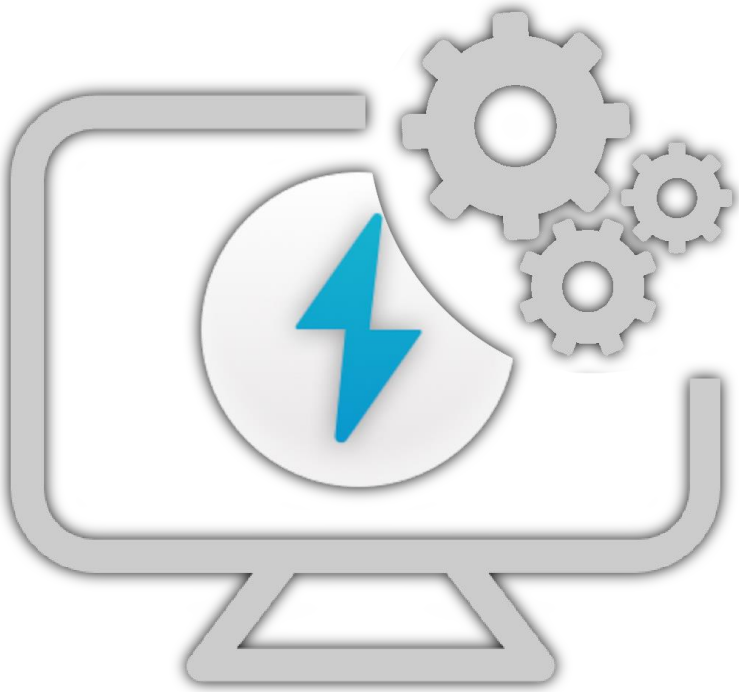

Input System Extension

Documentation

———

Creator: Lucas Gomes Cecchini

Pseudonym: AGAMENOM

Overview

This document provides guidance on using the **Input System Extension** for Unity, a powerful asset designed to streamline keyboard input configuration and management in your game.

With this extension, you can:

- **Easily customize and switch keyboard settings** directly from an in-game menu or editor interface.
- Utilize two types of **Scriptable Objects**:
    - One dedicated to **saving player preferences** persistently using PlayerPrefs.
    - Another designed for **programmatic access and manipulation** of keyboard configurations during gameplay.
- Benefit from **automatic saving**, ensuring all changes to input settings are instantly preserved without manual intervention.

The Input System Extension enables flexible, runtime keyboard remapping and management, providing a seamless player experience and simplifying input handling for developers.

Instructions

You can get more information from the Playlist on YouTube:

https://www.youtube.com/playlist?list=PL5hnfx09yM4Kqkhx0KHyUW0kWviPMTPCs

Script Explanations

GetAction Attribute

Overview
This script provides a custom way to select an input action in Unity's Inspector panel when using the new Input System. It defines a special attribute you can apply to fields, which makes those fields show a dropdown list of input actions that come from a pre-configured input asset. This helps you pick input actions more easily and reliably in the editor.

Key Components and Their Roles
1. The Custom Attribute

- **What it is:**
  A simple marker you add above a field in your script to tell Unity: "Show me a dropdown to pick an input action here."
- **How it works:**
  It doesn't contain logic itself. Instead, it signals the editor to draw that field in a special way.

2. The Drawer for the Attribute

- **What it is:**
  A special piece of code that tells the Unity Editor how to draw the dropdown list for any field marked with that attribute.
- **Important Variables:**
  - **extensionData:**
    This holds a reference to an object that contains the main input actions asset you want to pull actions from. This object stores the input configuration you made for your game.
- **Main Method:**
  - This method controls the drawing of the dropdown in the editor.
  - It receives information about where to draw the field, what property (variable) it is drawing, and the label to show.

How the Drawer Method Works Step-by-Step

1. **Validation:**
   It first checks if the field it's drawing is of the right kind (it must be a reference to an input action). If not, it shows an error message.
2. **Loading the Input Data:**
   If it hasn't already, it loads the ScriptableObject (a Unity asset that stores data) containing the default input actions. If this asset or the input actions inside it are missing, it shows an error.
3. **Gathering Actions:**
   It collects all the valid input actions from all the action maps (groups of input actions) inside the input actions asset.
4. **Handling No Actions:**
   If there are no input actions found, it shows a warning message instead of the dropdown.
5. **Finding Current Selection:**
   It checks what input action is currently assigned to the property being drawn. It tries to find this action's position in the list of all actions, so it can highlight the current choice in the dropdown.

6. **Creating Dropdown Options:**
   It creates a list of strings that combine the action map name and the action name for each action, so the user sees a clear path like "Player/Jump" or "UI/Submit".
7. **Drawing the Dropdown:**
   It draws a popup menu (dropdown) in the inspector with all the action names. The currently selected action is highlighted.
8. **Updating the Selected Action:**
   If the user picks a different action from the dropdown, the property is updated with the newly selected input action reference.

Usage Summary

- You add the **custom attribute** (like a tag) [GetAction] to any field in your script that holds an input action reference.
- When you look at that script in the Unity Inspector, instead of a simple field, you get a dropdown listing all available input actions from your main input asset.
- You pick the desired action from the dropdown, and your field automatically updates to reference that action.
- This avoids manual dragging and helps prevent mistakes by ensuring only valid input actions are assigned.

Why Use This Script?

- It simplifies working with input actions in the Unity Editor.
- It enforces consistency by making sure only input actions from your main input setup can be assigned.
- It improves workflow by showing a clear, easy-to-navigate list instead of manual object references or text strings.
- It helps catch errors early with validation and helpful messages directly in the inspector.

BindingId Attribute

Overview

This script allows you to link a string field in a Unity component to a list of input bindings that belong to a specific input action. It does this by creating a custom attribute you add to that string field, and a special editor interface that shows a dropdown menu with all the available bindings from the referenced input action. This makes selecting input bindings easier and less error-prone inside the Unity Editor.

Key Components and Their Roles

1. The Custom Attribute

- **Purpose:**
  This is a marker you put on a string field in your script to tell the editor, "This string represents a binding ID that comes from a specific input action."
- **Main Variable:**
  - **actionFieldName:**
    The name of another field in the same script that holds a reference to the input action this binding belongs to.
- **How it works:**
  When you apply this attribute, you tell it which input action's bindings should be shown as options for this string field.

2. The Custom Editor Drawer

- **Purpose:**
  This is the code that runs inside the Unity Editor and replaces the default text input for that string with a dropdown menu listing the input bindings from the referenced input action.
- **Main Method:**

This method draws the dropdown menu in the inspector, handles what is displayed, and updates the string field when a new binding is selected.

