Input System Extension

Documentation

Creator: Lucas Gomes Cecchini

Pseudonym: AGAMENOM

Overview

This document provides guidance on using the **Input System Extension for Unity**, a comprehensive asset designed to streamline input configuration, rebinding, and management for keyboards, gamepads, and other controllers in your game.

With this extension, you can:

- **Easily customize and remap input controls** directly from in-game menus, UI panels, or the editor interface.
- **Work with multiple types of Scriptable Objects**:
  - One for **persistent storage** of player preferences using PlayerPrefs, ensuring input settings are saved between sessions.
  - Another for **programmatic access**, allowing developers to read, modify, or reset input configurations during runtime.

- **Handle both text and icon-based input displays**, automatically switching between key labels and gamepad icons based on the connected controller.
- **Perform runtime rebinding with full lifecycle management**, including start, completion, cancellation, duplicate prevention, and UI feedback.
- **Reset individual or all input bindings to defaults** with a single click, ensuring players can easily revert any changes.
- **Automatically save changes** to input settings as they occur, reducing manual effort and preventing loss of configuration.

The Input System Extension provides **flexible, runtime input remapping and visualization**, delivering a seamless player experience while simplifying input management for developers.

Instructions

You can get more information from the Playlist on YouTube:

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

Script Explanations

GetAction Attribute

Purpose
This script provides a **custom attribute** for Unity inspectors that allows selecting an **InputAction** from a predefined default InputActionAsset using a dropdown. Instead of manually assigning InputActionReferences, developers can pick from a list of actions directly in the inspector, simplifying setup and reducing errors.

Variables

1. **extensionData**
    - Cached reference to the ScriptableObject containing the default InputActionAsset.
    - Ensures the asset is loaded only once for performance and prevents repeated resource lookups.

Classes and Methods
GetActionAttribute

- This is a marker attribute that is placed above a field in a Unity inspector.
- Purpose: Signals that the field should use the custom dropdown instead of the default object field.
- No variables or methods inside; it acts purely as a label for the drawer.

GetActionDrawer

- This is a **custom drawer** that handles how the field with the [GetAction] attribute is displayed in the Unity inspector.
- It overrides the default drawing method to show a dropdown of InputActions instead of the usual object picker.

OnGUI (Main Method)

- Parameters:
    - **position**: Area in the inspector to draw the field.
    - **property**: The serialized property being drawn (the field that uses [GetAction]).
    - **label**: Label shown in the inspector.

Workflow inside OnGUI:

1. **Property Type Check**

    - Ensures the property is an object reference because only InputActionReference fields are valid.
    - Shows an error message in the inspector if used incorrectly.
2. **Load the ScriptableObject**
    - Loads the InputSystemExtensionData if it hasn't been cached yet.
    - This contains the default InputActionAsset from which actions are displayed.
3. **Validate Asset Availability**
    - Checks if the ScriptableObject or the default InputActionAsset is missing.
    - Displays an error message in the inspector if they are missing.
4. **Gather Actions**
    - Loops through all action maps in the default InputActionAsset.
    - Collects all valid InputActions into a list for the dropdown.
5. **Handle Empty Actions List**
    - If no actions are found, shows a warning in the inspector.
6. **Get Current Selection**
    - Reads the currently assigned InputActionReference from the property.
    - Determines its index in the actions list.
7. **Prepare Dropdown Display**
    - Generates display strings for each action in the format: "ActionMapName/ActionName".
8. **Draw the Dropdown**
    - Displays a popup in the inspector.
    - Shows the current selection and allows changing it.
9. **Update Selection**
    - If the user selects a different action:
        - Finds the corresponding InputActionReference in the asset.
        - Assigns it to the property.
        - Marks the object as dirty so Unity knows it has changed.
        - Applies serialized property changes to save the selection.

Usage

- Place the attribute above a field in a MonoBehaviour or ScriptableObject that uses InputActionReferences. Example usage in an inspector:

```
[GetAction]
public InputActionReference jumpAction;
```

- Unity will display a dropdown for this field instead of the default object selector.
- The dropdown lists all actions in the default InputActionAsset defined in InputSystemExtensionData.
- Selecting an action automatically assigns the correct InputActionReference.

Summary

The script **enhances inspector usability** for input configuration:

- It avoids manual dragging of InputActionReferences.
- Ensures consistency with the default input setup.
- Handles missing assets or invalid fields gracefully by showing warnings and errors.
- Integrates with Unity's serialization system to automatically save changes in the inspector.

This makes it ideal for projects that rely heavily on the new InputSystem, providing a **clean and error-proof workflow** for selecting actions in the editor.

BindingId Attribute

Purpose

This script provides a **custom attribute** and **property drawer** that allows Unity developers to select a specific **binding from an InputAction** in the inspector. Instead of manually copying binding IDs, a dropdown is displayed that lists all available bindings from a referenced InputAction.

This is particularly useful for projects using the Unity Input System where multiple bindings exist, such as keyboard, gamepad, or composite controls.

Variables

1. **actionFieldName**
   - Defined in the custom attribute.
   - Holds the name of the InputActionReference field in the same class.
   - This links the binding ID field to its corresponding InputAction.
2. **extensionData (not used directly here)**
   - Not needed in this drawer because the InputActionReference is directly referenced in the class.
3. **bindings, options, optionValues, currentBindingId, selectedIndex**
   - Used inside the drawer to store all available bindings, the display names for the dropdown, the IDs for each option, and the currently selected binding.

Classes and Methods

BindingIdAttribute

- Custom attribute applied to a string field in a MonoBehaviour or ScriptableObject.
- Purpose: Links a field representing a **binding ID** to a specific **InputActionReference** field.
- Stores the name of the InputActionReference field.

BindingIdDrawer

- A **custom property drawer** that renders the string field as a dropdown of binding options.
- Ensures the dropdown dynamically reflects the bindings in the referenced InputAction.

OnGUI (Main Method)

1. **Validate Attribute**
   - Confirms that the attribute used on the field is a BindingIdAttribute.
   - If not, draws the default field instead.
2. **Resolve InputActionReference**
   - Finds the corresponding InputActionReference field in the same serialized object using the name stored in the attribute.
   - Shows an error if the field cannot be found.
3. **Get the InputAction**
   - Retrieves the actual InputAction from the InputActionReference.
   - Shows a warning if no InputAction is assigned.
4. **Collect Binding Data**
   - Retrieves all bindings of the InputAction.
   - Prepares a list of display strings for each binding, including device info, composite part names, and control scheme names.
   - Keeps track of the currently selected binding ID to show the correct selection in the dropdown.
5. **Draw Dropdown**
   - Displays a popup in the inspector using the prepared display strings.
   - Updates the string field with the binding ID corresponding to the selected option.
   - Applies the changes to the serialized object and marks it as modified so Unity saves the new selection.
6. **Formatting Enhancements**
   - Replaces '/' with '\' in the display string to prevent submenus in the dropdown.
   - Combines control scheme names in parentheses for clarity.
   - Adds composite part names as prefixes for multi-part bindings.

Usage

- Place the attribute above a string field that will hold a **binding ID**.
- Reference an InputActionReference field in the same class by passing its name to the attribute.

Example:

```
public InputActionReference jumpAction;

[BindingId(nameof(jumpAction))]
public string jumpBindingId;
```

- Unity will display a dropdown for jumpBindingId in the inspector.

- The dropdown lists all bindings from jumpAction, including device, scheme, and composite information.
- Selecting a binding automatically stores its ID in the string field.

Summary
This script **simplifies input binding selection in the Unity inspector**:
- Removes the need to manually copy binding IDs.
- Automatically displays all bindings from a specific InputAction.
- Supports composite bindings and control schemes for clarity.
- Updates serialized objects safely and marks them as dirty to persist changes.

It is ideal for projects using multiple control schemes and wanting a **clean, error-free workflow** for configuring action bindings in the editor.

Input System Extension Prefab Creator

Overview
This script is an **editor utility** that allows developers to quickly instantiate prebuilt UI prefabs related to the Input System Extension.
- It adds entries to the **GameObject menu** in the Unity Editor.
- Automatically handles **canvas creation**, **event system setup**, **parenting**, **prefab unpacking**, and **renaming**.
- Ensures that UI prefabs are ready to use in the scene without manual setup.

Main Class — InputSystemExtensionPrefabCreator
This class is static, meaning its methods can be called without creating an instance. It handles prefab creation, parenting, and editor integration.

Section 1 — Utilities
These methods handle the core logic for creating and configuring UI prefabs.

Method: CreateUICanvas
Purpose:
- Ensures that a Canvas and EventSystem exist in the scene for UI elements.
- Creates them automatically if missing.

Step-by-step behavior:
1. **Create Canvas GameObject**
   - Generates a new GameObject named "Canvas".
2. **Add Required Components**
   - Adds Canvas, Canvas Scaler, and Graphic Raycaster components, which are required for rendering UI elements.
3. **Configure Canvas Properties**
   - Sets rendering mode to screen overlay.
   - Assigns the UI layer, sorting order, and target display.
4. **Create EventSystem GameObject**
   - Adds a new "EventSystem" GameObject.
   - Adds components for handling user input (EventSystem and StandaloneInputModule).
5. **Undo Registration**
   - Registers both objects with Unity’s Undo system to allow undoing creation via editor commands.
6. **Return Canvas Reference**
   - Returns the Canvas component to use as a parent for UI prefabs.

