Custom Keyboard Settings

Documentation

Creator: Lucas Gomes Cecchini

Online Name: 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

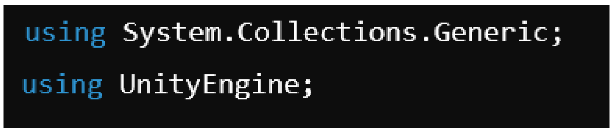
The `**KeyboardControlData**` script is a Unity script that defines a class used to manage a list of `**InputData**` objects. Here's a detailed explanation of the script and its components:

Script Purpose

The purpose of the `**KeyboardControlData**` script is to act as a container for multiple keyboard input configurations. This container is implemented as a `**ScriptableObject**`, which allows for the creation of an asset in Unity that can store persistent data. The script manages a list of `**InputData**` objects, each of which represents a specific keyboard input configuration.

Script Breakdown

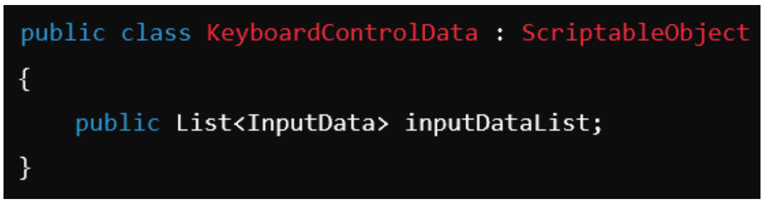
1. Namespace and Imports



```
using System.Collections.Generic;
using UnityEngine;
```

The script starts by importing the necessary namespaces:
- `**System.Collections.Generic**` is used to allow the use of generic collections such as `**List<T>**`.
- `**UnityEngine**` is the core namespace of Unity that provides access to Unity's engine features.

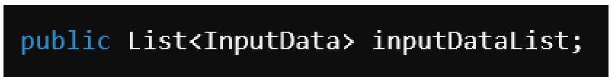
2. Class Declaration



```
public class KeyboardControlData : ScriptableObject
{
    public List<InputData> inputDataList;
}
```

- The class is declared as `**public**`, making it accessible from other scripts.
- The class inherits from `**ScriptableObject**`, which is a special class in Unity that allows you to create custom assets.

3. List of InputData



```
public List<InputData> inputDataList;
```

- `**inputDataList**` is a public field of type `**List<InputData>**`.
- This list holds multiple instances of the `**InputData**` class, each representing a specific keyboard input configuration.
- By making this field public, it will be visible and editable in the Unity Editor when the `**KeyboardControlData**` asset is selected.

ScriptableObject Explanation

`**ScriptableObject**` is a special type of object in Unity that allows you to create asset files to store data. Unlike MonoBehaviour scripts, `**ScriptableObject**` instances do not need to be attached to GameObjects. They are useful for storing data that can be shared across multiple scenes and objects.

Conclusion

The `**KeyboardControlData**` script provides a way to manage multiple keyboard input configurations in a structured and persistent manner using Unity's `**ScriptableObject**` system. This approach is beneficial for managing configurations that need to be shared across different parts of your game or application.

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

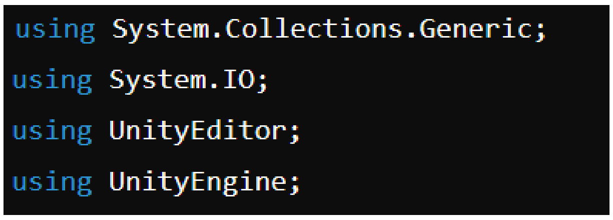
The `**KeyboardControlDataInspector**` script is a custom editor for managing `**KeyboardControlData**` ScriptableObject within the Unity Editor. It provides a user-friendly interface for adding, editing, and deleting `**InputData**` entries. Here's a detailed explanation of the script and its components:

Script Purpose

The purpose of the `**KeyboardControlDataInspector**` script is to enhance the Unity Editor by providing a custom inspector for `**KeyboardControlData**` ScriptableObject. This custom inspector allows users to manage a list of `**InputData**` objects, making it easier to configure keyboard input settings directly within the Unity Editor.

Script Breakdown

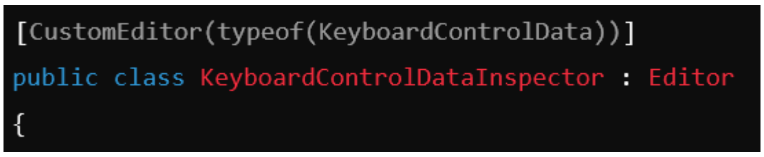
1. Namespace and Imports



```
using System.Collections.Generic;
using System.IO;
using UnityEditor;
using UnityEngine;
```

The script starts by importing necessary namespaces:
- `**System.Collections.Generic**` for using generic collections like `**List<T>**`.
- `**System.IO**` for handling file paths.
- `**UnityEditor**` for creating custom editor functionality.
- `**UnityEngine**` for core Unity features.

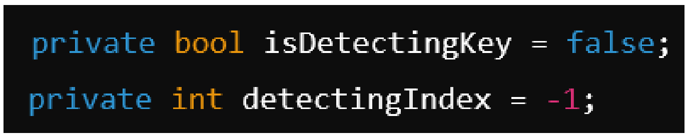
2. Custom Editor Declaration



```
[CustomEditor(typeof(KeyboardControlData))]
public class KeyboardControlDataInspector : Editor
{
```

- The `**[CustomEditor(typeof(KeyboardControlData))]**` attribute specifies that this custom editor is for the `**KeyboardControlData**` class.
- The class inherits from `**Editor**`, which is the base class for custom editors in Unity.