Step-by-Step Behavior of the Editor Drawer

1. **Check the Attribute:**
   The drawer confirms it is working on a field with the correct attribute. If not, it simply falls back to the normal field display.
2. **Find the Input Action Field:**
   It tries to find the other field in the same component that contains the input action reference, based on the name given in the attribute.
3. **Validate the Input Action Reference:**
   If the input action reference or the action itself is missing, it shows a warning or error message to help the developer understand what's wrong.
4. **Get the List of Bindings:**
   It fetches all the input bindings that belong to the input action.
5. **Prepare Dropdown Options:**
   For each binding, it creates a user-friendly label that includes:
   - The display string of the binding (e.g., which key or button it represents).
   - The binding's name if it's part of a composite (a group of bindings).
   - Any control schemes (groups of devices or setups) associated with that binding.
6. **Find the Currently Selected Binding:**
   It looks for which binding is currently assigned to the string, so it can highlight it in the dropdown.
7. **Show the Dropdown:**
   It draws the dropdown with all the prepared labels, allowing the user to select a binding easily.
8. **Update the Binding ID:**
   When the user chooses a different binding, it updates the string field with the corresponding binding's unique identifier.

Variables Summary

- **actionFieldName:**
  The name of the field that holds the input action reference you want to link bindings from.
- **property:**
  The current string field that holds the binding ID.
- **actionProperty:**
  The linked field containing the input action reference.
- **action:**
  The actual input action object containing the list of bindings.
- **bindings:**
  The list of all input bindings attached to the input action.
- **options:**
  The array of display names for each binding shown in the dropdown.
- **optionValues:**
  The array of the actual binding IDs corresponding to each dropdown option.
- **currentBindingId:**
  The current string value of the binding ID field.
- **selectedIndex:**
  The index of the currently selected binding in the list.

How You Use This Script

1. In your Unity component (script), you have:
   - A field that holds a reference to an input action.
   - A separate string field to hold a binding ID.
2. You mark the string field with this custom attribute, giving it the name of the input action field.
3. When you look at this component in the Unity Editor, instead of manually typing a binding ID, you get a dropdown list with all the binding options from the linked input action.
4. You pick the binding you want, and the script automatically stores its unique ID in the string field.

Why This Is Useful

- It helps avoid errors by only allowing valid bindings from the correct input action.
- It makes the inspector cleaner and easier to use by turning a string field into a dropdown menu.
- It shows detailed, human-readable names for each binding, including device and control scheme info.
- It supports composite bindings and displays them clearly.

- It ensures consistent linking between input actions and their bindings in your game's code.

Input System Extension Prefab Creator

Overview

This script is an editor utility designed to make it easy to add certain predefined user interface (UI) elements related to input management into a Unity scene. It adds options to Unity's **GameObject menu** so you can quickly insert special UI prefabs (ready-made UI objects) like control managers or reset buttons.

It automatically handles creating a canvas (the UI container) if one doesn't exist, ensures the new UI element is parented correctly, and sets things up to be ready for immediate editing.

Main Components

1. Utility for Creating a UI Canvas

- **Purpose:**
  This part makes sure there is a Canvas in the scene where UI elements can be placed. A Canvas is the basic container for all UI in Unity.
- **What it does:**
  - Creates a new Canvas object.
  - Adds necessary components for UI interaction, such as scaling and raycasting.
  - Sets the Canvas to overlay the screen.
  - Creates an EventSystem object, which is required to handle user inputs like clicks and touches.
  - Registers these creations with Unity's undo system so you can revert them if needed.
- **Returns:**
  The newly created Canvas object, so other methods can attach UI elements to it.

2. Method to Instantiate and Set Up Prefabs

- **Purpose:**
  This method places a specific UI prefab into the scene, making sure it's properly connected and parented.
- **Inputs:**
  - The name of the prefab to load.
  - The currently selected GameObject in the hierarchy (where the user might want to attach the new object).
  - A flag indicating if the prefab is a UI element (to decide whether it needs to be parented under a Canvas).
- **How it works:**
  - If the prefab is a UI element, it tries to find an existing Canvas or creates one if none is found.
  - Finds the prefab asset in the project by its name.
  - Determines where to put the new instance:
    - If a GameObject is selected, it uses that as the parent.
    - Otherwise, it parents to the Canvas if it's UI.
  - Instantiates the prefab under the chosen parent.
  - Calls another method to finalize the setup.

3. Method to Find a Prefab by Name

- **Purpose:**
  Searches through the Unity project assets to locate a prefab with a specific name.
- **How it works:**
  - Uses Unity's asset database to find all prefabs matching the name.
  - Returns the first prefab that exactly matches the given name.
  - If no prefab is found, logs an error message.

4. Method to Finalize Prefab Setup

- **Purpose:**
  Prepares the newly created prefab instance for editing.
- **What it does:**

- Registers the creation with Unity's undo system.
- Unpacks the prefab so it can be edited freely in the scene (instead of being locked as a prefab instance).
- Selects the new object in the editor.
- Opens a rename input so you can immediately rename the new object (like pressing F2 in a file explorer).

5. Menu Options for Creating Specific Prefabs

- **Purpose:**
  These are menu commands added to Unity's GameObject menu for easy access.
- **What they do:**
  - Each menu command corresponds to creating a specific UI prefab related to input management (like "Rebind Control Manager" or "Reset All" button).
  - When you select one, it calls the prefab creation method to add that UI element to your scene.

Variables Summary

- **canvasGO:**
  The new Canvas object created for UI elements.
- **canvas:**
  The Canvas component of the canvasGO object.
- **eventSystemGO:**
  The EventSystem object created to handle input events.
- **prefab:**
  The prefab asset found in the project by name.
- **selectedGameObject:**
  The GameObject currently selected in the editor, used to determine where to place the new prefab.
- **parent:**
  The transform (position in hierarchy) under which the new prefab will be placed.
- **instance:**
  The newly created instance of the prefab in the scene.

How to Use This Script

1. In the Unity Editor, open the **GameObject** menu.
2. Look for the **UI > Input System Extension** submenu.
3. Choose one of the options like "Rebind Control Manager (Legacy)" or "Button (Reset All) [TMP]".
4. The script will automatically:
   - Create a Canvas and EventSystem if needed.
   - Instantiate the chosen prefab.
   - Parent it correctly to the Canvas or selected object.
   - Unpack it so you can edit it freely.
   - Select the new UI element and allow renaming it immediately.

This makes adding input-related UI elements to your scene faster and easier without manually searching for prefabs or setting up canvases.

Input System Extension Data

Overview

This script defines a **configuration asset** for managing input settings and visual elements related to the input system in a Unity project. It is designed to store input binding data, keyboard key images, and gamepad button icons in a centralized place, making it easier to manage and display input controls in the game's user interface.