Usage:
- Called automatically when a UI prefab is being instantiated and no canvas exists.

Method: CreateAndConfigurePrefab
Purpose:
- Instantiates a prefab by name and sets it up properly in the scene.

Parameters:
- **fileName:** The prefab’s name without extension.

- **selectedGameObject:** The currently selected object in the hierarchy to use as a parent.
- **isUI:** If true, the prefab is considered a UI element and will be parented under a Canvas.

Step-by-step behavior:

1. **Find or Create Canvas**
   - If the prefab is UI and no canvas exists, it calls CreateUICanvas.
2. **Find Prefab Asset**
   - Calls FindPrefabByName to locate the prefab in the project.
   - Logs an error if the prefab is missing.
3. **Determine Parent**
   - If an object is selected in the hierarchy, the new instance is parented under it.
   - Otherwise, if it’s a UI prefab, it is parented under the Canvas.
4. **Instantiate Prefab**
   - Creates an instance in the scene.
5. **Finalize Setup**
   - Calls FinalizePrefabSetup to unpack the prefab, register Undo, select it, and allow renaming.

Usage:

- Central method for all prefab creation menu items. Handles both scene hierarchy and UI considerations.

Method: FindPrefabByName

Purpose:

- Searches the project for a prefab with a specific name.

Behavior:

1. Uses the asset database to search for prefab assets that match the name.
2. Checks each path to confirm the name matches exactly.
3. Returns the prefab GameObject if found, otherwise logs an error and returns null.

Usage:

- Called by CreateAndConfigurePrefab to locate the prefab asset before instantiation.

Method: FinalizePrefabSetup

Purpose:

- Handles final scene setup for a newly instantiated prefab.

Behavior:

1. **Undo Registration**
   - Registers the new GameObject with the Undo system.
2. **Unpack Prefab**
   - Converts the prefab instance into a fully editable GameObject in the scene.
3. **Select and Focus**
   - Sets the new instance as the active selection.
4. **Trigger Renaming**
   - Sends a delayed key event to simulate pressing F2, allowing the user to rename the object immediately.

Usage:

- Ensures prefabs are fully editable, visible, and easy to work with after instantiation.

Section 2 — Menu Items

This section defines **menu entries in the GameObject menu** for creating specific Input System Extension UI prefabs.

Each menu item:

1. Defines a hierarchical path in the menu (GameObject → UI → Input System Extension → [Prefab Name]).
2. Calls CreateAndConfigurePrefab with the prefab name and selected object.
3. Passes true for UI prefabs to ensure proper Canvas parenting.

Examples of Prefabs:

- "Rebind Control Manager (Legacy)"
- "Rebind Control Manager (TMP)"
- "Button (Reset All) [Legacy]"
- "Button (Reset All) [TMP]"

- "Input Display Manager"

Usage:

- Developers select a menu item to automatically instantiate a prefab in the scene.
- The prefab is placed in the hierarchy correctly, unpacked, and ready to use.

Variables and Their Purpose

Variable	Type	Purpose
canvasGO	GameObject	Temporarily holds the created Canvas.
canvas	Canvas	Component reference for UI parenting.
eventSystemGO	GameObject	Temporarily holds the created EventSystem.
fileName	String	Prefab name to search and instantiate.
selectedGameObject	GameObject	Current selection in hierarchy for parenting.
isUI	Boolean	Indicates if prefab should be parented under Canvas.
prefab	GameObject	Asset reference for the prefab found in the project.
parent	Transform	Target parent for the new instance in the hierarchy.
instance	GameObject	Newly instantiated prefab object.
newGameObject	GameObject	Prefab instance passed to finalization.

Practical Usage

1. **Developer Workflow**
   - Open the Unity Editor hierarchy.
   - Select a GameObject or leave nothing selected.
   - Go to the **GameObject → UI → Input System Extension** menu.
   - Click the desired prefab entry.
2. **Automatic Setup**
   - If the scene lacks a Canvas or EventSystem, they are created automatically.
   - Prefab is instantiated, parented, unpacked, and selected.
   - Renaming is enabled immediately.
3. **Benefits**
   - Eliminates repetitive manual prefab setup.
   - Ensures consistent UI hierarchy and Canvas usage.
   - Makes working with input system UI prefabs faster and less error-prone.

Input System Extension Data

Overview

This script defines a **ScriptableObject configuration** used to centralize and manage custom input data in a Unity project that uses the **new Input System**.

Its primary purposes are:

- To store references to the main input asset.
- To maintain a mapping of keyboard keys and gamepad buttons to specific sprites.
- To allow the Unity Editor to open and edit this configuration asset easily.

The configuration acts as a “hub” for input settings and UI-related data, so developers and designers can manage visual input representations (icons, button prompts, etc.) in one place.

Main Class: InputSystemExtensionData

This is the main ScriptableObject that holds all the data and structures related to the input system configuration.

Section 1 — Input Settings

defaultInputAction

- Holds a reference to the main **Input Action Asset** used by the player or game.
- It defines all input bindings such as move, jump, attack, pause, etc.
- This asset is typically created and edited using Unity’s Input System editor window.

playerPrefsKey

- A **unique string identifier** used to store input binding overrides in PlayerPrefs.
- PlayerPrefs is Unity’s local storage system for saving small amounts of data.
- The bindings are saved as **JSON**, meaning players can rebind controls and have them persist between play sessions.

Section 2 — Icon Settings
This section manages how input icons appear in the UI, especially for displaying prompts such as "Press A to Continue" or "Press Space to Jump".
defaultSprite
- A **fallback image** shown for keyboard keys if no specific icon exists for a given key.
- Prevents missing icons from breaking the UI.

KeyCodes
- A **list of key-to-sprite mappings** for keyboard keys.
- Each entry is represented by a InputSpriteList structure.
- Used for showing the correct icon in tutorials, menus, or HUDs.

xbox and ps4
- Both are **GamepadIcons structures** that contain all icon references for Xbox-style and PS4-style controllers, respectively.
- This allows the UI to dynamically display the correct images based on the type of gamepad connected.

Struct: InputSpriteList
A lightweight structure used for keyboard key mappings.
- **keyCode** — Represents a specific keyboard key (e.g., A, W, Space, etc.).
- **keyName** — A string used to match the key with input actions.
  It allows more flexibility than relying solely on the key code.
- **sprite** — The icon (image) representing that key visually.
  For example, an image of the space bar or an arrow key.

This list is displayed in the Unity Inspector, making it easy to assign or change icons visually.

Struct: GamepadIcons
A structured collection of **sprites representing common gamepad controls**.
It allows you to replace textual button names with matching images in the UI.
Categories and Variables
- **Face Buttons:**
  Represent the four main buttons on most controllers:
  - South (A / Cross)
  - North (Y / Triangle)
  - East (B / Circle)
  - West (X / Square)
- **Menu Buttons:**
  Represent Start/Options and Select/Share.
- **Triggers and Bumpers:**
  Include L1, R1, L2, R2.
- **D-Pad:**
  Includes separate sprites for each direction and an optional full D-Pad icon.
- **Analog Sticks:**
  Represent both left and right sticks, along with their "press" versions (when pushed in).

Method: GetSprite(controlPath)
This method is responsible for **returning the correct sprite** when given a control path string.
- For example, if the game needs the sprite for "buttonSouth", it returns the buttonSouth sprite.
- It uses a **switch expression** to match the string and return the correct icon.
- If no match is found, it calls another method (HandleUnknown).

Usage example:
The UI might call this when updating a button prompt to show the appropriate image for the player's controller type.

Method: HandleUnknown(path)
A private helper method that:
- Logs a **warning message** if the provided control path does not match any known button name.
- Returns **null** to indicate no valid sprite was found.

This helps developers identify missing or incorrect mappings in the configuration.

Editor Integration (Unity Editor Only)
The final section is wrapped in a preprocessor block so it only compiles in the Unity Editor.
Static Class: InputSystemExtensionDataWindow
This class adds a custom **menu option in the Unity Editor** that allows users to quickly open and inspect the configuration asset.
Menu Command
- Appears under **Window → Input System Extension → Input System Extension Data**.
- When selected, it calls the OpenInputSystemExtensionData() method.

Method: OpenInputSystemExtensionData()
This method does the following:
1. **Retrieves** the existing configuration asset using a helper function.
2. **Checks** if an inspector window for that asset is already open.
3. If it is open, **focuses** on it.
4. If not, **opens** a new property editor window showing the ScriptableObject data.

If the asset cannot be found, it logs an **error message** to alert the developer.
This makes it much easier to find and edit the input configuration without manually locating it in the Project window.

