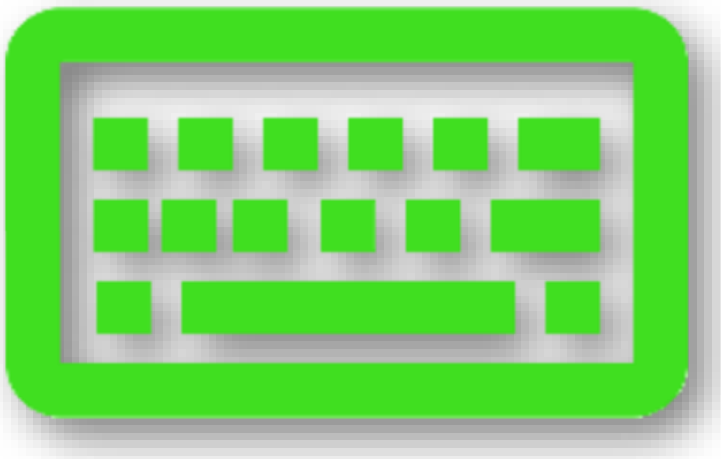

Custom Keyboard Settings

Documentation

———

Creator: Lucas Gomes Cecchini

Pseudonym: AGAMENOM

Overview

This document will help you use **Assets Custom Keyboard Settings** for **Unity**.

With it, you can easily configure and change your game's **Keyboards** from a **menu**.

It will create two types of **Scriptable Objects**, one for saving data to **PlayerPrefs** and another for accessing the Keyboard via script.

It also has **auto-save** capabilities as soon as settings are changed.

Instructions

You can get more information from the Playlist on YouTube:

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

Script Explanations

KeyboardControlData

This script provides a framework for managing keyboard control configurations within Unity. Here's a breakdown of its components and functionality:

1. KeyboardControlData Class

- **Purpose**: The primary role of this class is to store and manage keyboard control configurations, specifically through a **`ScriptableObject`**. This class allows for customizations of keyboard inputs, key code representations, and associated visual sprites.

- **Key Elements**:
  - **inputDataList**: A list of **`InputData`** objects. Each **`InputData`** represents a specific keyboard configuration (e.g., which key corresponds to a particular action). This list allows for the collection and organization of all keyboard configurations.
  - **defaultSprite**: A default sprite used to visually represent a key on the keyboard. This can be used as a fallback or default image for any input without a dedicated sprite.
  - **keyCodesSprites**: A list of **`InputSpriteList`** objects. Each entry in this list associates a particular sprite with a key code, allowing for visual customization of specific keys. It makes it easy to represent keys visually in the Unity editor.

2. Editor-Specific Functionality

- **Purpose**: The script contains editor-specific code that allows for the configuration of keyboard settings directly within Unity's Editor.

- **KeyboardControlDataEditor Class**: This class provides functionality to open and display the **`KeyboardControlData`** object in a Unity editor window. It is decorated with the **`#if UNITY_EDITOR`** directive, ensuring that it only runs in the Unity Editor and is excluded from builds.

- **OpenKeyboardSettingsData Method**: This method opens a custom editor window for the **`KeyboardControlData`** object.
  - It uses **`KeyboardTagHelper.GetKeyboardControlData()`** to fetch the **`KeyboardControlData`** object. If this object exists, it either:
    - Focuses an existing editor window displaying the **`KeyboardControlData`**.
    - Opens a new property editor window if none is currently open.
  - If no **`KeyboardControlData`** object is found, an error is logged.

3. Code Annotations

- **ScriptableObject**: A Unity class that allows data to be stored in assets within a project. It is ideal for settings and configuration objects, as it lets developers manage and reuse data across scenes.
- **Editor Tools**: The use of **`EditorWindow`** and **`EditorUtility.OpenPropertyEditor`** is a way to interact with Unity's built-in Property Editor, a graphical tool for inspecting and modifying objects in the editor.

4. Comment Block

- The comment block at the top of the script provides a summary of the script's functionality. It mentions that the script is used for managing and accessing keyboard control configurations through **`ScriptableObject`** objects, allowing customization of key inputs and their visual representation with sprites. Additionally, it provides some personal information about the script's author and pseudonym.

In short, this script is designed to handle customizable keyboard controls, providing an intuitive interface in the Unity Editor to configure keyboard mappings and their visual representation. The editor-related features help streamline this process, ensuring that developers can easily manage keyboard input settings.

