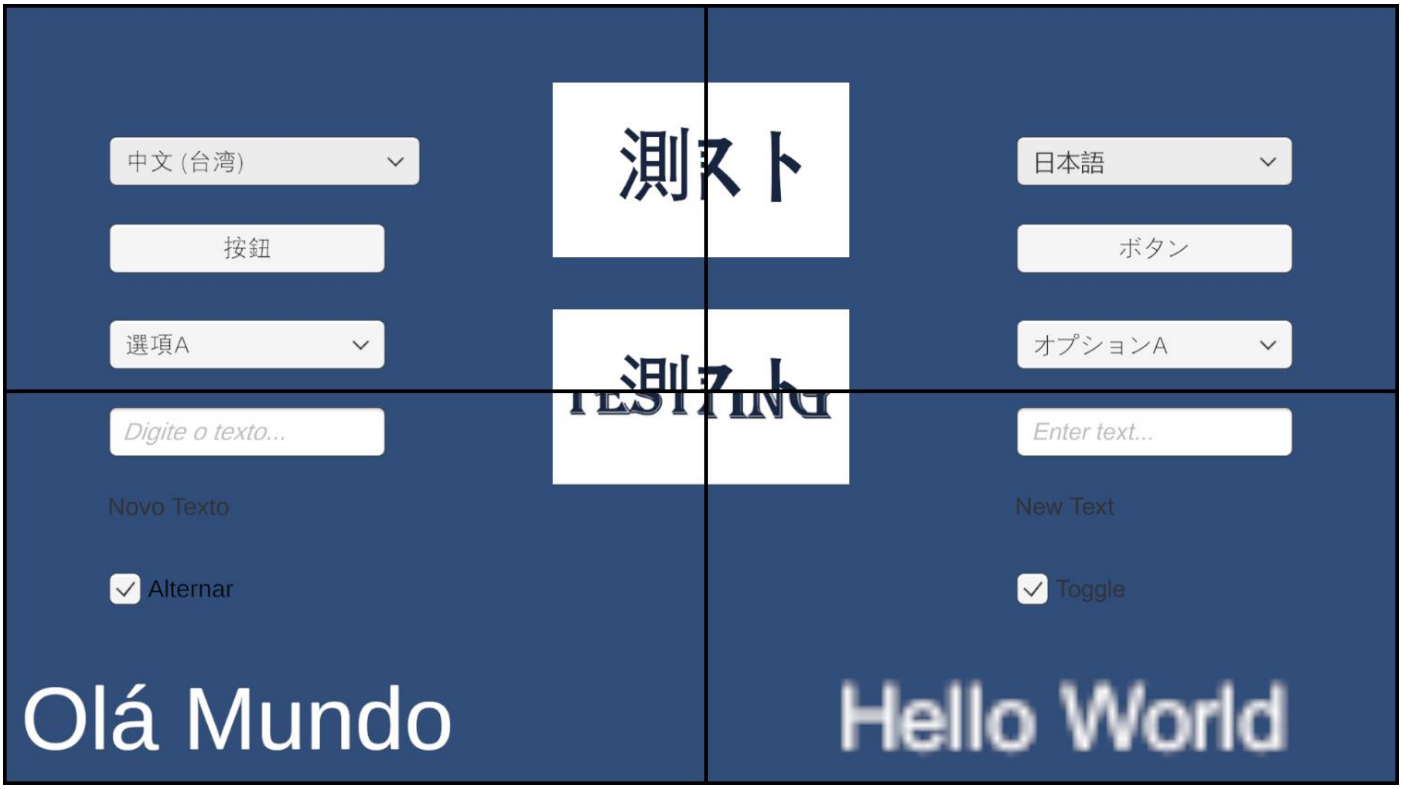

Language Tool

Documentation

_____

Creator: Lucas Gomes Cecchini

Online Name: AGAMENOM

Overview

This document will help you to use **Assets Language Tool** for **Unity**.

With it you can translate your game into several languages, automatic selection of languages (if you have a file available) and a simple text file to interpret, making it possible for anyone to translate the game into a new language.

It also has simple features to translate images and sound.

Instructions

You can get more information from the Playlist on YouTube:
https://youtube.com/playlist?list=PL5hnfx09yM4JkAyxrZWaFjhO3NMWxP_1F

Script Explanations

Language Manager Delegate

This script defines a centralized system for notifying other parts of an application when the active **language** changes. It uses a delegate and an event to allow external components to **subscribe** and automatically **react** whenever a language update occurs.

Let's go through it part by part:

Namespace: LanguageTools

- The script is placed inside a namespace called LanguageTools.
- Namespaces are used to organize code and avoid naming conflicts.
- This one specifically groups tools and utilities related to language management.

Class: LanguageManagerDelegate

- This is a **static class**, which means:
  - It cannot be instantiated (you can't create an object from it).
  - Its members are accessed directly using the class name.
- The purpose of this class is to **broadcast a notification** whenever the application's language changes.

Delegate: LanguageUpdateDelegate

- A **delegate** defines a specific method signature — in this case, methods that take **no parameters** and return **nothing**.
- It acts like a "type-safe pointer" to methods, allowing the event to call all registered methods automatically.

Meaning:
Any method that follows this format (no parameters, no return value) can be attached to the event.

Event: OnLanguageUpdate

- This is an **event** based on the LanguageUpdateDelegate delegate.
- Events are used to **notify subscribers** when something important happens.
- Here, it's triggered when the application language changes.
- Other classes or scripts can **subscribe** to this event, for example:
  - Updating UI text when the language changes.
  - Refreshing localized data.

- Reloading audio subtitles or other language-specific elements.

⚙️ Method: NotifyLanguageUpdate()

- This is a **public static method** that serves as the trigger mechanism.
- When called, it raises the OnLanguageUpdate event.
- The syntax OnLanguageUpdate?.Invoke() means:
  - "If there are any subscribers, call (invoke) all of them."
  - The ?. prevents errors if there are no subscribers.

Functionally:
When a language change is detected somewhere else in the application (for example, when the user selects a new language in settings), this method is called to inform all interested components that they should update themselves.

🧩 Usage Example (conceptual)

Imagine you have several parts of your app that display text:

- A main menu
- A settings panel
- A tooltip system

Each of them can **subscribe** to the OnLanguageUpdate event when they initialize.
When the user selects a new language:

1. The system calls LanguageManagerDelegate.NotifyLanguageUpdate().
2. The event is raised.
3. All subscribed components receive the notification and run their respective update methods — such as reloading text from a translation file.

💡 Summary

Element	Type	Purpose
LanguageTools	Namespace	Groups language-related utilities.
LanguageManagerDelegate	Static class	Manages and broadcasts language update notifications.
LanguageUpdateDelegate	Delegate	Defines the signature for update methods (no parameters, no return).
OnLanguageUpdate	Event	Allows external components to subscribe for language update notifications.
NotifyLanguageUpdate()	Method	Triggers the event, informing all subscribers.

In short, this script acts like a **message broadcaster** for language changes — whenever the language updates, every subscribed system automatically receives the message and updates itself accordingly.

Language Class Group

This script defines a **complete data model** for a Unity-based localization and UI management system.
Its purpose is to **organize, serialize, and store** all information related to languages, translations, and the visual layout of UI canvases — allowing Unity to easily **save**, **load**, or **edit** localized content and UI configurations.
Let's go through it section by section, explaining what each part does, what the variables represent, and how it's typically used in a project.

🧭 Overview

The script defines a set of **serializable classes** (which can be saved into files, JSON, or Unity assets).
They fall into four main categories:

1. **Language Structures** — store translation data and language info.
2. **Canvas & UI Structures** — describe UI layout elements.
3. **Canvas Component Data** — describe Unity UI component configurations.
4. **Language File Manager Data** — store language and UI data for editing, saving, and reloading.

🧠 1. Language Structures

LanguageAvailable

Defines the properties of a language available for selection in the app.

- culture: Language code such as "en-US" or "pt-BR".
- nativeName: The language name as written in its own form ("Español").
- name: The display name shown to users ("Spanish").
- isAvailable: Whether this language is currently active or usable.
- columnIndex: The position of this language in a localization table.

Usage:
Used to show available languages in a dropdown menu or to identify which translation column to use when reading from a file.

IdData

Links a **numerical ID** to its **localized text**.

- iD: Unique identifier for a text entry.
- text: The actual translated string.

Usage:
Used for fast lookup — scripts can find localized text by its ID instead of searching by name.

IdMetaData

Holds extra **display settings** associated with a text entry.

- iD: The same ID that ties this metadata to a specific text.
- alignment: Text alignment index (e.g., left, center, right).
- fontSize: The size of the text.
- fontListIndex: Which font from a predefined list is used.
- componentType: Identifies the type of UI component using it (e.g., TextMeshPro, standard Text, etc.).

Usage:
Ensures each localized text appears with the right formatting when loaded.

📦 2. Canvas & UI Structures

These structures describe how elements are positioned and scaled in the Unity UI system.

CanvasDataList
Represents a single **UI GameObject** in the scene and its layout data.
Main variables include:

- gameObjectName, instanceID: Identify the object.
- rectTransform: Reference to its RectTransform component (used for layout).
- localPosition, localRotation, localScale: The object's transform properties.
- anchorMin, anchorMax, anchoredPosition, sizeDelta, pivot: Layout settings that define the element's position within its parent.

Usage:
Used to recreate or save the position and state of UI elements for restoration or comparison.

CanvasStructure
Defines the entire layout of a UI canvas.

- canvasName: Name of the canvas being saved.
- canvasLayers: An array of CanvasLayers, grouping objects logically.
- rectTransform: Global RectTransform layout for the canvas root.
- canvas, canvasScaler, graphicRaycaster: Contain serialized data of the Unity components responsible for rendering and interaction.

Usage:
Used to save and restore full UI canvas structures, allowing exact replication across languages or resolutions.

CanvasLayers
Groups multiple UI objects together within a single canvas layer.

- CanvasObjectsLayers: Names of objects in this layer.
- rectTransforms: Layout data for each of those objects.

Usage:
Helps organize layered UI (e.g., background, middle, foreground).

RectTransformData
Stores layout information of a UI element (its shape and position in space).
Includes all layout properties:
localPosition, localRotation, localScale, anchors, pivot, and size delta.
Usage:
Acts as a lightweight serialized form of Unity's RectTransform for saving or restoring UI layouts.

🧩 3. Canvas Component Data

These structures represent **Unity UI component settings** (Canvas, CanvasScaler, and GraphicRaycaster).
They allow you to serialize component configurations and reload them as needed.

CanvasData
Describes settings of a **Canvas** component.

- renderMode: Defines how the UI is drawn (e.g., Screen Space, World Space).
- planeDistance: Distance from the camera.
- pixelPerfect, overrideSorting, overridePixelPerfect: Rendering precision options.
- sortingBucketNormalizedSize, vertexColorAlwaysGammaSpace: Rendering optimization values.
- additionalShaderChannels: Defines additional vertex data used by shaders.
- updateRectTransformForStandalone: Adjusts layout for standalone platforms.

Usage:
Used when reconstructing canvas components programmatically to preserve rendering behavior.

CanvasScalerData
Stores scaling behavior for different screen resolutions.

- uiScaleMode: Defines scaling method (constant pixel size, scale with screen size, etc.).
- referenceResolution: The base resolution used for scaling.
- matchWidthOrHeight: Balances between width and height scaling.
- physicalUnit, fallbackScreenDPI, defaultSpriteDPI: Handle scaling on different display types.
- dynamicPixelsPerUnit: Used for fine-tuning scaling.
- presetInfoIsWorld: Marks if the scaling applies to a world-space canvas.

Usage:
Ensures UI elements maintain consistent proportions on different screen sizes.

GraphicRaycasterData
Defines input interaction settings for UI.

- ignoreReversedGraphics: Determines if back-facing elements are ignored.
- blockingObjects: Specifies what can block raycasts (2D, 3D, or both).
- blockingMask: Layer mask used to filter which objects can block interactions.

Usage:
Helps preserve input detection settings when reloading or cloning canvases.

🌐 4. Language Components & Scripts
These define links between **language data** and Unity's **UI elements** or **events**.

LanguageOptions
Represents one selectable language in a UI menu.

- text: The visible name of the language.
- sprite: An icon or flag representing it.
- iD: Identifier associated with this option.

Usage:
Displayed in language selection menus. When chosen, triggers a language change event.

ScriptText
Stores a text entry and an event that can react to it.