How It's Used in a Project
1. **Create the ScriptableObject:**
   From Unity's Project window, the developer creates an instance of InputSystemExtensionData.
2. **Assign Input Asset and Sprites:**
   The main InputActionAsset, default sprite, key mappings, and gamepad icons are set up inside the ScriptableObject.
3. **Access at Runtime:**
   Game code can reference this asset to:
   - Load player bindings.
   - Show the correct input icons.
   - Handle custom rebinds.
4. **Editor Access:**
   Through the menu command, developers can easily open the asset in the inspector for editing.

Summary

Element	Type	Purpose
defaultInputAction	Asset Reference	Holds main input bindings.
playerPrefsKey	String	Identifies stored bindings.
defaultSprite	Sprite	Fallback for missing icons.
KeyCodes	List	Keyboard key-to-icon mappings.
xbox / ps4	Struct	Gamepad icon collections.
GetSprite()	Method	Returns matching icon for a control path.
HandleUnknown()	Method	Logs missing control mappings.
InputSystemExtensionDataWindow	Editor Class	Adds Unity menu shortcut.
OpenInputSystemExtensionData()	Method	Opens the asset in Inspector.

Input System Extension Data Inspector

Overview
This file defines a **custom inspector** for the InputSystemExtensionData ScriptableObject.
It extends the Unity Editor to add a **button that automatically generates icon mappings** for keyboard and gamepad controls.
The goal is to speed up setup by allowing developers to populate an input configuration asset with a predefined list of key and controller sprites, instead of assigning them manually.

Main Class — InputSystemExtensionDataInspector
This class customizes how the InputSystemExtensionData object appears in the Unity Inspector.
It also adds automation features to help fill out the data quickly.

The class uses Unity’s editor API and inherits from the base Editor type, which allows it to override how the inspector is drawn.

Inspector GUI Section
Purpose
Defines how the inspector window looks and behaves for this ScriptableObject.
It adds new buttons and retains the default fields from the original object.
Method: OnInspectorGUI
This method draws the entire inspector panel.
Here’s what happens step by step:

1. **Update Serialized Data**
   - The inspector fetches the current state of the object to ensure all serialized fields are synchronized.
2. **Create “Get Default Icons” Button**
   - A large button labeled “Get Default Icons” is displayed.
   - When pressed, it triggers a process that fills in all default icon data.
3. **Multi-Object Support**
   - The method loops over all selected ScriptableObjects in the inspector.
   - Each one is processed individually, allowing users to update multiple assets at once.
4. **Generate Default Data**
   - For every selected object, the method calls GenerateKeyCodeList.
   - This function populates gamepad and keyboard icon mappings automatically.
5. **Mark Object as Dirty**
   - After modification, each ScriptableObject is flagged as “dirty”.
   - This ensures Unity recognizes that changes were made and saves them properly.
6. **Add Space and Draw Default Inspector**
   - Adds a small space separator and then draws the default inspector fields (so the user can still view and edit all properties normally).
7. **Apply Modified Properties**
   - Saves all changes made during this process to the serialized object.

Default Sprite and Icon Generation Section
This part contains logic to **automatically populate the asset with icons**.
Method: GenerateKeyCodeList(script)
This is the core generation function that fills out default data for:

- Xbox icons,
- PS4 icons,
- Keyboard keys.

Step-by-Step Process

1. **Clear Old Data**
   - The existing list of keyboard key mappings is wiped clean to avoid duplicates.
2. **Store Default Sprite**
   - The script saves a reference to the default keyboard sprite (used as a fallback when an icon cannot be found).
3. **Generate Xbox Icon Set**
   - The script initializes a structure for all Xbox controller icons.
   - It assigns a sprite to each button (A, B, X, Y, triggers, bumpers, etc.).
   - The function GetSprite is used to search the project for a sprite matching each name (e.g., “XboxOne_A”, “XboxOne_B”).
   - If the sprite isn’t found, the default one is used.
4. **Generate PS4 Icon Set**
   - Similar to the Xbox setup, but uses PlayStation button names (e.g., “PS4_Cross”, “PS4_Triangle”).
   - Again, it uses the same sprite lookup method to locate the icons by name.
5. **Create Keyboard Key List**
   - A new list of keyboard mappings is created.
   - Each entry associates:
     - a **key code** (like “Space”, “Enter”, or “Tab”),
     - a **key name** (which matches the Input System’s control path, such as “/space”),

        - and a **sprite** (looked up by name, or default if not found).
    - This allows the input configuration to display icons for common keyboard keys in UI prompts.
6. **Add Keyboard Mappings to the Asset**
    - Once all default keys are prepared, they're appended to the KeyCodes list of the ScriptableObject.

Result:
After pressing the button in the inspector, the ScriptableObject is now pre-filled with ready-to-use gamepad and keyboard icon mappings.

Sprite Finder Utility Section
Method: GetSprite(spriteName, defaultSprite)
This utility function is responsible for locating sprite assets in the Unity project.
Function Steps

1. **Find All Sprites**
    - It searches the entire Unity project for assets of type "Sprite".
    - The search returns a list of unique identifiers (GUIDs) for each sprite asset found.
2. **Iterate Over Each Sprite File**
    - For every found GUID:
        - It converts the identifier into a file path.
        - It retrieves the asset importer for that path to confirm it's a valid sprite type.
3. **Load Sprite Sub-Assets**
    - Some texture files can contain multiple sprites (for example, a sprite sheet).
    - The method loads all sprite sub-assets contained within the file.
4. **Compare Sprite Names**
    - Each sprite's name is checked against the target name (spriteName).
    - If a sprite with the exact name is found, it is returned immediately.
5. **Debug Logging**
    - When a sprite is found, a log message is printed in the Unity console for confirmation.
6. **Return Default Sprite if None Found**
    - If no sprite with the given name is found anywhere in the project, the function returns the fallback sprite provided as a parameter.

This ensures that the generated mappings never result in null or missing icons — every field is always populated with a valid image, even if only the default one.

Usage Summary
When and How It's Used

- **Editor-Only Tool**
  This script does not run during gameplay. It's part of Unity's editor interface and exists solely to help developers configure input icons faster.
- **Developer Workflow**
    1. A developer selects one or more InputSystemExtensionData assets in the Project window.
    2. The custom inspector shows a "Get Default Icons" button.
    3. Clicking the button scans all project sprites and populates each asset with Xbox, PS4, and keyboard mappings automatically.
    4. The developer can then fine-tune or replace any icons manually.

Summary Table

Element	Type	Purpose
OnInspectorGUI()	Method	Customizes inspector layout and adds the "Get Default Icons" button.
GenerateKeyCodeList(script)	Method	Automatically fills gamepad and keyboard sprite mappings.
GetSprite(name, default)	Method	Searches the project for a sprite by name; returns default if missing.
defaultSprite (from script)	Reference	Fallback image used when no specific icon is found.
KeyCodes (from script)	List	Collection of keyboard key-to-sprite mappings.

xbox, ps4 (from script)	Structs	Store all controller button icons for Xbox and PS4 layouts.
targets	Editor Array	All selected ScriptableObjects being edited at once.
EditorUtility.SetDirty()	Unity Function	Marks an asset as changed so Unity saves it.

Practical Effect
This script transforms the input configuration process from a tedious manual task into a **one-click automation**.
It ensures consistency between multiple assets, eliminates missing icon issues, and helps standardize UI button prompts across an entire project.

Input System Extension Data Auto Creator

Overview
This script ensures that a **configuration asset** named *Input System Extension Data* always exists in the Unity project.
It does this in two ways:

1. **Automatically on Unity Editor startup**, creating the asset if it is missing.
2. **Manually via a menu option**, allowing the developer to create it on demand.

The asset created is a **ScriptableObject** used by the input system extension to store important data such as input bindings, icons, and configuration settings.
This ensures that the project always has a valid configuration asset available, preventing errors caused by missing data.

Section 1 — Asset Creation Menu
Static Class: InputSystemExtensionDataAutoCreator
This class defines the logic that actually creates the configuration asset and exposes it through a **menu item** in the Unity Editor.
Purpose
To let developers manually create or replace the configuration file from the editor menus when needed.

Method: CreateCustomObjectData
This is the central method that performs the asset creation process.
It is registered as a Unity Editor menu command under:
Assets → Create → Input System Extension → Input System Extension Data
Step-by-Step Explanation

1. **Define File Paths**
    - The method sets two strings:
        - One for the **directory** (Assets/Resources).
        - One for the **asset file path** (Assets/Resources/Input System Extension Data.asset).

These define where the ScriptableObject will be saved.
The *Resources* folder is used because Unity allows assets in this folder to be loaded at runtime.

2. **Ensure the Folder Exists**
    - The script checks if the *Resources* folder already exists inside the *Assets* directory.
    - If not, it creates the folder automatically.
    - This prevents errors during asset creation and guarantees a valid location.

3. **Check for Existing Asset**
    - The script searches for an existing file with the same path and name.
    - If one is found, a **confirmation dialog** is displayed.
    - The dialog asks the user whether they want to **replace the existing asset**.
    - If the user chooses “No”, the process is cancelled, leaving the existing asset intact.
    - This step protects against accidental overwriting.

4. **Create the New Asset**
    - If no asset exists (or the user allows replacement), the method:
        - Creates a **new instance** of the InputSystemExtensionData ScriptableObject.