InputData

The **`InputData`** script is a Unity script that defines a class to represent individual keyboard input configurations. Here's a detailed explanation of the script and its components:

Script Purpose

The purpose of the **`InputData`** script is to encapsulate data for a single keyboard input configuration. This configuration includes a tag to identify the input and a key code representing the specific keyboard key. The script is implemented as a **`ScriptableObject`**, allowing the creation of assets that store this data persistently within the Unity Editor.

Script Breakdown

1. Namespace and Imports

```
using UnityEngine;
```

The script starts by importing the **`UnityEngine`** namespace, which is essential for accessing Unity's core engine features, including **`ScriptableObject`** and **`KeyCode`**.

2. Class Declaration

```
public class InputData : ScriptableObject
{
    public string keyboardTag;
    public KeyCode keyboard;
}
```

- The class is declared as **`public`**, making it accessible from other scripts.
- The class inherits from **`ScriptableObject`**, a special Unity class that allows for the creation of custom asset files.

3. Public Fields

```
public string keyboardTag;
public KeyCode keyboard;
```

- **`keyboardTag`** is a public field of type **`string`**. This field is used to store a tag or identifier for the input data. It helps in distinguishing between different input configurations.
- **`keyboard`** is a public field of type **`KeyCode`**. **`KeyCode`** is an enumeration in Unity that represents all possible keys on a keyboard. This field stores the specific key associated with the input configuration.

ScriptableObject Explanation

`ScriptableObject` is a special type of object in Unity designed for holding data. Unlike **`MonoBehaviour`**, **`ScriptableObject`** instances do not need to be attached to GameObjects. They are useful for creating assets that store data which can be easily shared and reused across different parts of your game or application.

Conclusion

The **`InputData`** script provides a simple yet effective way to manage individual keyboard input configurations using Unity's **`ScriptableObject`** system. By storing input data in assets, you can easily

configure and reuse keyboard settings across different parts of your game or application. This approach also makes it easier to manage and update input configurations without hardcoding them into scripts.

KeyboardControlDataInspector

This script is a custom editor created for Unity's **`KeyboardControlData`** ScriptableObject. It enhances the Unity Editor's interface to make it easier to manage and edit a collection of keyboard input settings, including associated data such as key tags and sprites for key visual representation. Below is a detailed explanation of the script's functionality:

Purpose

1. **Simplified Input Management**: The script provides a graphical user interface (GUI) for managing **`InputData`** entries, which represent individual keyboard inputs and their metadata.
2. **Key Sprite Generation**: It includes tools to associate visual sprites with keyboard keys.
3. **Data Organization**: Allows renaming and saving input data assets systematically, ensuring a well-organized project structure.

Key Features

1. **Input Data Display and Editing**:
   - Shows a list of **`InputData`** entries.
   - Allows editing the **`keyboardTag`** (a label for the input) and **`keyboard`** (the key associated with the input).
   - Supports key detection by capturing a key press directly from the user.

2. **List Operations**:
   - **Add New Entries**: Users can create new **`InputData`** entries, stored as assets in a designated folder.
   - **Delete Entries**: Entries can be deleted with confirmation to prevent accidental removal.

3. **Save Operations**:
   - The **"Save All"** function updates and renames **`InputData`** assets based on their tags. Duplicate tags are managed by appending unique counters.

4. **Sprite Management**:
   - Users can generate a list of key codes and associate sprites with them.
   - The **"Default Sprite"** option specifies a fallback sprite if a specific sprite is unavailable.

5. **Key Detection**:
   - A **"Detect"** button allows users to press a key, which is then assigned to the selected **`InputData`**.

6. **Custom Asset Management**:
   - Automatically organizes **`InputData`** assets in a folder corresponding to the parent **`KeyboardControlData`** asset.

Core Methods

1. **`OnInspectorGUI`**:
   - Handles the rendering and interaction of the custom editor GUI.
   - Manages list display, input handling, and buttons for various operations.

2. **`DeleteInputData`**:
   - Deletes an **`InputData`** asset and removes it from the list.

3. **`CreateInputData`**:
   - Creates a new **`InputData`** asset with a default tag and no key assigned.

4. **`SaveAllInputData`**:
   - Iterates through all **`InputData`** entries to save and rename their corresponding assets.