- iD: Unique ID for the text.
- text: The script or line content.
- targetScripts: A UnityEvent that can trigger actions when the text changes.

Usage:
Used for dynamic scripts or dialogues where changing the text triggers updates elsewhere.

LanguageLines
Represents a single text line, with an option to mark if it should be translated.

- iD: The line's unique identifier.
- text: The content of the line.
- translateText: Whether the line should appear in translations.

Usage:
Used for managing which texts are part of translation files and which are not.

🗂️ 5. Language File Manager Data
These structures are used by a **Language File Manager tool**, allowing users to edit and save localization data.

LanguageForEditingSave

Represents editable text data linked to UI components.

- iD, text, alignment, fontSize, fontListIndex: Define the text and its style.
- textContext: Describes where this text is used.
- componentType: Identifies what UI component owns it.

Usage:
Used for editing text entries inside a custom editor window.

CanvasForEditingSave
Stores serialized canvas information.

- canvasID: Identifier for the canvas.
- textContext: Descriptive label of what it represents.
- json: Serialized data (layout, properties, etc.) in JSON format.

Usage:
Allows saving and restoring canvas setups during localization work.

LanguageFileManagerWindowData
Acts as the **main container** for everything used by the Language File Manager window.

- languageForEditing: Which language is being edited.
- componentSave: List of component texts and metadata.
- canvasSave: List of canvas data.
- availableLanguages: Languages defined in the system.
- firstTime: Marks if it's the first setup.
- idIndex: Used for generating new IDs.
- showTextData, showCanvasData: UI toggles for the editor.
- fileIsSaved: Tracks whether the file is currently saved.

Usage:
Stores the complete state of the localization editor window for persistence between sessions.

ManagerLanguageIdData
A lightweight structure for referencing or managing a single text entry.

- iD: Unique identifier.
- text: The localized or reference string.
- textContext: Describes how or where it's used.

Usage:
Used for quick lookups, linking components to their localized text in runtime.

Summary

Category	Purpose
Language Structures	Define language information, text IDs, and metadata.
Canvas & UI Structures	Describe positions and transformations of UI elements.
Canvas Component Data	Store rendering and scaling settings for UI components.
Language Components & Scripts	Link language data to Unity elements and events.
Language File Manager Data	Manage, edit, and save all language and canvas information.

In Practice

This script doesn't contain active behavior or logic — it's purely **data-driven**.
It provides the **foundation** for tools or systems that:

- Load localization data from files.
- Apply translations to UI.
- Save and restore complex UI layouts.
- Provide Unity editor tools for translation and layout management.

In essence, it's a **blueprint** describing how language and UI data are structured and stored throughout a localized Unity project.

Language File Manager

This script defines a **language file manager system** — the backbone for handling **localization data** in a Unity project.
Its main responsibilities are:

- Loading configuration data.
- Reading TSV files (tab-separated value files) that contain translations.
- Managing language selection and saving preferences.
- Extracting translated text and metadata.
- Providing easy access to localized strings and UI data during runtime.

It acts as a **bridge** between the localization data stored in files and the runtime environment of the game or app.

🧭 Overall Workflow

When the application starts, this manager:

1. Loads a LanguageSettingsData asset (contains default language info, folder paths, etc.).
2. Reads the LanguageData.tsv, MetaData.tsv, and CanvasData.tsv files.
3. Parses them into structured data for easy access.
4. Caches and stores everything for later use.
5. Retrieves localized text or metadata on demand.

⚙️ Section by Section Explanation

📁 Cached Data & Constants

This section defines internal variables and constants used by all methods.

- **cachedLanguageData** — Keeps the loaded language data in memory, so it doesn't need to reload from disk each time.
  This improves performance and prevents redundant file access.
- **CultureCodeKey** — The name of the key used in PlayerPrefs to save which language is currently selected by the player (e.g., "en-US").
- **assetsPath** — Stores the path to the folder where language files are located.
  This is built automatically from Unity's StreamingAssets directory.

🌍 Language Settings Loading

Method: LoadLanguageSettings()
Loads the language configuration file from Unity's *Resources* folder.
Process:

1. If it's already loaded (cachedLanguageData not null), it simply returns the cached version.
2. Otherwise, it looks for a resource called *"Language Data"*.
3. If not found, it logs an error and returns null.
4. If found, it stores it in the cache and returns it.

Usage:
Called before most operations to ensure the language system is ready.

🏷️ Culture Code Management

Handles saving and loading of the **selected language** (culture code).

GetSaveCultureCode()

Retrieves the saved or default culture code.

Steps:

1. Loads the language settings.
2. Starts with the default language.
3. If a saved language exists in PlayerPrefs, it overrides the default.
4. Updates the runtime variable selectedCulture.
5. Returns the resulting culture code.

Usage:

Called when the app starts to determine which language to load.

SetSaveCultureCode(newCode)

Saves a new culture code to PlayerPrefs and updates runtime data.

Steps:

1. Loads the language settings.
2. Updates selectedCulture with the new code.
3. Stores the value persistently using PlayerPrefs.SetString and PlayerPrefs.Save.

Usage:

Used when the user changes language from a settings menu.

📂 Asset Path Utilities

Method: GetLanguageAssetsPath()

Builds and returns the path to where localization files are stored.

Steps:

1. If already cached, returns the stored path.
2. Loads LanguageSettingsData to find the folder name.
3. Combines it with StreamingAssetsPath.
4. Ensures the directory exists (creates it if needed).
5. Returns the full path.

Usage:

Used internally when loading .tsv language files.

📊 Table Utilities

Method: RowAndColumnCounter(table, out rowCount, out columnCount)

Determines how many rows and columns exist in a loaded TSV table.

Behavior:

- If the table exists and has at least one entry,
  it counts the number of rows (vertical elements) and the number of columns from the first row.
- If empty or null, both counts return zero.

Usage:

Used before parsing to confirm the structure of the table is valid.

🌐 Available Languages Parsing

Method: GetAvailableLanguages()

Reads the LanguageData.tsv file and extracts a list of all available languages.

How it works:

1. Loads the settings and asset path.
2. Opens the LanguageData.tsv file.
3. Loads it into a VerticalTable structure (custom format from TSVTools).
4. Ensures the table has at least 4 rows and 3 columns.

5. For each language column:
    - Reads the culture code (row 0).
    - Reads the native name (row 1).
    - Reads the display name (row 2).
    - Reads a boolean value from row 3 that indicates if the language is enabled.
6. Adds the valid languages to availableLanguages in the settings data.

Usage:
Populates the language selection menu dynamically based on available languages in the TSV file.

🔢 ID Extraction
Method: ExtractIDs(table, languages, cultureCode)
Extracts a list of text entries (ID and translation) for a specific language.
Process:
1. Confirms that the table isn't empty.
2. Counts rows and columns to ensure validity.
3. Finds which column corresponds to the given culture code.
4. Reads from row 4 onward (skipping header rows).
5. Extracts:
    - Column 1 → The ID.
    - Column (culture column) → The localized text.
6. Converts the ID to a number and adds both to a list.

Returns:
A list of IdData objects, each containing:
- iD (unique number)
- text (localized string)

Usage:
Used to fill the in-memory list of translated text entries for the selected language.

💾 Data Loading
Method: GetAllData()
Loads all localization files (LanguageData.tsv, MetaData.tsv, CanvasData.tsv) and populates all language-related structures in memory.
Detailed behavior:
1. Loads the settings and ensures the asset path exists.
2. Checks for all three required TSV files.
3. Loads each one using LoadTableFile.
4. Checks which language is currently selected.
    - If the selected language doesn't exist, it falls back to the default.
    - Notifies the system about the language change.
5. Extracts:
    - idData from the LanguageData file.
    - idMetaData from the MetaData file.
    - idCanvasData from the CanvasData file.
6. Converts JSON metadata strings into structured IdMetaData objects.
    - If any JSON fails to parse, it logs a warning and adds a basic entry.

Usage:
Called once to fully load and initialize all localization content after language selection.

🔍 Data Retrieval
Provides methods to quickly look up localized text or metadata.
GetIDText(ids, id)

Finds a localized text string by its ID.
Steps:
1. Ensures the list isn't empty.
2. Searches for the object whose iD matches the given number.
3. Returns the corresponding text.

Usage:
Called by UI or game elements to get the localized version of a phrase.

GetIDMeta(ids, id)
Finds formatting and display metadata by ID.
Steps:
1. Ensures the metadata list isn't empty.
2. Searches for the entry with the matching ID.
3. Returns the associated IdMetaData object.

Usage:
Used to apply font, size, or alignment settings for specific localized text elements.

Summary of How It Works

Step	Purpose	Key Methods
1	Load basic settings	LoadLanguageSettings
2	Retrieve or save current language	GetSaveCultureCode, SetSaveCultureCode
3	Locate and validate TSV files	GetLanguageAssetsPath
4	Parse available languages	GetAvailableLanguages
5	Extract text and metadata for selected language	ExtractIDs, GetAllData
6	Provide access to localized content	GetIDText, GetIDMeta

Practical Example of Use
- When the app starts, GetSaveCultureCode() retrieves the previously chosen language.
- GetAllData() loads and parses the translation files into memory.
- Whenever a UI element requests text with a specific ID, GetIDText() returns the localized version.
- If that element needs its style, GetIDMeta() provides font, size, and alignment settings.
- When the user switches language, SetSaveCultureCode() updates PlayerPrefs and the language data.

In Essence
The **LanguageFileManager**:
- **Reads** localization files.
- **Parses** and **caches** them.
- **Manages** which language is active.
- **Provides** quick access to translations and UI settings.

It's a foundational tool in a localization system, keeping everything synchronized and structured — enabling seamless language switching and dynamic text rendering across the entire application.

Language Editor Utilities

Overview
Purpose:
This script defines a static utility class used **inside the Unity Editor** to manage **localization**

data (language files, translation tables, and language-related editor tools).
It provides methods for:

- Searching assets (prefabs, textures).
- Creating GUI styles for editor windows.
- Reading and writing TSV (tab-separated value) tables used for language data.
- Synchronizing and building tables that define available languages.
- Managing localized components and canvases inside Unity Editor tools.
- Searching scene objects by language ID.

It operates only inside the editor (never in runtime) because it's enclosed within a compiler directive that activates it only in editor mode.

📁 Constants & File Paths

fileData
A string constant that stores the path to a configuration file used by the language editor tools. This JSON file (LanguageFileData.json) contains serialized data for saved components and canvases.

📦 Prefab Utilities

Purpose:
To locate and load prefab assets by name directly from the Unity project.
Main Method:

- **FindPrefabByName**
  - Searches the Unity asset database for prefabs that match a given name.
  - Compares names ignoring case and file extension.
  - Returns the matching prefab as a loaded game object.
  - If nothing is found, logs an error.

Usage Example:
Used when an editor tool needs to instantiate or preview a prefab automatically by its name.

🎨 GUI Styles

Purpose:
To create reusable custom visual styles for Unity editor labels and buttons.
Main Methods:

1. **CreateLabelStyle**
   - Builds a custom label style.
   - Takes font size, a bold flag, and an alignment option.
   - Useful for creating consistent headers or section labels in custom editor windows.
2. **CreateCustomButtonStyle**
   - Creates a button style with larger text and color feedback (white normally, red on hover).
   - Used for navigation or highlighted actions inside the editor.

Usage Example:
Used when designing a custom editor window to make labels and buttons visually consistent.

📄 TSV Utilities

Purpose:
Handles importing, validating, and editing language tables stored in TSV format.
Main Methods:

1. **IsInvalidTSV**
   - Checks if a given file path exists and ends with .tsv.
   - Prevents invalid paths or wrong file types.
2. **ExtractIDsEditor**

- Reads translation entries from a loaded TSV table.
- It looks for columns corresponding to a given language code and extracts:
  - Text context (category or usage info).
  - Numeric ID.
  - Localized text.
- Returns a list of ID–text pairs for that language.

3. **InsertIDsEditor**
   - Takes a list of IDs and writes them into a specific column (language) of the TSV table.
   - Expands the table if needed.
   - Writes context, ID, and translation values in order.
4. **SyncTableWithIds**
   - Compares a TSV table with a new list of IDs.
   - Removes rows for missing IDs and adds rows for new ones, keeping order consistent.
   - Ensures the table stays synchronized with the current localization data.

Usage Example:
These methods are called when exporting or importing localization data from TSV files during development.

