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

🌟 What This Script Does

The purpose of this script is to let developers choose **Unity tags from a dropdown menu** instead of manually typing the tag name into a string field.
Adding this:

```
[TagDropdown]
```

above a string field makes Unity show:

- A dropdown listing all available tags
- A warning if the string does not match any existing tag
- Automatic adjustment of the field height when warnings appear
- Full support for multi-object editing

This makes tag selection faster, safer, and prevents typos.

🧩 The Attribute: TagDropdownAttribute

What it is
It is a small marker that tells Unity Editor:
“Whenever this attribute is on a string field, use the special drawer to draw it.”
Variables
The attribute contains **no variables**.
Methods
The attribute contains **no methods**.
It is simply a tag that activates the custom drawer.

🧩 The Drawer: TagDropdownDrawer

This is where all the logic lives.
It controls how the field looks in the Inspector.
The drawer contains two major methods:

1. A method that draws the field
2. A method that tells Unity how tall the field should be

🎨 Method: OnGUI

This method draws the dropdown menu and handles all Inspector logic.
It receives:

- **position** → The area where the field will appear
- **property** → The string variable being edited
- **label** → The text shown next to the field

🔍 Detailed Explanation of What Happens Inside

1️⃣ Ensure the field is a string

If the field is not a string type, the drawer displays a message telling the user the attribute is only for strings.

2️⃣ Begin property handling

This ensures:

- Prefab overrides work correctly
- Multi-object editing behaves properly

Unity requires this wrapper for custom drawers.

3️⃣ Retrieve all Unity tags

The drawer collects the list of all tags currently defined in the **Tags and Layers** window.
This becomes the basis for the dropdown list.
Variable involved:

- **tags** → The complete list of existing Unity tags

4️⃣ Collect information about the current field value
Two variables are used here:
- **hasMultipleDifferentValues**
  Checks whether the user selected multiple objects with different values
- **currentString**
  The current value of the field (the tag name stored in the object)

5️⃣ Determine if the tag is missing
If the current string does not exist in the tags list, the drawer builds a text explaining that the tag is missing.
Variables involved:
- **currentText**
  This is a formatted version of the missing tag text.
  If the string is empty → “Tag Missing”
  If the string has text → “Tag Missing (value)”
- **missingTagText**
  This is empty if the tag exists, or the missing text if it doesn't.

6️⃣ Build the dropdown list
The dropdown list always begins with a special item:
- The first item is the “missing tag” indicator
- The second item onward contains all actual Unity tags

Variable:
- **tagList** → The list that will be shown in the dropdown menu

7️⃣ Determine which list item is currently selected
Variable:
- **currentIndex**
  This is the index in the dropdown list that corresponds to the current string.

If the tag is missing, the index becomes 0 (the missing tag entry).

8️⃣ Prepare the dropdown menu
Unity requires each item to be wrapped in a GUI-friendly structure.
Variable:
- **options** → The list of elements used to draw the dropdown

9️⃣ Draw the dropdown
The menu is shown in the Inspector.
The drawer also enables Unity’s "mixed value" display if multiple objects have different tag values.
Variable:
- **newIndex**
  This is the index the user selects from the list.

🔟 Apply the new value when appropriate
If the user selects:
- A valid tag (not index 0)
- And it differs from the previous selection

Then the string field is updated to match the selected tag.

1️⃣1️⃣ Show warning when the selected value is invalid
If the dropdown’s current index is the “missing tag” entry and all selected objects share the same value, a warning appears below the dropdown.
This warning informs the user:
“String value does not match any tag!”
A help box is drawn below the field.

📏 Method: GetPropertyHeight

This method informs Unity how tall the entire field should be in the Inspector.
How it decides the height

- If the current string does not match any existing tag
  → The field height becomes larger to fit the warning box
- Otherwise
  → The height remains the default size

Variable used:

- **tags** → The list of Unity tags used to determine if the string is valid

✓ All Important Variables Explained

Variable	Description
tags	List of all Unity tags available in the project.
hasMultipleDifferentValues	True if multiple selected objects have different values.
currentString	The string stored in the field (current tag name).
currentText	Text explaining the missing tag state.
missingTagText	Final missing-tag message used in the list.
tagList	The full list shown in the dropdown (first item is the status).
currentIndex	The index that corresponds to the current tag in the dropdown.
options	The list formatted for the Unity dropdown UI.
newIndex	The index selected by the user in the dropdown.

🎉 Usage
To use this feature in a MonoBehaviour or ScriptableObject, simply place the attribute above a string field:

```
[TagDropdown]
public string tagName;
```

This gives the field:

- A dropdown with all Unity tags
- Automatic validation
- A warning when the selected tag does not exist
- Proper height adjustments
- Full support for selecting multiple objects

Scene Tag Dropdown Attribute

🌟 Overview of What the Script Does
This script creates:

1. **A custom attribute** called **SceneTagDropdown**
   – When you place it above a string field in the inspector, that field turns into a dropdown menu.
2. **A custom drawer**
   – This is what actually draws the dropdown menu in the Unity Inspector.
   – It lists all scenes that exist in the **Editor Build Settings**.
   – It also shows warnings when:
   - The scene is missing,
   - The scene exists but is disabled in the build settings,
   - Or the string value doesn't match any scene.
3. **Auto-height adjustment**
   – If a warning or error appears, the field becomes taller automatically.

📌 How You Use It in Your Scripts
You simply add the attribute above a string field:

```
[SceneTagDropdown]
public string myScene;
```

Now that string field shows a dropdown with all scenes from your Build Settings.