        - Saves this instance as an asset file in the *Resources* folder.
        - Marks it as “dirty,” which means Unity recognizes it as changed and needs to be saved.

5. **Save and Refresh Asset Database**
    - The script explicitly tells Unity to:
        - Save all modified assets.
        - Refresh the project view to ensure the new file appears immediately.

6. **Focus and Select the New Asset**
    - Finally, the editor automatically focuses the Project window and highlights the newly created asset.
    - This gives instant visual feedback, allowing the developer to start editing the asset right away.

Usage
This menu function is used when you want to **manually create** or **regenerate** the *Input System Extension Data* file.
It’s helpful if the file has been deleted, moved, or corrupted.

Section 2 — Automatic Asset Creation on Editor Startup
Static Class: InputSystemExtensionDataStartup
This class ensures that the configuration asset is **automatically created** whenever the Unity Editor launches the project.
It is decorated with a special Unity attribute that triggers the static constructor as soon as the editor finishes loading scripts.

Static Constructor
This section executes once when the Unity Editor starts or reloads scripts.
How It Works Step-by-Step
1. **Register a Delayed Call**
    - When the editor finishes loading, a small callback function is added to Unity’s **delay call queue**.
    - This ensures that the creation check runs only after the editor is fully initialized and all assets are available.
2. **Define Asset Path**
    - It defines the expected location of the configuration file: Assets/Resources/Input System Extension Data.asset.
3. **Check if File Exists**
    - It checks the project’s file system directly to see whether this file exists.
4. **Create Asset If Missing**
    - If the file does **not** exist, the script calls the CreateCustomObjectData method from the first class.
    - This triggers the same asset creation logic described earlier.

Purpose
The goal of this system is to ensure **project stability**.
Developers do not have to remember to create the asset manually — if it is missing, it will be automatically generated on editor startup.
This guarantees that the input system extension always has the data it requires to operate.

Practical Workflow
1. **When starting the Unity Editor:**
    - The script checks automatically whether the *Input System Extension Data* file exists.
    - If it doesn’t, it creates one instantly in the correct location.
    - This ensures the input configuration is always available.
2. **When using the menu manually:**
    - Developers can recreate or replace the configuration file anytime.
    - This is especially useful if the existing file becomes corrupted or needs resetting.

3. **When working on multiple machines or repositories:**
    - Even if someone forgets to commit the asset to version control, the project will still work, since the file is automatically regenerated on startup.

Summary Table

Element	Type	Purpose
InputSystemExtensionDataAutoCreator	Static Class	Provides menu-based manual creation of the configuration asset.
CreateCustomObjectData()	Method	Handles creation, saving, and selection of the asset.
path	String	Directory path for the Resources folder (Assets/Resources).
assetPath	String	Full file path to the ScriptableObject asset.
AssetDatabase	Unity API	Manages creation, saving, and refreshing of assets.
EditorUtility.DisplayDialog()	Unity API	Displays a dialog box asking whether to replace an existing file.
Selection.activeObject	Unity API	Highlights and selects the newly created asset in the Project window.
InputSystemExtensionDataStartup	Static Class	Runs automatically when the editor starts, ensuring asset existence.
InitializeOnLoad	Unity Attribute	Makes the class execute as soon as scripts are loaded.
delayCall	Editor Event	Schedules the creation check after the editor has initialized.
File.Exists()	System Function	Checks if the asset file already exists on disk.

Result and Benefits

- **Automatic Initialization:**
  The input configuration file is always available without manual setup.
- **Safety Checks:**
  Prevents accidental overwriting of existing assets through confirmation prompts.
- **Convenient Workflow:**
  Provides both automatic creation (on startup) and manual creation (via menu).
- **Consistency Across Projects:**
  Ensures every Unity project using this system follows the same directory and asset structure.

On Input System Event

Overview
This script is a **generic static dispatcher** for the Unity Input System. Its purpose is to **listen to input actions** of any type (like boolean, float, or vectors) and **trigger callbacks** for different input states: pressed, hold, and released.
It is designed to allow **multiple scripts** to share the same input action safely without interfering with each other, because each callback is tied to a specific **owner**, usually a MonoBehaviour.

Main Structure
The system has two main parts:
1. **The static dispatcher** (OnInputSystemEvent<T>) which monitors input actions and calls callbacks.
2. **The fluent configuration wrapper** (OnInputSystemEventConfig<T>) which allows easy registration of callbacks for a specific action and owner.

Variables and Data Structures
- **States dictionary**: Stores the runtime state for each registered input action. The key is the input action, and the value is a State object containing all callbacks and last value.
- **State object**: Represents the runtime state of a single input action. Contains:

  - **Action**: The input action being tracked.
  - **Last value**: The last read input value.
  - **OnHold / OnPressed / OnReleased**: Lists of tuples containing the owner, the callback, and an optional predicate to check if that owner is currently enabled.
  - **OwnerHeld dictionary**: Tracks whether each owner was previously holding the input, to detect releases.
- **Owner**: Any object (usually a MonoBehaviour) that registers callbacks, so multiple scripts can subscribe independently.
- **Enabled predicate**: Optional function provided by the owner that determines whether the owner's callbacks should currently run.

How the Dispatcher Works
Update State

1. For each action, the system reads the **current input value**.
2. Determines whether the action is currently being held (not zero for float/vector, true for bool).
3. Collects all **owners** that have registered callbacks for that action.
4. For each owner:
   - Checks if the owner is enabled (using the predicate).
   - If the owner was previously holding the input but is now disabled, it triggers the **released** callback.
   - If enabled and currently held:
     - If the owner was not previously held, trigger **pressed**.
     - Always trigger **hold**.
   - If currently not held but previously held, trigger **released**.
5. Stores the current value as the **last value** for the next update.

Determining Non-Zero Input

- A helper function checks if the input value is considered "active":
  - For floats: any value above a small threshold.
  - For vectors: if magnitude is above a threshold.
  - For booleans: if true.
  - For quaternions: if not identity.
  - Default: any value not equal to its type default.

Registering Callbacks

- **RegisterHold**: Adds a callback to be triggered while the input is held.
- **RegisterPressed**: Adds a callback for when the input transitions from not held to held.
- **RegisterReleased**: Adds a callback for when the input transitions from held to not held.
- **UnregisterAll**: Removes all callbacks for a specific owner and clears its held state.
- **Clear**: Removes all callbacks and state for a specific action.

State Management

- **GetOrCreateState**: Retrieves an existing state for an action or creates a new one.
- Hooks into Unity Input System's **after-update event** to call the state's update method every frame.

Fluent Configuration Wrapper
The configuration wrapper allows a **chained syntax** to register callbacks more conveniently:

- Bind a specific action and owner.
- Optionally provide an **enabled predicate**.
- Methods:
  - **OnHold(callback)**: Registers a hold callback.
  - **OnPressed(callback)**: Registers a pressed callback.
  - **OnReleased(callback)**: Registers a released callback.
  - **Dispose()**: Unregisters all callbacks for the owner and action.

This allows statements like:
WithAction(action, owner).OnPressed(doSomething).OnHold(doOtherThing);

Usage Workflow

1. Define or reference an input action in Unity (boolean, float, vector, etc.).
2. Call WithAction on the dispatcher, providing the action and an owner.
3. Use OnPressed, OnHold, and OnReleased to register callbacks.
4. Optionally provide an enabled predicate to enable or disable callbacks dynamically.
5. The dispatcher automatically updates every frame and triggers callbacks per owner.
6. Call Dispose or UnregisterAll to remove callbacks when no longer needed (e.g., when a script is disabled or destroyed).

Key Advantages

- **Multiple owners per action**: Different scripts can respond independently.
- **Safe release detection**: Automatically handles cases where an owner becomes disabled while holding an input.
- **Generic for any input type**: Works for floats, booleans, vectors, and more.
- **Fluent API**: Easy to read and maintain code for input callbacks.
- **Automatic update loop**: No need to manually call update for each action.

This script is ideal for projects that need **centralized input management** while allowing multiple scripts to safely react to the same input actions without conflict.

Example	Description	Code
Basic pressed callback	Trigger a method when an input is pressed.	OnInputSystemEvent<bool>.WithAction(jumpAction, this).OnPressed(_ => Jump());
Hold callback with enabled predicate	Trigger continuously while held, only if a condition is true.	OnInputSystemEvent<Vector2>.WithAction(moveAction, this, () => isPlayerActive).OnHold(dir => Move(dir));
Released callback	Trigger a method when input is released.	OnInputSystemEvent<bool>.WithAction(shootAction, this).OnReleased(() => StopShooting());
Chained callbacks	Combine pressed, hold, and released for one action.	OnInputSystemEvent<float>.WithAction(zoomAction, this).OnPressed(v => StartZoom(v)).OnHold(v => Zoom(v)).OnReleased(() => EndZoom());
Dispose / unregister all	Remove all callbacks for an owner and action.	var config = OnInputSystemEvent<bool>.WithAction(jumpAction, this).OnPressed(_ => Jump()); config.Dispose();
Multiple owners	Different scripts responding independently to the same action.	OnInputSystemEvent<Vector2>.WithAction(moveAction, player1).OnHold(dir => player1.Move(dir)); OnInputSystemEvent<Vector2>.WithAction(moveAction, player2).OnHold(dir => player2.Move(dir));

Input System Utility

Overview

The script defines a *static utility class* inside the *Input System Extension* namespace.
Its purpose is to detect whether the player has interacted with **any type of input device** during the current frame using Unity's **new Input System**.
It supports:

- **Keyboard**
- **Mouse**
- **Gamepads**
- **Touchscreens** (mobile devices)

Being *static* means:

- You don't need to create an instance of the class.
- You call the methods directly, e.g. InputSystemUtility.IsAnyInputPressedThisFrame().