Table Builders
Purpose:
Creates and manipulates structured tables that describe available languages or serve as translation templates.
Main Methods:

1. **BuildTableFromAvailableLanguages**
   - Builds the first few rows of a new TSV table using the available language list.
   - Inserts culture code, display name, and availability flags.
   - Ensures proper column count based on the number of languages.
2. **ReplaceTopRows**
   - Takes an existing table and replaces its first rows (header info) with another set.
   - Used when refreshing metadata but keeping existing translations.
3. **OpenEditorWindowWithComponent**
   - Opens the *Language File Manager* window and adds a new editable component.
   - Waits for the window to initialize, then injects data such as ID, font, and alignment.
4. **OpenEditorWindowWithCanvas**
   - Similar to the above but loads an entire saved canvas layout from JSON data.
5. **CreateLanguageDataBase**
   - Builds a default language database with predefined identifiers and names.
   - Used to initialize a project's language settings when none exist yet.
6. **IsIDInLanguageList**
   - Checks whether a given numeric ID exists in the saved component list.
7. **IsIDInCanvasList**
   - Checks whether a given canvas ID is present in saved JSON data.
8. **FindDuplicateIDs**
   - Scans a list of IDs and returns any duplicates.
9. **LoadLanguageIDs**
   - Reads all component IDs from the stored JSON file.

Usage Example:
These are used to build, save, and reload the internal data structure of the localization system.

Asset & GUI Utilities
Purpose:

Provides general-purpose tools for custom Unity editor GUIs.

Main Methods:

1. **FindTextureByName**
   - Searches for a texture by filename in the Unity assets and loads it.
   - Used to find icons for custom editor components.
2. **DrawLabeledTextField**
   - Displays a label with a text field horizontally aligned.
   - Returns updated text from user input.
3. **DrawLabeledIntField**
   - Like the above, but specifically for integer input.
4. **DrawColoredBox**
   - Draws a colored rectangular area (background) and executes a block of UI code inside it.
   - Useful for visually grouping related content.
5. **DrawArrowButton**
   - Renders a left or right arrow button for navigating between IDs.
   - Automatically disables itself if at min or max.
   - Supports Undo operations and executes a custom action when pressed.

Usage Example:

Used throughout custom editor windows (for navigation, grouping, and visual clarity).

🧩 Component Management

Purpose:

Handles the default set of localized components and canvases used by the language editor.

Main Methods:

1. **AddDefaultLanguageComponents**
   - Adds default component and canvas entries to lists when creating a new configuration.
   - These entries represent prebuilt localization elements ready to edit or duplicate.
2. **DisplayComponentIcon**
   - Draws a small icon for each type of language component (text, dropdown, input field, etc.).
   - Chooses the correct icon and tooltip automatically.
3. **SearchByID**
   - Searches all GameObjects in the active scene for a matching localization ID.
   - Checks several component types that can contain IDs (text, dropdown, file, etc.).
   - If found, selects the objects in the hierarchy.
   - If not found, offers to open a scanner window.

Usage Example:

Used for debugging or quickly finding where a localization entry is used in the scene.

🔍 ID Matching Helpers

Purpose:

Internal helper methods used by the SearchByID feature.

Main Methods:

1. **TryMatchComponentID**
   - Checks if a given GameObject has a single component with an integer ID field matching the target.
2. **TryMatchListComponentID**
   - Checks components that contain a *list* of sub-items, and finds if any of those items has the matching ID.
3. **FindNearestValidID**
   - Finds the closest available ID before or after the current one.

    - Useful for navigation arrows (previous/next ID).

Usage Example:
Used when scanning scenes for specific translations or navigating between them efficiently.

⚙️ Usage Summary

When to Use:
This utility class is not attached to GameObjects directly.
It's meant to be used by **other editor scripts and windows**, such as:

- Language File Manager
- Language ID Scanner
- Custom localization editors

Common Tasks It Helps With:

- Loading or creating translation tables (TSV files).
- Synchronizing language IDs between files and components.
- Designing custom editor UI with better formatting.
- Quickly finding and editing localized elements in the scene.
- Managing all supported languages and their availability.

🧠 Summary

In short:
The class LanguageEditorUtilities is a **Swiss Army Knife** for Unity localization tools.
It centralizes all repetitive editor-related operations — file loading, GUI creation, ID validation, and scene searching — into a single helper module.
It's essential for maintaining multilingual data consistency and editor usability when building custom localization systems.

Canvas Manager

🧩 Overview

This script defines a **Canvas Manager**, a helper used to **save**, **rebuild**, and **validate** user interface hierarchies made with Unity's **Canvas system**.
It extracts all the important layout information from a Canvas object — positions, anchors, scaling, and components — and can later reapply that information to **recreate** or **synchronize** the same layout programmatically.
It's designed mainly for editor or tool automation, ensuring that a Canvas layout can be exported, stored, and reconstructed exactly the same way later.

⚙️ Core Concept

The manager deals with a **data structure** (called *CanvasStructure*) that represents everything about a Canvas:

- The **name** of the Canvas.
- Its **layers** (a list of UI elements organized by hierarchy).
- Layout information of every element, such as position, size, rotation, anchors, and pivot.
- Metadata about the main Canvas components (Canvas, CanvasScaler, and GraphicRaycaster).

So, the script is essentially a **serializer** and **rebuilder** for Canvas hierarchies.

📁 Constants and Ignored Prefixes

Variable: ignoredPrefixes

- A static list containing a few words:
    - "TMP SubMeshUI", "Dropdown List", and "Blocker".

- These represent temporary or internal objects created automatically by TextMeshPro or Unity's UI system.
- Objects whose names start with any of these are **ignored** when analyzing the Canvas hierarchy.
  This prevents unnecessary internal objects from being included in exported data.

📦 Main Public Methods

These are the entry points of the system — what an editor tool or script would call.

1. Extract Canvas Data

Purpose:

Takes a Canvas object from the scene and converts it into a *CanvasStructure* that represents all its metadata and hierarchy.

How it works:

1. It begins by creating an empty CanvasStructure with all its sub-sections initialized.
2. It checks the Canvas hierarchy for duplicate child names — duplicates can cause issues because they make it impossible to match elements when reapplying data later.
3. It stores the Canvas name.
4. It temporarily activates all children in the hierarchy so that no hidden element is skipped during extraction, remembering their previous active states.
5. It then extracts metadata from the main Canvas components (Canvas, CanvasScaler, GraphicRaycaster) and stores their configuration.
6. Next, it builds a full list of hierarchy paths — from root to every visible child — and stores their RectTransform information (position, anchors, rotation, scale, etc.).
7. Finally, it restores all the objects to their original active or inactive state.

Usage:

Used when saving or exporting the layout of a Canvas for later reconstruction.

2. Apply Canvas Data

Purpose:

Takes a previously extracted *CanvasStructure* and applies its layout back to an existing Canvas object in the scene.

How it works:

1. It checks that the provided structure is valid — all layers must have the same number of names and transform data.
2. It reconfigures the Canvas object's components to match the stored settings (Canvas, CanvasScaler, GraphicRaycaster).
3. For each layer path in the stored data:
   - It searches for objects in the existing Canvas hierarchy that match the stored names.
   - If it finds a matching element, it re-applies its stored RectTransform settings (position, anchors, size, rotation, etc.).
   - If an element is missing, it logs a warning message identifying the missing layer.

Usage:

Used when synchronizing an existing Canvas to match a stored configuration — for instance, restoring a UI layout from a saved file.

3. Create Canvas From Structure

Purpose:

Generates an entirely new Canvas GameObject based on the data from a *CanvasStructure*.

How it works:

1. It ensures that the provided data structure is valid and has a defined name.

2. It creates a new GameObject that includes the essential Canvas components (RectTransform, Canvas, CanvasScaler, GraphicRaycaster).
3. It applies all stored metadata to these components.
4. It then reconstructs the full hierarchy layer by layer:
    - For each layer, it creates GameObjects with the stored names.
    - It assigns RectTransform data to replicate their exact layout.
    - Each created element is given an Image component (with a random debug color for visibility) and an Outline component (for better visualization in the Editor).
    - The hierarchy is properly nested to mirror the original structure.

Usage:
Used to completely rebuild a Canvas layout programmatically, for example, generating a UI dynamically or previewing saved UI templates.

Hierarchy Processing Methods
These methods are internal helpers that handle hierarchy traversal and validation.

1. Generate Canvas Layers
- Scans the Canvas hierarchy recursively.
- For every path from root to leaf (a sequence of nested UI elements), it builds a record that includes:
    - The names of each object in the path.
    - The RectTransform data for each one.
- Returns an array of *CanvasLayers*, each representing a full branch of the UI tree.

Usage:
Called by the extraction process to build the internal hierarchy representation.

2. Contains Duplicate Siblings
- Checks every level of the hierarchy to ensure no two sibling objects share the same name.
- Ignores internal or system-generated objects.
- If duplicates are found, it logs a warning and highlights the duplicate in the Unity Editor.

Usage:
Used to validate the hierarchy before extraction or rebuilding.

Utility Methods
These are supporting internal methods used by the main functions.

1. Convert To RectTransform Data
- Takes a RectTransform component and reads its position, rotation, scale, anchor values, pivot, and size.
- Returns a simplified data object (RectTransformData) that can be serialized easily.

Usage:
Used during extraction and when building the hierarchy structure.

2. Activate All Children
- Recursively goes through all child objects under a parent.
- Temporarily enables each inactive GameObject while recording its previous active state.
- Skips ignored elements.

Usage:
Used during data extraction to make sure all elements, even hidden ones, are captured.

3. Find Child By Name
- Searches for a direct child of a given parent with a specific name.

- Returns that child if found, otherwise null.

Usage:
Used when applying or rebuilding the hierarchy to find matching objects.

4. Is Ignored Name
- Checks whether an object's name starts with any of the prefixes in the ignored list.
- Returns true if it should be skipped.

Usage:
Used throughout the hierarchy processing and activation functions.

🧩 Metadata Extraction and Application
These methods handle component-specific data — extracting and reapplying the settings of Canvas, CanvasScaler, and GraphicRaycaster.

1. Populate Canvas Metadata
- Reads the main components attached to a Canvas object.
- Stores their parameters into a CanvasStructure object:
  - RectTransform layout.
  - Canvas rendering options (render mode, sorting, pixel settings).
  - CanvasScaler settings (scaling mode, reference resolution, match mode).
  - GraphicRaycaster configuration (blocking mask and behavior).
- Each group of settings is stored in a sub-structure: CanvasData, CanvasScalerData, and GraphicRaycasterData.

Usage:
Called during extraction to capture all configuration data of the Canvas.

2. Update Canvas Metadata
- Applies all stored component configurations back to a Canvas object.
- Writes every property from the saved structure to the corresponding components.
- Ensures that after rebuilding, the new or updated Canvas behaves exactly like the original.

Usage:
Used when applying saved data or creating a new Canvas from a structure.

🧠 Summary of the Workflow
1. **Extract Phase**
   - The tool takes a Canvas GameObject and turns it into a structured, serializable object (*CanvasStructure*).
   - It includes the hierarchy layout, RectTransform information, and component settings.
2. **Save Phase (optional)**
   - The resulting structure can be serialized to a file or stored in memory for later use.
3. **Apply Phase**
   - A previously stored structure can be applied back to an existing Canvas to synchronize layout and metadata.
4. **Create Phase**
   - The same structure can also be used to generate a completely new Canvas that visually matches the original.

🧭 Practical Usage
- **In a localization or UI editor tool:**
  The manager can back up, clone, or restore UI canvases exactly as they were.

- **In procedural UI generation:**
  It can build entire UI screens at runtime or in the editor from structured data.
- **In version control or data comparison:**
  Developers can verify that two canvases share identical layout and component metadata.
- **In debugging UI problems:**
  The system can detect duplicate element names that may cause hierarchy mismatches.

In essence, the **Canvas Manager** acts as a **bridge** between Unity's visual UI layout and structured data representation.
It ensures that every element of a Canvas can be safely extracted, stored, validated, and reconstructed with precision — maintaining complete integrity of the layout and its core components.

ID Exists Attribute

This script adds a **custom validation system** inside Unity's Inspector to ensure that integer IDs used by localized UI elements are **unique** across the project. It combines a **custom attribute** and a **property drawer** to automatically check IDs from a configuration file and display a warning if a duplicate is found.
Let's break it down carefully and explain every element, variable, and method, including its usage:

📦 Purpose Overview

The goal of this system is to help developers **avoid duplicate identifiers** in multilingual UI systems.
When working with localization, each component or canvas may have a unique ID. This script automatically verifies that IDs are not repeated, by comparing them with data stored in a JSON file located in ProjectSettings/LanguageFileData.json.
If an ID is duplicated, Unity's Inspector will visually warn the user by showing a **yellow highlight** and a **warning message** below the field.