3. Fields for Key Detection

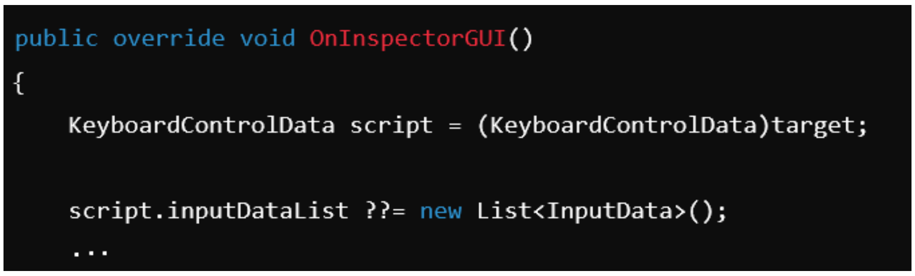


```
 private bool isDetectingKey = false;
private int detectingIndex = -1;
```

These fields manage the state of key detection:

- `**isDetectingKey**` indicates if the script is currently detecting a key press.
- `**detectingIndex**` holds the index of the `**InputData**` being edited.

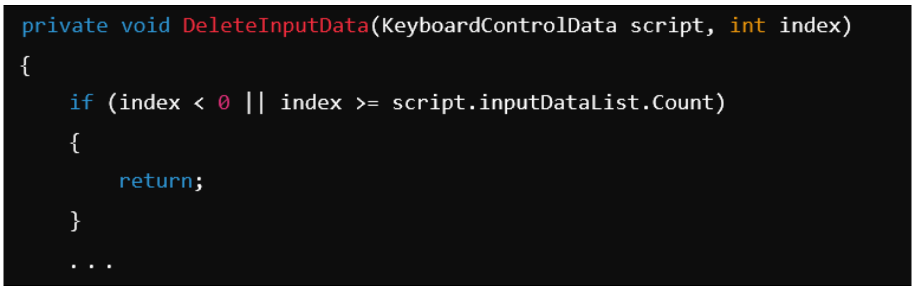
4. OnInspectorGUI Method



```
public override void OnInspectorGUI()
{
    KeyboardControlData script = (KeyboardControlData)target;

    script.inputDataList ??= new List<InputData>();
    ...
```

The `**OnInspectorGUI**` method is where the custom inspector's UI is defined:
- ****Initialization****: The `**inputDataList**` is initialized if it's null.
- ****Script Field****: Displays the script name in a read-only field.
- ****Headers****: Adds labels for better organization.
- ****Input Data List****:
  - If the list is empty, displays a message.
  - If the list has entries, iterates through each one to display and allow editing of the `**keyboardTag**` and `**keyboard**` fields, along with options to delete or detect keys.
- ****Save and Create Buttons****: Provides buttons to save all entries or create a new `**InputData**` entry.
- ****Key Detection****: Handles key detection, updating the relevant `**InputData**` entry when a key is pressed.

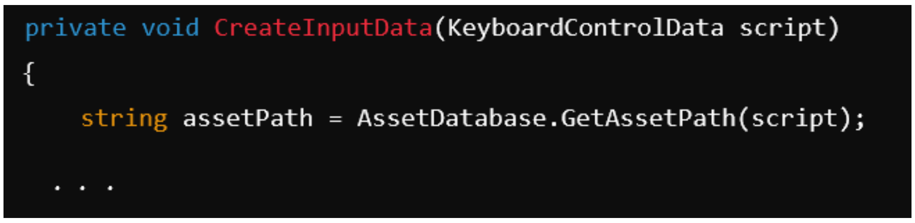
5. DeleteInputData Method



```
private void DeleteInputData(KeyboardControlData script, int index)
{
    if (index < 0 || index >= script.inputDataList.Count)
    {
        return;
    }
    ...
```

The `**DeleteInputData**` method handles the deletion of an `**InputData**` asset:
- Validates the index.
- Retrieves the asset path and deletes the asset if it exists.
- Removes the entry from the list and repaints the inspector.

6. CreateInputData Method



```
private void CreateInputData(KeyboardControlData script)
{
    string assetPath = AssetDatabase.GetAssetPath(script);

  . . .
```

The `**CreateInputData**` method handles the creation of a new `**InputData**` asset:
- Determines the folder path for the new asset.
- Ensures the folder exists or creates it.
- Creates a new `**InputData**` instance and assigns default values.
- Generates a unique file path for the new asset.
- Adds the new asset to the list and saves the changes.

7. SaveAllInputData Method

```
private void SaveAllInputData(KeyboardControlData script)
{
    string assetPath = AssetDatabase.GetAssetPath(script);
 . . .
```

The `**SaveAllInputData**` method saves all `**InputData**` assets and renames them based on their tags:
- Iterates through each `**InputData**` entry.
- Marks the asset as dirty to ensure changes are saved.
- Generates a new file path based on the `**keyboardTag**` and ensures it's unique.
- Moves the asset to the new path if necessary.

Conclusion

The `**KeyboardControlDataInspector**` script provides a comprehensive and user-friendly way to manage `**KeyboardControlData**` ScriptableObject entries within the Unity Editor. It allows for easy creation, editing, detection, and deletion of `**InputData**` entries, enhancing the workflow for configuring keyboard input settings.

KeyboardControlDataCreator

The `**KeyboardControlDataCreator**` script is designed to facilitate the creation of a `**KeyboardControlData**` asset within the Unity Editor. This script adds a menu item to the Unity Editor, allowing users to create and manage a `**KeyboardControlData**` asset easily. Here's a detailed explanation of each part of the script:

Script Purpose

The purpose of the `**KeyboardControlDataCreator**` script is to provide a simple way to create a `**KeyboardControlData**` asset through a custom menu item in the Unity Editor. This makes it more convenient for developers to generate and manage keyboard control configurations.

