Custom Attributes

Documentation

———

Creator: Lucas Gomes Cecchini

Pseudonym: AGAMENOM

Overview

This document will help you use **Custom Attributes** for **Unity**.

With it you can customize your **Inspector**.

Just like Unity's **Attributes**, it adds functions that make it easier to use.

Such as easier selection of **Tag** or **Scene name**, **viewing properties**, etc.

Instructions

You can get more information from the Playlist on YouTube:

https://www.youtube.com/playlist?list=PL5hnfx09yM4I_6OdJvShZ0rRtYF9jv6Cd

Script Explanations

Tag Dropdown Attribute

This script is designed for Unity and allows for a custom property drawer that shows a dropdown of Unity tags in the Inspector. It uses a custom attribute (TagDropdownAttribute) and a corresponding property drawer (TagDropdownDrawer). Here's a breakdown of the script's components:

TagDropdownAttribute Class

- **Purpose**: This is a custom attribute class used to mark a property in Unity to use the custom property drawer (TagDropdownDrawer).
- **Implementation**: It doesn't implement any behavior by itself; it simply acts as a marker for the PropertyDrawer to identify which fields should use the custom dropdown.

TagDropdownDrawer Class (Custom Property Drawer)

This class is where the main functionality resides. It defines how the property will appear and behave in the Unity Inspector.

1. OnGUI Method

This method is responsible for rendering the property and handling interactions in the Inspector.

- **Parameters**:
  - position: The rectangle on the screen where the property is displayed.
  - property: The serialized property that is being drawn.
  - label: The label for the property.
- **Flow**:
  - First, the method checks if the property is of type string using property.propertyType. If it's not, it shows an error message: "Use [TagDropdown] com strings.".
  - **Fetching Tags**: It retrieves all available tags from Unity using UnityEditorInternal.InternalEditorUtility.tags.
  - **Handling the "Tag Missing" Option**:
    - If the current value of the property (property.stringValue) doesn't match any of the tags, it creates an option called "Tag Missing". The option includes the current string value in parentheses to indicate the missing tag (e.g., "Tag Missing (SomeTag)").
    - The tagList is built with this "Tag Missing" option at the beginning, followed by the list of available tags.
  - **Dropdown Display**: It uses EditorGUI.Popup to show a dropdown menu with the tags. The dropdown is populated with the tagList.
  - **Updating Property**: If a tag is selected (and not the "Tag Missing" option), it updates the serialized property.stringValue with the newly selected tag.
  - **Warning for Missing Tag**: If the "Tag Missing" option is selected, it shows a warning below the dropdown that indicates the value does not match any tag.

2. GetPropertyHeight Method

This method is used to adjust the height of the property field in the Inspector, especially if the warning message about a missing tag is shown.

- **Flow**:
  - It checks if the current property value exists in the list of Unity tags.
  - If the value does not match any tag, it adds extra height for the warning message by returning a larger height value. Otherwise, it returns the default height.

Usage

1. **TagDropdownAttribute**:
   - This attribute is applied to a string field in a Unity script to indicate that it should use the custom dropdown. For example:
   - [TagDropdown]
   - public string myTag;
2. **TagDropdownDrawer**:

- This is automatically invoked by Unity's editor when the property has the TagDropdownAttribute.
- The dropdown will display all the available tags, with the option to select "Tag Missing" if the current value doesn't match any tags. A warning message will appear if the "Tag Missing" option is selected.

Summary

- **TagDropdownAttribute** is used to mark a string property to use the tag dropdown in the Unity Inspector.
- **TagDropdownDrawer** is responsible for rendering the dropdown and handling interactions. It ensures the property is a string, shows a list of Unity tags, and provides a warning if the selected tag is missing.
- This custom drawer enhances the Unity Inspector experience by providing an intuitive way to select Unity tags while also warning when the tag is invalid.

Scene Tag Dropdown Attribute

This script defines a custom attribute and property drawer for Unity, allowing string fields in the Inspector to display a dropdown of scene names from the Editor Build Settings. The dropdown menu supports additional functionality such as warnings for missing or disabled scenes, and dynamically adjusts the UI height based on conditions. Here's a breakdown of how each part of the script works:

SceneTagDropdownAttribute Class

- **Purpose**: This is a custom attribute used to mark a string property in Unity to trigger the custom property drawer (SceneTagDropdownDrawer).
- **Implementation**: It is a simple marker with no behavior beyond notifying Unity's editor that the property should use a custom drawer.

SceneTagDropdownDrawer Class (Custom Property Drawer)

This is the core of the script, providing the logic to render and interact with the dropdown menu in the Unity Inspector.