□ Part 1 — The Attribute: IDExistsAttribute

Description

The attribute marks integer variables that must be validated for uniqueness.
By adding [IDExists] above an integer field, Unity will trigger the custom validation behavior from the drawer defined later.

Variable

- **searchCanvas**:
  A boolean value that defines which category of IDs should be checked.
  - false: check against component IDs.
  - true: check against canvas IDs.

Constructor

- **Purpose:** Initializes the attribute when applied to a field.
- **Parameter:**
  searchCanvas — optional, defaults to false. If set to true, validation targets canvas IDs instead of component IDs.

Example Usage

```
[IDExists] public int componentID = 0;     // Checks component IDs.
[IDExists(true)] public int canvasID = 0;  // Checks canvas IDs.
```

When these fields are displayed in the Unity Inspector, the drawer (explained below) runs automatically.

Part 2 — The Custom Drawer: IDExistsDrawer

This part only runs inside the Unity Editor.
It defines **how the attribute behaves visually** and **how the validation is performed**.

Constants and Cached Data

1. **FilePath**
   - A string constant defining where the JSON configuration file is located: "ProjectSettings/LanguageFileData.json".
   - This file contains the stored list of IDs used in the project.
2. **cachedData**
   - A static variable that holds the parsed content of the JSON file.
   - It prevents repeatedly reading the file every frame, improving performance.
3. **lastFileWriteTime**
   - Stores the timestamp of the last time the JSON file was modified.
   - Used to detect changes and reload the data only when necessary.

Rendering the Inspector (Method: OnGUI)

Purpose: Controls how the field is drawn in Unity's Inspector, including the warning message if the ID is duplicated.

Steps Inside OnGUI:

1. Retrieve the custom attribute assigned to the field.
2. Save the current interface color to restore later.
3. Check if the field's integer value already exists in the ID list using CheckIDExists().
4. If the ID is already used:
   - Change the GUI color to yellow for emphasis.
   - Draw a help box message below the field warning about the duplicate ID.
5. Draw the integer field normally using Unity's editor tools.
6. Restore the GUI color after rendering.

Effect in Editor:

- The field will appear highlighted and show a warning if the ID already exists in the system.

Adjusting Field Height (Method: GetPropertyHeight)

Purpose:

Ensures the Inspector layout has enough vertical space to show both the field and the warning message.

Logic:

1. If the field is not an integer, return the normal height.
2. If it's an integer and a duplicate ID is detected:
   - Add extra vertical space to fit the warning help box.
3. Otherwise, use the default height.

This ensures the Inspector layout remains tidy and readable.

Checking for Duplicates (Method: CheckIDExists)

Purpose:

Validates whether a given integer ID already exists in the JSON data.

Step-by-Step Process:

1. **File Existence Check:**
   - If the JSON file doesn't exist, stop the check and return false.

2. **File Timestamp:**
    - Retrieve the last modification time of the file.
3. **Conditional Reload:**
    - Compare the current write time with the cached one.
    - If the file has been modified since the last read:
        - Reload the JSON content.
        - Parse it into a structured data object (LanguageFileManagerWindowData).
        - Update the cached timestamp.
4. **Duplicate Search:**
    - Depending on checkCanvas, search in the correct list:
        - If true → compare against the list of canvasID.
        - If false → compare against the list of component IDs.
    - Returns true if a matching ID is found, meaning it's already in use.

How It Works in Practice

1. A developer adds [IDExists] above a variable in a MonoBehaviour script.
2. When Unity displays the field in the Inspector:
    - The drawer (IDExistsDrawer) activates.
    - It reads the ID value entered by the user.
    - The script checks the JSON configuration for duplicates.
3. If the ID already exists:
    - The field turns **yellow**.
    - A warning message appears below it indicating the conflict.
4. If the ID is unique:
    - The field remains normal.

This gives **instant visual feedback**, helping maintain a clean and conflict-free ID system for localized elements.

Summary Table

Element	Type	Purpose
IDExistsAttribute	Custom Attribute	Marks integer fields for uniqueness validation.
searchCanvas	Boolean	Chooses between validating canvas or component IDs.
FilePath	Constant String	Path to the JSON file storing registered IDs.
cachedData	Static Variable	Stores the loaded JSON content to avoid re-reading.
lastFileWriteTime	Static DateTime	Detects when the file has changed to refresh the cache.
OnGUI	Method	Draws the property and shows a warning if duplicated.
GetPropertyHeight	Method	Adjusts layout height to fit the warning box.
CheckIDExists	Method	Performs the validation by reading JSON data.

In Short

This script acts as a **real-time validator** inside Unity's Inspector that:

- Reads a configuration file with registered IDs.
- Checks whether the current field's ID already exists.
- Displays an automatic warning if there's a duplicate.

It's a **useful safety mechanism** for maintaining consistency in complex, multilingual user interface systems.

Font And Alignment Utility / Font And Alignment Utility TMP

Purpose Overview

Both utility classes provide functionality for **managing fonts and text alignment** in Unity projects with multilingual support. They serve two main purposes:

1. **Font Access**: Retrieve a font or font asset either by its position in a language-specific list (index) or by providing the font asset itself to find its index.
2. **Alignment Conversion**: Translate between **integer codes** stored in localization or configuration data and the respective Unity alignment enums:
    - Legacy Unity UI: TextAnchor
    - TextMeshPro: TextAlignmentOptions

This ensures consistent text formatting across different languages and UI components, while allowing a simple integer representation for storage or localization purposes.

Font Management

1. Retrieving a Font by Index

- **Input:** 1-based index of the font in the language-specific font list.
- **Process:**
    - Load the current language settings.
    - Access the list of available fonts (fontList for legacy UI, TMPFontList for TextMeshPro).
    - Return the font corresponding to the index.
- **Output:** The requested font asset, or null if the index is invalid.

2. Retrieving the Index of a Font

- **Input:** A font asset.
- **Process:**
    - Load the current language settings.
    - Iterate over the font list to find a match.
    - Return the **1-based index** of the font if found.
- **Output:** Index of the font or 0 if the font is not in the list.

Usage Example (neutral):

- Get the third font in the list for UI text: font = GetFontByIndex(3).
- Determine the index of a specific font in the language settings: index = GetFontIndex(myFont).

Alignment Conversion

1. From Integer Code to Alignment Enum

- **Input:** An integer representing the alignment.
- **Process:**
    - Each integer maps to a specific alignment value.
    - For legacy UI, integers map to TextAnchor values (like UpperLeft, MiddleCenter, LowerRight).
    - For TextMeshPro, integers map to TextAlignmentOptions values (like TopLeft, Center, MidlineRight, CaplineFlush, etc.).
- **Output:** The corresponding enum value used by the UI system.

Example:

- Integer 1 → Legacy UI: UpperLeft, TMP: TopLeft.
- Integer 8 → Legacy UI: MiddleCenter, TMP: Center.

If the integer is invalid or unrecognized, a warning is logged, and a default alignment is used.

2. From Alignment Enum to Integer Code

- **Input:** An alignment enum value (TextAnchor or TextAlignmentOptions).
- **Process:**
    - Each enum value is mapped back to its stored integer representation.

    - This allows the system to **serialize** alignment in a simple, language-independent format.
- **Output:** Corresponding integer code.
- **Fallback:** Returns a default integer if the enum is unrecognized and logs a warning.

Usage Example (neutral):

- Save the alignment of a text field as an integer for storage in localization data.
- Restore a saved integer alignment code to apply it to a UI element.

Variables Overview

Variable / Property	Purpose
fontListIndex	1-based index to select a font from the list.
font / TMPFont	The font asset to search for its index.
alignment	Integer code representing a text alignment.
newAlignment	Enum value corresponding to an integer code.
alignmentValue	Integer code corresponding to an enum value.
fonts	List of fonts or TMP font assets loaded from language settings.

Method Summary

Method	Description	Input	Output
GetFontByIndex	Returns a font by its position in the list	Font index	Font asset (or null)
GetFontIndex	Returns the index of a given font	Font asset	Index (1-based) or 0
ConvertToTextAnchor / ConvertToTextAlignmentOptions	Converts integer code to UI alignment enum	Integer	Enum value
ConvertToAlignmentCode	Converts UI alignment enum to integer code	Enum value	Integer

Usage Scenario

1. **Localization System**:
    - Each language may define its own font list.
    - Text elements store alignment as integers in a language file or database.
    - When displaying text, integer codes are converted to actual alignment enums.
2. **Font Selection**:
    - UI designers can reference fonts by index, simplifying configuration.
    - Code can also retrieve the index of a font for storage in settings.
3. **Compatibility**:
    - Supports both legacy Unity UI and TextMeshPro, ensuring smooth migration or coexistence of both systems.

✅ **Key Takeaways**

- These utilities **bridge numeric data storage and actual font/alignment objects** used by the UI.
- They support **bidirectional conversion**: integer ↔ alignment enum.
- They provide **safe access to fonts** without hardcoding assets in scripts.
- Both systems are **interchangeable in logic**, only differing in the specific enum types they use for alignment.

Language Settings Data

Purpose Overview

This script defines a **central configuration for localization** in Unity projects. It uses a data container (ScriptableObject) to store settings for languages, fonts, culture, UI elements, and metadata for dynamically localizable text. It ensures that both standard Unity UI and TextMeshPro components can access language-specific assets and configurations at runtime. It also includes an editor utility to quickly open the asset in the Unity Editor for modification.

Main Components

1. Folder and Language Settings

These variables define where the language files are stored and the default language used by the project:

- **Folder Name**: Directory where all language files are located. This helps the system know where to load localized text from.
- **Default Language**: The fallback language if the player has not selected a language yet.

Usage: The system uses these settings to locate files and determine which language to display by default.

2. Font Configuration

These variables manage font assets used for different UI systems:

- **Font List Data**: Holds the list of fonts for standard Unity UI elements. This allows different languages to use specific fonts that support required characters.
- **Font List Data TMP**: Holds the list of fonts for TextMeshPro UI elements. TMP requires its own asset format, so a separate list ensures proper rendering.

Usage: When displaying text, the localization system retrieves the correct font from these lists by index or asset reference.

3. Canvas Configuration

These settings are related to UI components:

- **Error Language Tool**: A UI prefab shown when language loading fails or configuration errors occur. This provides a visual fallback so the user knows something went wrong.

Usage: Shown automatically if the system cannot load the selected language or font configuration.

4. Extracted Data

This section stores runtime and metadata information:

- **Selected Culture**: The currently active culture or locale, such as "en-US" or "pt-BR".
- **Available Languages**: A list of all languages that are defined and optionally available in the build.
- **Id Data**: Stores dynamic text IDs used outside of Unity Canvas UI components. These IDs help the system find and replace text dynamically.
- **Id MetaData**: Additional information about each ID, such as font size, tags, or special formatting instructions.
- **Id Canvas Data**: Similar to Id Data but specifically for Unity Canvas components. This allows precise control over UI text localization.

Usage: During runtime, the localization system reads these lists to update text dynamically, switch languages, and ensure the correct font and alignment are applied.

Editor Utility

The script includes a small editor helper to quickly open the LanguageSettingsData asset:

- **Menu Option**: Adds a menu item in Unity under Window > Language > Language Settings.
- **OpenLanguageSettingsData Method**:
    1. Loads the LanguageSettingsData asset.
    2. Checks if the editor window for the asset is already open.
    3. If it exists, brings it to focus.
    4. If not, opens it in the default property editor.

Usage: This is purely an editor convenience to make it easy for developers to modify localization settings without searching for the asset manually.

Variables Overview

Variable	Purpose
folderName	Specifies where language files are stored.
defaultLanguage	The fallback language when no selection exists.
fontListData	Fonts for legacy Unity UI.
fontListDataTMP	Fonts for TextMeshPro UI.
errorLanguageTool	Prefab to show when loading fails.
selectedCulture	Currently selected culture/locale.
availableLanguages	List of all defined languages.
idData	IDs for dynamic text outside Canvas UI.
idMetaData	Metadata for each localized ID.
idCanvasData	IDs for text inside Canvas UI.

How It Works in Practice

1. At project start or language change:
    - The system reads selectedCulture and loads the corresponding language files from folderName.
    - It retrieves the proper font from fontListData or fontListDataTMP for rendering.
    - Dynamic text IDs are matched with entries in idData or idCanvasData to update UI and non-UI elements.
2. If an error occurs:
    - The errorLanguageTool prefab is displayed to indicate a configuration problem.
3. Editor convenience:
    - Developers can open this asset quickly via the menu item and edit languages, fonts, or culture settings.

