugging.

2. Sequencing and Execution Modes

- **Command Sequences**: Organize commands into sequences, enabling batch execution and undo of multiple commands in a defined order.
- **Execution Modes**:
    - ExecuteKeep: Commands remain in the sequence after execution.
    - ExecuteDelete: Commands are removed after execution.
    - ExecuteSendUndoList: Executed commands are moved to an undo list.
    - Loop: Sequences can loop a specified number of times or indefinitely.

3. Conditional Execution

- **In-Conditions**: Commands wait for a specific condition to be met before execution begins.
- **Out-Conditions**: Commands ensure exit criteria are satisfied before completing.
- **Custom Conditions**: Easily define custom conditions, such as time delays, user-defined functions, or state checks.

4. Asynchronous Operations

- Fully supports asynchronous execution using Task and CancellationToken to handle:
    - Long-running processes.
    - Waits for specific conditions.
    - Graceful cancellation of ongoing tasks.

5. Undo Management

- Tracks executed commands in an undo list.
- Provides seamless undo functionality for individual commands or entire sequences.
- Supports complex operations, such as undoing nested sequences.

6. Enhanced Editor Integration

- **Custom Inspector for Commands**: Displays command details, including name and state, with color-coded visualizations.
- **CommandManager Editor**:
    - Interactive controls to execute or undo sequences.
    - Expandable UI for viewing and managing commands in each sequence.
    - Real-time feedback on command states.

Architecture

1. Core Components

- **Command**:
    - Encapsulates an individual action.
    - Supports execution, undo, and condition checks.
- **CommandAction**:
    - Defines how commands are executed (method or coroutine).

    - Includes callbacks for start and completion.

- **CommandManager**:
    - Manages collections of sequences.
    - Provides a central system for executing and undoing commands.

2. Sequencing

- **Sequence**:
    - Groups commands into ordered collections.
    - Supports execution and undo of the entire group.
    - Handles looping and advanced execution modes.
- **Execution Modes**:
    - Flexible behavior when running commands (e.g., retaining, removing, or moving commands to an undo list).

3. Execution Logic

- **ExecuteType**:
    - Modular execution logic, defining conditions for starting and ending commands.
    - Supports out-of-the-box conditions like immediate execution or time-based delays.
    - Customizable to fit any gameplay or editor need.
- **ExecuteCondition**:
    - Base class for defining in and out conditions.
    - Extendable for custom logic.

Customization and Extensibility

1. Custom Conditions

- Create your own ExecuteCondition for unique gameplay needs.
- Examples:
    - “Wait until the player reaches a checkpoint.”
    - “Only execute if the player has enough resources.”

2. Flexible Execution Modes

- Easily swap between execution behaviors (e.g., retain commands or automatically move them to undo lists).

3. Editor Integration

- Extend the custom editors to fit specific workflows.
- Enhance debugging by visualizing command states and sequences in the editor.

Why Use CommandSystem?

- **Decoupled Design**: Commands are self-contained, making your code more modular and maintainable.
- **Scalability**: Add new commands, conditions, or sequences without modifying existing logic.
- **Flexibility**: Supports a wide variety of use cases, from gameplay mechanics to editor tools.
- **Ease of Use**: Provides ready-to-use tools for managing commands and sequences with minimal boilerplate.