1. OnGUI Method

This method is called to render the property and handle the user interaction in the Inspector.

- **Parameters**:
  - position: The position and size where the property is drawn on the screen.
  - property: The serialized property that this drawer is working with.
  - label: The label of the property.
- **Flow**:
  - **Check Property Type**: First, the method checks if the property is a string. If it's not, it shows an error message ("Use [SceneTagDropdown] with strings").
  - **Fetch Scene List**:
    - allScenes: The list of all scene names (without file extensions) in the build settings is retrieved.
    - enabledScenes: A filtered list of scenes that are enabled in the build settings is created.
  - **Handling "Missing Scene" Option**:
    - If the current property value (property.stringValue) doesn't match any scene, the option "Missing Scene" is displayed in the dropdown.
    - If the scene is in the build but disabled, the name of the scene is shown with the label "[Disabled]".
  - **Building Scene List**: A list of scenes is created with the "Missing Scene" option (if applicable) at the beginning, followed by all enabled scenes.
  - **Find the Current Scene**: The script checks if the current value of the property is in the list of scenes. If it is not, it selects the "Missing Scene" option.
  - **Dropdown Display**: The EditorGUI.Popup function displays the dropdown menu in the Inspector with the list of scenes.
  - **Update Property**: If a new scene is selected from the dropdown, the string value of the property is updated to reflect the selection.
  - **Warnings**:

- If "Missing Scene" is selected, and the scene exists but is disabled, a warning message appears indicating that the scene is in the build settings but disabled.
- If the selected value doesn't match any scene, an error message is shown.

2. GetPropertyHeight Method

This method is used to adjust the height of the property field in the Inspector, especially if warnings are shown.

- **Flow**:
  - **Check Property Value**: It checks if the current property value is either empty or doesn't match any scene in the build settings. If either condition is true, it increases the height to accommodate the warning.
  - **Check for Disabled Scene**: It checks if the scene exists but is disabled in the build settings. If it is, the height is also increased to make room for the warning.
  - **Default Height**: If there are no warnings, the default height for the property is used.

Usage

1. **SceneTagDropdownAttribute**: This custom attribute is applied to a string field in a Unity script to indicate that it should use the custom scene dropdown. For example:
2. [SceneTagDropdown]
3. public string sceneName;
4. **SceneTagDropdownDrawer**: Unity automatically uses this drawer for any field with the SceneTagDropdownAttribute. It will display a dropdown of all enabled scenes in the build settings, allow the user to select one, and show warnings if the selected scene is missing or disabled.

Key Features

- **Dropdown List**: The dropdown shows a list of scene names, sourced from Unity's build settings.
- **Warnings**: The script provides feedback if the selected scene is missing or disabled in the build.
- **Dynamic Height**: The UI adjusts its height based on whether a warning message needs to be displayed, ensuring that there is enough space in the Inspector to show all necessary information.

Summary

- **SceneTagDropdownAttribute**: Marks a property to use the custom scene dropdown.
- **SceneTagDropdownDrawer**: Handles rendering the dropdown, updating the property value, and showing warnings for missing or disabled scenes. It also adjusts the height of the property field to accommodate any warnings.

Read Only Attribute

This script defines a custom attribute and corresponding property drawer in Unity that allows you to make a field read-only in the Inspector, preventing users from editing it while still displaying its value. Here's an explanation of how each part of the script works:

ReadOnlyAttribute Class

- **Purpose**: This class defines a custom attribute named ReadOnlyAttribute. It acts as a marker to indicate that a field should be displayed as read-only in the Unity Inspector. It does not contain any logic or behavior by itself.
- **Usage**: This attribute is applied to a field in Unity scripts to make that field non-editable in the Inspector, while still allowing the field's value to be visible. For example, you might use it like this:
- [ReadOnly]
- public int myField;

ReadOnlyDrawer Class (Custom Property Drawer)

This is the class that controls how the ReadOnlyAttribute is rendered in the Unity Inspector. The PropertyDrawer ensures that the field is displayed in a way that prevents edits.

1. OnGUI Method

The OnGUI method is overridden to control how the field with the ReadOnlyAttribute is displayed in the Inspector.

- **Parameters**:

    - position: Specifies where the property should be displayed in the Inspector, including its size.
    - property: The serialized property that this drawer is rendering. This contains the actual data of the field.
    - label: The label or name of the property in the Inspector.
- **Flow**:
    - **Disable Editing**: The GUI.enabled = false; line disables all user interactions with the property, making it read-only. No input from the user can change the value of the field.
    - **Render the Property**: The EditorGUI.PropertyField(position, property, label); function draws the field in the Inspector. It displays the property's current value but does not allow modification because the GUI is disabled.
    - **Restore GUI State**: The GUI.enabled = true; line restores the normal GUI state, allowing further interactions with other properties in the Inspector that are not read-only.