Main Components

1. Configuration Data Storage

- This asset stores several key things related to input:

- **Default Input Actions:**
  A reference to the main set of input actions used in the game. This is the input configuration that can be saved or loaded with custom bindings.
- **Persistence Key:**
  A unique identifier used for saving and loading player-customized input bindings from the player's preferences storage, typically saved as JSON.
- **Default Keyboard Sprite:**
  An image used as a fallback when no specific keyboard key image is found.
- **List of Keyboard Key Images:**
  A collection where each keyboard key (like a letter or number key) is paired with a specific sprite (an image) that visually represents that key in the UI.
- **Gamepad Icons:**
  Separate groups of images representing buttons and controls for two popular gamepad styles: Xbox and PS4 (DualShock). These images are used to replace text labels with icons in UI elements.

2. Keyboard Key Sprite Mapping

- Each entry associates a particular keyboard key with a sprite image.
- This lets the UI show a small picture of the key instead of just text, which improves clarity and aesthetics.

3. Gamepad Icons Group

- This is a collection of sprites grouped by common gamepad controls such as face buttons (A, B, X, Y or Cross, Circle, Square, Triangle), triggers, bumpers, D-Pad directions, sticks, and menu buttons.
- **Key features:**
  - The icons are separated for two controller styles: Xbox and PS4.
  - There is a method to get the correct sprite based on a string describing the control path, like "buttonSouth" or "dpad/up".
  - If an unknown control path is requested, the system logs a warning to alert the developer and returns no image.

4. Editor Integration

- In the Unity Editor (the development environment), a special menu option is added under **Window > Input System Extension**.
- Selecting this menu item opens the configuration asset in the Unity Inspector window, allowing easy editing of all settings and icon assignments.
- If the asset is not found or cannot be loaded, it logs an error to help diagnose the issue.

Variables Summary

- **defaultInputAction:**
  Holds the main input action set to manage bindings.
- **playerPrefsKey:**
  The unique key used for saving/loading binding overrides.
- **defaultSprite:**
  Default fallback image for keyboard inputs.
- **KeyCodes:**
  List pairing keyboard keys with their representative sprites.
- **xbox:**
  Group of sprites representing Xbox controller buttons.
- **ps4:**
  Group of sprites representing PS4 controller buttons.

How It Works in Practice

- You create or reference this configuration asset in your project.
- The system uses it to:
  - Load and save custom input bindings to/from player preferences, ensuring player choices persist.

    - Display appropriate key or button icons in your UI instead of plain text, making controls visually intuitive.
- When a UI element needs to show which button or key is assigned to an action, it asks this configuration for the correct image to show.
- The editor menu option lets you open and modify this asset easily, adjusting bindings or icon sets without hunting through project folders.

Why This Is Useful

- Centralizes input-related visual and data configuration in one asset.
- Improves UI by showing icons instead of text for controls.
- Supports different controller types seamlessly.
- Simplifies binding persistence for player-customizable controls.
- Provides easy editor access to tweak settings without manual asset searching.

Input System Extension Data Inspector

🌟 **Purpose**

This script customizes the visual editor (Inspector) of a configuration asset used in a Unity project. Specifically, it is for configuring input bindings and their visual representations (icons). Its main goal is to:

- Add a button in the editor called **"Get Default Icons"**.
- When clicked, this button will automatically assign appropriate icon images (sprites) to:
    - Xbox controller buttons
    - PS4 controller buttons
    - A large list of keyboard keys

This setup enhances the visual clarity of your user interface by showing button images rather than plain text.

📦 **Script Sections & Key Concepts**

🔷 **Inspector Setup**

- The script is linked to a special asset used to store input-related data.
- It tells Unity to **override the default display** for this asset in the editor.
- It also allows **multiple assets to be edited at once** (multi-object editing support).

🔷 **Main Interface Logic**

▶ **Display Method**

- When the asset is viewed in Unity's editor:
    - The script adds a **large clickable button** labeled "Get Default Icons".
    - When clicked:
        - For every selected configuration asset:
            - It calls a function to automatically fill in the icon mappings.
            - It tells Unity to **mark the asset as modified**, so changes are saved.
- Below the button, the regular layout for public fields is still shown.

🧠 **Icon Auto-Assignment Logic**

▶ **Icon Population Method**

When the user clicks the button, this method runs:

🎮 **Gamepad Icon Assignment**

- The method sets up the icon collections for two gamepads:
    - **Xbox-style controllers**
    - **PS4-style controllers**
- For each button (like A, B, X, Start, L1, etc.), it tries to find an image in your project with a specific name like "XboxOne_A" or "PS4_Triangle".
- If the image exists, it gets used.
- If not found, the script falls back to using a **default image** defined in the asset.

This ensures that even if some icons are missing, your UI will still show something useful.

⌨️ Keyboard Key Icon Assignment

- The method clears any previous keyboard mappings.
- It then builds a **very large list** of key-to-image mappings — for nearly every key on a typical keyboard:
    - Numbers, letters, function keys (F1–F15)
    - Symbols like “+”, “-”, “@”
    - Navigation keys like “Home”, “Insert”, “Page Down”
    - Control keys like “Alt”, “Ctrl”, “Shift”, “Caps Lock”, etc.
- Each key tries to match with a sprite image by name (e.g., “F2” tries to find a sprite named “F2”).
- If a specific image can’t be found, the fallback image is used.
- The completed list is stored back in the asset for later use in UI generation.

🧩 Sprite Lookup Method

This is the method that tries to **find a sprite by its name**:

1. It searches the entire Unity project for image assets that are used as sprites.
2. For each sprite found:
    - It compares the name of the sprite to the one it’s looking for.
    - If the name matches, it uses that sprite.
    - If no match is found after searching everything, it falls back to a default.

A message is printed in the console when a matching sprite is successfully found, which helps during development.

⚙️ How to Use This Script

1. Create or select your **input configuration asset** (the one that stores default actions and icons).
2. Open it in Unity’s Inspector panel.
3. You will see a **button labeled “Get Default Icons.”**
4. Click the button:
    - Gamepad and keyboard icon slots will automatically be filled in.
    - The asset will be marked as changed and ready to be saved.

This drastically speeds up the process of mapping hundreds of buttons to their visual representations — especially useful when building input visualizations or remapping screens.

✅ Summary

- **Enhances** Unity’s inspector interface for input config assets.
- **Adds a button** that fills in keyboard and gamepad icon data in one click.
- **Automatically searches** your project for matching icon images based on name.
- **Supports multiple asset selections** for batch editing.
- **Fallback system** ensures no icons are left blank if a specific one is missing.

Input System Extension Data Auto Creator

🧭 Overview

This script ensures that a **special configuration asset** required for your input system (called Input System Extension Data) is always available in the Unity project. It achieves this in two ways:

1. **Automatically** creates the asset when Unity’s editor starts (if it's missing).
2. Allows the developer to **manually create it** using a menu option in the editor.