🧩 Attribute: SceneTagDropdownAttribute

Purpose
This is a “marker attribute.”
It doesn’t contain logic.
Its only job is to tell Unity:
“The drawer SceneTagDropdownDrawer should handle this field.”
Variables
It has **no variables**.
Methods
It has **no methods**.

🧩 Drawer: SceneTagDropdownDrawer

This is where all the actual logic happens.
It customizes how the field looks in the Inspector.

🎨 Method: OnGUI

This method is responsible for **drawing the field** in the Inspector.
It receives:

- **position** → The area where the field will be drawn.
- **property** → The serialized field being edited.
- **label** → The label displayed next to the field.

🔍 What the method does step by step

1️⃣ Check if the field is a string

If the field is not a string, a message appears telling the user the attribute is only for strings.

2️⃣ Start property handling

This ensures:

- Prefab overrides work,
- Multi-object editing works.

3️⃣ Load all scenes from Build Settings

Two important lists are created:
allScenes
A list of *all* scene names from Build Settings.
enabledScenes
A list of scenes that are actually enabled (checkbox checked).
These lists are used to populate the dropdown.

4️⃣ Check if multiple objects are selected

If the user selected multiple objects and they have different string values, Unity enables the “mixed value” state (displaying “—”).
Variable involved:

- **hasMultipleDifferentValues**

5️⃣ Get the current value

Variable:

- **currentString**

This is the string stored in the field.

6️⃣ Handle missing or disabled scenes

The script compares the stored string with the existing scenes.
It prepares:

- A label describing the **missing** scene,
- A label describing a **disabled** scene,
- A system fallback for empty fields.

These texts go into:

- **missingScene**
- **missingText**

Then it creates:

- **sceneList** → The list used by the dropdown.
  The first item in the list is always the “status” item (missing/disabled message).

7️⃣ Find the index currently selected

Variable:

- **currentIndex**

If the current value is not found inside the dropdown list, index 0 is used (the status item).

8️⃣ Display the dropdown

Variables:

- **options** → The dropdown entries converted to GUI-friendly elements.
- **newIndex** → The option the user selects.

If the user chooses a valid option, the string field is updated.

9️⃣ Show warnings or errors

This happens when values are consistent (no mixed selection).
Three situations are checked:
1. Scene exists but is disabled
→ A warning box appears.
2. Scene name doesn't exist anywhere
→ An error box appears.
3. Everything OK
→ No help box shown.
The boxes are drawn **below** the main field.

🔚 End of OnGUI

The method finishes handling the property and exits.

📏 Method: GetPropertyHeight

This method tells Unity how tall the property should be.
Why is this needed?
Because sometimes a warning box or error box must be shown, so the field needs extra height.
Logic used:

- If the string is empty → extra height
- If the string does not match any scene → extra height
- If the string matches a disabled scene → extra height
- Otherwise → normal height

It uses **EditorGUIUtility.singleLineHeight** to calculate these sizes.

✓ Summary of All Important Variables

Variable Name	Purpose
allScenes	List of all scene names from Build Settings.
enabledScenes	List of scenes that are enabled in the Build Settings.
hasMultipleDifferentValues	Detects multi-object selection differences.
currentString	The value stored in the string field.
currentText	Helper text for missing scene formatting.
missingScene	Text explaining a missing or disabled scene.
missingText	Final status message for the first list item.
sceneList	The final list of options used in the dropdown.
currentIndex	Index of the currently selected item in dropdown.
options	GUI-friendly converted list for Unity’s popup.
newIndex	The index selected by the user.

🎉 Practical Usage Example

When you mark a string field with the attribute:

```
[SceneTagDropdown]
public string sceneToLoad;
```

You get:

- A dropdown listing all enabled scenes
- A first option showing the status (missing/disabled)
- Automatic warnings if you type a wrong scene name
- Automatic height adjustments
- Support for multiple object editing

This makes your inspector much safer and avoids typos.

Read Only Attribute

🧩 General Overview

This script adds a feature to Unity's Inspector that allows certain fields to be **visible but not editable**.
When you apply the special marker [ReadOnly] to a variable in a Unity component, that variable will still appear in the Inspector, but the user will not be able to change its value.
This is very useful when you want to **display runtime information, debug data, or internal state variables** without risking accidental edits that could break functionality.

🏷️ 1. ReadOnlyAttribute

Purpose
This is a **custom attribute** used to tell Unity that a field should be **displayed but locked** in the Inspector.
It doesn't contain any code logic or data.
It acts purely as a **marker**, meaning that its presence tells another script (the custom drawer) how to handle the field's display behavior.
How It's Used
You apply it above a field in any MonoBehaviour or ScriptableObject.
For example, if you want to show a variable's value during play mode but not allow editing, you mark it with [ReadOnly].
Then, when Unity draws this field in the Inspector, it will appear greyed out and uneditable, but still visible.

🧭 2. ReadOnlyDrawer

This is a **custom drawer** that defines what happens in the Inspector when Unity sees a field marked with the ReadOnly attribute.
It exists only inside the Unity Editor, meaning it's not included in the final game build.
The drawer takes control of how the field is drawn on the screen.

🖼️ Method: OnGUI

Purpose
This method controls **how the property is displayed** in the Inspector.
When Unity is about to draw a field that has the [ReadOnly] attribute, it calls this method instead of its default drawing behavior.

Step-by-Step Explanation

1. **Disable editing in the graphical user interface**
   - Before drawing the field, the script tells Unity's interface system that the next controls should be **disabled**.
   - This means the user can see the field's value but cannot type or change anything inside it.
   - The field becomes visually greyed out to indicate that it is locked.
2. **Draw the field normally**
   - The drawer then instructs Unity to render the field as usual, with its label and current value.
   - Because editing is disabled, the user can only observe the data.
3. **Restore the GUI state**
   - After drawing, it immediately re-enables the user interface.
   - This is important because the Inspector might draw many other fields afterward, and we only want the [ReadOnly] field to be locked — not everything else.