Script Breakdown

1. Namespace and Imports

```
using UnityEditor;
using UnityEngine;
```

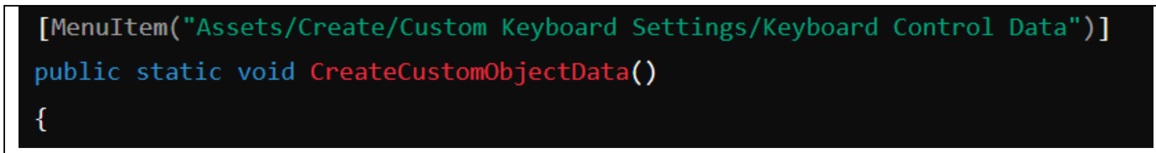
The script begins by importing necessary namespaces:
- `**UnityEditor**` for using editor-related functionalities.
- `**UnityEngine**` for core Unity features.

2. Class Declaration

```
public class KeyboardControlDataCreator
{
```

The `**KeyboardControlDataCreator**` class is declared without inheriting from any base class, as it only contains static methods for creating assets.

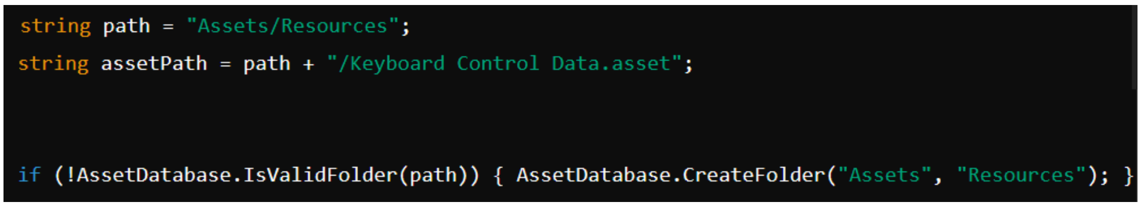
3. MenuItem Attribute



```
[MenuItem("Assets/Create/Custom Keyboard Settings/Keyboard Control Data")]
public static void CreateCustomObjectData()
{
```

- The `**[MenuItem("Assets/Create/Custom Keyboard Settings/Keyboard Control Data")]**` attribute specifies that the `**CreateCustomObjectData**` method will be available as a menu item in the Unity Editor under the path `**Assets/Create/Custom Keyboard Settings/Keyboard Control Data**`.
- The method `**CreateCustomObjectData**` is static, making it callable without needing an instance of the class.

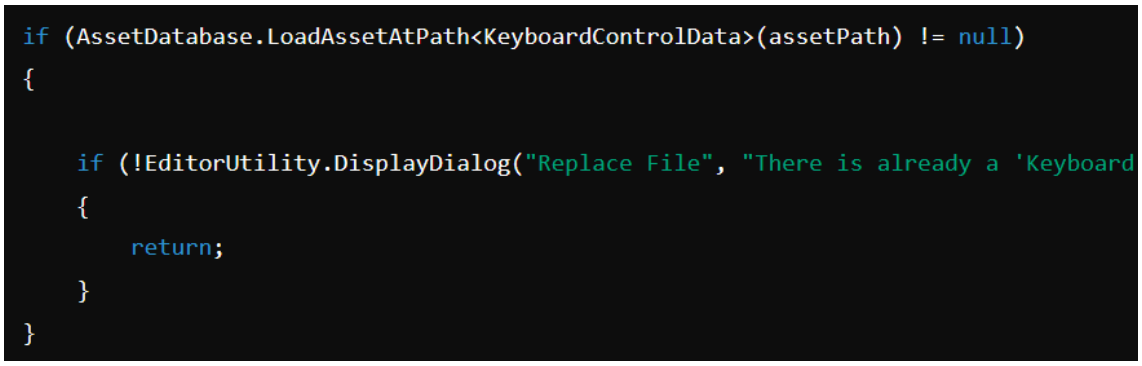
4. Asset Path and Folder Creation



```
string path = "Assets/Resources";
string assetPath = path + "/Keyboard Control Data.asset";


if (!AssetDatabase.IsValidFolder(path)) { AssetDatabase.CreateFolder("Assets", "Resources"); }
```

- `**string path = "Assets/Resources";**` defines the directory where the asset will be saved.
- `**string assetPath = path + "/Keyboard Control Data.asset";**` specifies the full path and filename for the asset.
- The script checks if the `Resources` folder exists using `**AssetDatabase.IsValidFolder(path)**`. If the folder does not exist, it creates the folder with `**AssetDatabase.CreateFolder("Assets", "Resources");**`.

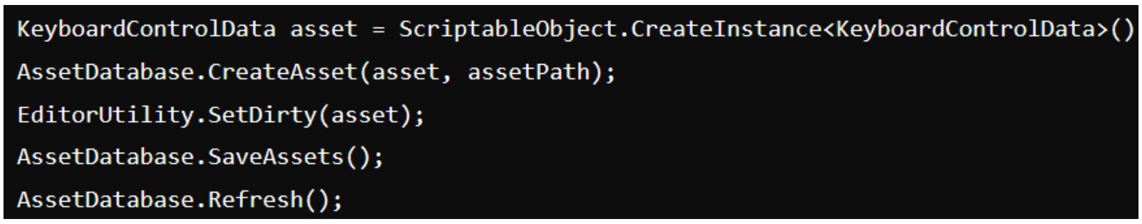
5. Checking for Existing Asset



```
if (AssetDatabase.LoadAssetAtPath<KeyboardControlData>(assetPath) != null)
{

    if (!EditorUtility.DisplayDialog("Replace File", "There is already a 'Keyboard
    {
        return;
    }
}
```

- The script checks if an asset with the specified path already exists using `**AssetDatabase.LoadAssetAtPath<KeyboardControlData>(assetPath)**`.
- If an asset is found, it displays a confirmation dialog using `**EditorUtility.DisplayDialog**` to ask the user if they want to replace the existing asset. If the user selects "No", the method returns without creating a new asset.