In short, this script **centralizes all localization data**, ensuring that fonts, languages, and text IDs are managed consistently for both runtime usage and editor modifications.

Language Settings Data Editor

Purpose Overview

This script provides a **custom interface in the Unity Editor** for managing the localization settings stored in the LanguageSettingsData asset. It allows developers to:

- Choose the default language for the project.
- Assign and manage font lists for both standard Unity UI and TextMeshPro components.
- Set the folder path where language files are stored.
- Visualize runtime-extracted localization data like selected culture, available languages, and IDs.
- Work within Unity's editor workflow, supporting undo operations and asset saving.

Essentially, it makes the LanguageSettingsData asset **easier and more intuitive to configure** without manually editing fields in the inspector.

Main Components

1. Fields

- **availableCultureDisplayNames**: A list of human-readable names for all available system cultures. Used to populate a dropdown menu for selecting the default language.
- **availableCultures**: All the system-supported cultures, which include language and regional variations.
- **currentSelectedCultureIndex**: Tracks which culture is currently selected in the dropdown menu.

Usage: These fields are internal to the editor and manage how the dropdown for selecting the default language is displayed and updated.

2. Editor Initialization

- When the editor is opened or the asset is selected, it runs an initialization process to **load all available system cultures** and store their display names.
- This ensures the dropdown menu for selecting the default language is fully populated and user-friendly.

Usage: Automatically populates the language selection dropdown without requiring manual input.

3. Inspector GUI Layout

The main interface is organized into sections for clarity:

a) Script Reference

- Displays the reference to the asset script itself in a **read-only field**.
- Ensures developers know which script is being edited.

b) Archives Location

- Provides a text field for the folder where localization files are stored.
- Helps the system know where to load language data from.

c) Default Language Selection

- Dropdown menu using system cultures.
- Automatically sets the defaultLanguage in the asset based on the selection.
- Updates the internal tracking index to match the chosen culture.

d) Font List Data (Regular Unity UI)

- Field to assign a ScriptableObject that contains standard font references.
- Displays the list of fonts in an expandable view.
- Allows easy editing of font references without opening the asset separately.

e) Font List Data (TextMeshPro)

- Similar to the regular font section, but specifically for TextMeshPro fonts.
- Supports multi-language text rendering with TMP assets.

f) Canvas Log Reference

- Field to assign a prefab used to display errors in the UI if language loading fails.
- Optional but improves the feedback workflow in the game.

g) Extracted Runtime Data

- Read-only fields for **visualizing data that exists at runtime**:
  - Selected culture
  - List of available languages
  - Text IDs and metadata
  - Canvas-specific IDs
- These fields are disabled to prevent accidental edits, as they reflect runtime-extracted information.

4. Saving Changes

- The editor automatically **marks the asset as modified** when fields are changed.
- Saves the changes to disk to ensure they persist between editor sessions.

- Integrates with Unity's undo system so changes can be reverted if necessary.

How It Works in Practice

1. **Open Asset in Editor**
   - Developer selects the LanguageSettingsData asset.
   - The editor automatically loads available system cultures for language selection.
2. **Modify Settings**
   - Set the folder for language files.
   - Choose a default language.
   - Assign font lists for both Unity UI and TMP text components.
   - Optionally assign a prefab to display errors.
3. **View Runtime Data**
   - The editor shows the currently active culture and extracted data such as IDs, metadata, and canvas text IDs.
   - These fields cannot be edited in the editor but provide a live overview.
4. **Save Changes**
   - Changes are automatically tracked and saved when the user modifies fields.
   - The editor ensures integration with Unity's asset and undo systems.

Variables and Their Purpose

Variable	Purpose
availableCultureDisplayNames	Display names for the culture dropdown.
availableCultures	All system-supported cultures for selection.
currentSelectedCultureIndex	Tracks the currently selected culture in the dropdown.

Asset Fields (Editable in Inspector via this editor)

Field	Purpose
folderName	Path to language file archives.
defaultLanguage	Default language for the project.
fontListData	Reference to standard font list asset.
fontListDataTMP	Reference to TMP font list asset.
errorLanguageTool	Prefab used for showing errors in UI.
selectedCulture	Currently active culture at runtime (read-only).
availableLanguages	List of all languages detected (read-only).
idData	IDs for runtime localization (read-only).
idMetaData	Metadata for IDs (read-only).
idCanvasData	Canvas-specific localization IDs (read-only).

Summary

This editor script enhances workflow by:

- Centralizing all language, font, and folder settings in a single interface.
- Allowing easy selection of default language with system culture support.
- Providing intuitive management of font assets for both Unity UI and TMP.
- Visualizing runtime data without risking accidental edits.
- Ensuring changes are saved and tracked seamlessly in the Unity Editor.

It essentially acts as a **user-friendly bridge** between the raw localization data in LanguageSettingsData and the developer, making configuration faster, safer, and more visual.

Language Data Creator
Purpose Overview

This script is designed to **ensure that a LanguageSettingsData asset exists** in a Unity project. This asset is essential for managing localization settings such as available languages, default language, fonts, and UI localization data. The script provides two main functionalities:

1. **Manual asset creation** via a menu option in the Unity Editor.
2. **Automatic asset creation** when the project loads, if the asset does not already exist.

This ensures that developers always have a ready-to-use LanguageSettingsData asset, streamlining the setup for localization systems.

Main Components

1. Manual Asset Creation

Purpose: Allows developers to explicitly create the LanguageSettingsData asset from a Unity Editor menu.

Key Steps:

- **Define the folder and asset path:**
  - The asset is stored under the Resources folder, so it can be loaded at runtime without hardcoding a path.
  - Example path: Assets/Resources/Language Data.asset.
- **Check for the folder existence:**
  - If the Resources folder does not exist, it is created automatically.
- **Check for an existing asset:**
  - If an asset with the same name already exists, the user is prompted with a dialog asking whether to overwrite it.
- **Create the asset if approved or missing:**
  - Generates a new instance of the LanguageSettingsData asset.
  - Registers it in the project and marks it as modified so Unity knows it needs to save changes.
- **Save and refresh:**
  - Saves the asset to disk and refreshes the project view so the new asset appears immediately.
  - Highlights the newly created asset in the Project window for easy access.

Usage:

- Open the Unity Editor.
- Navigate to the menu option labeled “Assets → Create → Language → Language Data.”
- Confirm any overwrite prompts if an asset already exists.
- The asset is created and ready for editing.

2. Automatic Asset Initializer

Purpose: Ensures the asset is automatically present whenever the Unity project loads, without requiring manual intervention.

Key Steps:

- **Run on project load:**
  - A special mechanism is used to execute code automatically when the editor loads.
- **Delay the execution slightly:**
  - Ensures that other editor systems are fully initialized before creating the asset.
- **Check if the asset exists on disk:**
  - If the asset is missing, it calls the same manual creation method to generate it automatically.

Usage:

- Simply open the Unity project.
- If no LanguageSettingsData asset exists, it will be created automatically in the Resources folder.
- No action is needed from the developer for initial setup.

Variables and Their Purpose

Variable / Constant	Purpose
folder	Defines the location where the asset should be stored (Resources folder).
assetPath	Full path to the asset file, used to check existence and create the asset.
asset	Represents the newly created LanguageSettingsData instance.

Methods and Their Purpose

Method / Function	Purpose	Usage
CreateLanguageDataAsset	Handles creation of the LanguageSettingsData asset manually, including folder creation, overwrite prompts, asset registration, saving, and selection in the editor.	Called by the menu option or by the automatic initializer if asset is missing.
Static initializer of the automatic asset initializer	Runs automatically when the project loads, checking for the asset and creating it if missing.	Ensures the project always has a LanguageSettingsData asset ready for use.

How It Works in Practice

1. **Manual workflow:**
    - Developer chooses the menu option to create the asset.
    - If a previous asset exists, the user decides whether to overwrite it.
    - The asset is generated, saved, and highlighted in the Project view.
2. **Automatic workflow:**
    - Upon loading the project in Unity Editor, the system checks for the asset.
    - If missing, it creates the asset automatically in the same location.
    - The developer can then immediately start configuring the asset without manual creation.

Summary

This script **simplifies the setup of localization in Unity projects** by guaranteeing the presence of the LanguageSettingsData asset. It provides:

- A **manual menu option** to create or overwrite the asset.
- An **automatic initializer** that creates the asset if missing on project load.
- Automatic folder creation and project integration, so the asset is ready to use in runtime and editor workflows.

It is a **setup tool** rather than a runtime tool, ensuring that localization systems have their required asset in place with minimal effort.

Language Font List Data / Language Font List Data TMP

Purpose Overview

These scripts provide a system for **managing font assets in Unity projects** for localization purposes. They define two types of assets:

1. **Legacy Unity Fonts** – for standard Unity UI text components.
2. **TextMeshPro Fonts** – for advanced text rendering using TMP components.

Each asset type is stored as a **ScriptableObject**, which is a Unity object type that can persist data in the project. Additionally, each has a **custom editor interface** that allows **drag-and-drop of font assets**, making it easier for developers to populate the lists without manually selecting each font.

Components of Each Script

1. Data Containers

These are ScriptableObjects that hold lists of fonts.

Variables:

Variable	Purpose
fontList (for legacy fonts)	Stores a list of standard Unity Font objects used for localization. Initialized to an empty list to prevent errors.
TMPFontList (for TMP fonts)	Stores a list of TextMeshPro font assets. Also initialized empty.

Usage:

- The project maintains a single asset per type, which is referenced in localization systems to apply fonts to UI elements or text components dynamically.

2. Custom Editors

The custom editors enhance the default Unity Inspector for these ScriptableObjects, providing a **visual, interactive interface**.

Features and Variables:

Feature	Purpose
Drag-and-drop area	A large rectangle in the inspector where fonts can be dropped. It visually indicates where developers should drag font assets.
Event handling	Detects when the user drags or drops assets in the area and updates the list accordingly.
Undo recording	Ensures that adding fonts can be undone using Unity's undo system.
Duplicate prevention	Only adds fonts to the list if they are not already present, avoiding repeated entries.
Mark dirty	Marks the asset as modified so that Unity knows to save changes to disk.
Default inspector	Still draws the regular inspector fields for any other properties or future additions.

Usage Workflow:

1. Open the ScriptableObject in the Unity Editor.
2. Drag one or more font assets into the designated drop area.
3. The fonts are added to the list, duplicates are ignored automatically.
4. Changes are saved and tracked by Unity's system for undo/redo and asset persistence.

Note: The same logic applies to both legacy fonts and TMP fonts, just with their respective font types.

3. Shared Utilities

Both editors rely on small utility functions (e.g., CreateLabelStyle) to customize the appearance of the drag-and-drop label. These handle font size, boldness, and alignment within the drop area for better UX.

How It Works in Practice

- **Legacy Fonts:**
  Developers create a ScriptableObject for standard fonts. Dragging fonts into the editor automatically updates the fontList, making them available for UI components or localization systems.

- **TMP Fonts:**
  Similarly, TMP fonts are managed via a separate ScriptableObject. The same drag-and-drop logic allows adding TMP_FontAsset objects, which are then referenced by TextMeshPro-based UI elements.
- Both systems are **fully editor-only enhancements**; at runtime, the lists are read by localization tools to assign fonts dynamically according to the current language.

Summary
These scripts provide a **centralized, user-friendly way to manage fonts** for multilingual Unity projects. The key advantages are:

1. **ScriptableObject containers:** Keep font references persistent and reusable.
2. **Drag-and-drop editors:** Streamline adding and managing fonts.
3. **Duplicate prevention and undo support:** Ensure data integrity and safe editing.
4. **Separate handling for legacy and TMP fonts:** Provides flexibility depending on the text system used.

In short, they make font management **visual, efficient, and robust**, reducing setup errors in localization workflows.

Language Initialization

Purpose Overview
This script is responsible for **initializing language settings** and managing **font assets** for a Unity project. It handles both **legacy Unity fonts** and **TextMeshPro fonts**, ensuring that they are properly loaded, stored, and maintained in memory for use in localization systems.
It also provides functionality to save and reload font lists from text files, as well as to load fonts from **asset bundles**, which is useful for distributing fonts separately from the main game build.