Parameters Explained

- **position** → The rectangular area in the Inspector window where the field will appear. It defines its size and placement.
- **property** → The serialized representation of the variable being shown. It contains the value and metadata that Unity uses to draw it.
- **label** → The text label that appears next to the field in the Inspector.

⚙️ Internal Logic Summary

When Unity processes a field with [ReadOnly]:

1. It disables editing.
2. Draws the field in a non-interactive state.
3. Re-enables the interface for subsequent elements.

This simple sequence ensures that only the chosen variable is locked.

🧪 Usage Example (Conceptual)

You can apply the [ReadOnly] marker above any public or serialized field.
In the Inspector, the variable will appear greyed out and cannot be modified.
For instance, developers often use this to:

- Show the current health or position of a character during runtime.
- Display calculated values that are updated by scripts but shouldn't be changed manually.
- Protect internal data that should only be modified through code logic.

📦 Variables Involved

There are **no internal variables** stored in this drawer because it doesn't need to remember or manipulate data — it only changes the *display state* of the Unity Editor temporarily while drawing.
However, three parameters are used during drawing:

- **position** (visual layout)
- **property** (the serialized data)
- **label** (the text caption next to the field)

Additionally, Unity's built-in **GUI.enabled** flag is toggled to disable and re-enable interactivity.

✅ Benefits

- Prevents accidental changes to critical or runtime-only data.
- Makes debugging easier by displaying values directly in the Inspector.
- Keeps code safe and consistent by separating "viewable" and "editable" data.
- Works seamlessly with all serialized field types (numbers, strings, objects, etc.).

🧭 Summary of the Script's Flow

1. The [ReadOnly] attribute marks specific fields.
2. The Unity Editor detects that attribute and invokes the custom drawer.
3. The drawer temporarily disables editing for that field.
4. It renders the field's value in a non-editable (greyed-out) state.
5. The drawer then re-enables normal editing for all other fields.

Highlight Empty Reference Attribute

🧩 General Overview

This script extends Unity's Inspector so that when a variable marked with a special tag ([HighlightEmptyReference]) has **no assigned object**, it gets visually highlighted with a **red background** and a **warning message**.
It helps developers quickly identify **unassigned references** in the Editor, preventing common runtime errors such as missing components, prefabs, or script references.

🏷️ 1. HighlightEmptyReferenceAttribute

Purpose

This is a **marker attribute** that you attach above a field in a Unity script.
Its role is to tell the Unity Editor that this variable should be visually checked for null or missing references.

This attribute doesn't contain any internal logic or data — it only serves as a tag that the Unity Editor recognizes through the matching drawer.

Example Usage

When placed above an object reference (for example, a reference to a GameObject, Transform, or ScriptableObject), Unity will automatically highlight it in the Inspector if nothing is assigned to it.

Example use case:

You might mark fields that must always be assigned, such as "player reference", "enemy prefab", or "UI manager".

If you forget to assign something, the field turns red and a clear message appears underneath, reminding you to fill it in.

🧭 2. HighlightEmptyReferenceDrawer

This is the **custom property drawer** that defines how Unity draws fields marked with the [HighlightEmptyReference] attribute.

It only runs inside the Unity Editor, never in the game build.

This drawer modifies how the field looks and adds logic to detect and warn about unassigned object references.

🖼️ Method: OnGUI

Purpose

This method controls **how the field appears and behaves** in the Unity Inspector.

It is called every time Unity renders the Inspector window for an object that uses the [HighlightEmptyReference] attribute.

Step-by-Step Breakdown

1. **Store the original background color**
   Before changing any visual settings, the script saves the current background color of the Inspector interface.
   This ensures that the interface can be restored to normal afterward.

Variable involved:

- **previousColor** → Holds the Inspector's color before modification.

2. **Check if the field is an object reference and whether it's empty**
   The drawer verifies that the variable type is an object reference (meaning it can hold things like components, prefabs, or other assets).
   Then it checks if the current reference value is null or missing.

Variable involved:

- **isEmpty** → Becomes true if the field is an object reference and nothing has been assigned.

3. **Change the background color if the field is empty**
   If the reference is missing, the field's background color changes to red, visually alerting the user.
   Otherwise, it keeps the default color.
4. **Define the position for the field**
   A rectangular area (called a "rect") is created to tell Unity where on the screen to draw the field.
   It specifies horizontal position, vertical position, width, and height.

Variable involved:

- **propertyRect** → Defines the space where the reference field is drawn.

5. **Draw the property field**
   The field is drawn as usual, but if it's empty, the background appears red.
6. **Restore the previous background color**
   After drawing the field, the background color is set back to what it was before.
   This prevents the red color from affecting any other elements in the Inspector.
7. **If the field is empty, display an error message**
   When the reference is missing, a help box appears below the field with a message that says what type of object should be assigned.

To do this:

- The script tries to find the **type of the field** using reflection (a system that allows the script to look at the field's metadata).
- If the field type can't be found, it defaults to showing "Unknown".

Variables involved:

    - **typeName** → Stores the type of object that should be assigned (for example, “Transform”, “AudioSource”, or “ScriptableObject”).
    - **helpBoxRect** → Defines the position and size of the warning box below the field.
8. **Draw the warning box**
   The warning box shows a red message such as:
   “Put an item of type 'Transform' here!”
   This gives the user a clear indication of what is missing.

Parameters Explained

- **position** → Defines where in the Inspector this element is drawn (its location and size).
- **property** → Represents the serialized version of the variable being edited. It holds both the data and metadata Unity uses to display and modify fields.
- **label** → The field’s display name or caption in the Inspector.