6. Creating and Saving the Asset



```
KeyboardControlData asset = ScriptableObject.CreateInstance<KeyboardControlData>()
AssetDatabase.CreateAsset(asset, assetPath);
EditorUtility.SetDirty(asset);
AssetDatabase.SaveAssets();
AssetDatabase.Refresh();
```

- The script creates a new instance of `**KeyboardControlData**` using `**ScriptableObject.CreateInstance<KeyboardControlData>()**`.
- It then saves this new instance as an asset at the specified path with `**AssetDatabase.CreateAsset(asset, assetPath)**`.

- `**EditorUtility.SetDirty(asset)**` marks the asset as dirty, ensuring any changes are saved.
- `**AssetDatabase.SaveAssets()**` saves all modified assets to disk.
- `**AssetDatabase.Refresh()**` refreshes the AssetDatabase to reflect the changes in the Unity Editor.

7. Focusing and Selecting the New Asset

```
EditorUtility.FocusProjectWindow();
Selection.activeObject = asset;
```

- `**EditorUtility.FocusProjectWindow()**` brings the Project window to the front.
- `**Selection.activeObject = asset**` selects the newly created asset in the Project window, making it easy for the user to locate and interact with it.

Conclusion

The `**KeyboardControlDataCreator**` script streamlines the process of creating a `**KeyboardControlData**` asset in the Unity Editor. By providing a custom menu item, it simplifies the workflow for developers who need to generate and manage keyboard control configurations. The script ensures the necessary folders are in place, handles potential conflicts with existing assets, and makes the newly created asset easily accessible in the Project window.

KeyboardTagHelper

The `**KeyboardTagHelper**` script provides utility functions for managing `**KeyboardControlData**`, a ScriptableObject containing a list of `**InputData**` for keyboard input configurations. This helper class includes methods for retrieving, setting, saving, and loading keyboard control data. Additionally, there are two serializable classes for storing and retrieving data in a structured format. Below is a detailed explanation of each part of the script:

Script Breakdown

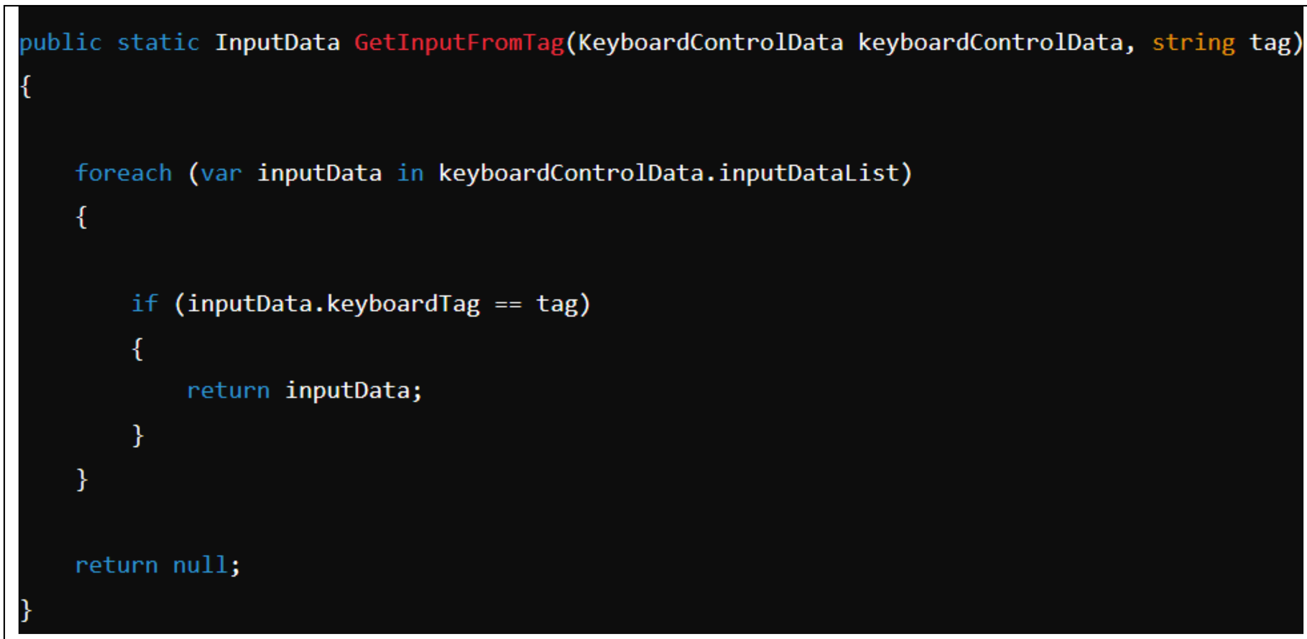
1. Namespace and Class Declaration

```
using System.Collections.Generic;
using UnityEngine;

public static class KeyboardTagHelper
{
```

The script starts by importing necessary namespaces:
- `**System.Collections.Generic**` for using lists.
- `**UnityEngine**` for core Unity functionalities.

The `**KeyboardTagHelper**` class is declared as a static class, indicating that it contains only static methods and cannot be instantiated.