Key Variables

Variable	Purpose
settingsData	Stores the main language settings object, which contains information like default language, available languages, and references to font lists.
fontListData	Holds the list of legacy Unity Font objects, either loaded from resources or asset bundles.
fontListDataTMP	Holds the list of TextMeshPro fonts, similar to fontListData, but for TMP text components.
folderPath	The system path where font files and asset bundles are stored, used to save and load font lists.

Core Methods and Their Functionality
1. Initialize Language Settings

- **Purpose:** Called automatically at game start to prepare language and font systems.
- **Workflow:**
  1. Defines the folder path for font data.
  2. Loads the language settings from resources.
  3. Sets the application's language based on saved preferences or system culture.
  4. If running outside the editor, it ensures the folder exists, saves font lists to files, and loads fonts from asset bundles.

Usage: This method runs automatically when the game starts, initializing all language and font-related data.

2. Load Language Settings

- **Purpose:** Loads the LanguageSettingsData asset from resources.
- **Functionality:**

    - Checks if the asset exists; logs an error if not.
    - Extracts both regular and TMP font lists for later use.

Usage: Provides access to language and font configurations for runtime operations.

3. Setup Default Language

- **Purpose:** Determines which language the application should use at startup.
- **Functionality:**
    - Retrieves the system culture (e.g., "en-US").
    - Checks if a saved language exists in player preferences.
    - If not, selects either the system culture or the default language.
    - Saves the chosen culture for future sessions.
    - Loads all relevant language data into memory.

Usage: Ensures the application displays text in the correct language when first run or when user preferences exist.

4. Load Fonts From Asset Bundles (Legacy and TMP)

- **Purpose:** Loads fonts packaged in asset bundles into the font lists.
- **Workflow:**
    - Scans the designated folder for asset bundles (.ltbundle for legacy fonts, .tmpltbundle for TMP fonts).
    - Opens each bundle and retrieves all font assets.
    - Adds the fonts to the respective font lists if they are not already present.
    - Logs errors if a bundle cannot be loaded or if the list is missing.

Usage: Allows fonts to be distributed and loaded separately from the main build, useful for reducing build size or updating fonts dynamically.

5. Save Font Lists to Files

- **Purpose:** Creates text files containing the names of all fonts in the lists.
- **Functionality:**
    - Generates file paths for both legacy and TMP font lists.
    - Writes font names to text files if the files do not already exist.

Usage: Provides a persistent record of available fonts that can be used to validate or reload fonts later.

6. Load Font Lists from Files

- **Purpose:** Reads previously saved font list files and updates the in-memory lists.
- **Functionality:**
    - Checks if the folder exists.
    - Reads the text files containing font names.
    - Filters the current font lists to include only the fonts listed in the files.

Usage: Ensures that only valid, expected fonts are loaded into the system, preventing missing or invalid fonts from being used at runtime.

How the System Works in Practice

1. **Startup Initialization:** The script automatically runs when the game starts.
2. **Language Determination:** Checks saved preferences or system culture and sets the default language.
3. **Font Loading:** Loads fonts from resources or asset bundles for both legacy and TMP text.
4. **Persistence:** Saves font lists to files and reloads them to ensure consistency.
5. **Error Handling:** Logs missing settings, missing font bundles, or invalid data to help developers troubleshoot issues.

Summary

This script is a **central manager for localization and font systems** in Unity:

- Handles both **legacy and TMP fonts**.
- Loads fonts from **resources, files, and asset bundles**.
- Maintains a **persistent record of font lists** for reliability.
- Automatically sets the **default language** based on preferences or system culture.
- Designed to work **both in editor and at runtime**, ensuring a smooth workflow for multilingual projects.

Access Permission Checker

Purpose Overview

This script is designed to **verify runtime access permissions** for critical folders in a Unity project. Specifically, it checks:

- The Assets folder, which contains all project files.
- The StreamingAssets folder, which is often used to store runtime data.

If access is restricted, it **displays a warning in the game** so the user or developer can take corrective action. This helps prevent runtime errors due to missing read/write permissions.

Key Variables

Variable	Purpose
None are directly public, but internal data includes:	The script primarily operates on **paths to project folders** (Assets and StreamingAssets) and uses a temporary file to test access.
settings	Holds the language settings data, which includes a reference to a UI warning prefab (errorLanguageTool).
warning	Stores the instantiated warning UI that is shown in-game if access is denied.

Core Methods and Their Functionality

1. Initialize Check Settings

- **Purpose:** Called automatically at game start to verify folder access.
- **Functionality:**
  - Triggers the folder access verification process.
- **Usage:** Ensures that the check happens **as soon as the game runs**, without needing manual activation.

2. Check Access Permissions

- **Purpose:** Determines whether the critical folders can be accessed.
- **Functionality:**
  - Checks both the Assets and StreamingAssets folders.
  - If either folder is inaccessible, it logs an error and calls the method to display a warning.
  - If access is available, it logs a confirmation message.
- **Usage:** Central method that validates folder availability before the game relies on them for reading or writing data.

3. Check Folder Access

- **Purpose:** Performs the actual test to see if a folder is readable and writable.
- **Functionality:**
  - Verifies that the folder exists.
  - Attempts to create a temporary file in the folder.
  - Attempts to delete the file afterward.

    - Returns true if both operations succeed, false if an exception occurs.
- **Exceptions Handled:**
    - Input/output errors (e.g., disk problems).
    - Unauthorized access errors (permission denied).
- **Usage:** Provides a reliable way to check whether the game can safely use a folder for saving or reading files.

4. Show Warning

- **Purpose:** Displays an in-game warning if folder access is restricted.
- **Functionality:**
    - Loads the language settings to find the UI warning prefab.
    - Instantiates the warning in the scene.
    - Makes the warning persistent across scene changes.
    - Logs whether the warning was successfully displayed or if there was an error.
- **Usage:** Ensures the user sees a clear notification when critical folder access is denied, allowing troubleshooting before the game encounters further errors.

How the System Works in Practice

1. **Automatic Runtime Check:** As soon as the game starts, it runs the initialization method to check folder access.
2. **Permission Verification:** For each critical folder, the script tests if it can write and delete a temporary file.
3. **Error Handling:** If any folder cannot be accessed, an error is logged, and a visual warning is displayed.
4. **Warning Display:** Uses a UI prefab defined in the language settings to ensure a localized warning message can be shown. The warning persists even if the player changes scenes.
5. **Logging:** All successes or failures are logged to help developers understand and fix permission issues.

Summary

This script acts as a **runtime safety checker** for critical folders in a Unity project:

- Validates read/write access to Assets and StreamingAssets.
- Uses temporary files to reliably test permissions.
- Provides an in-game visual warning if access is restricted.
- Ensures warnings persist across scenes.
- Logs errors and confirmations for developer awareness.

Its main goal is to **prevent runtime crashes** or unexpected behavior caused by inaccessible folders, which is crucial for projects relying on dynamic data, assets, or localization systems.

Language Text / Language Text TMP

General Overview

The component is attached to a text object (either a Text or TMP_Text) and links that object to the LanguageTools localization data.
When the active language changes, or when the object becomes active, it automatically updates:

- The text content (if translation is enabled).
- The font.
- The font size.
- The text alignment.

This makes it possible to switch languages dynamically in the application without manually editing each text element.

Serialized Fields (Visible in Inspector)

Variable	Purpose
textComponent	Reference to the UI text element that will display localized content.
translateText	Boolean flag that decides whether the actual text string should be replaced with a translation. If disabled, only font and style are updated.
iD	A numeric identifier that points to the corresponding text entry and metadata inside the language data file. Each entry in the LanguageTools database has its own ID.

These fields are serialized so that they can be edited directly in the Unity Inspector.

Private Fields (Internal Use)

Variable	Purpose
languageData	Stores the current set of loaded language settings. It contains all localized text and metadata (like alignment, font index, etc.) that the system will apply to the component.

This variable is loaded dynamically whenever a language update occurs.

Properties (Public Accessors)

The component provides **getters and setters** for all serialized fields, allowing other scripts to read or modify them at runtime:

- TextComponent → Accesses the assigned UI element.
- TranslateText → Enables or disables text translation dynamically.
- ID → Changes the ID used for fetching translations and style data.

Unity Lifecycle Methods

Method	Description
OnEnable	Runs when the object becomes active. It subscribes to the global language update event (LanguageManagerDelegate.OnLanguageUpdate) and immediately calls LanguageUpdate() to apply the current language settings.
OnDisable	Runs when the object is deactivated or destroyed. It unsubscribes from the language update event to prevent memory leaks or unwanted updates.

This ensures the UI text always matches the selected language when the scene loads or when language changes during gameplay.

Localization Logic

LanguageUpdate()

This is the core function where localization happens.

1. **Validation**
   Checks whether the assigned text component is valid.
   If not, an error message appears in the Unity console, and the method stops.
2. **Load Language Data**
   Uses a LanguageTools utility (LoadLanguageSettings) to retrieve the current language configuration file (for example, English, Spanish, Portuguese, etc.).
   If it fails to find the file, it logs an error and exits.
3. **Translate Text (optional)**
   If translateText is enabled, it calls GetIDText() using the assigned ID to find the corresponding translated text in the language database.
   If a translation exists, it updates the text component's content.

4. **Apply Metadata**
   Retrieves style and formatting information via GetIDMeta().
   This metadata contains:
   - **Alignment code** → Converted to a proper alignment setting using a helper (ConvertToTextAnchor).
   - **Font size** → Sets the text size directly on the component.
   - **Font list index** → Points to a font in the LanguageTools font database, fetched with GetFontByIndex().
5. **Result**
   The text now matches the target language's text, font, and style.

🧰 Editor-Specific Section (Unity Editor Only)

In the Unity Editor, both scripts include a **custom inspector** to make localization setup easier.

Features of the Custom Editor:

- Adds an **"Import Settings" button** above the default inspector.
- When clicked:
  1. It checks whether the current ID already exists in the language database.
  2. If so, it prompts the user for confirmation before overwriting it.
  3. Then it extracts current properties (text, alignment, font, font size, font reference) from the selected component.
  4. Finally, it opens the **LanguageTools editor window** with those values pre-filled, so developers can save or modify that entry easily.

This speeds up workflow by letting developers capture an existing UI element's configuration and register it directly into the localization system without typing everything manually.

🧭 Usage Summary

1. Attach **LanguageText** to a Unity UI Text element, or **LanguageTextTMP** to a TextMeshPro element.
2. Assign the respective text component reference.
3. Set the ID that corresponds to an entry in the LanguageTools language database.
4. Enable or disable translateText depending on whether you want the system to replace the text automatically.
5. When the game runs, or when the active language changes, the system automatically updates:
   - The displayed text content (if enabled).
   - Font, alignment, and size according to metadata.
6. (Optional) In the editor, click "Import Settings" to capture the component's current layout and register it into the localization tool.

⚖️ Difference Between the Two Versions

Version	UI Type	Font Utility Used	Text Component Type
LanguageText	Legacy Unity UI	LanguageTools.Legacy.FontAndAlignmentUtility	UnityEngine.UI.Text
LanguageTextTMP	TextMesh Pro	LanguageTools.TMP.FontAndAlignmentUtilityTMP	TMPro.TMP_Text

They are functionally identical — both pull data from the same language database and respond to the same language update events.
The only distinction is compatibility: one targets **classic UI**, and the other targets **TextMesh Pro** elements.

Language Text Input Field / Language Text Input Field TMP

Purpose and Overview

The **Language Text Input Field** components are tools that automatically localize **input fields** and their **placeholder texts** according to data provided by the *LanguageTools* system.
They ensure that when the player changes the language, the text placeholder, font type, font size, and alignment update dynamically to match the language's visual and textual requirements.
Two versions exist:

- **Legacy version** – Works with the classic Unity UI InputField component.
- **TMP version** – Works with the TextMeshPro TMP_InputField component.

Both execute the same logic with equivalent internal behavior.

Serialized Fields (User-Configurable Variables)

1. **inputField**
   - Reference to the text input element in the scene.
   - This is the UI object that will be localized (either Legacy or TMP version).
2. **translateText**
   - A switch (true or false) that determines if the placeholder text should be translated automatically when the language updates.
   - When off, it will only apply font and alignment metadata without modifying the placeholder's written text.
3. **iD**
   - The numeric identifier used by *LanguageTools* to find the correct translation and formatting metadata in the loaded language data file.
   - Each localized element in the interface has its own unique ID.

Private/Internal Fields

1. **languageData**
   - Stores the currently loaded language configuration (texts, fonts, alignments, and metadata).
   - Loaded from the *LanguageTools* data file each time the component updates.
2. **text**
   - A reference to the field that displays what the user types (the input text area).
3. **placeholder**
   - A reference to the placeholder element (the text shown when the input is empty).
   - Used to update localized placeholder content and styling.