📏 Method: GetPropertyHeight

Purpose

This method determines **how tall the field’s space** should be in the Inspector.
If the field is missing a reference, extra vertical space is added to make room for the help box below it.

Step-by-Step Breakdown

1. **Check if the field is empty**
   It performs the same logic as before — checking if the variable is an object reference and if it has no value assigned.
2. **Return the correct height**
    - If the reference is missing, the method adds extra height (approximately triple the normal field height) so the help box can fit below the field.
    - If the field is correctly assigned, it returns the default single-line height.

This ensures that the warning message doesn’t overlap with other elements in the Inspector.

Variables Involved in Height Calculation

- **isEmpty** → Boolean that indicates whether the field has no reference assigned.
- The method then returns either a larger or smaller height value depending on that condition.

🧪 Usage Summary

1. Add the [HighlightEmptyReference] attribute above any object reference field (e.g., a GameObject, Transform, ScriptableObject, or Component reference).
2. In the Unity Editor, open the object containing this script.
3. If the field is **not assigned**, it will be highlighted with a **red background** and a **warning box** telling you what type of object should go there.
4. Once you assign a valid reference, the highlight and warning automatically disappear.

✅ Benefits

- Makes it **visually obvious** when an important reference is missing.
- Prevents runtime errors caused by unassigned objects.
- Improves debugging and scene setup speed.
- Provides clear type information about what is expected in the field.
- Automatically adapts its height to keep the Inspector layout clean and readable.

🧭 Summary of Logic Flow

1. The attribute [HighlightEmptyReference] marks a field for special handling.
2. When Unity draws the Inspector, it detects this attribute and uses the custom drawer.
3. The drawer checks if the field is an object reference and whether it’s empty.
4. If empty:
    - The field background becomes red.
    - A help message appears below it with the expected object type.
5. If filled:
    - The field appears normal.
    - No warning is shown.
6. The drawer automatically adjusts the visual height to fit any messages.

Gizmo Transform Attribute

✅ High-Level Overview

This script adds a **custom attribute** that you can put on any 3D vector field inside Unity.
When applied, the field gains:

- Buttons inside the Inspector to enable *position* and optionally *rotation* editing.
- Scene View gizmos that let you drag handles directly in the scene.
- Support for lists/arrays.
- Proper undo behaviour.
- Automatic reset when scripts reload.

The system is made of two parts:

1. **The attribute**, which marks a Vector3 field.
2. **The drawer**, which handles GUI, Scene View handles, internal state, and gizmo rendering.

📌 The Attribute (GizmoTransformAttribute)

Purpose
Marks a Vector3 so that the editor draws gizmos for it.
Variables

- **rotationPropertyName**
  Stores the *name* of a Quaternion field (optional).
  If present, rotation editing is allowed in addition to position.

Constructors

- No-argument constructor → position only.
- Constructor with string → links to a Quaternion field.

Usage examples
Basic:

```
[GizmoTransform]
Vector3 position;
```

With rotation:

```
[GizmoTransform(nameof(rotation))]
Vector3 position;
Quaternion rotation;
```

📌 The Custom Drawer (GizmoTransformDrawer)

Handles all editor behaviour.

1) Internal Editing State
Dictionaries:
These track which property is currently being edited:

- **editingPosition**
  Stores whether a certain Vector3 field is currently in "position editing mode".
- **editingRotation**
  Stores whether rotation editing is active.

Each entry uses the property's *unique path* as the key.
Active References:

- **activeObject**
  Points to the component currently being edited.
- **activePosition**
  The specific Vector3 field being manipulated.
- **activeRotation**
  The related Quaternion field (may be null).

GUI Content:
Two preconfigured button definitions:

- Position button
- Rotation button
  Each contains icon + tooltip.

Layout constants:
Defines the size and spacing of the inspector buttons.

2) Scene GUI (OnSceneGUI)
Runs every frame while in Scene View.
Responsibilities:
- Checks if the selection is valid and cancels editing when needed.
- Updates the serialized object to keep data in sync.
- Gets the world-space position of the Vector3.
- Calculates world-space rotation.
- Draws and handles the Scene View transform handles:

✓ Position handle
✓ Rotation handle
- Applies modified values back to local coordinates.
- Records undo and marks object as dirty.
- Draws visual gizmos (sphere + colored axes).

3) Editing Control Functions
TogglePositionEditing
- Activates or deactivates position editing.
- Ensures no other field remains active.
- Stores references to the active fields.
- Registers OnSceneGUI so Unity begins showing handles.

ToggleRotationEditing
Same as above, but for rotation editing.
PropertyIsValid
Checks if the serialized property still exists (important for list items that can be deleted).
ResetOnScriptsReload
Automatically stops editing whenever scripts recompile, preventing Unity from keeping invalid references.
StopEditing
- Clears the dictionaries.
- Removes the SceneGUI callback.
- Refreshes Scene View.

4) Gizmo Drawing
DrawGizmo
Draws:
- A small sphere that marks the point.
- Three axis lines (red, green, blue).
- Ensures a broken quaternion doesn't crash the drawer.

DrawAxis
Draws a single colored axis line from the gizmo center.

5) Inspector GUI (OnGUI)
Draws buttons + the Vector3 field.
Responsibilities:
1. Tries to find the linked rotation property (handles arrays, nested objects, and absolute paths).
2. Ensures state dictionaries have valid entries.
3. Draws:
    - Position button
    - Rotation button (if available)
    - The Vector3 field itself

Clicking a button toggles its editing mode.