Namespace
InputSystemExtension
A grouping container that organizes this utility so it doesn't conflict with other classes.

Class
InputSystemUtility
A fully static class.
Its role is to provide simple, easy-to-read helper methods for checking inputs.
It does not store internal data; it only performs checks against Unity's input devices.

Methods (Explained in Detail)
1. IsAnyInputPressedThisFrame()
Purpose
Check whether *any* input from keyboard, mouse, gamepad, or touchscreen was pressed in the current frame.
How it works
It simply calls four other methods:

- Check keyboard
- Check mouse
- Check gamepad
- Check touch

If **any** of those return true, then it returns true.
Use Case
Useful for:

- Determining if the player interacted with the game at all.
- Auto-hiding UI until the player presses something.
- Detecting "wake up" actions.

2. IsKeyboardPressed()
Purpose
Detect if **any key** on the keyboard was pressed this frame.
Internal Logic

1. Check if a keyboard exists.
2. Ask the Input System if *any key* reported a "pressed this frame" event.

What counts as a press?
Any physical keyboard key, including:

- letters
- numbers
- arrows
- modifiers (Shift, Ctrl)
- special keys

3. IsMousePressed()
Purpose
Detect if **any mouse button** was pressed this frame.
Internal Logic

1. Check if a mouse device exists.
2. Check the three primary mouse buttons:
    - Left
    - Right
    - Middle

If any of them report "pressed this frame", the method returns true.
Notes
This does **not** check scroll wheel or movement, only button clicks.

4. IsGamepadPressed()
Purpose
Detect if **any button** on **any connected gamepad** was pressed this frame.
Internal Logic

1. Get the list of all connected gamepads.
2. For each gamepad:
    - Loop through all controls (buttons, triggers, etc.)
    - Check if the control is a button.

  - If the button reports “pressed this frame”, return true.

What counts as a gamepad button?
Any of the following, on any gamepad model:

- A/B/X/Y
- Cross/Circle/Square/Triangle
- Bumpers
- Triggers (treated as buttons if they act digitally)
- D-Pad directions
- Stick clicks

5. IsTouchPressed()
Purpose
Detect if a touchscreen registered a “touch pressed” event this frame.
Internal Logic

1. Check if a touchscreen device exists.
2. Check the *primary touch*’s "press" control.
3. If it was pressed this frame, return true.

Who uses this?
Mobile devices such as Android phones and tablets.

Usage Guide
You would typically use this utility in places where you need to detect general player input, such as:
1. Waking up UI
If the game hides its HUD until the player interacts:
```
if (InputSystemUtility.IsAnyInputPressedThisFrame())
   ShowUI();
```
2. Pausing
To unpause when *any* button is pressed.
3. Idle Detection
To reset an inactivity timer.
4. Menu Navigation
To detect initial input for focusing controls.
5. Multi-platform input support
The tool abstracts platform differences, so the same logic works for:

- PC (keyboard, mouse)
- Consoles (gamepad)
- Mobile (touch)

Summary
This utility is a clean, simple, device-agnostic input detector that:

- Checks multiple device types.
- Is easy to call from anywhere.
- Uses Unity’s new Input System.
- Does not require prior knowledge of which device the player is using.

It’s ideal for general input detection in menu systems, wake-up screens, or games supporting multiple input devices.

Input System Extension Helper

Purpose
The script provides a convenient way to access a specific data asset used by the input system extension. This asset is stored as a ScriptableObject in Unity’s Resources folder, and the script ensures it is loaded efficiently and only once.

Variables

1. **extensionData**
   - Type: ScriptableObject holding input system extension data.
   - Purpose: Acts as a cache so that the asset is not loaded from disk repeatedly. Once loaded, it keeps a reference in memory for faster access.

Methods

1. **GetInputSystemExtensionData**
    - Purpose: Retrieves the InputSystemExtensionData asset, loading it from the Resources folder if it hasn't been loaded yet.
    - Behavior:
        - Checks if the cached variable (extensionData) already contains the asset. If so, it returns it immediately.
        - If not, it attempts to load the asset from the Resources folder using a specific name.
        - If loading fails (the asset is missing or misnamed), it logs an error message to help the developer diagnose the problem.
        - Returns the cached or newly loaded asset so that it can be reused.

Usage

- This helper is intended to be used anywhere in the project where scripts need access to the input system extension data.
- Instead of manually loading the asset each time, a script can call the helper method to get a ready-to-use reference.
- It simplifies code and improves performance by ensuring the asset is loaded only once.

Example usage in practice:

- A script wants to read default input bindings or configurations. It calls the helper, receives the data asset, and reads the required information.
- Multiple scripts can call this method safely without worrying about repeatedly loading the same asset.

In short, this script is a **singleton-like accessor** for a shared data asset. It caches the asset in memory for performance and provides a simple, centralized way to retrieve input system configuration data.

Input Binding Saver

Purpose

This script manages saving and loading input binding overrides for the game. It uses Unity's PlayerPrefs system to persist changes to input controls, allowing players' custom input configurations to be stored and restored automatically.

Variables

1. **No public variables** are defined in the script.
2. The script primarily interacts with **InputSystemExtensionData**, which contains:
    - The default input action asset (the set of all input controls).
    - The PlayerPrefs key used to identify saved binding data.

Methods

Default Save and Load

1. **SaveDefaultBindings**
    - Retrieves the InputSystemExtensionData instance.
    - Ensures the asset exists.
    - Saves the current input bindings into PlayerPrefs using the key specified in the data.
    - Purpose: Provides a convenient way to save all default bindings at once.
2. **LoadDefaultBindings**
    - Automatically executes when the game starts.
    - Retrieves the InputSystemExtensionData instance.
    - Ensures the asset exists.
    - Loads any saved binding overrides from PlayerPrefs and applies them to the input actions.
    - Purpose: Ensures that players' custom input settings are applied automatically at startup.

Save and Load Overrides

3. **SaveBindings**

- Takes a specific input action asset and a PlayerPrefs key.
- Converts the current binding overrides into a JSON string.
- Stores the JSON string in PlayerPrefs under the given key.
- Logs a confirmation message.
- Purpose: Allows saving individual input action assets to PlayerPrefs.

4. **LoadBindings**
   - Takes a specific input action asset and a PlayerPrefs key.
   - Checks if a saved binding exists for that key.
   - Retrieves the JSON string and applies the saved overrides to the asset.
   - Logs a confirmation message.
   - Purpose: Allows loading and applying saved bindings to an input action asset.

Usage

- **Automatic usage at game startup:**
  - The script automatically loads any saved input binding overrides when the game begins, ensuring player preferences are applied without extra code.
- **Manual saving:**
  - Developers can call the default save method to persist all bindings from the InputSystemExtensionData asset.
  - They can also save or load individual input action assets manually using the dedicated methods with a custom PlayerPrefs key.
- **Persistence:**
  - Players' custom controls are stored as JSON in PlayerPrefs, which survives game sessions.
  - This allows consistent input configurations between sessions without requiring external files.

Summary

The script is a **binding persistence manager**. Its main goal is to store input overrides to PlayerPrefs and reload them automatically at startup. It works with a central data asset (InputSystemExtensionData) but also allows flexibility for custom saving and loading of individual input assets. This ensures that player input preferences are maintained across sessions without extra manual handling.

Rebind Control Manager and Rebind Control Manager TMP

Purpose

This component is designed to **manage interactive input rebinding** in Unity. It allows users to change control bindings at runtime using UI elements, supports **composite bindings** (like WASD for movement), shows icons representing controls, prevents duplicate bindings, and handles cancel input. It's primarily a **UI and input management tool** for customizing controls.

Key Variables and Fields

Input Settings

1. **cancelRebindActionReference**
   - Input action that can cancel the rebinding process if performed during rebind.
2. **inputActionReference**
   - The input action that will be rebinding target.
3. **targetBindingId**
   - Unique identifier for the specific binding in the action that will be modified.
4. **displayOptions**
   - Options that control how the binding is displayed in the UI (for example, showing the device layout, path, or control name).

UI Elements

5. **actionLabel**
   - Text element that shows the name of the action being bound.
6. **autoLabelFromAction**
   - Boolean flag to automatically update the label with the action's name.
7. **startRebindButton**
   - Button that starts the interactive rebinding process.