Properties (Getters and Setters)

Each main variable (inputField, translateText, and iD) has a public property that allows external scripts or editor tools to read or modify their values safely.
This is mostly useful in automated editors or custom language management systems.

Unity Event Methods (Lifecycle Behavior)

OnEnable

- Runs when the component becomes active or visible in the scene.
- It connects the component to the global **language update event** provided by *LanguageManagerDelegate*.
- Immediately calls the localization process once at startup to apply the correct language.

OnDisable

- Runs when the component is disabled or removed.
- It disconnects from the language update event to prevent unnecessary updates or errors.

有 Main Localization Logic

LanguageUpdate

This is the core method that updates the text and styling of the input field.
It performs the following steps:

1. **Validation**
   - Checks if the input field reference is assigned.
   - Checks if the text and placeholder components exist and are of the expected type.
2. **Load Language Data**
   - Requests the latest language configuration from *LanguageTools* through the data manager.
   - If no data is found, the update stops with an error message.
3. **Translate Placeholder**
   - If the translation option is active, the system retrieves the localized text linked to the provided ID.
   - If a valid translation is found, the placeholder text is updated.
4. **Apply Metadata Styling**
   - Retrieves visual information (metadata) associated with the same ID.
   - Metadata can include:
     - **Text alignment** (left, center, right, justified).
     - **Font size** (numeric scaling for text).
     - **Font type** (which font asset to use).
   - The same styling is applied to both the placeholder and the active text display.

This ensures the entire input field maintains consistent language-dependent formatting.

Editor Integration (Custom Inspector)

Both versions have a custom editor tool that enhances workflow inside the Unity Editor.

Purpose

- Provides an **"Import Settings"** button that allows the current input field settings to be sent directly into the *LanguageTools* editor window.
- This captures current text, font, alignment, and size to create or update corresponding localization entries.

Behavior

When the button is clicked:

1. It verifies if the language ID already exists in the system.
2. If it does, it asks for confirmation before replacing existing data.
3. It extracts text, alignment, size, and font details from the current placeholder.
4. Sends this data to the *LanguageTools* editor interface for storage or modification.

This makes it easy to register or adjust localized UI components directly from the Unity Inspector.

Typical Usage Example (Conceptual)

1. Attach the **Language Text Input Field** component to any UI element that contains an input field (either Legacy or TMP).
2. Assign the correct **Input Field** reference in the inspector.
3. Set the **LanguageTools ID** for that field (each UI element must have its own ID).
4. Optionally enable **Translate Text** if the placeholder text should change automatically.
5. When the game starts or when the language changes, the system updates:
   - Placeholder text → localized translation.
   - Font type → language-appropriate font (e.g., Arabic, Japanese, etc.).
   - Font size → scaled based on metadata.
   - Alignment → adjusted for language direction or style.

⚖️ Comparison: Legacy vs TMP

Aspect	Legacy Version	TMP Version
UI Type	InputField	TMP_InputField
Text Component	Text	TMP_Text
Font Data Source	Standard Unity fonts	TMP FontAssets
Metadata Application	Alignment, Font Size, Font	Alignment, Font Size, Font
Functionality	Identical logic and structure	Identical logic and structure

The TMP version simply uses the TextMeshPro text system, which supports richer text rendering and more advanced font handling, but both scripts synchronize the same localization data and event-driven updates.

🧠 Summary

In essence, both scripts:

- Listen for **language changes**.
- Retrieve **localized text and font settings** from *LanguageTools*.
- Update both **placeholder** and **input text** with that information.
- Optionally allow importing and managing localization data directly through the Unity Editor.

They ensure that text inputs always reflect the active language and maintain consistent styling across all supported languages.

Language Dropdown / Language Dropdown TMP

🧩 Purpose and Overview

The scripts integrate the **LanguageTools localization system** with dropdown UI elements. Their function is to:

- Automatically **translate the dropdown options** when the game language changes.
- **Update text styling** (font, alignment, size) based on language metadata.
- **Keep user selection intact** after translation updates.
- Provide an **import utility in the editor** to sync dropdown data with the language system.

One version is for the **legacy Unity UI system**, and the other is for **TextMesh Pro**. Otherwise, both behave the same way.

📦 Serialized Fields (Configurable in the Inspector)

These are editable properties visible in Unity's Inspector.

Variable	Description
dropdown	Reference to the dropdown UI component that this script will localize. It's the target dropdown whose text will change when the language updates.
translateText	Boolean toggle that decides whether the dropdown's options will be translated automatically or not. When off, only font and alignment will update.
options	A list of custom data structures (LanguageOptions) representing the dropdown's content. Each entry contains: the text label, an optional sprite, and a localization ID that links to the correct translated string in the language file.

Example:

Option A → ID - 2
Option B → ID - 3
Option C → ID - 4

Each ID connects to a specific text entry in the LanguageTools database.

⚙ Private Fields (Internal References)

Variable	Purpose
languageData	Stores the currently loaded language settings, including all translation data and metadata (such as font or alignment info).
captionText	Reference to the dropdown's main display text (the label showing the selected option).
itemText	Reference to the text element used for dropdown list options.

These are automatically found and stored during runtime.

🔧 Properties (Getter/Setter Access)

These provide access to the serialized fields programmatically.
They're useful if another script wants to configure the dropdown dynamically.

- **Dropdown** → Get or assign the dropdown component.
- **TranslateText** → Enable or disable automatic translation.
- **Options** → Read or modify the list of dropdown options.

🔁 Event Subscription (OnEnable / OnDisable)

The dropdown needs to react whenever the game's language changes.
This is done through a **delegate-based event system** provided by LanguageManagerDelegate.

Method	Role
OnEnable()	Subscribes the LanguageUpdate() method to the language update event. Immediately triggers an update to ensure the dropdown displays the correct language when activated.
OnDisable()	Unsubscribes from the event to avoid memory leaks or invalid references when the object is disabled or destroyed.

🌍 LanguageUpdate() — Core Localization Logic

This is the **main method** that refreshes the dropdown based on the current language data.
Here's the step-by-step breakdown:

1. **Verify the dropdown is assigned.**
   If not, logs an error and exits early.
2. **Retrieve text references.**
   Gets the dropdown's caption and item text components.
   If either is missing, logs an error and exits.
3. **Load the current language data.**
   Pulls translation text and styling metadata from the LanguageTools system.
4. **Check if the options list is valid.**
   If it's empty, logs a warning and stops (no options to translate).
5. **Update text content.**
   If translateText is enabled, calls UpdateLocalizedOptions() to replace option texts with localized ones.
6. **Apply style metadata.**
   Reads the font, alignment, and size data associated with the first option's ID.
   Then applies them to both caption and item text components.

This method ensures both **visual style** and **text content** match the selected language.

🏛 UpdateLocalizedOptions() — Translation Handler

This private method performs the actual text replacement in the dropdown list.
Steps:

1. Loops through each entry in options.
   For each, retrieves its localized text using the language ID.
2. Replaces the text value with the translated version.
3. Stores the dropdown's **current selected index** (so the user's choice is not lost).
4. Clears all dropdown options.
5. Rebuilds the dropdown with updated, translated entries.
6. Restores the previously selected index and updates the visible caption.

This ensures that when a language changes:

- The dropdown options are replaced with the correct translations.
- The user's selection remains consistent.
- The visible text updates instantly.

Editor Integration (Custom Inspector)

Both scripts include an **editor-only section** that adds a custom button to the Unity Inspector.

Feature	Function
Import Settings button	Imports the dropdown's current data into the LanguageTools system. It registers the options' IDs, text, and style settings (font, alignment, size).
Conflict handling	If an ID already exists in the system, it prompts the user before overwriting.
Multiple option support	The first option imports style metadata; subsequent ones import only text and ID information.

This feature makes it easier for developers to synchronize dropdown data with the localization system without manually editing language files.

Usage Summary

Step	What You Do
1	Attach the script to a GameObject that contains a dropdown (legacy or TMP version, depending on your UI).
2	Assign the dropdown reference in the Inspector.
3	Fill in the options list with text, optional icons, and localization IDs.
4	When the language changes in-game, the dropdown automatically updates.
5	In the Editor, use the "Import Settings" button to sync dropdown content into the LanguageTools database.

Difference Between the Two Versions

Aspect	Legacy Dropdown	TMP Dropdown
Text component	UnityEngine.UI.Text	TMPro.TMP_Text
Dropdown type	Dropdown	TMP_Dropdown
Font handling	Standard Unity font system	TextMesh Pro font assets
Visual quality	Basic	High-quality text rendering
Other behavior	Identical	Identical

Both share identical logic; the only difference lies in **which text rendering system they operate on**.

In summary, both scripts are **modular localization bridges** between the LanguageTools system and dropdown UI elements.
They automatically handle:

- Translating option texts,

- Updating font and alignment per language,
- Preserving user selections, and
- Simplifying editor setup through a custom import tool.

Language Manager / Language Manager TMP

Purpose

These components provide a user interface element (a dropdown menu) that allows players or users to select their preferred language during runtime.
They load all available languages from a configuration file, populate the dropdown options, automatically apply the previously saved selection, and notify other parts of the system when the language changes.

Key Variables

languageDropdown

- This represents the user interface element used to choose a language.
- In one system, it is a **Dropdown** from the standard UI.
- In the other, it is a **TMP_Dropdown** from TextMeshPro.
- It displays the list of available languages for the player to select.

availableLanguages

- A list containing all languages supported by the project.
- Each entry includes:
  - The **name** of the language (for internal reference).
  - The **nativeName**, which is displayed to the player (e.g., “Español”, “Português”).
  - The **culture** code (e.g., “en-US”, “pt-BR”), which uniquely identifies the language variant.

languageData

- A data container that holds all information about the available languages and their configuration.
- It’s loaded from an external resource (for example, a file or ScriptableObject).
- Contains both translation data and metadata for the localization system.

Core Behavior and Workflow

1. Initialization (Start event)

When the scene loads and the component becomes active:

1. It checks whether the dropdown component has been correctly assigned.
2. It loads the **language configuration data** from the system.
3. It retrieves the **list of available languages** defined in that configuration.
4. It calls the **PopulateDropdown** method to display these options in the UI.
5. It calls **SetupDropdownSelection** to prepare the event listener that reacts to user selection changes.

If any of these steps fail (for example, missing dropdown or missing configuration), the system logs an error and stops further execution to prevent unexpected behavior.

2. Populating the Dropdown

The **PopulateDropdown** process does the following:

- Clears any existing options to ensure a fresh start.
- Creates a list of all native language names to be shown in the dropdown.
- Sorts the list alphabetically for cleaner presentation.
- Compares each available language’s **culture code** with the previously **saved culture** (retrieved from storage) to find which one should be preselected.

- Fills the dropdown with the names and sets the initial selection accordingly, but without triggering any events yet.

This ensures the UI reflects the user's previously chosen language when the application starts.

3. Dropdown Selection Setup

The **SetupDropdownSelection** step connects the dropdown's "value changed" event to the component's internal **OnLanguageChanged** function.
This means whenever the user selects a new language, the system reacts automatically — it removes any old listeners to avoid duplication, then adds a fresh one.

4. Responding to Language Changes (OnLanguageChanged)

When a new language is chosen:

1. The selected index is verified to ensure it's valid (within range).
2. The system retrieves the **culture code** for that selected language.
3. This culture code is then **saved** (so it persists across sessions, typically in PlayerPrefs or an equivalent storage system).
4. The entire language dataset is **reloaded**, ensuring that all texts and assets are updated for the new language.
5. A **global update notification** is triggered (via LanguageManagerDelegate.NotifyLanguageUpdate()), which tells all other components in the game that they need to refresh their localized text or visuals.

Persistent Language Storage

Both systems use two helper operations:

- **GetSaveCultureCode()** → Retrieves the last selected language's culture code.
- **SetSaveCultureCode()** → Saves the newly chosen culture code for future use.

This ensures the game or app remembers the user's preference between sessions.

Integration with Other Systems

These managers work hand-in-hand with:

- **LanguageFileManager**, which handles loading and managing all localization data.
- **LanguageManagerDelegate**, which broadcasts updates so that text components (like LanguageTextMesh or LanguageTextMeshTMP) can automatically refresh.

This makes the language switching system fully dynamic and global — all localized elements update in real time whenever the user selects a different language.