6) Property Height
Makes space for:
- One row of buttons
- One Vector3 field

🎮 **How You Use It**

Basic position editing
Apply the attribute to a Vector3:

```
[GizmoTransform]
public Vector3 point;
```

Now in the Inspector you get:

- A "Move" icon button → enables gizmo mode.
- Scene View handles to drag the point.

Position + Rotation editing
If your object also has a Quaternion:

```
[GizmoTransform(nameof(rotation))]
public Vector3 localOffset;
public Quaternion rotation;
```

Now you get:

- Position edit button
- Rotation edit button
- Scene View handles for both

🧠 **How It Works Internally (Short Summary)**

1. You click a button in the Inspector.
2. The drawer stores a reference to that property.
3. SceneView hook starts running.
4. Handles appear at the correct world-space location.
5. Your interaction modifies the Vector3 or Quaternion.
6. Values are converted from world space → local space.
7. Unity's undo system and serialization record the change.
8. Custom gizmo graphics show the axes and sphere.

Gizmo Sphere Attribute

📌 **Purpose of the Tool**
This script creates a **custom attribute + custom editor drawer** that lets you edit a **3D sphere gizmo directly in the Unity Scene View**.
You apply the attribute to a Vector3 field (the center offset), and link it to a float field (the radius).
Then, inside the scene, you can drag small handles to change the radius visually.
The tool supports:

- Live editing in Scene View
- Persistent editing state (only one sphere edited at a time)
- Snapping with the Control key
- Undo support
- Custom sphere color
- Safe handling when the field is inside arrays or nested objects
- Editor-only execution

The attribute goes above your Vector3 field like this:

```
[GizmoSphere(nameof(radius))]
public Vector3 sphereOffset;
public float radius;
```

📌 **Part 1 — The Attribute**
GizmoSphereAttribute
This is the annotation you place above a Vector3 field.
Its purpose is to link:

- The **offset** → the Vector3 being annotated

- The **radius** → another float field whose name is given
- An optional **custom color** used to draw the gizmo sphere

Variables inside the attribute

Variable	Meaning
radiusPropertyName	The text name of the field that stores the sphere radius.
customColor	The RGB color applied to Sphere handles and drawing.

Constructors

There are two constructors:

1. One that receives only the name of the radius → default color = green.
2. One that receives the name + RGB values → custom color.

📌 Part 2 — The Custom Drawer

This is the logic that runs inside the Unity Editor when drawing the inspector and the scene gizmo.

💠 Persistent Editing State (shared static data)

These variables control what is currently being edited. They ensure only one sphere is edited at a time.

editingAttributes

A dictionary linking the property path of the offset field → the attribute instance.
Used to retrieve its color and radius link.

editingSphere

A dictionary linking offset property paths → "true or false" meaning the gizmo is currently active for that field.

activeObject

The serialized Unity object currently being edited (e.g., the Component instance).

activeOffset

The serialized Vector3 field that stores the center offset inside the component.

activeRadius

The serialized float field storing the radius.

Editor UI constants

Numbers used to size the edit button:

- button width
- button height
- spacing
- button icon

📌 Part 3 — Scene GUI (The visual sphere and handles)

OnSceneGUI

This method runs every time the Unity Scene View redraws.
It:

1. Validates that the editing target still exists
2. Reads the component and its transform
3. Calculates the world position of the sphere (local offset → world position)
4. Reads the radius
5. Draws 6 draggable handles:
   - +Y
   - -Y
   - +X
   - -X
   - +Z
   - -Z
6. When a handle moves:
   - The radius becomes the distance from the sphere center to that handle
   - If Control is pressed → values snap
7. Draws the sphere with:
   - 3 wire circles
   - 3 faint solid discs

📌 Part 4 — Drawing the Sphere

DrawSphereGizmo

This function draws:

1. Wireframe edges
2. Transparent solid discs (for visual volume)
3. Restores the original color of Unity handles

It uses three perpendicular planes to simulate a sphere because Unity's editor does not provide a rotated wire-sphere function.

📌 **Part 5 — Inspector GUI (the button + the field)**

OnGUI

This draws the actual inspector UI for the Vector3 field.
It performs:

1. Reads the attribute attached to the field
2. Attempts to locate the radius field, including special handling for arrays
3. Draws a button labeled with the "EditCollider" icon
4. When the button is clicked → toggles the gizmo editing
5. Below the button, draws the normal Vector3 field

GetPropertyHeight

The drawer increases the vertical height in the inspector so the button and field fit.

📌 **Part 6 — Editing State Control**

These methods manage whether editing is active.

ToggleEditing

This is called whenever the edit button is pressed.
It:

1. Turns off editing for every other sphere
2. If this sphere was already being edited → stop editing
3. Otherwise:
    - Marks the current sphere as active
    - Stores attribute, object, offset field, and radius field
    - Registers the Scene GUI callback so handles become visible
    - Forces scene repaint

PropertyIsValid

Some SerializedProperties become unsafe, for example after recompilation.
This method tries to access the property type; if it throws, the property is invalid.

ResetOnReload

Automatically executed after script reloads.
Ensures editing is stopped so no corrupted state persists.

StopEditing

Completely clears:

- Active sphere
- Dictionaries
- Scene GUI callback

And repaints the scene so the handles disappear.

📌 **Summary of How You Use It**

1. Add attribute to a Vector3 field

You place the attribute above the offset field.
You tell it the name of the float radius field:

```
[GizmoSphere(nameof(radius))]
public Vector3 sphereOffset;
public float radius;
```

2. In the Inspector

A small button appears.
When pressed:

- The Scene View activates sphere handles

- You can drag the handles to change the radius

3. In the Scene View

You see:

- Wireframe sphere
- Transparent discs
- Six handles to resize the radius
- Control key for snapping

4. The offset never changes

Only the radius is modified.
The center is always the Vector3 offset field.