The asset is placed in the **Resources folder**, so it can be loaded at runtime using Unity’s built-in resource loading system.

🔷 Primary Responsibilities

✅ **Ensure that a configuration file exists.**
✅ **Automatically create it on startup if it’s missing.**
✅ **Provide a menu shortcut to manually create or replace it.**

📦 Components of the Script

1. Manual Asset Creation via Menu

This section allows developers to manually create the configuration asset via Unity’s **"Assets" menu**.

🔶 **Menu Path:**
- The menu option will appear under:
  Assets > Create > Input System Extension > Input System Extension Data

🔶 **Key Variables:**
- **path**: This stores the target folder path where the asset will be saved (Assets/Resources).
- **assetPath**: This defines the full file path of the asset to be created.

🔶 **Logic Breakdown:**
1. **Folder Check**:
   - It first checks whether the Resources folder exists inside Assets.
   - If not, it creates that folder.
2. **File Check**:
   - If the asset already exists at the target path, it opens a dialog asking whether to replace the file.
3. **Asset Creation**:
   - If no file exists (or if the user agrees to replace it), a new instance of the configuration asset is created.
   - The asset is saved to the Resources folder.
   - Unity is informed that the asset has been modified, so it will be saved properly.
4. **Selection**:
   - Once created, Unity focuses the Project window and automatically selects the newly created asset to make it visible to the user.

2. Automatic Asset Creation on Editor Startup

This part ensures that the configuration asset is always present whenever Unity is opened.

🔶 **What triggers it:**
- Unity automatically runs a specific block of code right after the editor finishes loading.

🔶 **How it works:**
1. **Define the target path** where the asset should exist (Assets/Resources/Input System Extension Data.asset).
2. **Check if the file exists** at that location using the file system.
3. **If the file is missing**, it calls the same creation logic used by the menu command — ensuring the file is created immediately.

💡 **How to Use**

Scenario A: Asset is missing
- The next time Unity opens, the system detects the missing file and **automatically creates it**.
- The new configuration asset is added to Assets/Resources.

Scenario B: Developer wants to reset or manually create it
- The developer goes to:
- Assets > Create > Input System Extension > Input System Extension Data
- Unity asks if the existing asset should be replaced (if it exists).
- If confirmed, the old asset is replaced with a fresh one.

📄 **Why "Resources" Folder?**

Unity's **Resources folder** is special:
- It allows loading assets at runtime via code using their file name.
- This makes it ideal for global configuration files like this one, especially in input systems or UI frameworks.

🔄 **Summary of Features**

Feature	Description
🏁 Startup Initialization	Automatically creates the config file if it's missing on Unity editor launch.
📁 Folder Check & Creation	Makes sure the Resources folder exists before saving the asset.
📝 Menu Option	Lets the developer manually generate or replace the asset.
❓ Dialog Safety Check	Prevents accidental overwrites by asking before replacing.

🎯 Auto-Selection	Highlights the newly created asset in Unity for visibility.

✅ **Final Result**
By using this script:
- Your project will **never be without the required configuration file**.
- The setup process becomes **automated and reliable**, minimizing manual mistakes.
- Your development workflow gains a **safe and repeatable way** to regenerate the input data asset.

On Input System Event

🧭 **Purpose**
This script acts as a **generic and reusable input dispatcher** for Unity's Input System. It allows developers to bind callbacks for **pressed**, **held**, and **released** events from any InputAction, no matter the data type (like float, vector2, or boolean). It's built for flexibility, minimal coupling, and runtime control.

🏗️ **Core Concept**
- Each input action is tracked individually.
- You can **register callbacks** for when an input:
  - Is just pressed (activation starts)
  - Is being held (held down continuously)
  - Is released (input ends)
- The dispatcher tracks these states **frame by frame** and invokes the appropriate logic.

🔐 **Type Parameter**
The script is generic and works with **any input value type**, such as:
- A 2D vector for a joystick
- A float for a trigger axis
- A boolean for buttons

The type is defined at the top using a placeholder symbol to make the dispatcher reusable across different value types.

📦 **Data Structure: states**
- This is a dictionary that links each input action to its **internal tracking object**.
- Each key is a Unity InputAction.
- Each value is a State object, which holds:
  - The last value read from the input
  - Whether the action was held during the previous frame
  - All the registered callbacks (pressed, held, released)
  - A condition that controls whether the action should currently be active

🔁 **Internal State Object**
Each action has its own internal structure that tracks:
- The **input action** itself (what control it belongs to).
- The **last known input value**.
- Whether it was **held** during the previous update cycle.
- The **callback functions** for each input state (pressed, hold, release).
- A boolean check (function) to determine if this action is currently enabled.

Update Method
- Called every frame **after the Input System updates**.
- Reads the current value of the input.
- Compares it with the previous state to determine if:
  - It was just pressed
  - It is still being held
  - It has just been released
- Calls the appropriate callback depending on what transition occurred.

🔍 **How It Knows If Input is "Active"**

The script includes a utility method to determine if the input value is **non-zero**, which is what it considers as "pressed/held." The logic depends on the type:

- A float must be greater than a small threshold.
- A Vector2 or Vector3 must have a noticeable magnitude.
- A quaternion must not be the identity rotation.
- A boolean is simply checked as true/false.
- Any unknown type is compared to its default value.

🔧 Registration Methods

The following functions allow developers to attach their logic to specific events:

Method Name	When It's Triggered
RegisterHold	Called every frame while the input is held
RegisterPressed	Called once when the input first becomes active
RegisterReleased	Called once when the input becomes inactive

There are also matching **unregister** functions to remove listeners from any of the above.

Clear Method

- Fully removes all listeners and state tracking for an input.
- Useful for memory cleanup or temporary input suspension.

🌐 Runtime Configuration with WithAction

This function returns a **configuration object** that allows for **fluent chaining** of callback bindings.
For example, a developer might call:

```
WithAction(myAction).OnPressed(...).OnHold(...).OnReleased(...)
```

This:

- Enables the input action
- Sets a custom condition to enable/disable updates dynamically (if provided)
- Lets you bind all events in one place using chained calls

🔄 Internal Attach to Unity Lifecycle

When a new action is added to the system, the script **automatically hooks** into Unity's input system update loop. Specifically:

- After each input system update, it triggers the update of that specific action.
- This ensures that the input values are always read **after Unity updates them** and **before your gameplay logic runs**.

📘 Configuration Wrapper: OnInputSystemEventConfig

This is the fluent interface returned when using WithAction.
It stores the input action and provides the following methods:

- OnPressed: Binds a function to the press event.
- OnHold: Binds a function to the hold event.
- OnReleased: Binds a function to the release event.
- UnbindAll: Removes all bindings for that action.

This makes input registration **clean, centralized, and readable**.