2. Effect of ReadOnly
By setting GUI.enabled to false, the property becomes visually disabled in the Inspector, meaning the user can view its value but cannot modify it. Once the field is rendered, the GUI state is restored to ensure that other properties are not affected by the read-only behavior.

Usage
1. **ReadOnlyAttribute**: To use this functionality, you apply the ReadOnly attribute to any field in a Unity script. The field will then be displayed as read-only in the Inspector. For example:
2. [ReadOnly]
3. public int score;
4. **ReadOnlyDrawer**: The ReadOnlyDrawer automatically takes over the rendering of any field marked with the ReadOnlyAttribute in the Unity Inspector. It ensures that such fields are displayed as read-only.

Summary
- **ReadOnlyAttribute**: Marks a property as read-only, preventing editing in the Unity Inspector.
- **ReadOnlyDrawer**: Controls how the ReadOnlyAttribute is rendered, ensuring that the property is displayed without allowing edits. It disables the GUI state during the property field rendering and restores it afterward to avoid affecting other fields.

This script is useful for displaying data that should not be modified, such as internal values or constants, while still giving users the ability to view the information in the Inspector.

Highlight Empty Reference Attribute

This script defines a custom attribute and property drawer in Unity to highlight fields with null references in the Inspector. When applied to a field, the script visually marks those fields with a red background and provides an error message to help with debugging. Here's a detailed explanation of how each part of the script works:

HighlightEmptyReferenceAttribute Class
- **Purpose**: This class defines a custom attribute called HighlightEmptyReferenceAttribute. It serves as a marker to indicate that a property should be highlighted if it contains a null reference.
- **Implementation**: The attribute itself doesn't contain any behavior or logic, it is only used to identify which fields should be handled by the custom property drawer (HighlightEmptyReferenceDrawer).
- **Usage**: To use this attribute, it is applied to fields in a Unity script that are object references and can potentially be null. For example:
- [HighlightEmptyReference]
- public GameObject myGameObject;

HighlightEmptyReferenceDrawer Class (Custom Property Drawer)
This is the class that defines how properties marked with HighlightEmptyReferenceAttribute should be displayed in the Unity Inspector. It provides the logic to highlight null references and display an error message.

1. OnGUI Method

The OnGUI method is responsible for rendering the property in the Inspector and modifying the appearance based on whether the reference is null.

- **Parameters**:
  - position: Specifies where the property should be drawn on the screen, including its size.
  - property: The serialized property being drawn, which contains the actual data of the field.
  - label: The label for the property in the Inspector.
- **Flow**:
  - **Save the Current Background Color**: The script saves the current background color to restore it later. This ensures that only the highlighted property is affected.
  - **Check for Null Reference**: The script checks if the property is an object reference (SerializedPropertyType.ObjectReference) and if its value is null (property.objectReferenceValue == null).
  - **Change Background Color**: If the property is null, the background color is set to red. Otherwise, it restores the previous background color.
  - **Render the Property**: The EditorGUI.PropertyField method is used to draw the property in the Inspector, using the defined position and label.
  - **Restore Background Color**: After drawing the property field, the background color is reset to its previous state to ensure that only this property is highlighted.
  - **Show Error Message**: If the property is null, the script determines the expected type of the field and displays an error message in a help box below the property. This message tells the user to assign an object of the correct type to the field.
    - If the reference is null, the script attempts to retrieve the type of the field from the target object or the serialized property.
    - The message is displayed using EditorGUI.HelpBox, and it provides guidance on what type of object should be assigned to the field.

2. GetPropertyHeight Method

This method is responsible for determining the height of the property field in the Inspector, especially when the help box for the error message needs to be displayed.

- **Flow**:
  - The method checks if the property is empty (i.e., if it is an object reference and its value is null).
  - If the property is null, it returns a height that accommodates both the property field and the help box that displays the error message. This ensures the Inspector has enough space for the warning message.
  - If the property is not null, it returns the default height for a single line of the property field.