8. **bindingDisplayText**
    - Text element that shows the current binding's display string.
9. **actionBindingIcon**
    - Icon representing the binding visually.
10. **rebindOverlayUI**
    - UI overlay shown during the rebinding process to block other inputs and indicate a rebind is in progress.
11. **rebindPromptText**
    - Text element guiding the user during the rebind (e.g., "").
12. **actionRebindIcon**
    - Icon representing the current input being pressed during the rebind.
13. **resetButton**
    - Button that resets the binding to its default value.

Events

14. **onRebindStart**
    - Triggered when a rebinding process starts.
15. **onRebindStop**
    - Triggered when a rebinding process ends, either completed or canceled.
16. **onUpdateBindingUI**
    - Triggered whenever the binding UI should refresh after a change.

Private/Internal Variables

17. **currentRebind**
    - Holds the ongoing interactive rebinding operation.
18. **allRebindManagers**
    - Static list containing all active rebind managers to facilitate global updates.
19. **currentEventSystem**
    - Cached reference to Unity's EventSystem to safely manage UI focus during rebinding.

Properties

- Allow external access to the input action, display options, binding text, icons, and prompt text.
- Automatically refresh the UI when changed.

Unity Callbacks

1. **Awake**
    - Checks for missing references and logs warnings.
    - Ensures essential UI and input elements are assigned.
2. **Start**
    - Hides the overlay UI initially.
    - Registers click events for the rebind button and reset button.
    - Caches the current EventSystem.
3. **OnEnable / OnDisable**
    - Adds/removes the manager to/from a global list of managers.
    - Subscribes/unsubscribes to input system change events for automatic UI updates.
4. **OnValidate**
    - Used in the Unity Editor to keep the UI in sync when references change.

Core Methods

1. **TryResolveBinding**
    - Finds the input action and the index of the binding to rebind using the unique binding ID.
    - Returns false if the binding cannot be found.
2. **RefreshBindingDisplay**
    - Updates the UI to show the current binding's display string.
    - Checks if the binding has been modified from its default and enables or disables the reset button accordingly.
    - Invokes an event to notify external listeners.
3. **ResetToDefault**
    - Resets the selected binding (and any composite parts) to its original default.
    - Updates the UI and persists changes.
4. **StartInteractiveRebind**

    - Starts a rebinding operation for the selected binding.
    - If the binding is a composite, it starts from the first part.
5. **ExecuteInteractiveRebind**
    - Core logic for performing the interactive rebind:
        - Disables the action map to prevent other inputs.
        - Hides the EventSystem to block UI interaction.
        - Displays overlay and prompts.
        - Monitors cancel input.
        - Handles completion, cancellation, duplicate input prevention, and composite continuation.
        - Updates the UI and persists changes.
6. **CheckDuplicateBindings**
    - Ensures the newly selected input isn't already assigned to another action.
    - Cancels the rebind if a duplicate is detected.
7. **GenerateCustomPrompt**
    - Generates the text shown to the player during rebinding.
    - Handles composite bindings by highlighting the part currently being modified.
8. **MonitorCancelInput / OnCancelInputPerformed**
    - Listens for cancel input during a rebind and cancels the operation when triggered.
9. **HandleActionChange**
    - Global handler for input system changes.
    - Updates the UI of any manager affected by action, map, or asset changes.
10. **UpdateActionLabel**
    - Updates the action name in the UI label if auto-update is enabled.

Events

- **InteractiveRebindEvent**
    - Triggered when a rebind starts or stops, providing the manager and the operation.
- **UpdateBindingUIEvent**
    - Triggered when the UI should refresh, sending the binding's display string, device layout, and control path.

Usage

1. Attach the component to a UI element representing a binding.
2. Assign references for:
    - Input action to rebind.
    - Cancel action.
    - Binding ID.
    - UI elements (text, buttons, icons, overlay).
3. Enable auto-label if you want the action name displayed automatically.
4. Click the **start rebind button** to begin modifying the binding.
5. Press a key or button to rebind.
6. Press cancel to stop the operation.
7. Reset button returns the binding to default.
8. The UI updates automatically, including for composite bindings, duplicate prevention, and icon representation.

Summary

This script provides a **complete, user-friendly system** for input rebinding in Unity. It handles all aspects of interactive rebinding, including UI updates, cancel actions, composite controls, duplicate prevention, and persistence of default bindings. It's ideal for games that want players to customize controls at runtime with a polished interface.

Gamepad Icon Rebind Handler and Gamepad Icon Rebind Handler TMP

Purpose

This component is designed to **replace text-based input displays with gamepad icons**. It works together with RebindControlManager components to automatically swap a binding's text with the correct icon depending on the controller being used (like PlayStation or Xbox). It helps make input prompts in the game more intuitive and visually clear for players using controllers.

Key Variables
Inspector Fields

1. **rebindControlGroup**
    - A reference to a parent object containing one or more input rebinding managers.
    - This is used to locate all rebinding elements that may need their display updated with icons.

Private Fields

2. **extensionData**
    - Holds a reference to a data object that contains mappings from input controls to their corresponding sprites (icons).
    - For example, it includes icons for PS4 buttons or Xbox buttons, linked to specific controller inputs.

Unity Lifecycle Methods

1. **Start**
    - Initializes the handler when the scene begins.
    - Loads the icon mappings from the data object.
    - Finds all rebinding managers inside the specified parent group.
    - Subscribes to each manager's **update event**, so that whenever a binding changes, the handler can refresh the visual display.
    - Immediately refreshes all bindings on start to ensure the correct icon or text is shown.

Event Handlers

1. **OnBindingUIUpdate**
    - This method is called whenever a RebindControlManager updates its binding UI.
    - It receives the manager that triggered the update, the text display for the binding, the controller layout, and the specific control path.

The method works in the following steps:

3. **Validation**
    - Checks if the controller layout, control path, and icon data are valid.
    - Exits early if any required data is missing.
4. **Determine the Correct Icon**
    - Checks the controller type:
        - If it's a PlayStation controller, it looks up the PS4 icon mapping.
        - If it's a generic or Xbox-style controller, it looks up the Xbox icon mapping.
    - If no icon is mapped, it will fall back to displaying text.
5. **Retrieve UI References from the Manager**
    - Gets the text element showing the binding.
    - Gets the image element showing the binding icon.
    - Gets the rebind prompt text and icon elements.
    - Logs an error if any of these references are missing.
6. **Update UI Display**
    - If an icon was found:
        - Shows the icon images.
        - Hides the text elements.
    - If no icon was found:
        - Shows the text elements.
        - Hides the icon images.

Usage

1. **Attach the Component**
    - Place this component on a GameObject in the scene.
2. **Assign the Rebind Control Group**
    - Specify the parent object that contains all RebindControlManager components for which you want to swap text with icons.
3. **Provide Icon Data**

- Ensure the InputSystemExtensionData object exists and contains the appropriate controller icons.
    - This data is used to automatically select the correct sprite for each control.
4. **Automatic Updates**
    - Whenever a binding changes through the rebinding system, this component automatically:
        - Checks the controller type.
        - Retrieves the correct icon.
        - Updates the UI to show the icon instead of text (or vice versa if no icon exists).
5. **Result**
    - Players always see intuitive input prompts: gamepad buttons are represented by their icons instead of text labels, improving the user experience.

Summary
This script acts as a **visual bridge between the rebinding system and the UI**, dynamically swapping text for controller icons whenever bindings are updated. It handles different controller types (PlayStation, Xbox, etc.), ensures the correct icon is displayed for each input, and falls back to text when an icon isn't available. This is ideal for games with controller support, providing a clean and visually clear input display for players.

Reset All Rebind Control

Purpose
This component is a **UI utility** designed to reset all input bindings managed by rebinding managers within a specified group. It ensures that players can restore every control to its default configuration by pressing a single button. This is useful for games that allow input remapping, providing an easy way to undo all changes.

Key Variables
Inspector Fields
1. **resetAllButton**
    - A reference to a user interface button.
    - When clicked, it triggers the reset process for all rebindable controls within the group.
2. **rebindControlGroup**
    - A reference to a parent object containing all the rebinding components.
    - This parent is searched to find every rebinding manager that should be reset.

Private Fields
3. **rebindControls**
    - A list that stores all the standard rebinding managers found in the group.
    - These managers handle input rebinding for normal text-based UI elements.
4. **rebindControlsTMP**
    - A list that stores all the TMP-based rebinding managers found in the group.
    - These are for UI elements using TextMeshPro instead of standard text.

Properties
1. **ResetAllButton**
    - Provides access to the reset button so it can be read or assigned from other scripts if necessary.
2. **RebindControlGroup**
    - Provides access to the parent group object containing the rebinding managers.

Unity Lifecycle Methods
1. **Start**
    - This method runs when the scene starts.
    - It first ensures that the reset button is assigned. If it is missing, a warning is logged.
    - If the button is present, it attaches a listener so that clicking the button triggers the reset process.
    - It then searches the parent group for all standard rebinding managers and TMP-based rebinding managers, storing them in their respective lists.