🧪 Example Use Case (in concept, not code)

Imagine a player's jump input:

1. When the button is first pressed — trigger the jump animation (OnPressed).
2. While the button is held — apply upward force (OnHold).
3. When the button is released — stop the force (OnReleased).

You bind all of this to a single input action, without needing to write a Monobehaviour to manually track state.

🛠️ Summary Table

Feature	Description

Generic Input Type Support	Works with any struct value like floats, vectors, or bools
Automatic Update Hook	Evaluates input state changes every frame after Unity's input system runs
Press / Hold / Release Tracking	Differentiates transitions and triggers the right callback
Fluent API for Clean Setup	Use WithAction for easy chaining of input logic
Runtime Enable/Disable Support	Optional check to enable/disable input dynamically
Safe Unregistration	Can remove any callback cleanly at runtime
Self-contained State	No need for external tracking or timers

Input System Extension Helper

Purpose of the Script

This script provides **centralized and global access** to a data file named Input System Extension Data, which is a **ScriptableObject** stored inside Unity's **Resources** folder.
It ensures that any part of your project can **quickly retrieve the configuration asset** required for your custom input system extensions.

What It Depends On

- The data file (Input System Extension Data) must be placed in the **Assets/Resources** folder.
- This file is a **ScriptableObject** that contains configuration for things like gamepad icons, keyboard mappings, or custom input rules.

Variable Overview

extensionData (private static)

- This is a **cached reference**.
- The script stores the data here after the first time it is loaded, so the system **does not repeatedly search the file system**.
- It is **shared globally** because it is marked as static, meaning all calls to the script will use the same instance.

Method Breakdown

Method: GetInputSystemExtensionData

This is the only public method in the script. It works as follows:

1. **Checks if the data has already been loaded:**
   - If the internal cache variable already contains the data, it immediately returns it.
2. **If not cached, attempts to load it:**
   - It uses Unity's Resources.Load system to locate a file called Input System Extension Data in the **Resources folder**.
   - If found, it stores the reference in the internal variable and returns it.
3. **If loading fails:**
   - Logs an error message to the Unity console.
   - This helps the developer realize the file might be missing or misnamed.
   - It then returns nothing (null), so the caller knows the asset was not available.

How the Script Is Used in a Project

This helper is a **utility script** and is typically used by other systems that rely on the input extension data. For example:

- A custom input handler might need the list of icons or key mappings stored in the asset.
- Instead of manually referencing or searching for that asset, the system simply calls this method.

Usage Flow Example (in plain logic):

1. A gameplay system wants to draw an icon for the "jump" key.
2. It calls: **"Get the Input System Extension Data."**
3. The helper:
   - Checks if it already has the data → returns it.
   - If not → loads it from disk → returns it.
4. The gameplay system uses the returned asset to get the sprite or key mapping.

✅ Advantages of This Design

Feature	Description
Central Access Point	All other systems can access the config from one place.
Memory Efficient	Uses a cached variable to avoid loading the asset multiple times.
Safe Loading	Gracefully handles failure by logging an error message.
Editor-Friendly	Assumes standard Unity workflow with Resources folder usage.
Reusable	Static method allows it to be called from anywhere without instantiation.

🧪 Example Use Case in Concept

Imagine a UI system that shows keyboard and gamepad icons:

- It doesn't need to know where the icon data lives.
- It just calls this helper and asks: “Give me the icon for the ‘Jump’ button.”
- The helper gives it access to the preloaded Input System Extension Data, which has all the mappings ready.

🛠️ Summary Table

Component	Role
extensionData	Stores the loaded asset to avoid redundant loading.
GetInputSystemExtensionData()	Loads and returns the ScriptableObject from Resources.
Resources.Load	Unity’s built-in method to find assets stored in the Resources folder.
Error logging	Helps identify if the asset is missing or misnamed.

Input Binding Saver

🧭 Purpose of the Script

This script allows your application or game to **save and restore input key bindings**, so users can remap controls and have them remembered between sessions. It works by using Unity’s built-in **PlayerPrefs system**, which acts like a persistent storage area for small pieces of data, such as strings. The script is part of a larger system that uses an InputSystemExtensionData configuration asset to manage input settings and override behaviors.

📦 Variables and Their Roles

extensionData (retrieved at runtime)

This is not declared inside the script but is accessed using a helper method. It contains:

- The input configuration data.
- A reference to the InputActionAsset, which contains all input bindings.
- A unique **storage key** used for PlayerPrefs (like a name tag for saving and loading the correct data).

🔧 Method-by-Method Explanation

SaveDefaultBindings

- Retrieves the current extension data from the helper system.
- Verifies that the data exists. If it doesn’t, it logs an error and stops.
- If the data is valid, it calls a method to save the bindings of the default input asset using the configured storage key.

🔁 Usage:

This is called manually (for example, from a pause menu or settings screen) when you want to save the user’s remapped keys.

LoadDefaultBindings

- This is a method that is automatically executed when the game or application starts (it is marked for auto-loading).
- Just like the save method, it retrieves the extension data and checks if it exists.

- If the data is valid, it loads the previously saved bindings using the given key.

🧠 **Usage:**
This method ensures that user-defined keybindings are **restored automatically** on every app start without requiring user input.

SaveBindings
- Accepts two parameters:
  - An input binding asset (which defines the controls).
  - A key string used to identify where in PlayerPrefs to save the data.
- First, checks if the asset is valid. If not, it logs a warning and exits.
- Converts the asset's binding overrides into a JSON string.
- Stores the JSON string in PlayerPrefs using the key.
- Calls Save() to persist the changes.
- Logs a success message for debugging.

🕹️ **Key Concept:**
Unity's input system allows you to override default bindings. This method **serializes** those overrides into a JSON format so they can be written to disk.

LoadBindings
- Accepts the same parameters as the save method.
- Verifies that the asset is valid.
- Checks if PlayerPrefs contains data for the provided key.
- If data is found:
  - It reads the JSON string.
  - Applies it to the input asset, restoring all overridden bindings.
- Logs a success message.

🕹️ **Key Concept:**
This is the opposite of the save method — it **loads the stored input mappings** from PlayerPrefs and applies them back to the current control setup.

🔁 **How the System Works Together**
1. At startup:
   - The load method is executed automatically.
   - If saved bindings exist, they are loaded and applied to the input system.
2. During gameplay:
   - A player can rebind a control through an input UI.
3. At any moment (e.g., when the player clicks "Save Settings"):
   - The SaveDefaultBindings method is called, storing the new mappings.
4. On the next launch:
   - The system automatically reloads the saved mappings using LoadDefaultBindings.

📁 **Summary Table**

Method	Purpose
SaveDefaultBindings	Saves default input bindings from the central config asset.
LoadDefaultBindings	Automatically loads saved input bindings at startup.
SaveBindings	Converts and saves an input asset's bindings to PlayerPrefs.
LoadBindings	Reads binding data from PlayerPrefs and applies it to input.