Usage

1. **HighlightEmptyReferenceAttribute**: The HighlightEmptyReference attribute is applied to fields that can have null references and that you want to highlight in the Inspector. For example, you can use it to mark fields that require a reference to a GameObject or a ScriptableObject:
2. [HighlightEmptyReference]
3. public GameObject player;
4. **HighlightEmptyReferenceDrawer**: The HighlightEmptyReferenceDrawer automatically handles the rendering of fields marked with the HighlightEmptyReferenceAttribute. It highlights the background of the field in red when it is null and displays a helpful error message beneath the field to guide the user in assigning a valid reference.

Summary

- **HighlightEmptyReferenceAttribute**: Marks a property to be visually highlighted when its value is null in the Unity Inspector.
- **HighlightEmptyReferenceDrawer**: Customizes the rendering of the marked property in the Inspector by changing the background color to red and displaying an error message when the property is null. It also adjusts the height of the property to accommodate the error message.

This script is useful for debugging and improving the user experience by ensuring that developers are aware of missing or incorrect references in the Inspector. It helps to prevent errors caused by unassigned references by providing immediate visual feedback.

Conditional Hide Attribute

This script defines a custom attribute, ConditionalHideAttribute, and a corresponding PropertyDrawer, ConditionalHidePropertyDrawer, in Unity. The purpose is to conditionally hide properties in the Unity Inspector based on the values of other fields, allowing for more dynamic and context-sensitive UI rendering. Here's a detailed explanation of how the script works, covering each variable, method, and its usage:

ConditionalHideAttribute Class

This class defines a custom attribute that is used to mark properties in Unity that should be hidden based on the values of other properties. The attribute can be configured to hide a property depending on the value of one or more fields.

1. ConditionalSourceFields:

- This is an array of strings that specifies the names of the properties that will be used to determine whether or not the property should be hidden.
- For example, if you want to hide a field based on the value of another boolean field, the name of the boolean field will be listed here.

2. HideIfAnyFalse:

- A boolean flag that determines whether the property should be hidden if *any* of the conditional source fields are false. If this is true, the property will be hidden if any of the conditions are false; otherwise, all conditions must be true for the property to be visible.

3. Constructors:

- There are two constructors in the ConditionalHideAttribute class:
  - The first constructor accepts an array of field names and hides the property if more than one condition exists and any of them are false.
  - The second constructor allows specifying whether the property should be hidden if any condition is false (via the hideIfAnyFalse flag) or only if all conditions are false.

ConditionalHidePropertyDrawer Class

This class defines how properties with the ConditionalHideAttribute should be rendered in the Unity Inspector. It handles the logic to conditionally hide properties based on the conditions set in the attribute.

1. OnGUI Method:

- **Parameters**:
  - position: Defines the area where the property will be drawn in the Inspector.
  - property: Represents the actual property being drawn.
  - label: The label of the property that will be displayed next to the property field in the Inspector.
- **Flow**:
  - The OnGUI method starts by retrieving the ConditionalHideAttribute applied to the current property.
  - Based on the attribute settings (ConditionalSourceFields and HideIfAnyFalse), the script checks if the property should be hidden or displayed. It does this by calling one of the condition-checking methods (CheckSingleCondition, CheckAllConditions, or CheckAnyCondition).
  - If the condition to hide the property is met, the property is not drawn (return is called early). Otherwise, the property is rendered using EditorGUI.PropertyField.

2. GetPropertyHeight Method:

- This method controls the height of the property field in the Inspector.
- **Flow**:
  - Similar to the OnGUI method, it first checks whether the property should be hidden by evaluating the conditions in the ConditionalHideAttribute.
  - If the property is hidden, the height is returned as 0 to prevent any space being allocated for the property.
  - If the property is not hidden, the default height for the property is returned using EditorGUI.GetPropertyHeight.

3. CheckSingleCondition Method:

- This method checks if a single condition (property) should hide the field.
- **Flow**:
  - It retrieves the condition property using GetConditionProperty.
  - If the condition property is null or is not a boolean, it assumes the condition is false (hiding the property).
  - If the condition is a boolean and it evaluates to false, the property will be hidden.

4. CheckAllConditions Method:
- This method checks multiple conditions and hides the property if *any* condition is false.
- **Flow**:
  - It iterates through the list of conditional source fields and checks each one.
  - If any of the condition properties are false (or not boolean), it returns true to indicate the property should be hidden.
  - If all conditions are true, it returns false to allow the property to be displayed.

5. CheckAnyCondition Method:
- This method checks if *any* of the conditions are true (hiding the property only if all conditions are false).
- **Flow**:
  - It iterates through the conditional source fields and checks each condition.
  - If any condition is true, it returns false (the property should be displayed).
  - If all conditions are false, it returns true (the property should be hidden).