Gizmo Cube Attribute

✅ Overview — What This Tool Does

This system allows you to take:

- a **Vector3 offset**
- a **Vector3 size**

and edit them visually in the Scene View as a **3D cube gizmo** with interactive handles.
You apply the attribute above your offset field, linking it to the size field.
Inside the Scene View, six handles appear around the cube (left, right, up, down, forward, back).
Dragging a handle updates the cube's size and center automatically.
It supports:

- color customization
- snapping with the control key
- undo
- safe behavior in arrays and nested serialized objects
- persistent "one cube at a time" editing
- editor-only execution

✅ Part 1 — The Attribute

The attribute is what you place above your **offset** Vector3 field.
It tells Unity this field can be edited as a gizmo cube.

Stored Values

sizePropertyName

A string containing the name of the field that stores the cube's size.
Used by the drawer to locate the size property inside Unity's serialization system.

customColor

The color used to draw the cube gizmo in the Scene View.

Constructors

There are two versions:

1. One that just receives the name of the size field → default color is cyan.
2. One that receives name + RGB values → custom color.

✅ Part 2 — The Property Drawer

This is the editor logic that:

- draws the button in the Inspector
- draws the cube in the Scene View
- calculates size based on handle movement

Everything happens inside this custom drawer.

🔷 Persistent Editing State (Shared Static Data)

These variables ensure only one cube can be edited at a time and keep track of what is being modified.

editingAttributes

A map:

- Key → property path of the offset field
- Value → the attribute instance associated with that field

Used to know the custom color and size link.

editingCube

A map:

- Key → path of the offset property
- Value → Boolean indicating whether that cube is being edited

activeObject
The serialized Unity object containing the offset and size fields.
activeOffset
The specific serialized Vector3 field storing the offset.
activeSize
The specific serialized Vector3 field storing the size.

UI Constants
Defines:

- width of the inspector button
- height of the inspector button
- spacing
- icon used on the button

These control how the Inspector layout looks.

🔷 **Part 3 — Scene GUI: How Cube Editing Works**
The core editing happens inside the Scene View callback.
OnSceneGUI
Runs every frame while the Scene View is visible.
Its responsibilities:

1. Validate everything
Checks:

- whether anything is selected
- whether the target object still exists
- whether the offset and size properties are still valid

If something is invalid → editing stops.

2. Read the component
Gets the object that owns the fields and retrieves its transform.

3. Calculate world positions

- Converts the local offset into a world-space center.
- Converts the local size into world-space half extents.

This is necessary because handles operate in world space.

4. Prepare handle positions
Six handle positions are computed:

- Right
- Left
- Up
- Down
- Forward
- Back

Each position sits at the outer face of the cube.

5. Draw and update the handles
Every handle is drawn as a small cube in the Scene View.
User can drag them freely:

- If the control key is held → snapping is applied
- Otherwise → smooth movement

Each dragged handle updates its corresponding face.

6. Recalculate size and offset
When any handle changes:

- The distance between opposite handles determines size

- The mid-point between them determines the new offset

The values are transformed back into local space before being assigned.
Finally:

- Values are applied
- Undo is recorded
- Object is marked dirty to update properly

7. Draw the cube in the Scene View
The drawer displays:

- the cube wireframe
- translucent faces for visibility

This gives the user clear feedback about the cube’s shape and orientation.

🔷 Part 4 — Drawing the Cube

DrawGizmo
This helper method:

1. Computes all eight cube corners
2. Draws a wireframe cube around them
3. Draws six translucent faces (front, back, top, bottom, left, right)

It uses the transform’s rotation so the cube respects the object's orientation.

🔷 Part 5 — Inspector GUI

OnGUI
This method draws everything inside the Inspector panel.
It performs:

1. Searches for the size field linked by name
    - If inside arrays/nested objects, it reconstructs the full path
2. Determines whether this field is currently being edited
3. Draws the edit button
    - If clicked, toggles cube editing
4. Draws the normal Vector3 field below the button

GetPropertyHeight
Ensures the Inspector reserves enough vertical space for both the edit button and the property.

🔷 Part 6 — Editing Control

ToggleEditing
Switches editing on or off.

- Only one cube can be edited at the same time
- Disables all others when a new one is activated
- Stores active offset and size fields
- Registers the Scene View callback

PropertyIsValid
Tests whether the serialized property can be accessed safely, preventing Unity from throwing errors.

ResetOnReload
Executed automatically after script recompilation.
Clears the editing state to avoid broken references.

StopEditing
Fully resets everything:

- Clears dictionaries
- Nullifies references
- Removes Scene View callback
- Repaints editors to remove handles

✅ How You Use It

1. Apply the Attribute
Above a Vector3 offset field, link it to a Vector3 size field:

```
[GizmoCube(nameof(cubeSize))]
public Vector3 cubeOffset;
public Vector3 cubeSize;
```

2. In the Inspector
A button appears next to the offset field.
Pressing the button activates gizmo editing.
3. In the Scene View
A cube appears, and six movable handles let you adjust:

- Width
- Height
- Depth
- Center position

Snapping works with the control key.
4. Edits are written back automatically
When you move handles:

- size changes
- offset changes
- undo works
- properties update in real time

Conditional Hide Attribute

🧩 General Overview

This script allows you to **hide or show fields in the Unity Inspector dynamically**, depending on the state of other variables in the same object.
It defines a **custom rule tag** called ConditionalHide, which you can attach to variables. When Unity draws the Inspector window, it checks whether the specified conditions are met. If the conditions are not satisfied, the field disappears from view.
This system is especially helpful when building complex Inspector interfaces — for example, showing advanced settings only when a toggle is enabled or hiding optional parameters when not needed.