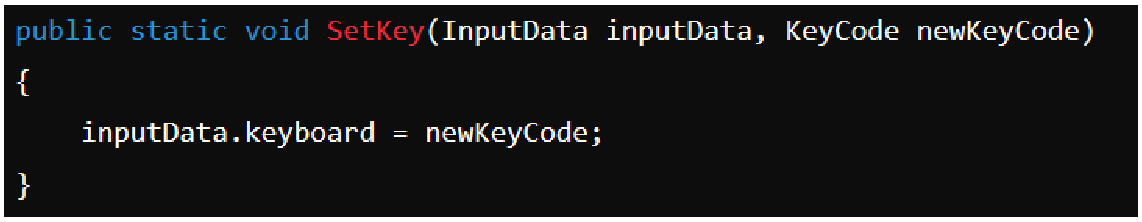
2. GetInputFromTag Method



```
public static InputData GetInputFromTag(KeyboardControlData keyboardControlData, string tag)
{

    foreach (var inputData in keyboardControlData.inputDataList)
    {

        if (inputData.keyboardTag == tag)
        {
            return inputData;
        }
    }

    return null;
}
```

- This method takes a `**KeyboardControlData**` object and a string `**tag**`.
- It iterates through the `**inputDataList**` in the `**keyboardControlData**` and returns the `**InputData**` object whose `**keyboardTag**` matches the specified tag.
- If no matching `**InputData**` is found, the method returns `**null**`.

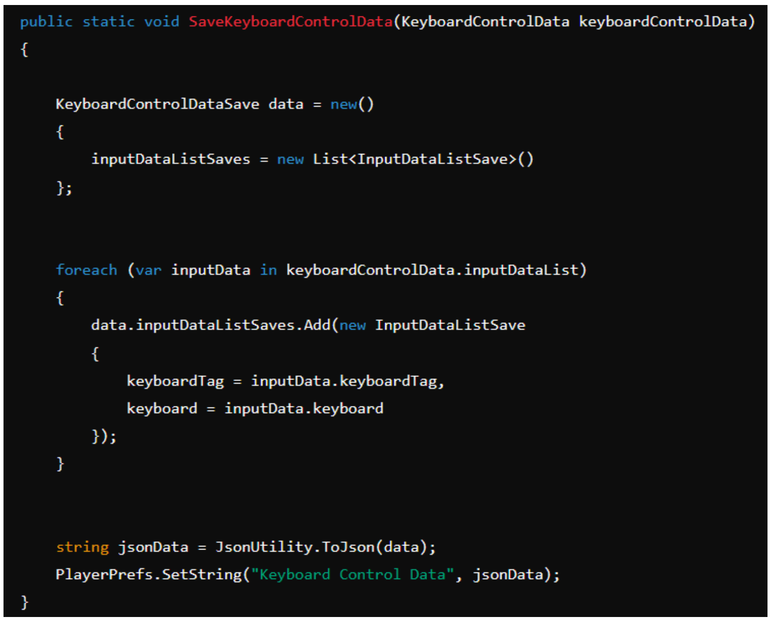
3. SetKey Method



```
public static void SetKey(InputData inputData, KeyCode newKeyCode)
{
    inputData.keyboard = newKeyCode;
}
```

- This method sets a new `**KeyCode**` for the given `**InputData**`.
- It takes an `**InputData**` object and a `**KeyCode**` and updates the `**keyboard**` property of the `**InputData**` object with the new `**KeyCode**`.

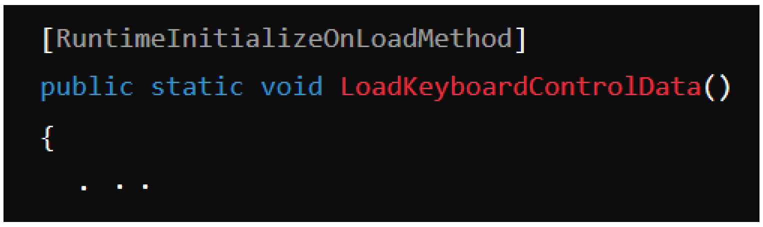
4. SaveKeyboardControlData Method



```
public static void SaveKeyboardControlData(KeyboardControlData keyboardControlData)
{

    KeyboardControlDataSave data = new()
    {
        inputDataListSaves = new List<InputDataListSave>()
    };


    foreach (var inputData in keyboardControlData.inputDataList)
    {
        data.inputDataListSaves.Add(new InputDataListSave
        {
            keyboardTag = inputData.keyboardTag,
            keyboard = inputData.keyboard
        });
    }


    string jsonData = JsonUtility.ToJson(data);
    PlayerPrefs.SetString("Keyboard Control Data", jsonData);
}
```

- This method saves the `**KeyboardControlData**` to `**PlayerPrefs**`.
- It creates a new instance of `**KeyboardControlDataSave**` and populates its `**inputDataListSaves**` property with converted `**InputData**` objects.
- Each `**InputData**` is converted to an `**InputDataListSave**` object and added to the list.
- The list is then serialized to JSON format using `**JsonUtility.ToJson**` and stored in `**PlayerPrefs**` with the key `**"Keyboard Control Data"**`.