5. **`GenerateKeyCodeList`**:
   - Populates a list of **`InputSpriteList`** objects linking key codes to sprites.

6. **`GetSprite`**:
   - Retrieves a sprite asset by its name. If not found, the default sprite is returned.

Behavior in the Unity Editor
- **Initialization**: Ensures the **`inputDataList`** is ready for use.
- **Dynamic Interaction**: Updates the editor dynamically as users interact with it (e.g., adding/removing items or detecting keys).
- **Data Persistence**: Saves any changes to the assets immediately to maintain consistency.

Use Case
This script is ideal for projects requiring a customizable keyboard input system with visual feedback, such as games, interactive applications, or simulations. It simplifies the setup of input mappings and allows quick adjustments directly within the Unity Editor.

KeyboardControlDataCreator

This script is designed to facilitate the creation and management of custom **`KeyboardControlData`** assets within the Unity Editor. Here's a detailed explanation of how it works:

Overview

The script automates the process of creating a **`KeyboardControlData`** asset, which is a type of custom data used to store keyboard control configurations. It also ensures that this asset is created automatically if it doesn't already exist in the project.

Components

1. **KeyboardControlDataCreator Class**:
   - **Purpose**: Provides functionality to create a **`KeyboardControlData`** asset through a custom menu option in Unity.

   - **CreateCustomObjectData Method**:
     - **Menu Item**: Adds a new option to the Unity Editor's Asset menu under **"Create/Custom Keyboard Settings/Keyboard Control Data"**.
     - **Asset Path**: Defines the path where the asset will be saved (**`Assets/Resources/Keyboard Control Data.asset`**).
     - **Folder Check**: Verifies if the **`Resources`** folder exists in the project. If not, it creates the folder.
     - **Duplicate Check**: Checks if an asset with the same name already exists at the specified path. If it does, it prompts the user to confirm if they want to replace the existing asset.
     - **Asset Creation**: Creates a new instance of **`KeyboardControlData`**, saves it as an asset at the specified path, and marks it as dirty to ensure it is saved correctly.
     - **Focus Editor**: Automatically focuses on the Project window and selects the newly created asset.

2. **KeyboardControlDataCreatorStartup Class**:
   - **Purpose**: Ensures that the **`KeyboardControlData`** asset is created automatically when the Unity Editor starts, if it doesn't already exist.

   - **Static Constructor**:
     - **Delayed Execution**: Uses a delayed call to check for the asset after the Unity Editor has started up.
     - **Asset Existence Check**: Checks if the asset already exists at the specified path. If it doesn't, it triggers the **`CreateCustomObjectData`** method to create it.

Functionality

1. **Creating the Asset**:
   - The script provides a menu option in the Unity Editor to create a **`KeyboardControlData`** asset.

- When this menu option is selected, it ensures that the necessary folder structure is in place and handles the creation and saving of the asset.
- If the asset already exists, the user is prompted to confirm if they want to replace it, avoiding accidental overwrites.

2. **Automatic Asset Creation**:
- On editor startup, the script checks for the presence of the **`KeyboardControlData`** asset.
- If the asset does not exist, it automatically creates it, ensuring that the project always has this essential asset available without requiring manual intervention.

Summary

This script streamlines the process of managing **`KeyboardControlData`** assets in Unity by providing a convenient way to create the asset through the editor menu and ensuring its presence in the project through automatic creation on editor startup. This automation simplifies asset management, especially in larger projects where such assets are crucial for keyboard control settings.

KeyboardTagDropdownAttribute

This script introduces a custom attribute and property drawer for Unity's Inspector. The attribute is designed to be used with string properties to provide a dropdown list of keyboard tags. The property drawer manages the rendering of this dropdown and handles validation and warnings.

1. `KeyboardTagDropdownAttribute` Class

```
#if UNITY_EDITOR
using System.Collections.Generic;
using CustomKeyboard;
using UnityEditor;
using UnityEngine;

public class KeyboardTagDropdownAttribute : PropertyAttribute
{
    // This attribute is just a marker, it does not need additional implementation.
}
#endif
```

Description

- **`KeyboardTagDropdownAttribute`**: This class is a custom attribute derived from **`PropertyAttribute`**. It doesn't need any additional implementation beyond marking fields in Unity's Inspector. Its presence indicates that the property should use a custom property drawer.