6. GetConditionProperty Method:
- This method retrieves the property corresponding to the condition field.
- **Flow**:
  - It tries to find the property directly by its name.
  - If the property is nested (e.g., part of a structure or array), it splits the property name into parts and navigates through the object to find the correct property.
  - The method returns the SerializedProperty object that represents the condition, or null if the property cannot be found.

Usage
To use this system in Unity, you apply the ConditionalHideAttribute to any property that you want to conditionally hide in the Inspector. You pass in the names of the fields that will control the visibility of the property.

Example Usage:
```
public class MyScript : MonoBehaviour
{
   public bool showExtraOptions; // Controls visibility of the next field.

   [ConditionalHide("showExtraOptions")]
   public string additionalInfo; // This will only be visible if `showExtraOptions` is true.
}
```

Example with Multiple Conditions:
```
public class MyScript : MonoBehaviour
{
   public bool isAdmin; // Controls visibility of the next field.
   public bool isEditor; // Controls visibility of the next field.

   [ConditionalHide(true, "isAdmin", "isEditor")]
   public string secretSettings; // This will only be visible if both `isAdmin` and `isEditor` are true.
}
```

Summary
- **ConditionalHideAttribute**: Marks properties that should be conditionally hidden based on other properties. The conditions can either require all specified properties to be true or just one.

- **ConditionalHidePropertyDrawer**: Controls how properties with ConditionalHideAttribute are drawn in the Inspector. It hides or shows the properties based on the conditions defined in the attribute.
- **Methods for Condition Checking**: Methods like CheckSingleCondition, CheckAllConditions, and CheckAnyCondition handle the logic of evaluating the conditions set by the attribute and deciding whether to display the property.

This script improves the user experience in the Unity Inspector by allowing you to show or hide properties dynamically based on other properties, keeping the interface cleaner and more relevant.

Conditional Hide Attribute

This script provides a custom solution for Unity Editor that enables developers to add interactive buttons in the Unity Inspector to execute parameterless methods directly from the editor. It's particularly useful for debugging or quickly triggering game logic during development.

Usage Overview

To use this system, a developer simply needs to apply the [Button] attribute above a method within a MonoBehaviour. For example:

```
[Button("Do Something")]
void DoSomething() { ... }
```

When this script is attached to a GameObject, Unity's inspector will display a button labeled "Do Something", which, when clicked, will execute that method in edit mode.

Explanation of Components

1. ButtonAttribute (Custom Attribute)

This class defines a custom attribute that can be used to mark methods which should appear as buttons in the Unity Inspector.

- **Purpose**: Marks a method so the editor knows it should create a button for it.
- **Label (string)**: Optional text that determines what the button will display. If not provided, the method's name will be transformed into a readable label.
- **Constructor**: Stores the label string (or null if none is given).

2. ButtonDrawerEditor (Custom Editor - Only Compiled in Editor)

This class customizes the Unity Inspector interface for all MonoBehaviour components.

Class Attributes

- **[CustomEditor(typeof(MonoBehaviour), true)]**: Makes this editor apply to all MonoBehaviour-derived scripts.
- **[CanEditMultipleObjects]**: Allows this editor to be used even when multiple objects are selected.

Methods Inside ButtonDrawerEditor

- **OnInspectorGUI()**
  - Called by Unity to draw the custom inspector.
  - Calls DrawButtons() to render the buttons for marked methods.
  - Then draws the default inspector (usual fields in Unity).
- **DrawButtons(targetObject)**
  - Uses *reflection* to find all parameterless methods in the target object that are marked with [Button].
  - Filters these methods to only those that:
    - Belong to the current MonoBehaviour.
    - Are marked with the ButtonAttribute.
    - Have **no parameters** (so they can be safely invoked from the editor).
  - For each such method:
    - Retrieves the label from the attribute, or falls back to a prettified version of the method name.
    - Draws a button in the inspector.
    - If the button is clicked:

    - Uses Unity's **Undo system** to track changes.
    - Invokes the method via reflection.
    - Marks the object as **dirty**, signaling that it was modified (helps save changes in scenes or prefabs).

How It Works Together

1. When Unity loads the editor:
    - It checks all MonoBehaviour scripts for methods with [Button].
    - It replaces the default inspector with one that includes these buttons (as well as the regular fields).
2. During design time:
    - The developer sees buttons for all eligible methods.
    - Clicking a button will instantly invoke the corresponding method.

Benefits and Use Cases

- **Debugging**: Trigger game logic without needing UI or entering play mode.
- **Setup/Reset Helpers**: Quickly reset object states or generate data in the editor.
- **Convenience**: Call development-only tools directly from the Inspector.