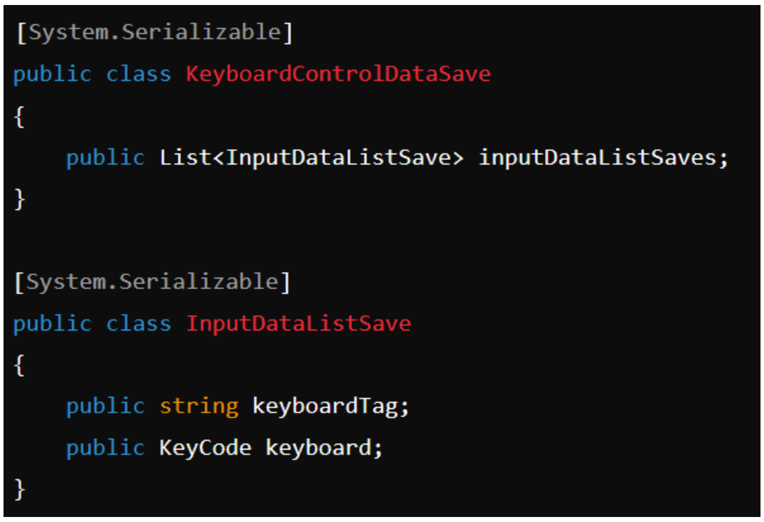
5. LoadKeyboardControlData Method



```
[RuntimeInitializeOnLoadMethod]
public static void LoadKeyboardControlData()
{
    . . .
```

- This method loads the `**KeyboardControlData**` from `**PlayerPrefs**` and updates the existing `**KeyboardControlData**` instance.
- It is marked with `**[RuntimeInitializeOnLoadMethod]**`, which ensures it runs when the game starts.
- It checks if `**PlayerPrefs**` contains data for the key `**"Keyboard Control Data"**`.
- If data exists, it loads and deserializes it from JSON using `**JsonUtility.FromJson**`.
- It then loads the `**KeyboardControlData**` instance from the `**Resources**` folder.
- If the instance is found, it updates its `**inputDataList**` with the deserialized data.
- If no matching `**InputData**` is found for a tag, an error message is logged.

6. Serializable Classes



```
[System.Serializable]
public class KeyboardControlDataSave
{
    public List<InputDataListSave> inputDataListSaves;
}

[System.Serializable]
public class InputDataListSave
{
    public string keyboardTag;
    public KeyCode keyboard;
}
```

- These two classes are marked with `**[System.Serializable]**`, allowing them to be serialized and deserialized.
- `**KeyboardControlDataSave**` contains a list of `**InputDataListSave**` objects.
- `**InputDataListSave**` stores the `**keyboardTag**` and `keyboard` properties of each `**InputData**` object.

Conclusion

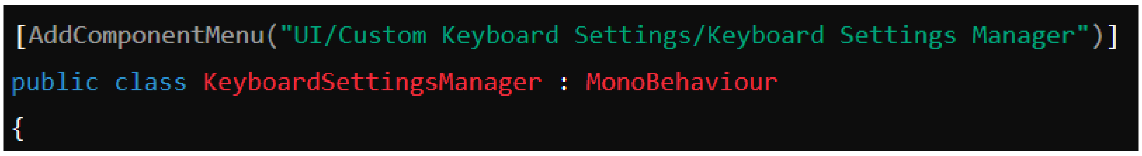
The `**KeyboardTagHelper**` script provides essential utility functions for managing keyboard control configurations in Unity. It includes methods to retrieve, set, save, and load keyboard input data, facilitating easy management of input configurations. The use of serialization allows for the persistence of input data across sessions, making it a powerful tool for developers working on projects with customizable keyboard controls.

KeyboardSettingsManager and TMP_KeyboardSettingsManager

The `KeyboardSettingsManager` script is a Unity MonoBehaviour designed to manage keyboard input settings through a UI interface. It allows users to select, reset, and save key bindings for game controls. Below is a detailed breakdown of the script's functionality:

Script Breakdown

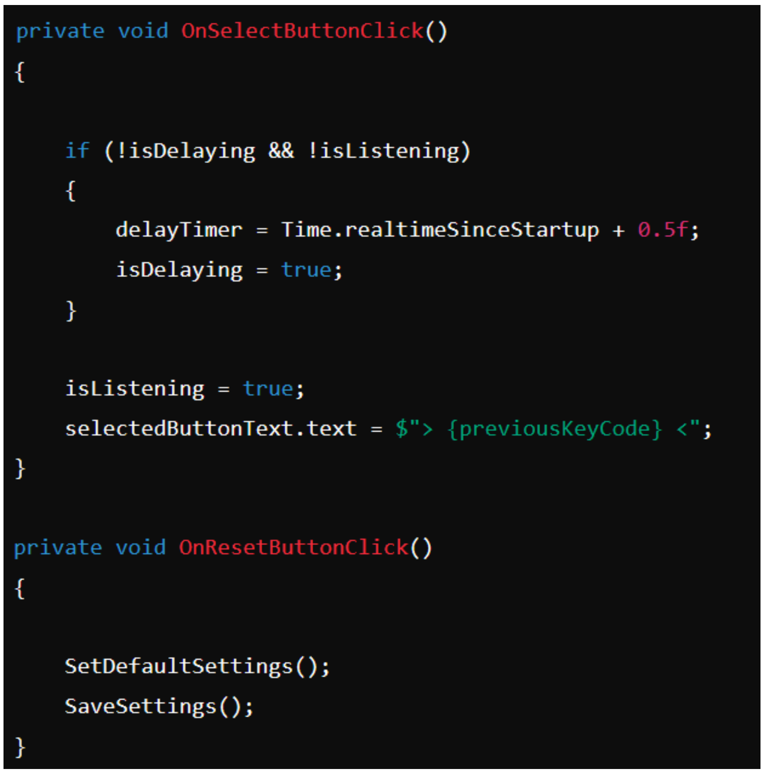
1. Class Declaration and Fields



```
[AddComponentMenu("UI/Custom Keyboard Settings/Keyboard Settings Manager")]
public class KeyboardSettingsManager : MonoBehaviour
{
```

- **[AddComponentMenu]**: Adds the script to the Unity component menu under "UI/Custom Keyboard Settings/Keyboard Settings Manager", making it easily attachable to GameObjects.
- **Serialized Fields**:
  - `selectButton`, `selectedButtonText`, `resetButton`: UI elements for selecting and displaying the key code, and for resetting to default.
  - `defaultKeyCode`: The default key binding.
  - `inputData`: An instance of `InputData` to hold the current key settings.
  - `keyboardTag`: A string tag used to identify the key binding in `KeyboardControlData`.
  - `keyboardControlData`: The `KeyboardControlData` instance used for saving and loading settings.
- **Private Fields**:
  - `currentKeyCode`, `previousKeyCode`: To keep track of the currently selected and previously selected key codes.
  - `isListening`: A flag to indicate if the script is currently waiting for new keyboard input.
  - `otherManagers`: A list to keep track of other `KeyboardSettingsManager` instances to prevent key conflicts.
  - `delayTimer`, `isDelaying`: To manage the delay before listening for new key inputs.