2. `KeyboardTagDropdownDrawer` Class

```
#if UNITY_EDITOR
[CustomPropertyDrawer(typeof(KeyboardTagDropdownAttribute))]
public class KeyboardTagDropdownDrawer : PropertyDrawer
{
    ...
```

Description

- `**[CustomPropertyDrawer(typeof(KeyboardTagDropdownAttribute))]**`: This attribute specifies that `**KeyboardTagDropdownDrawer**` is the custom drawer for properties marked with `KeyboardTagDropdownAttribute`.

Methods

- `**OnGUI(Rect position, SerializedProperty property, GUIContent label)**`: This method handles the drawing of the property in the Inspector.
  - **Property Type Check**: Ensures the property is a string. If not, it shows an error message.
  - **Retrieve Keyboard Tags**: Uses `**KeyboardTagHelper.GetKeyboardControlData()**` to fetch the list of keyboard tags from `**KeyboardControlData**`.
  - **Dropdown Options**: Constructs a list of tags including a **"Missing Tag"** option to handle cases where the current string doesn't match any tag.
  - **Dropdown Rendering**: Uses `**EditorGUI.Popup**` to display the dropdown and handle user selection.
  - **Warning Message**: Shows a warning if the selected value is **"Missing Tag"**.

- `**GetPropertyHeight(SerializedProperty property, GUIContent label)**`: Determines the height of the property field in the Inspector.
  - **Additional Height**: Adds extra height if the current string value does not match any of the keyboard tags to accommodate the warning message.

Summary

This script enhances Unity's Inspector by providing a custom dropdown for string properties marked with `**KeyboardTagDropdownAttribute**`. It ensures that only valid keyboard tags are selectable and provides feedback if the current value does not match any existing tags. This custom editor functionality improves the usability and reliability of keyboard tag assignments within Unity's development environment.

KeyboardTagHelper

This script provides functionality for managing keyboard control data within a Unity project, specifically for persisting, updating, and retrieving configurations associated with keyboard inputs. It uses Unity's `**PlayerPrefs**` to save and load settings between sessions. Here's a detailed breakdown of the various components and their roles:

1. KeyboardTagHelper Class

- **Purpose**: A utility class for handling the keyboard control data. It provides methods for retrieving, updating, saving, and loading keyboard configurations associated with specific tags and key codes.

Key Methods:

- **GetKeyboardControlData**: This method loads a `**KeyboardControlData**` resource from the Unity `**Resources**` folder. The resource must be named **"Keyboard Control Data"**. If the resource isn't found, an error is logged.

- **GetInputFromTag**: This method retrieves an `**InputData**` object based on a specific tag (a string identifier for the input). It searches through the `**KeyboardControlData**` object's list of inputs and returns the one matching the provided tag. If no match is found, it returns `**null**`.

- **SetKey**: This method updates the `**KeyCode**` of an `**InputData**` object. The `**InputData**` must be provided, and the new key is set directly.

- **SetKeyFromTag**: This method updates the `**KeyCode**` of an `**InputData**` object identified by a tag. If the tag doesn't exist, an error is logged.

- **GetKeySprite**: This method returns a sprite representing a particular `**KeyCode**`. It checks the `**KeyboardControlData**` for an associated sprite and returns it. If no specific sprite is found for the key, it defaults to a global `**defaultSprite**`.

- **SaveKeyboardControlData**: This method serializes the current keyboard control data (including all **`InputData`** objects) into a JSON format and stores it in **`PlayerPrefs`**. It essentially saves the data for persistence across Unity sessions. Each **`InputData`** object is converted into a simpler **`InputDataListSave`** format that stores the tag and the associated **`KeyCode`**.

- **LoadKeyboardControlData**: This method loads keyboard control data from **`PlayerPrefs`** when the application starts. If there is saved data, it deserializes the JSON and updates the **`KeyboardControlData`** object in the project with the saved key codes. The method is marked with the **`[RuntimeInitializeOnLoadMethod]`** attribute, meaning it runs automatically when the game starts.

2. Supporting Serializable Classes

- **KeyboardControlDataSave**: A class used to store the serialized keyboard control data in a format that can be saved to **`PlayerPrefs`**. It contains a list of **`InputDataListSave`** objects that store the tag and the key code of each input.