🧪 **Example Use Case (in plain logic)**
1. The player remaps "Jump" from the **Spacebar** to the **Enter key**.
2. The input system updates the override in memory.
3. The game calls **SaveDefaultBindings** when the player clicks "Save".
4. The override is stored in PlayerPrefs as a JSON string.
5. The next time the game runs:
   - The JSON is retrieved.
   - The override is reapplied.
   - The player's remap is preserved without needing to reconfigure.

✅ **Benefits**

Feature	Description
Persistence	Remapped controls stay saved between sessions.
User-Friendly	No need to manually rebind after restarting the game.
Minimal Setup	Automatically initializes if the data asset is correctly configured.
Modular	Can be extended to support multiple profiles or backup systems.

Rebind Control Manager and Rebind Control Manager TMP

This script is a Unity MonoBehaviour component designed to provide advanced runtime input rebinding features for games using the Input System package. It allows players to remap controls during gameplay using UI elements, supports composite bindings (e.g., WASD), provides visual feedback (icons and labels), handles duplicate detection, and saves overrides persistently. Here's a detailed breakdown of how it works:

🔧 Purpose and Usage

This component is added to a GameObject in the UI. It connects UI elements like buttons, labels, and icons to specific input actions. At runtime, the player can:

- Start a rebind operation via a UI button.
- Cancel the rebind using a specific input (like Escape).
- Reset a binding to its default.
- See updated UI (icon, label, control text) for the selected input.

It handles all of this automatically, including saving/loading rebinds via PlayerPrefs.

🔍 Explanation by Component

=== Serialized Fields ===

These fields are configured in the Unity Editor.

Input Settings

- **cancelRebindActionReference**: An InputAction that can cancel the rebinding process mid-operation.
- **inputActionReference**: The action whose binding the user wants to modify.
- **targetBindingId**: A GUID that identifies which specific binding inside the InputAction is being edited.
- **displayOptions**: Controls how the input name is shown (e.g., simplified name, full path, etc).

UI Elements

- **actionLabel**: Displays the name of the input action (e.g., "Jump").
- **autoLabelFromAction**: If true, the label updates automatically based on the InputAction's name.
- **startRebindButton**: The button to begin rebinding.
- **bindingDisplayText**: Text element showing the current key/button assigned.
- **actionBindingIcon**: Visual icon representing the current input device (e.g., keyboard icon).
- **rebindOverlayUI**: Overlay that appears while rebinding is in progress.
- **rebindPromptText**: Shows a prompt like “Press a key…”.
- **actionRebindIcon**: Icon shown during rebinding (can highlight the active device).
- **resetButton**: Allows user to revert the binding to default.

Events

- **onRebindStart / onRebindStop**: Triggered when a rebind begins or ends.
- **onUpdateBindingUI**: Allows external scripts to react when the UI is updated.

🔐 Private Fields

- **currentRebind**: Stores the active rebinding operation, used to cancel or continue it.
- **allRebindManagers**: A global list of all active instances of this script.
- **currentEventSystem**: Reference to Unity’s EventSystem, cached to avoid UI input during rebinding.

🔄 Public Properties

These give access to some private fields and also auto-refresh the UI when changed.

- **ActionReference**: Sets the input and refreshes the UI.
- **DisplayOptions**: Sets how the input name is displayed.

- **BindingDisplayText**, **ActionBindingIcon**, **RebindPromptText**, **ActionRebindIcon**: Access UI elements for customization.

🧩 Key Methods

🔄 Unity Lifecycle

- **Awake()**: Logs warnings if fields are misconfigured.
- **Start()**: Hides the overlay UI and attaches button click events.
- **OnEnable() / OnDisable()**: Manages shared static list and registers input system change callbacks.
- **OnValidate()**: Ensures UI stays up to date when modified in the Editor.

🎮 Core Logic

TryResolveBinding(out action, out index)

- Finds the correct InputAction and binding index using the stored binding ID.
- Returns false if invalid.

RefreshBindingDisplay()

- Updates the binding string shown in the UI.
- Checks if the binding is overridden and enables the reset button accordingly.

ResetToDefault()

- Reverts the binding (and any composite parts) to their default path.
- Calls InputBindingSaver.SaveDefaultBindings() to persist the reset.

StartInteractiveRebind()

- Starts the rebind process from a button click.
- Supports composite rebinding by finding and starting from the first part.

🔁 ExecuteInteractiveRebind(...)

- Begins a rebind operation for a specific binding.
- Shows overlay UI and prompt.
- Temporarily disables the EventSystem and current action map.
- Checks for duplicate bindings during the OnPotentialMatch step.
- If rebinding composite controls, continues rebinding each part in sequence.

👄 CheckDuplicateBindings(...)

- Prevents assigning the same input to multiple bindings.
- Scans all actions in the asset and compares effective paths.

✏️ GenerateCustomPrompt(...)

- Builds a prompt string like <Up>/<Down>/<Left>/<Right> for composite bindings.
- Highlights the currently active rebind part.

✋ MonitorCancelInput() / OnCancelInputPerformed(...)

- Listens for a cancel input (e.g., Escape key) during rebind and cancels if pressed.

🔄 HandleActionChange(...)

- Reacts to any action/binding changes at runtime.
- Refreshes UI when something is modified in the Input System.

🔤 UpdateActionLabel()

- Auto-fills the action label with the action name if autoLabelFromAction is enabled.

📜 Nested Event Classes

- **UpdateBindingUIEvent**: Used to notify listeners when the binding display updates.
- **InteractiveRebindEvent**: Used when rebind starts or stops, providing context (like the operation).

✅ Summary

🎯 What this script does:

- Allows rebinding inputs in a UI-friendly way.

- Supports composite inputs (like WASD).
- Automatically updates UI.
- Validates input to prevent duplicates.
- Provides canceling and resetting.
- Persists changes via PlayerPrefs.

🧑‍💻 **Typical Usage:**

1. Attach this script to a UI prefab.
2. Assign the correct InputAction, Binding ID, and UI elements in the Inspector.
3. Customize events like onRebindStart to provide audio or feedback.
4. Let users change controls at runtime.

Gamepad Icon Rebind Handler and Gamepad Icon Rebind Handler TMP

This script is a Unity component designed to automatically replace the text shown for input bindings (like "Button South" or "Right Trigger") with corresponding **gamepad icon sprites**, enhancing UI clarity for controller players. It works in conjunction with the RebindControlManager component and listens for updates when the player rebinds controls.

🎯 **Purpose and Usage**

- Attach this script to a UI GameObject and assign a parent GameObject that contains one or more RebindControlManager components.
- It listens for input binding updates and dynamically replaces text with appropriate icons based on the type of gamepad (PlayStation or Xbox).
- Supports multiple managers in the same group.