2. Event Handlers

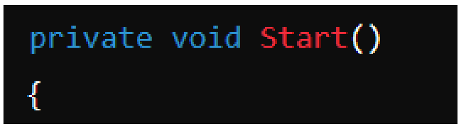


```
private void OnSelectButtonClick()
{

    if (!isDelaying && !isListening)
    {
        delayTimer = Time.realtimeSinceStartup + 0.5f;
        isDelaying = true;
    }

    isListening = true;
    selectedButtonText.text = $"> {previousKeyCode} <";
}

private void OnResetButtonClick()
{

    SetDefaultSettings();
    SaveSettings();
}
```

- **OnSelectButtonClick**: Initiates the process of listening for new key input. It sets a delay if not already delaying or listening and updates the `selectedButtonText` to show the previous key code.

- **OnResetButtonClick**: Resets the key binding to the default and saves these settings.

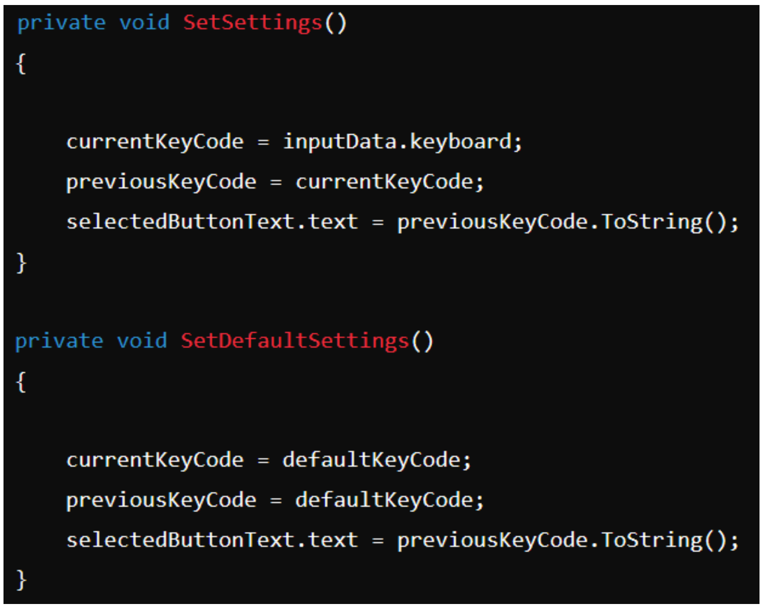
3. Initialization



```
private void Start()
{
```

- **Start**:
  - Initializes `otherManagers` with all instances of `KeyboardSettingsManager` in the scene to check for key conflicts.
  - Sets up button click listeners.
  - Loads existing settings or initializes default settings based on available data.

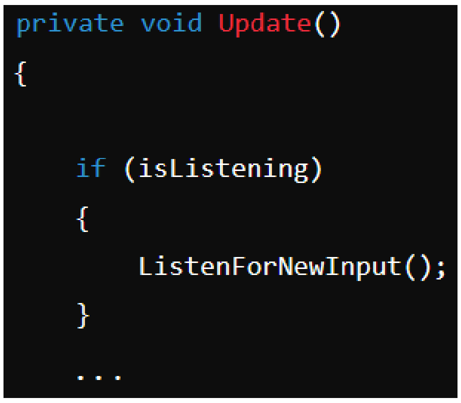
4. Set Settings Methods



```
private void SetSettings()
{

    currentKeyCode = inputData.keyboard;
    previousKeyCode = currentKeyCode;
    selectedButtonText.text = previousKeyCode.ToString();
}

private void SetDefaultSettings()
{

    currentKeyCode = defaultKeyCode;
    previousKeyCode = defaultKeyCode;
    selectedButtonText.text = previousKeyCode.ToString();
}
```

- **SetSettings**: Updates the UI and `currentKeyCode` with the key code from `inputData`.
- **SetDefaultSettings**: Resets the key code to the default value and updates the UI.

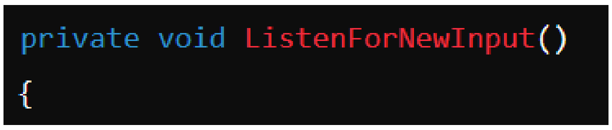
5. Update Method



```
private void Update()
{

    if (isListening)
    {
        ListenForNewInput();
    }
    ...
```

- **Update**:
  - Checks if the script is in listening mode and processes new input.
  - Updates the interactability of the reset button based on whether the current key code is different from the default.
  - Manages the delay before starting to listen for new inputs.