- **InputDataListSave**: This is a lightweight class that represents a single keyboard input (tag and **`KeyCode`**). It is used to store the key configuration in a way that can be easily serialized and saved to **`PlayerPrefs`**.

- **InputSpriteList**: This class associates a **`KeyCode`** with a specific **`Sprite`** to visually represent the key in the UI. Each entry in the **`KeyboardControlData`** includes a **`KeyCode`** and its corresponding sprite.

3. Persistence System Using PlayerPrefs

- **Saving Data**: The **`SaveKeyboardControlData`** method serializes the keyboard control data (list of **`InputData`** objects) into a JSON string and stores it using Unity's **`PlayerPrefs`**. This ensures that the configuration persists between sessions.

- **Loading Data**: The **`LoadKeyboardControlData`** method loads the saved keyboard configuration data from **`PlayerPrefs`**. It checks for existing saved data, deserializes it into the appropriate format, and updates the **`KeyboardControlData`** object accordingly, restoring the saved key configurations.

4. Error Handling and Debugging

- Throughout the script, there is a focus on error handling. If certain actions fail (like failing to load the **`KeyboardControlData`** resource or finding a matching tag), appropriate error messages are logged using **`Debug.LogError`**. This helps developers identify issues with missing or incorrect configurations.

5. Automatic Data Loading

- The **`LoadKeyboardControlData`** method is annotated with **`[RuntimeInitializeOnLoadMethod]`**, which means it is automatically executed when the game starts, ensuring that any previously saved keyboard settings are loaded and applied as soon as the game is run.

6. Overall Functionality

- The script provides a central utility for managing customizable keyboard settings. It allows developers to modify key configurations, save them for later use, and load them at runtime. This can be particularly useful for games or applications that allow users to customize their keyboard controls.
- The system is flexible, as it handles both the keyboard control data (the keys) and their visual representations (sprites). The saved data can be reloaded later, maintaining consistency in user preferences across game sessions.

In short, this script is part of managing customizable keyboard controls in Unity. It provides an interface for updating key mappings, saving them for future sessions, and visually representing the keys with sprites. The use of PlayerPrefs ensures that changes made to key configurations are persistent and easily recoverable, allowing for a personalized user experience.

KeyboardSettingsManager and TMP_KeyboardSettingsManager

This script is a Unity MonoBehaviour component that manages customizable keyboard input settings for a game. It allows players to select or reset specific keys associated with actions in the game and persist those settings across sessions. Here's a detailed explanation:

Core Purpose

The script facilitates the management of keyboard input configurations by allowing the user to:
1. Select a key (KeyCode) for a specific action.
2. Reset the key to its default value.
3. Persist the customized or default settings using Unity's **`PlayerPrefs`**.

Primary Components

1. UI Elements

- **Select Button**: Allows the user to start selecting a new key.
- **Reset Button**: Resets the key to a predefined default value.
- **Text or Image Display**: Shows the current key either as text or an image, depending on the configuration.

2. Keyboard Tag

Each action or setting is associated with a unique **`keyboardTag`** that identifies it. This tag links the UI to the corresponding data in a central input settings file.

3. Default KeyCode

A predefined default key is provided for each action, which can be restored by the reset functionality.

4. InputData

This is an external data structure retrieved using the **`KeyboardTagHelper`**. It holds the actual key settings for actions defined by the **`keyboardTag`**.

5. Other Managers

To avoid conflicts, the script keeps track of all instances of the **`KeyboardSettingsManager`** in the scene. This ensures that the same key isn't assigned to multiple actions.

Script Workflow

1. Initialization (`Start` Method)

- **Retrieve Other Managers**:
  The script finds all other instances of **`KeyboardSettingsManager`** in the scene to ensure that key assignments are unique.

- **Setup UI Listeners**:
  Button listeners for selecting and resetting keys are assigned.

- **Load Settings**:
  - If settings exist (from **`KeyboardTagHelper`**), they are loaded and applied.
  - Otherwise, default settings are applied.

- **UI Configuration**:
  Depending on whether the key should be displayed as text or an image, the corresponding UI elements are activated or hidden.

2. Key Selection (`OnSelectButtonClick` Method)

- When the **"Select"** button is clicked:
  - A delay of 0.5 seconds is started to prevent accidental key detection.
  - The script enters a **"listening"** state to detect the next key press.