🔍 **Component and Variable Breakdown**

🔧 **Serialized Field**

- **rebindControlGroup**: A reference to the parent object that contains one or more RebindControlManager components. This group will be scanned to attach update listeners.

📦 **Private Variable**

- **extensionData**: Stores a reference to the InputSystemExtensionData asset, which holds all the sprite mappings for control paths (e.g., icons for buttonSouth, leftTrigger, etc.).

🧠 **Main Behavior**

🔄 **Start()**

- This method is automatically called when the object becomes active.
- First, it loads the InputSystemExtensionData asset using the helper method GetInputSystemExtensionData().
- Then, it retrieves all RebindControlManager components that are children of the rebindControlGroup.
- For each manager:
  - It adds the OnBindingUIUpdate method as a listener to their onUpdateBindingUI event.
  - It calls RefreshBindingDisplay() to immediately apply icons if possible.

✅ This setup ensures that icon replacements occur as soon as the UI initializes and whenever bindings are changed.

🖼️ **OnBindingUIUpdate(...)**

This method is called **every time a binding UI is updated** (e.g., after rebinding or loading saved settings). It performs the logic of deciding whether to display **an icon or fallback text**.

Parameters:

- **manager**: The RebindControlManager that triggered the update.

- **bindingDisplay**: The default text string for the control (e.g., "Button South").
- **deviceLayoutName**: The type of controller (e.g., "Gamepad", "DualShockGamepad").
- **controlPath**: The specific control identifier (e.g., <Gamepad>/buttonSouth).

Behavior:

1. **Validation**:
   - Checks if deviceLayoutName, controlPath, and extensionData are valid.
   - If any are missing, it skips processing.
2. **Icon Lookup**:
   - If the layout is a **PlayStation-style gamepad**, it tries to fetch the icon from the ps4 icon map.
   - Otherwise, if the layout is any generic **gamepad**, it uses the xbox icon map.
3. **Access UI Elements from the Manager**:
   - BindingDisplayText: Text component for the main UI.
   - ActionBindingIcon: Image used to show the icon in the main UI.
   - RebindPromptText: Text shown while rebinding is active.
   - ActionRebindIcon: Icon shown while rebinding is active.
4. **UI Replacement Logic**:
   - If an icon was found:
     - It assigns the icon sprite to the two image components.
     - It enables the image GameObjects and hides the text GameObjects.
   - If no icon was found:
     - It hides the images and shows the text instead.

🎮 Example:
If the player binds Button South on a DualShock controller, and that matches a cross icon, it shows the image of the ✖ button instead of the word "Button South".

✅ **Summary of How It Works**

Feature	Description
🎯 **Purpose**	Replace control names with gamepad icons (PS/Xbox) dynamically.
🧠 **Listens to**	RebindControlManager.onUpdateBindingUI events.
🗺️ **Icon Mapping**	Uses InputSystemExtensionData.ps4 or .xbox maps to find control icons.
🔁 **Automatic Setup**	Scans all rebind managers under a parent group and auto-hooks into events.
📺 **UI Behavior**	Shows icon if found, otherwise falls back to original binding text.
🛠️ **Usage Requirements**	Requires setup of icon mapping ScriptableObject and properly configured managers.

📦 **Example Use Case**

1. Create InputSystemExtensionData with correct icons mapped.
2. Add RebindControlManager to UI elements that control input rebinding.
3. Add this script to the same UI screen, assign the container object with those managers.
4. When the game starts or rebinds change, the icons automatically appear for the appropriate inputs.

Reset All Rebind Control

This script is a **UI utility component** used in Unity for resetting all **input rebind controls** within a specific group to their original, default bindings. It is meant to be attached to a GameObject containing a **UI button** and a reference to a **group of rebind managers**.

🎯 **Purpose and Usage**

- Use this component when you want to provide a **"Reset to Default"** functionality in your input settings menu.

- It detects all RebindControlManager and RebindControlManagerTMP components under a specified GameObject.
- When the assigned button is clicked, it resets **all found components** to their default input settings.
- It supports both **standard Unity Text-based** and **TextMeshPro-based** input rebind UI components.

🔧 Variable Breakdown

resetAllButton

- UI button that the player clicks to trigger the reset.
- The script checks if this is assigned and then connects its onClick event to a method.

rebindControlGroup

- A GameObject in the scene that acts as a container.
- All child objects under this group will be searched for rebind managers.

rebindControls

- A list that holds all the RebindControlManager components found inside the control group.
- These handle rebind logic using **Unity's legacy UI system (Text)**.

rebindControlsTMP

- A list that holds all RebindControlManagerTMP components found inside the same group.
- These handle rebind logic using **TextMeshPro-based UI**.

🔁 Method Explanation

Start()

- Called automatically when the component is initialized.
- It performs the following:
    1. **Event Registration**:
        - If resetAllButton is assigned, it adds the ResetAll method as a click listener.
        - Logs a warning if no button is assigned.
    2. **Component Detection**:
        - Searches through all children of rebindControlGroup and stores found components of both types in the respective lists.
        - These will later be used to reset bindings.

ResetAll()

- Called when the button is clicked.
- This method:
    1. Iterates through all components in rebindControls.
        - For each one that exists, calls its ResetToDefault() method.
    2. Repeats the process for all components in rebindControlsTMP.

This guarantees that **every control rebinding UI element under the group is reset** back to its default input mapping in one action.

🧠 How It Works in Practice

1. In the Unity Editor:
    - Drag this script onto a UI object.
    - Assign a Button component to the resetAllButton field.
    - Assign a GameObject containing all rebind UI elements to the rebindControlGroup field.
2. When the player clicks the button:
    - The script resets **every** rebindable control (both Text-based and TMP-based) under the group.
3. Each control individually reverts to the default binding saved in the input action asset.
    - Any PlayerPrefs override is effectively discarded.

✅ Summary Table

Element	Purpose
resetAllButton	Triggers the reset when clicked.
rebindControlGroup	Container where all input rebind components are located.
rebindControls	Stores all Text-based rebind UIs (standard Unity Text).

rebindControlsTMP	Stores all TMP-based rebind UIs (TextMeshPro components).
Start()	Initializes button listener and gathers all rebind UI components.
ResetAll()	Calls the reset method on all detected rebind components.

Dependencies

- Requires both RebindControlManager and/or RebindControlManagerTMP components in child objects.
- Assumes each of these components has a method called ResetToDefault() that reverts the input binding.

Input Display Manager

Purpose of the Script

This script dynamically updates and animates UI icons that represent input controls (like a button or joystick direction). It switches the icons depending on whether the player is using a keyboard or a gamepad. It also animates those icons (e.g., color changes) when inputs are pressed or released.
It supports:

- **Single input displays** (e.g., "Jump" button),
- **Multiple direction displays** (e.g., arrows for movement),
- **Color transitions** for visual feedback,
- **Automatic control detection** between keyboard and gamepad.