Core Method

1. **ResetAll**
   - This method performs the actual reset operation.
   - It iterates over each standard rebinding manager in the list and calls its reset function to restore its default bindings.
   - It then does the same for all TMP-based managers.
   - The result is that all input bindings in the specified group are returned to their default values in a single operation.

Usage

1. **Attach the Component**
   - Place this component on any GameObject in the scene, typically in the same panel as the UI button.
2. **Assign the Reset Button**
   - Drag the UI button that should trigger the reset into the resetAllButton field.
3. **Assign the Rebind Group**
   - Drag the parent object that contains all the rebinding managers into the rebindControlGroup field.
4. **Result**
   - When the player clicks the assigned button, all rebinding managers within the group are reset to their default input bindings.
   - This includes both standard text UI managers and TextMeshPro-based managers.

Summary

This script provides a **centralized, user-friendly way to restore all custom input bindings to defaults**. It works automatically with a button click, finds all relevant managers in the specified group, and resets them efficiently. It’s a convenience feature for games with customizable input controls, ensuring players can easily undo all changes.

Input Display Manager

Overview

This script manages **input display UI elements** in a Unity scene. It automatically updates **input icons** to match the current input control type (keyboard or gamepad) and responds to input events.

- Supports both **single inputs** (like a jump button) and **directional inputs** (like movement arrows or a joystick).
- Handles **color transitions**, fading, and visibility of icons dynamically.
- Automatically detects connected devices or allows manual control type selection.

Main Class — InputDisplayManager

The class is attached to a Unity GameObject and provides runtime behavior for updating UI based on user input.
It contains **enums, serializable classes, inspector fields, private fields, properties, Unity event methods, input detection, icon updates, view control, color helpers, and cleanup utilities**.

Enums

ControlType

- Represents the type of input device: **Keyboard** or **Gamepad**.
- Used throughout the script to decide which sprites or icons to show.

Serializable Classes

InputViewerData

- Represents a single input control (like a jump button).
- **Variables**:
  - nameTag: Unique identifier for this viewer.
  - enable: Enables or disables the viewer.
  - inputActionReference: Reference to the input action (button, key, etc.).
  - keyboardId and gamepadId: IDs used to select the correct icon sprite.
  - inputIcon: UI Image component that displays the icon.

- inputEvent: Stores the event configuration to handle input changes.

InputMultipleViewsData

- Represents directional input controls (like D-Pad or arrow keys).
- **Variables**:
  - nameTag, enable, inputActionReference, keyboardId, gamepadId similar to single input.
  - inputIconUp, inputIconDown, inputIconLeft, inputIconRight: Images for each direction.
  - inputEvent: Event configuration for 2D inputs (Vector2).

Inspector Fields

- automatic: If true, automatically detects input type at runtime.
- controlType: Manually selected input type if automatic detection is off.
- activatedColor and disabledColor: Colors for active or inactive states.
- transitionTime: Duration of color transition when input state changes.
- hideTransitionTime: Duration of fade-out when hiding icons.
- inputViewerData: List of single input viewers.
- inputMultipleViewsData: List of directional input viewers.

Private Fields

- extensionData: Stores icon sprites and input mappings.
- colorTransitions: Tracks active color transition coroutines for each icon.

Public Properties

- SetAutomatic: Get or set automatic detection of input type.
- SetControlType: Get or set the current input control type. Changing it updates all icons.
- InputViewerDataEditor and InputMultipleViewsDataEditor: Provide access to modify input data at runtime or in the editor.

Unity Event Methods

Awake

- Validates that all required input data is assigned and logs errors if not.

OnEnable

- If automatic detection is enabled, detects control type.
- Subscribes to device change events.
- Initializes input viewers and directional viewers.
- Updates all icons to match the current control type.

OnDisable / OnDestroy

- Performs cleanup: unsubscribes from events and stops coroutines.

Validation

ValidateData

- Checks every single and multiple input viewer:
  - Ensures nameTag is assigned.
  - Ensures inputActionReference is assigned.
  - Ensures UI Image components are assigned.
- Logs errors for missing assignments to help catch setup issues early.

Initialization Helpers

InitializeInputViewerData

- For each single input viewer:
  - Disposes existing input events to avoid duplicates.
  - Creates new event handlers for pressed and released actions.
  - Sets the initial icon color depending on whether the viewer is enabled.

InitializeInputMultipleViewsData

- For each directional input viewer:
  - Disposes existing events.
  - Creates event handlers for press, hold, and release states.
  - Updates icons based on input vector and control type.
  - Sets initial colors or hides icons if disabled.

Input Detection & Control Type
DetectControlType
- Checks if any gamepads are connected.
- Defaults to **Gamepad** if found; otherwise, defaults to **Keyboard**.

OnDeviceChange
- Triggered when an input device is connected or disconnected.
- If automatic detection is on, updates control type and refreshes icons.

Icon Updates
UpdateIcons
- Updates all icons for both single and directional inputs.
- Chooses sprites according to current control type and binding IDs.

UpdateInputViewerIcons
- For each single input viewer, fetches the sprite corresponding to keyboard or gamepad binding.

UpdateInputMultipleViewsIcons
- For directional inputs:
  - Keyboard: assigns each directional icon its corresponding key.
  - Gamepad: highlights only the primary direction icon (usually "up").

GetSpriteForBinding
- Determines the correct sprite for a binding ID.
- Handles **keyboard/mouse bindings** and **gamepad bindings** separately.
- For gamepads, detects PS4 or Xbox and retrieves the correct sprite.
- Falls back to default sprite if no match is found.

SetIconsActive
- Activates or deactivates directional icons based on booleans.

View Control
EnableAndDisableViewer
- Enables or disables a viewer by nameTag.
- Updates icon visibility and triggers color transitions.

HandleIconEnableDisable
- Helper method that starts a fade-in or fade-out transition and updates the GameObject active state.

FadeOutAndDeactivate
- Coroutine to gradually fade out an icon and deactivate it.

Icon Color Helpers
StartColorTransition
- Initiates a smooth color change for an icon.
- Cancels any existing transition to avoid conflicts.

ColorTransitionCoroutine
- Handles gradual color change over time.

SetInitialColor / SetTransparent
- Prepares icon to be visible or hides it smoothly.

SetDirectionActive
- Activates or deactivates a directional icon with a color transition.

Cleanup & Utility
OnClean
- Centralized cleanup called on disable or destroy.
- Unsubscribes from device events, unbinds input events, stops color transitions.

UnbindAllEvents
- Disposes all input events to prevent memory leaks or unintended callbacks.

StopAllColorTransitions
- Stops all ongoing color transitions and clears tracking.

Usage
1. Attach this script to a UI GameObject.
2. Configure **single input viewers** and **directional viewers** in the inspector.

3. Choose **automatic detection** or set a control type manually.
4. At runtime:
   - Icons update automatically based on input events.
   - Pressing a key or button triggers color changes on icons.
   - Directional inputs highlight the corresponding arrows.
   - Switching between keyboard and gamepad updates sprites accordingly.
5. Developers can enable/disable specific viewers dynamically via EnableAndDisableViewer.

This system ensures **dynamic, responsive, and visually consistent input icons** for both keyboard and gamepad, with smooth transitions and automatic device detection.

Input Display Manager Inspector

Overview

This script is a **custom editor** for a Unity component called InputDisplayManager. Its main purpose is to add a **button in the Unity inspector** that allows you to automatically create or configure input icon images for both single and directional inputs.
It simplifies setup by generating UI elements based on the **assigned input actions**, ensuring consistent naming and hierarchy for images.

Main Class — InputDisplayManagerInspector

This class is attached as a **custom editor** in Unity. It only runs inside the editor, not at runtime in the game. It extends the default inspector GUI with custom functionality.

- Provides a **button** called “Create Automatic Presets”.
- Automatically generates or renames UI Image elements for input icons.
- Handles **undo operations** and ensures all changes are recorded in Unity's serialization system.

Inspector GUI
OnInspectorGUI

- This method **overrides the default inspector layout**.
- **Steps it performs**:
  1. Updates the serialized fields of the component to reflect any changes.
  2. Draws a button labeled **“Create Automatic Presets”**.
  3. When the button is pressed, it iterates over all selected objects and calls the preset creation logic.
  4. Registers undo operations for both the main component and any created/renamed GameObjects.
  5. Marks the object as dirty so Unity knows to save changes.
  6. Draws the rest of the default inspector fields normally.

Preset Creation Logic
CreateAutomaticPresets

- Automatically creates or updates UI icon images for the component.
- **Input:** The InputDisplayManager component.
- **Process:**

1. **Single input viewers** (non-directional)
   - Iterates over each input viewer in the component.
   - Skips entries with no assigned input action.
   - Updates the viewer’s nameTag to match the input action name.
   - Checks if a UI Image already exists:
     - If it exists, renames it according to a standard format.
     - If it doesn’t exist, tries to find an image with the expected name in the hierarchy.

        - If not found, creates a new image with a default size and assigns it to the viewer.

2. **Directional input viewers** (e.g., D-Pad or movement arrows)
    - Iterates over each directional input viewer.
    - Skips entries with no assigned input action.
    - Updates the nameTag to match the input action name.
    - Calls a helper function to **create or rename four directional images**: Up, Down, Left, Right.
    - Ensures each directional image has a valid UI Image component.