🏷️ ConditionalHideAttribute (the rule definition)

This part defines the behavior rules. It's the **tag** you attach to your fields to tell Unity *when* they should appear or disappear.

🔷 Purpose

It stores the **names of other fields** that determine whether a specific property should be visible.
It also defines **how** those conditions should be combined — whether *any* false condition hides the property, or whether it hides only if *all* conditions fail.

🔷 Variables inside

- **ConditionalSourceFields**
  This holds one or more text values that represent the names of the fields Unity will check.
  Each of these source fields must be a true/false type in the Inspector.

Example: if you write [ConditionalHide("isActive")], it means "show this field only if isActive is true".

- **HideIfAnyFalse**
  This determines the logic used when multiple conditions are listed.
  If it is true, the target field hides whenever **any** of the conditions are false.
  If it is false, the field hides only when **all** conditions are false.

🔷 Constructors (setup methods)

The script provides two ways to set up the rule when you apply it:

1. **Simplified usage** –
   You can pass one or more condition names directly.
   If you list multiple names, the rule automatically hides the field if any of them are false.

Example:
[ConditionalHide("isVisible", "canEdit")]
→ The field disappears if either condition is false.

2. **Detailed usage** –
   You can explicitly define whether the property should hide when *any* or *all* conditions fail.
   The first parameter is a true/false value that controls this behavior.

Example:
[ConditionalHide(false, "option1", "option2")]
→ The field will hide only when both conditions are false.

🧭 **ConditionalHidePropertyDrawer (the Inspector behavior)**
This section tells Unity **how to draw** the fields that use the ConditionalHide tag.
It runs only inside the Unity Editor and controls both **visibility** and **space allocation** in the Inspector.

🖼️ **Method: OnGUI (controls the visual display)**
This part decides **whether to show or skip** the field during Inspector rendering.

Step-by-step logic:
1. It first retrieves the rule information from the field (the conditions and how they should be evaluated).
2. It checks whether the rule should hide the field.
   - If there is **one** condition, it checks whether that condition is false.
   - If there are **multiple** conditions, it checks all of them using the rule's HideIfAnyFalse setting.
3. If the result of that check means the field should be hidden, it stops there and skips drawing that field.
4. Otherwise, the field is displayed normally in the Inspector.

Parameters explained:
- **position** → Defines where on the screen Unity should draw the element.
- **property** → Represents the specific variable that Unity is currently drawing.
- **label** → The field's visible name or caption inside the Inspector.

📏 **Method: GetPropertyHeight (controls layout space)**
This part ensures that when a field is hidden, it also **doesn't occupy any vertical space** in the Inspector.
Unity's Inspector layout engine asks how tall each field should be.
- If the field is hidden, this method returns **zero**, so nothing is drawn and no gap appears.
- If the field is visible, it returns the normal height.

This keeps the Inspector layout clean and compact.

⚙️ **Hidden logic: Condition checking**
Several internal checks determine whether the field should appear.
They work together to interpret the rules stored in the attribute.

1. CheckSingleCondition
- Looks up one specific field (the condition) by its name.
- If that field cannot be found or is not a true/false value, it assumes the condition is false.
- If the field is found and its value is false, the target field will be hidden.

2. CheckAllConditions
- Used when there are multiple condition names and the rule says "hide if any condition fails".
- It loops through all listed condition fields.
- If any of them are missing or false, it decides that the property should be hidden.
- Only when *all* conditions are true does the property remain visible.

3. CheckAnyCondition
- Used when there are multiple condition names but the rule says "hide only if all are false".
- It loops through all condition fields, looking for at least one that is true.
- If it finds one true condition, the field stays visible.

- If all conditions are false, the field hides.

🔍 Method: GetConditionProperty (finding the target fields)

This helper part is responsible for **locating the field** that controls visibility.
It takes the name provided in the ConditionalHide tag and tries to find the corresponding variable inside the same object.
If it doesn't find it immediately, it supports **nested searches** — for example, if the condition is written like myClass.myToggle, it will look for the inner property within another object.
It returns the matching condition data or nothing if the path doesn't exist.

🧪 How to Use It

You attach the ConditionalHide tag above a variable in your script, referencing one or more true/false fields in the same component.
Example Scenarios:

1. **Single toggle control**
2. [ConditionalHide("showAdvanced")]
3. public int advancedSetting;

→ The "advancedSetting" field appears only when "showAdvanced" is true.

4. **Multiple controls (any false hides)**
5. [ConditionalHide("isVisible", "isActive")]
6. public GameObject target;

→ The "target" field appears only if both "isVisible" and "isActive" are true.

7. **Multiple controls (hide only if all are false)**
8. [ConditionalHide(false, "optionA", "optionB")]
9. public GameObject optionalPanel;

→ The "optionalPanel" is visible if *at least one* of the options is true.

10. **Nested reference**
11. [ConditionalHide("settings.enableSound")]
12. public AudioClip backgroundMusic;

→ The field will only be visible if enableSound inside settings is true.

✅ Summary of How It Works

1. The [ConditionalHide] tag is attached to a variable.
2. The tag stores a list of condition names and how to interpret them.
3. When Unity draws the Inspector:
   - It checks those condition fields.
   - Based on their values, it decides whether the field should appear.
4. If hidden, the field is skipped visually and occupies no space.
5. If visible, it's drawn normally as if no condition existed.

🎯 Benefits

- Keeps the Unity Inspector clean and context-sensitive.
- Prevents users from changing irrelevant settings.
- Makes editor interfaces more intuitive and professional.
- Supports multiple conditions and nested properties.
- Works dynamically — updates instantly as conditions change in the Inspector.