3. Resetting to Default (`OnResetButtonClick` Method)
- When the **"Reset"** button is clicked:
  - The KeyCode is reset to the predefined default value.
  - The new settings are saved.

4. Listening for New Inputs (`Update` Method)
- **Key Detection**:
  When in the **"listening"** state:
  - The script iterates through all possible **`KeyCode`** values to check for a pressed key.
  - If a key is pressed:
    - It verifies whether the key is already in use by another **`KeyboardSettingsManager`**.
    - If unused, the key is assigned as the new KeyCode.
    - UI is updated to reflect the change.
    - The settings are saved.

- **Reset Button State**:
  The reset button is enabled only if the current KeyCode differs from the default.

- **UI Updates**:
  The visibility of the text or image UI elements is toggled based on the current configuration.

5. Avoiding Key Conflicts (`IsKeyCodeUsedByOtherManagers` Method)
- The script checks all other instances of **`KeyboardSettingsManager`** in the scene.
- If another instance is already using the newly selected key, the key assignment is rejected, and a warning is logged.

Saving Settings
- When a new key is selected or reset, the script:
  - Updates the **`KeyCode`** in the **`InputData`** structure.
  - Persists the data using the **`KeyboardTagHelper`**.

Error Handling
- If the associated **`InputData`** for the **`keyboardTag`** is missing, the script logs an error and falls back to default settings.
- If a KeyCode conflict is detected, a warning is displayed.

Key Features
1. **Persistence**: The script uses **`PlayerPrefs`** (via **`KeyboardTagHelper`**) to save and load settings, ensuring customization is retained across game sessions.
2. **Conflict Prevention**: Ensures unique KeyCodes across multiple **`KeyboardSettingsManager`** instances.
3. **UI Integration**: Seamlessly updates the UI (text or image) to reflect the current key settings.
4. **Customization**: Allows players to change key bindings dynamically at runtime.

Practical Use
This script is ideal for games or applications that require customizable controls. It ensures:
- Player-friendly customization of input settings.
- Smooth integration with Unity's UI components.
- Robust handling of potential errors and conflicts in input settings.

MovementHandler

This script defines a class, **`MovementHandler`**, which manages directional input for player movement within a game. The class tracks the state of input keys (like forward, backward, left, and right) and calculates the resulting movement direction based on the player's key presses.

Here's a breakdown:

1. **Variables:**
   - **`forwardLastPressedTime`**, **`backLastPressedTime`**, **`rightLastPressedTime`**, **`leftLastPressedTime`**:
     These variables store the time when each directional key (forward, back, right, left) was last pressed. They start with a value of **`-1f`** to indicate no key has been pressed initially.

   - **`forwardInputActive`**, **`backInputActive`**, **`rightInputActive`**, **`leftInputActive`**:
     These boolean flags track whether a specific direction key is currently being pressed.

   - **`forwardInput`**, **`backInput`**, **`rightInput`**, **`leftInput`**:
     These are instances of **`InputData`**, which represent the keyboard keys used for the respective directions. They are initialized in the constructor, and their fields contain information on which key triggers each movement.

2. **Constructor:**
   The constructor takes four parameters—**`forward`**, **`back`**, **`right`**, and **`left`**—which represent the input data for each direction. These are then assigned to the corresponding variables.

3. **`GetMovementInput` Method:**
   This method returns the movement direction based on the active inputs. It handles the key press and release events and computes the player's movement accordingly.

   - Key Down Events:
   - If a key is pressed (like the forward key), the script records the current time and marks that input as active (e.g., **`forwardInputActive = true`**).

   - Key Events:
   - If the key is released, the active flag for that direction is set to **`false`** (e.g., **`forwardInputActive = false`**).

   - Movement Calculation:
   - The script checks whether the forward or backward key is active and prioritizes the most recently pressed key by comparing the time values (**`forwardLastPressedTime`** and **`backLastPressedTime`**).
   - Similarly, it compares right and left key inputs, giving priority to the more recently pressed key.
   - Based on which key was pressed last, the method adjusts the movement direction by subtracting the corresponding vector (e.g., **`Vector3.forward`** for forward, **`Vector3.back`** for backward, etc.).

   - Normalization:
   - After determining the direction based on active inputs, the resulting vector is normalized. This ensures consistent speed regardless of the direction the player is moving in.