Control Type

The script can work in **automatic** mode (detects if a gamepad is present) or **manual** mode (you set the device type yourself).
There are two types of controls:

- **Keyboard**: Uses individual keys.
- **Gamepad**: Uses joystick or controller buttons.

Main Data Structures

1. Input Viewer Data

Each entry in this list is for a **single input**, like a "Jump" or "Shoot" action.
Each entry includes:

- A **name tag**: identifier.
- A flag to determine whether this input is currently **enabled**.
- A reference to the **input action**.
- Binding identifiers for **keyboard and gamepad** (used to fetch icons).
- A reference to the **UI image** that shows the icon.
- A float-type event handler to **listen to the input and respond visually**.

2. Input Multiple Views Data

This represents **directional inputs** like Up, Down, Left, Right.
Each entry includes:

- A **name tag**.
- An enable flag.
- Input action reference.
- Binding IDs for keyboard and gamepad.
- References to **four directional icons**.
- A vector2-type event handler to track directional input and respond accordingly.

Inspector Fields

These variables are editable in Unity's inspector and define how the system behaves.

- **Auto detection toggle**: Enables/disables automatic input detection.
- **Default control type** if auto is off.
- **Colors**: One for activated (pressed), one for disabled (released).
- **Transition times**: Defines how fast colors change or icons fade in/out.
- **Lists**: Define what inputs to show on screen, either individually or as directions.

Internal Workflow

1. Startup Phase

- Checks if input viewer data is valid (no missing references or empty names).
- Initializes the event handlers for both single inputs and directional inputs.

2. Device Detection

- On enable, if auto detection is on, the script checks if a gamepad is connected.
- It also listens to device change events from Unity's input system.
- If a new gamepad is plugged in or removed, it switches the input control type and updates icons.

3. Input Event Handling

- For single inputs:
  - When the button is pressed: the icon transitions to the **active color**.
  - When released: the icon transitions to the **disabled color**.
- For directional inputs:
  - On press: shows directional icons.
  - On hold: checks which direction is being pressed and highlights that direction.
  - On release: resets all direction icons to the disabled color.

🎨 Icon Updates

When control type changes or icons need to be refreshed:

- For single inputs:
  - It gets the correct icon based on control type and assigned binding ID.
- For directional inputs:
  - For keyboard, it assigns icons per direction.
  - For gamepad, it uses one icon (usually “Up”) to represent the composite direction.

Icon sources come from a shared data asset (extensionData), which maps input paths or key codes to specific sprites.

🧩 Helper Functions

- **Set icons active/inactive**: Turns direction icons on/off depending on context.
- **Color transitions**: Animates color from one state to another for visual feedback.
- **Fade out & disable**: Smoothly hides icons when they're disabled.
- **Unbind events**: Cleans up when the component is destroyed to avoid memory leaks or repeated listeners.

🧪 Usage

You can use this component in a UI canvas to:

- Display the current active input icons.
- Show feedback when the player presses a button or moves a joystick.
- Dynamically respond to player hardware changes (e.g., switching from keyboard to controller).
- Support both single inputs (like “Jump”) and directional inputs (like “Move”).

✅ Customization

You can:

- Add new inputs by filling in the lists with tags, actions, and image references.
- Assign your own sprites in the data asset for specific keys or gamepad controls.
- Use EnableAndDisableViewer to show or hide icons dynamically during gameplay.

Input Display Manager Inspector

🔍 Purpose of the Script

This script customizes the Unity **Inspector panel** for the InputDisplayManager component.
Its primary function is to **automatically create or assign UI Image elements** to be used as input icons, based on the input actions defined in that component.
It adds a custom **button labeled “Create Automatic Presets”**, which generates the appropriate child UI elements (Images) in the scene hierarchy if they do not already exist.

🧱 Structure Breakdown

🧭 Target Component

The script operates on the InputDisplayManager component in Unity. It customizes how that component is displayed and edited in the Inspector window.

📦 Inspector GUI Logic

🔘 "Create Automatic Presets" Button

When this button is clicked in the Unity Editor:

1. It loops through all selected instances of InputDisplayManager in the scene.
2. It stores an undo point so changes can be reverted.
3. It calls the **automatic generation function** to create or link Image UI elements.
4. It marks the target object as modified so Unity knows to save the change.

🖼️ Default Inspector

After handling the custom button, the script continues to draw the standard Unity fields for the component so that other properties (like color or lists) remain visible and editable.

🧠 Main Logic: Automatic Preset Creation

📌 Purpose

This core method scans each input entry (both single and directional inputs) and:

- **Renames** existing images to follow a clean naming convention.
- **Creates** new UI Image elements if none exist.
- **Re-links** those images into the component's data structure so they're used at runtime.

📜 Single Input Entries

For each input:

- It sets the **name tag** based on the input action name (for identification).
- It either:
  - Renames the existing linked image.
  - Searches the scene hierarchy for a matching image by name.
  - Creates a new image under the component's transform if needed.
- The image is given a default size and linked back to the component.

🔼 Directional Input Entries

For inputs like arrow keys or D-pad:

- The name tag is also updated from the input action.
- Each direction (Up, Down, Left, Right) is handled independently:
  - The method either **renames an existing image**, **finds a matching child**, or **creates a new one**.
  - The created or found image is returned and assigned to the corresponding direction.

🧩 Directional Image Creation Helper

This helper method is responsible for managing each individual directional image:

- It ensures naming follows a standard format like: "Image (Jump_Up)", "Image (Move_Left)", etc.
- If an existing image is passed in, it is renamed and returned.
- If not, it tries to find an existing GameObject under the parent transform.
- If no match is found, a new GameObject is created with a default image and size.

🔄 Undo Support

Throughout the process, the script uses Unity's undo system to:

- Allow developers to revert changes through **CTRL+Z / CMD+Z**.
- Ensure all GameObject creations, modifications, and additions are tracked in the editor's undo history.

🧪 Usage Summary

When to Use:

- After assigning input actions in the InputDisplayManager component.
- If you want to quickly generate corresponding UI images for those inputs.
- To avoid manually creating and linking UI icons in the scene.

How to Use:

1. Select a GameObject that has the InputDisplayManager component.

2. Click the **“Create Automatic Presets”** button.
3. It will create or update child GameObjects with image components.
4. These images will be linked back to the component automatically.
5. You can then assign sprites or customize their visuals.

✅ **Benefits**

- **Fast setup** for prototyping or production.
- **Consistent naming convention** makes it easy to organize UI elements.
- Ensures **no missing references** or misconfigured icons.
- Designed with **undo safety**, making it non-destructive.