Button Attribute

🧩 General Overview

This system adds **interactive buttons inside the Unity Inspector** that directly call specific actions (methods) from a component.
Normally, Unity's Inspector only shows fields (numbers, text, toggles, etc.), but not buttons tied to your script's logic.
With this setup, any method marked with a special **custom tag** becomes a clickable button.
Example usage inside a component:
You can write [Button(nameof(MyMethod))] above a method, and when viewing that component in the Inspector, a button labeled **"My Method"** will appear. Clicking that button immediately runs the method.

This is extremely useful for **testing**, **debugging**, or **quickly triggering behaviors** in the Editor without running the full game.

🏷️ **Custom Tag Definition — "ButtonAttribute"**
This is the **marker** that tells Unity which methods should appear as clickable buttons.
When Unity loads the Inspector for an object, it scans all its methods looking for this marker.

🔷 **Purpose**
It identifies methods that should become buttons in the Inspector.
It also allows you to optionally define a **custom label** for each button.

🔷 **Variable inside**

- **Label**
  This holds the name that should appear on the button.
  If it is left empty, the button automatically uses the name of the method itself (but formatted nicely for readability).

🔷 **How it works conceptually**
When the Editor scans a script:

1. It looks for methods marked with [Button].
2. It checks if that method has **no parameters** (since buttons can't pass arguments).
3. For each valid one, Unity draws a button in the Inspector.
4. Clicking the button calls that method directly in the editor environment.

🔷 **Recommended usage**
You can safely link the button's label to the method name using Unity's built-in *nameof* feature.
This ensures that if you later rename the method, the label automatically updates and never breaks.
Example idea:
[Button(nameof(ResetPosition))]
→ Creates a button labeled "Reset Position".

🧭 **Custom Editor — "ButtonDrawerEditor"**
This part is what actually **draws the buttons** inside Unity's Inspector window.
It replaces the default inspector drawing process for all components (because it extends Unity's base editor class).
This custom editor only works **inside the Unity Editor**, not in a built game.

🖼️ **Method: OnInspectorGUI**
This is Unity's main process for displaying the Inspector interface.
Here, it does two things in sequence:

1. **Draws all custom buttons** that correspond to methods using the [Button] marker.
2. **Draws the regular inspector fields** (the normal variables and properties).

It also adds a small vertical gap between the custom buttons and the standard fields to keep the interface clean and readable.

⚙️ **Method: DrawButtons**
This is the **core** of the system — it searches for methods with the button tag and creates visual buttons for them.
Here's what happens step by step:

1. **Finds all suitable methods** inside the component being viewed.
   It filters them by the following rules:
   - The method must belong to the component being inspected.
   - It can be public or private.
   - It must not require parameters.
   - It must be marked with the [Button] tag.
2. **For each matching method**, it:
   - Reads the tag information to get the label text (if one was provided).
   - If no label is set, it automatically formats the method name into a readable button title.
3. **Draws a clickable button** in the Inspector using Unity's layout system.

4. **When clicked**, the button does the following:
    - Records the action for Unity's undo system (so the change can be reversed).
    - Calls (executes) the method directly on the component instance.
    - Marks the object as "changed" so Unity updates it visually in the editor.

This allows you to safely run editor-only actions — for example:

- Resetting values
- Creating or clearing objects
- Triggering setup or cleanup routines
- Testing AI or animation logic directly in the editor

✨ Method: FormatMethodName

This helper converts raw method names into a **readable label format** for the buttons.
It takes a name like GenerateLevelMap and turns it into **"Generate Level Map"**.
How it does this conceptually:

1. It detects transitions from lowercase to uppercase letters and inserts spaces.
   Example: "ResetPosition" → "Reset Position".
2. It separates sequences of uppercase letters from normal words to improve readability.
   Example: "GetIDText" → "Get ID Text".
3. It trims any extra spaces and returns the clean, formatted version.

This ensures that all buttons look consistent and user-friendly in the Inspector.

🧱 Supporting Concepts and Tools

The script also uses Unity's built-in systems to make everything robust and editor-safe:

- **Undo System**
  Records the state before running a button action, allowing you to revert changes easily using the Undo command.
- **Dirty Flag System**
  Marks the component as "modified" after the button is pressed, ensuring the editor properly saves any changes made by the method.
- **Reflection**
  The system looks through the script's methods at runtime to identify which ones have the [Button] marker.
  This allows it to find and display them automatically without manual configuration.

🧪 Example of Usage in Unity

Imagine you have a script controlling a player:

```
[Button(nameof(ResetStats))]
void ResetStats() {
   // logic to reset values
}
```

In the Inspector, you will now see a button labeled "Reset Stats".
Clicking it will immediately run that method on the selected object, even when the game is not playing.
You can also add multiple buttons for different purposes:

```
[Button(nameof(SpawnEnemies))]
void SpawnEnemies() { ... }

[Button(nameof(ClearEnemies))]
void ClearEnemies() { ... }
```

Both buttons will appear stacked in the Inspector and trigger the respective methods.

✅ Summary of Behavior

1. You mark one or more methods with [Button].
2. The editor scans your component and finds those methods.
3. It draws corresponding buttons in the Inspector, one per method.
4. When you click a button:
    - The method is immediately called on the selected object.
    - Unity records and tracks changes made by that method.
5. The rest of the inspector content is drawn normally below the buttons.

🎯 Benefits and Uses

- **Improves workflow:** lets you test or reset values without entering Play mode.
- **Reduces repetitive setup:** automate initialization or debugging tasks.
- **Keeps editor clean:** no need for special menus or editor windows.
- **Safe to use:** supports Undo and change tracking.
- **Automatic formatting:** turns method names into readable button labels.