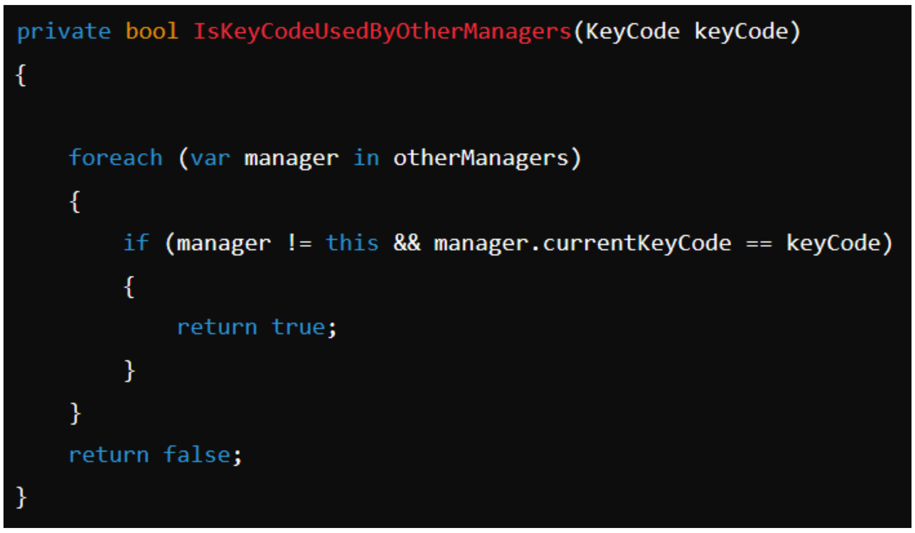
6. ListenForNewInput Method



```
private void ListenForNewInput()
{
```

- **ListenForNewInput**:
  - Checks for a new key press if `isListening` is true.
  - Verifies that the key code is not used by other `KeyboardSettingsManager` instances.
  - Updates the key code and UI, then saves the new settings.

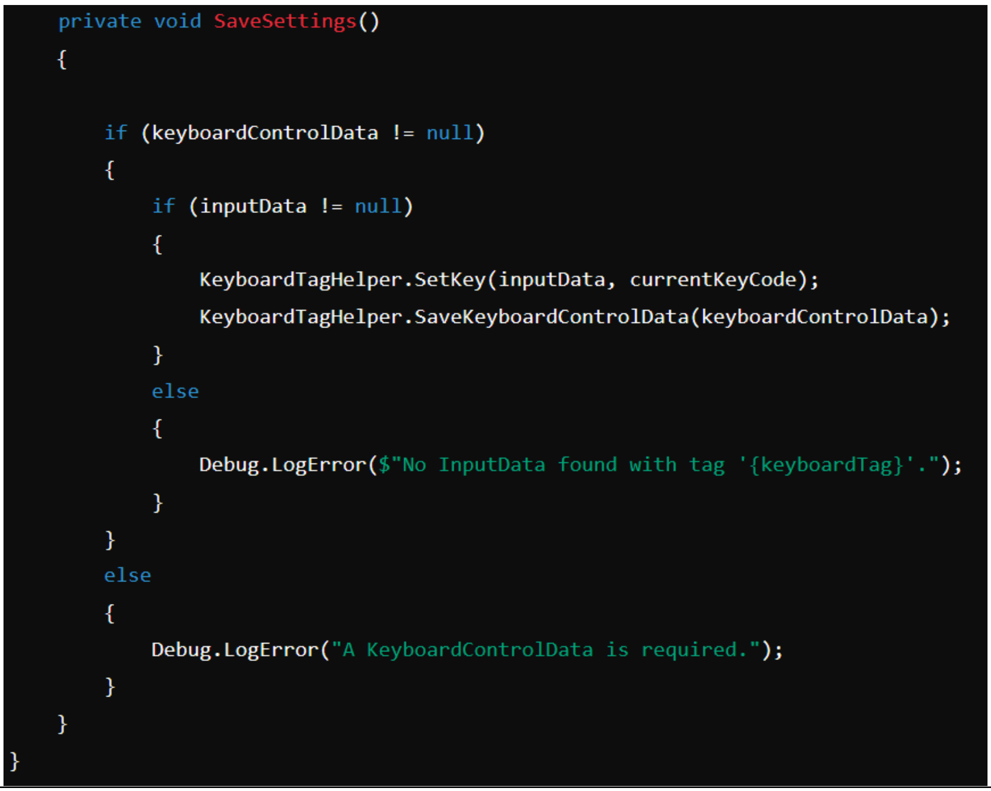
7. IsKeyCodeUsedByOtherManagers Method



```
private bool IsKeyCodeUsedByOtherManagers(KeyCode keyCode)
{

    foreach (var manager in otherManagers)
    {
        if (manager != this && manager.currentKeyCode == keyCode)
        {
            return true;
        }
    }
    return false;
}
```

- **IsKeyCodeUsedByOtherManagers**:
  - Checks if the `keyCode` is already in use by other `KeyboardSettingsManager` instances to avoid conflicts.

8. SaveSettings Method



```
    private void SaveSettings()
    {

        if (keyboardControlData != null)
        {
            if (inputData != null)
            {
                KeyboardTagHelper.SetKey(inputData, currentKeyCode);
                KeyboardTagHelper.SaveKeyboardControlData(keyboardControlData);
            }
            else
            {
                Debug.LogError($"No InputData found with tag '{keyboardTag}'.");
            }
        }
        else
        {
            Debug.LogError("A KeyboardControlData is required.");
        }
    }
}
```

- **SaveSettings**:
  - Saves the current key settings to **`KeyboardControlData`** using the **`KeyboardTagHelper`**.
  - Logs errors if **`keyboardControlData`** or `input

Data` is not found.

Summary

The **`KeyboardSettingsManager`** script is responsible for managing keyboard settings in a Unity project. It provides UI elements for selecting, displaying, and resetting key bindings. It includes functionality for detecting new key inputs, checking for conflicts with other managers, and saving the settings. The script ensures that the settings are updated and saved correctly, and it handles any conflicts or errors that may arise.