Summary

In essence, the script allows players to move in four directions (forward, back, left, right), and it handles cases where multiple keys are pressed by prioritizing the key pressed most recently. The movement is then output as a normalized vector, which can be used to move the player in a consistent direction and speed.

KeyboardPrefabCreator

This script facilitates creating and configuring keyboard-related UI prefabs in Unity directly from the Editor's menu. It includes options for both legacy UI prefabs and TextMeshPro (TMP) prefabs, streamlining the process of integrating these components into Unity projects. Here's a breakdown of its functionality:

General Overview

The script provides tools to:
1. Create UI prefabs specifically designed for managing keyboard settings.
2. Automatically set up the necessary parent-child relationships for prefabs in the scene hierarchy.
3. Handle the creation of a **`Canvas`** (if needed) for UI elements.
4. Unpack prefabs after instantiation to allow customization.
5. Use Unity's **`Undo`** system to make changes reversible.
6. Provide Editor integration by adding options to Unity's **"GameObject/UI"** menu.

Key Functionalities

Menu Options for Prefab Creation

- **Legacy Prefab Option**: Adds a menu item labeled **"Keyboard Settings Manager (Legacy)"** under **`GameObject > UI`**. Selecting this creates a prefab using legacy Unity UI components.
- **TMP Prefab Option**: Adds a menu item labeled **"Keyboard Settings Manager (TMP)"** for creating prefabs that utilize TextMeshPro components.

Both options invoke a shared method, **`CreateAndConfigurePrefab`**, to handle the prefab creation process.

Prefab Creation Process

1. **Determine Parent Object**:
   - If the prefab is UI-based, the script checks for an existing **`Canvas`** in the scene.
   - If no **`Canvas`** is found, it automatically creates one using the **`CreateUICanvas`** method.
   - For non-UI prefabs, the script uses the currently selected object in the Unity hierarchy as the parent (if applicable).

2. **Find the Prefab**:
   - Uses Unity's **`AssetDatabase`** to locate the prefab file based on its name.
   - Logs an error if the prefab cannot be found.

3. **Instantiate the Prefab**:
   - The prefab is instantiated under the determined parent object.
   - If the prefab is UI-based, it will be placed under the **`Canvas`** hierarchy.

4. **Finalize the Setup**:
   - Registers the prefab's creation with Unity's **`Undo`** system, allowing developers to revert changes.
   - Unpacks the prefab instance to enable further modifications.
   - Automatically selects the prefab in the Unity hierarchy and triggers the rename action for easy identification.

Creating a Canvas

The **`CreateUICanvas`** method ensures a proper setup for UI prefabs:
- Creates a new GameObject named **"Canvas"**.
- Adds essential components:
  - **`Canvas`**: To render UI elements.
  - **`CanvasScaler`**: To handle resolution-independent scaling.
  - **`GraphicRaycaster`**: For detecting UI interactions.
- Configures the **`Canvas`**:
  - Sets the render mode to **`ScreenSpaceOverlay`**.
  - Assigns it to the **"UI"** layer for appropriate rendering.
- Creates an accompanying **`EventSystem`** to manage input events (e.g., mouse clicks, keyboard presses).

- Registers both the **`Canvas`** and **`EventSystem`** for undo functionality.

Finding Prefabs by Name

The **`FindPrefabByName`** method searches the Unity project for a prefab matching a given name:
- Scans the project using **`AssetDatabase.FindAssets`**.
- Verifies the file name of each result and returns the first match.
- Logs an error if no matching prefab is found.

Unpacking and Selection

Once a prefab is created:
1. It is unpacked to allow direct editing of its components.
2. The newly created prefab is selected in the Unity hierarchy for immediate interaction.
3. A rename action is triggered to allow developers to customize the name of the new prefab easily.

Error Handling and Debugging

- Logs meaningful error messages when prefabs cannot be found or if certain actions fail.
- Provides fallback mechanisms (e.g., creating a new **`Canvas`** when none exists) to ensure smooth prefab creation.

Integration and Usability

This script enhances productivity by:
1. Automating repetitive tasks such as creating and configuring prefabs and UI elements.
2. Reducing errors by handling parent assignment and prefab unpacking programmatically.
3. Improving the user experience with intuitive menu options and automated selection/renaming.

In summary, the script simplifies the process of adding and managing keyboard settings UI prefabs in Unity, making it more efficient for developers to implement and customize these components.