CreateOrRenameDirectionalImage

- Handles **creating or renaming a directional image** under a parent transform.
- **Inputs:**
    - Parent object (usually the component's GameObject).
    - Expected name for the image.
    - Existing Image reference (can be null).
- **Logic:**
    - If the image exists, rename it.
    - If not, search for an existing object in the hierarchy.
    - If not found, create a new GameObject with a RectTransform, CanvasRenderer, and Image.
    - Sets default size for new images.
    - Returns the Image component.

Variables

- **serializedObject**: Unity's system to edit and track serialized fields for components.
- **targets**: All selected objects in the Unity editor that share this component.
- **script**: Reference to the InputDisplayManager being modified.
- **parentTransform**: The GameObject under which images are created (usually the component's transform).
- **data**: A single input or directional input viewer struct from the component.
- **imageComponent**: The UI Image being created or updated.
- **currentName / expectedName**: Standardized names for image GameObjects to keep hierarchy organized.

Usage

1. Attach InputDisplayManager to a GameObject.
2. Add this custom editor script to the project.
3. Select the GameObject in the Unity inspector.
4. Click **"Create Automatic Presets"**.
5. The script will automatically:
    - Create or rename single input icon images based on input actions.
    - Create or rename directional images (Up, Down, Left, Right) for multi-directional inputs.
    - Ensure all images are placed under the component's GameObject.
6. Images are automatically assigned to the corresponding input viewer fields in the component.
7. Any changes are recorded for **undo** and **saved** in the scene.

Summary

This editor script automates the tedious process of creating and organizing UI images for input icons. It ensures that:

- Each input action has a corresponding image.
- Directional inputs have separate images for each direction.

- Objects are named consistently and correctly parented.
- Changes are undoable and serialized properly.

It is **strictly an editor utility** to improve workflow and does not affect runtime behavior of the game.

Input Debug Logger

Purpose

The script is a **debugging tool** for Unity projects using both the legacy Input system and the new Input System. It logs pressed keys and input paths side by side, showing how each key or button maps to both systems. Additionally, it checks whether each key or path exists in a preconfigured list from InputSystemExtensionData and logs extra info, such as index in the list and whether a sprite is associated with the key.
This is useful for developers who want to **verify input mappings** or debug input detection issues.

Variables

1. **ManualMap**
   - A dictionary linking legacy KeyCodes to new Input System paths.
   - Covers keys that do not have direct name correspondence between the two systems (e.g., LeftControl maps to <Keyboard>/leftCtrl).
   - Includes extra keys like mouse buttons.
2. **data**
   - Reference to InputSystemExtensionData, which contains a list of key configurations including KeyCodes, names, and sprites.
   - Used to check if a key/path exists in the project's input database and to log extra information.

Unity Methods

1. **Start**
   - Runs when the component is first enabled.
   - Loads the InputSystemExtensionData instance.
   - Logs a warning if the data cannot be found but allows the script to continue.
2. **Update**
   - Runs every frame.
   - Performs three main steps:
     1. Collect pressed paths from the new Input System.
     2. Check all legacy KeyCodes for presses and log information comparing legacy and new paths.
     3. Log keys detected only by the new Input System.

Input Collection

- **CollectPressedNewInputPaths**
  - Returns a list of all currently pressed input paths from the new Input System.
  - Checks three types of devices: keyboard, mouse, and gamepad.
  - Converts pressed controls into standard path strings like <Keyboard>/a or <Mouse>/leftButton.

Input Logging

1. **ProcessLegacyInputKeys**
   - Iterates through every possible KeyCode.
   - Checks if each key was pressed using the legacy Input system.
   - Resolves the corresponding path in the new Input System using ResolveNewPathForKey.
   - If InputSystemExtensionData is available, retrieves additional info:
     - Index in the key list by KeyCode and by new path name.
     - Associated key name and whether a sprite is assigned.
   - Logs detailed messages with all this information.
2. **LogNewOnlyInputs**
   - Iterates over new Input System paths that were pressed but **not detected by legacy KeyCode checks**.

- Checks if each path exists in InputSystemExtensionData.
- Logs presence/absence and sprite status.

Path Resolution

- **ResolveNewPathForKey**
    - Finds the best match for a KeyCode in the new Input System.
    - Steps for resolution:
        1. **ManualMap**: If a manual mapping exists, use it.
        2. **Letters**: Map A-Z to <Keyboard>/a to <Keyboard>/z.
        3. **Top-row numbers**: Map Alpha0-Alpha9 to <Keyboard>/0 to <Keyboard>/9.
        4. **Numpad numbers**: Map Keypad0-9 to <Keyboard>/numpad0 to <Keyboard>/numpad9.
        5. **Function keys**: Map F1-F15 to <Keyboard>/f1 to <Keyboard>/f15.
        6. **Heuristic match**: Compare the KeyCode string to pressed paths from the new Input System to find likely matches.
        7. **Fallback**: If no match is found, returns "Unknown mapping".
- **NormalizeForMatch**
    - Helper function to normalize strings for comparison.
    - Converts to lowercase and removes non-alphanumeric characters to improve matching accuracy.

Usage

1. Attach this component to any GameObject in a Unity scene.
2. When running the game, press any key or button.
3. The console will display messages for each detected input:
    - **Legacy KeyCode** pressed.
    - **New Input System path**.
    - **Index** in the key database (if available).
    - Whether a **sprite** is assigned for visual reference.
4. Also logs inputs that are detected only by the new Input System.

This is helpful for:

- Verifying that all input devices are mapped correctly.
- Ensuring legacy and new input systems are consistent.
- Debugging missing or incorrect input mappings in InputSystemExtensionData.

Test Script

Purpose

This script is a **test/demo component** for Unity that demonstrates how to use the new Input System to handle **jumping** and **movement**. It uses input actions to control a Rigidbody's velocity and forces, while optionally interacting with an **input display manager** to show input feedback in the UI.

Variables and Fields

1. **actionJump**
    - Input action reference for the jump action.
    - Expected to produce a float value when pressed.
    - Configured via a custom attribute that allows selection from a predefined list of InputActions in the inspector.
2. **actionMove**
    - Input action reference for movement.
    - Expected to produce a Vector2 value representing directional input (e.g., WASD or joystick).
3. **targetRigidbody**
    - The Rigidbody component that will receive jump forces and horizontal movement velocities.
4. **speed**
    - Multiplier for horizontal movement speed.
    - Applied to the movement vector to control how fast the Rigidbody moves.
5. **displayManager**

  - Reference to an **Input Display Manager**, which can show or hide input UI elements.
  - Used to visually display input interactions in the game.

6. **nameTag**
   - Tag used to identify input UI elements in the display manager.
7. **enable**
   - Boolean flag to toggle whether the input display viewer should be enabled or disabled.
8. **jumpInputEvent**
   - Event configuration for jump input.
   - Stores the actions that should occur when the jump input is pressed.
9. **moveInputEvent**
   - Event configuration for movement input.
   - Stores actions that occur when movement input is held or released.

Properties
The script exposes properties to **get or set** the main fields like jump/move actions, Rigidbody target, movement speed, display manager, name tag, and enable flag. These allow external scripts to modify or read these values at runtime.

Unity Lifecycle Methods
1. **Start**
   - Runs once when the component is initialized.
   - Configures input events for jump and movement:
     - **Jump**:
       - Resets the Rigidbody's vertical velocity to zero.
       - Applies an upward impulse to simulate jumping.
     - **Move**:
       - Updates horizontal velocity based on input direction multiplied by speed.
       - Preserves vertical velocity while moving.
       - Stops horizontal movement when input is released while keeping vertical velocity intact.
2. **OnDestroy**
   - Runs when the component is destroyed.
   - Cleans up input event subscriptions to prevent memory leaks.

Public Methods
1. **EnableAndDisableViewer**
   - Toggles the visibility of input UI elements associated with the specified name tag.
   - Uses the display manager and the enable flag to show or hide the input display viewer.
   - Can be triggered from the context menu in the Unity inspector.

How It Works
1. The component listens to input actions configured in the new Input System.
2. When the jump action is triggered, it modifies the Rigidbody's vertical velocity and applies a jump force.
3. When movement input is detected, it updates the Rigidbody's horizontal velocity based on input direction and speed.
4. Movement stops when the input is released, but vertical motion from gravity or jumping continues.
5. Optional input UI elements can be toggled to show what input is being received, useful for debugging or demonstrations.

Usage
1. Attach this script to any GameObject in a Unity scene.
2. Assign the **jump** and **movement** InputAction references in the inspector.
3. Assign a Rigidbody to the **targetRigidbody** field.
4. Optionally, assign an **Input Display Manager** to show input feedback.
5. Adjust the **speed** value to control movement intensity.
6. Run the scene and press jump or movement keys to see the Rigidbody respond and, if configured, the UI update.

This script is primarily a **demonstration tool** to show how to use the new Input System to drive Rigidbody movement and integrate with a visual input display system, making it useful for testing, tutorials, or prototyping player controls.