Usage Summary

- Attach the **LanguageManager** (for legacy UI) or **LanguageManagerTMP** (for TextMeshPro UI) to a GameObject that includes the corresponding dropdown component.
- Assign the dropdown field in the inspector.
- Ensure that the localization configuration (LanguageSettingsData) is properly set up and available.
- At runtime, the dropdown will:
  - Display all supported languages.
  - Restore the last chosen language.
  - Allow switching languages instantly.
  - Notify all text elements to update automatically.

Functional Parity

Both versions:

- Load and display available languages.

- Remember the last chosen language.
- Trigger global updates upon user selection.
- Operate identically, differing only in the UI component type they interface with:
    - One targets **Unity's built-in UI Dropdown**.
    - The other targets **TextMeshPro's TMP_Dropdown**.

Thus, functionally they are interchangeable depending on which UI framework your project uses.

Automatic Language Font Validator / Automatic Language Font Validator TMP

Overview

These scripts automatically validate and update the font of a UI text element to ensure that all characters in the displayed text are supported by the current font. If a font cannot display some characters, the script will attempt to replace it with a fallback font. Additionally, if no font fully supports the text, and language fallback is enabled, it can replace the displayed text with a language name that is guaranteed to be supported.

One script is designed for legacy UI text elements, while the other is designed for TextMeshPro elements. Functionally, they behave the same.

Key Variables / Data

1. **Target Text / Font Settings**
    - **textComponent:** The text element being monitored and validated.
    - **isLanguageManager:** Enables fallback logic for replacing native language names with supported alternatives.
2. **Private / Runtime Data**
    - **supportedLanguages:** List of available languages for fallback replacement if some characters are unsupported.
    - **localizationSettings:** Contains loaded language configuration, including font settings and fallback lists.
    - **lastValidatedText:** Stores the most recently validated text to avoid redundant processing.

Core Methods / Logic

1. **Initialization (Start)**
    - Ensures that the text element is assigned.
    - Loads language and font configuration data.
    - If fallback logic is enabled, stores the list of supported languages.
    - Performs an initial validation on the text content to make sure all characters are supported.
2. **Continuous Monitoring (Update)**
    - Checks each frame whether the text content has changed.
    - If the text changes, it revalidates whether the current font can display all characters.
3. **Font Validation (ValidateFontSupport)**
    - Iterates through each character in the text to see if it is supported by the current font.
    - If the current font supports all characters, no action is taken.
    - If some characters are unsupported, it iterates through fallback fonts, checking each one.
    - If a fallback font supports all characters, it replaces the current font with that fallback.
    - If no font fully supports the text, and fallback language logic is enabled, it replaces the text with a supported language name.

Usage

1. Attach the component to a UI text element (legacy Text or TextMeshPro).
2. Assign the text element to be monitored in the inspector.
3. Optionally enable the language fallback feature to automatically replace unsupported native names.
4. The script automatically:
    - Checks for unsupported characters in the current font.
    - Replaces the font with a valid fallback if necessary.
    - Replaces the text with a fallback language name if no fonts support all characters and fallback logic is enabled.

Key Features

- Works with both legacy UI Text and TextMeshPro text elements.
- Automatically detects unsupported characters in the displayed text.
- Dynamically applies fallback fonts when necessary.
- Supports language fallback replacement for native names.
- Continuously monitors text changes to ensure font validity in real time.
- Prevents broken or missing characters in multilingual UI.

Force Import Of Fonts TMP

✅ **High-Level Purpose of the Script**

Unity sometimes *fails to include TextMeshPro font assets* in the final build unless they have been explicitly used in a scene or referenced by a TMP component.
This script forces Unity to **pre-load and embed every TMP font** defined in the Language System by:

1. Loading the list of fonts from the LanguageSettingsData.
2. Creating temporary TextMeshPro objects in the scene (in Edit Mode).
3. Assigning each font to one of these objects so Unity is forced to import it.

These objects are created **only in the Unity Editor**, never during gameplay.
It is meant to be placed in an initialization scene (such as a splash screen or main menu) so all multilingual fonts are guaranteed to be included in the build.

✅ **Explanation of Variables and Data Structures**

1. "FontText" (a small data container)

A structure called **FontText** holds two things:

- A **sample text string** (for example: "こんにちは" or "Olá mundo").
- A **specific font asset** that this text belongs to.

This allows you to override the default preview text for certain fonts — useful for fonts that need special characters (Japanese, Arabic, Chinese, etc.).
Unity shows this text during the font-forcing import process.

2. "textMesh"

This is a reference to a **TextMeshPro GameObject** already present in the scene.
The script uses this object as a **template** ("clone source").
Every temporary text object created by the script is a copy of this one.
It must contain a TextMeshPro component, or the script will log an error.

3. "fontTexts"

This is a list of **FontText entries**.
Each entry maps:

- A font
  → to

- A sample text string

If a font appears here, the script will display the custom text instead of using the font's name.
If the list is empty, the script uses the font name as default preview text.

4. "gameObjects"
This is an internal array storing every TMP clone created by the script.
It exists to prevent duplicate clones when the script reloads or recompiles.
Every time the script recreates the clones, the old clones stored here are destroyed.

5. Public Properties
There are three public getters/setters:

- **TextMesh** → exposes the base TMP object
- **FontTexts** → exposes the list of custom sample texts
- **GameObjects** → exposes the created clone objects

These allow easy access in the inspector or through other tools.

✅ **How the Script Behaves (Lifecycle and Logic)**
"Start" method behavior
The script's main logic lives in the method named "Start".
It executes automatically when:

- The script is added to an object,
- The editor reloads,
- You press the "Reload" button in the custom inspector.

It does **NOT** execute in the final build.

✓ Step 1 — Validates the template object
If the template TMP object (textMesh) is missing:

- The method logs an error.
- The process stops.

✓ Step 2 — Deletes previously created clones
If there are old clones stored in "gameObjects", each one is destroyed immediately.
This prevents the script from creating hundreds of redundant TMP objects every time Unity recompiles.

✓ Step 3 — Loads the language system data
The method calls LanguageFileManager to load **LanguageSettingsData**.
This data is expected to contain:

- A list of all TMP fonts used in the project.

If the data cannot be loaded, an error appears and the process stops.

✓ Step 4 — Validates the font list
If the font list is empty or missing:

- A warning is shown.
- The script stops (no fonts to process).

✓ Step 5 — Creates one TMP clone per font
For each font in the language system:

1. The script clones the template TMP object.
2. It finds the TextMeshPro component in the clone.
3. It checks if the font has a custom preview text in "fontTexts".
4. It sets the preview text.

5. It assigns the current font asset to the TMP component.
6. It renames the object for easier identification.
7. It ensures the clone is active.
8. It stores the clone in the "gameObjects" array.

Unity will now recognize this font as being used and include it in the final build.

✓ Step 6 — Logs the result
The script prints a summary message such as:
“Successfully created X TMP font import instances.”

✓ Step 7 — Removes itself in builds
If running outside of the Unity Editor (like in a build):

- The script automatically destroys itself to avoid wasting memory.

✅ Custom Inspector (Editor Extension)
A custom inspector adds functionality when you select this script in Unity.
✓ It shows a yellow explanation box
This box explains why TMP needs forced importing and what the script does.

✓ It adds a “Reload (Font List Data)” button
Clicking this button manually re-triggers the “Start” method so you can rebuild the TMP clones on demand.

✓ It draws the default inspector afterwards
All your serialized fields remain visible and editable.

✅ How to Use This Script

1. **Create or locate a TextMeshPro object** in your scene that will act as the template.
2. **Attach this script** to any GameObject in your initialization scene.
3. **Assign the template TMP object** to the “TextMesh” field.
4. (Optional) Add custom font preview text entries in “FontTexts”.
5. Press **Reload** in the inspector to test font cloning.
6. Ensure the LanguageSettingsData contains all your fonts.
7. When you build the game, all TMP fonts will now be properly included.

This ensures multilingual text will always render correctly on every device and platform.

Adjust Size To Dropdown / Adjust Size To Dropdown TMP

Overview
These two scripts dynamically adjust the width and positioning of a UI element (usually a dropdown container) based on the text content inside it. They monitor whether text lines break into multiple lines and ensure that the element does not overflow beyond the boundaries of the parent canvas. They also adjust scroll behavior to make sure the selected item is visible in a scrollable list.
One script works with legacy UI text components, while the other is built for TextMeshPro elements. Functionally, both perform the same operations.

Key Variables / Data

1. **Settings / Manual Adjustments**
    - **manualSizeAdjustment:** Extra width added on top of the calculated size.
    - **sizeMultiplier:** Additional width applied when a text element wraps to multiple lines.

    - **margin:** Minimum horizontal margin to prevent the UI element from touching the canvas edges.
    - **fitDirection:** Determines whether the element expands to the left or right.
2. **Automatic / Runtime Data**
    - **parentRect:** The UI element being resized.
    - **canvasRectTransform:** The parent canvas used to calculate boundaries.
    - **scrollRect:** The scrollable container for the dropdown list.
    - **canvasWidth:** The width of the canvas, used for overflow detection.
    - **objectWidth:** The width of the UI element itself.
    - **textList:** A collection of text items inside the element, each paired with a flag indicating if the text has wrapped.
    - **textIsBroken:** Boolean indicating whether any text has wrapped into multiple lines.
    - **sizeAdjustment:** The final computed width adjustment based on wrapping and manual offsets.

Nested Classes

- Store references to individual text elements (either legacy Text or TextMeshPro) and whether each text has broken into multiple lines. This allows the system to dynamically detect wrapping and adjust the UI element accordingly.

Core Methods / Logic

1. **Initialization (Start)**
    - Finds the parent canvas to determine layout boundaries.
    - Collects all child text elements and allows their overflow, ensuring line break detection works.
    - Sets scroll sensitivity of the list dynamically, based on the number of items.
    - Automatically scrolls to the currently selected item in the dropdown.
2. **Scroll to Selected (ScrollToSelected)**
    - Identifies which item is selected in the list.
    - Computes a normalized scroll position to bring the selected item into view.
    - Applies clamping to avoid exceeding scroll boundaries.
3. **Dynamic Size Adjustment (Update)**
    - Checks if any text item wraps into multiple lines.
    - Updates the total width adjustment by combining line wrapping and manual offsets.
    - Applies width changes to the UI element in the chosen fit direction.
    - Converts the element's position relative to the canvas to detect overflow.
    - Switches expansion direction if the element would exceed the canvas edges.
4. **Line Break Detection (CheckLineBreaks)**
    - Iterates through all text items and updates the flag indicating if a text wraps.
    - Optionally logs a warning when a text element breaks into multiple lines.

Usage

1. Attach the component to a dropdown container or scrollable UI element.
2. Assign references to either Text components (for legacy UI) or TextMeshPro components.
3. Set optional adjustments such as manual width addition, line wrap multiplier, and canvas margin.
4. The script automatically:
    - Adjusts the width of the element based on text content.
    - Monitors and handles line breaks.
    - Ensures the UI element stays within canvas boundaries.

- Adjusts scroll sensitivity and auto-scrolls to the selected item.

Key Features

- Works for both legacy text and TextMeshPro.
- Dynamically adapts UI width based on content.
- Automatically detects line breaks and updates layout.
- Prevents elements from overflowing the canvas.
- Adjusts scroll behavior for better usability in long dropdown lists.
- Switches fit direction automatically when necessary.

Language Image

Overview

This component dynamically updates a Unity UI Image element with a language-specific sprite. It determines the appropriate image based on the active language or culture, loads it from disk asynchronously, and applies it to the Image component. This allows different images to appear for different languages without manual intervention.

Inspector Fields / Settings

- **fileName:** The name of the image file to load, including its extension (like .png or .jpg). This identifies which image to display for the current language.
- **useImage:** A toggle that specifies whether the Image component should be updated with the loaded sprite. If disabled, the sprite can still be loaded but not displayed automatically.
- **image:** The target Unity UI Image component that will display the loaded sprite. Must be assigned if useImage is true.
- **imageTexture:** Stores the loaded texture from disk. This is automatically updated when a new image is loaded.
- **spriteTexture:** The sprite generated from the loaded texture, which is applied to the Image component.

Private Fields

- **languageData:** Holds the currently active language settings, including which culture is selected. This is used to resolve the correct path for the localized image.
- **previousFilePath:** Tracks the path of the last loaded image to prevent redundant reloading if the image hasn't changed.
- **filePath:** The full path to the localized image for the current language.

Properties

- **FileName:** Allows external scripts or tools to get or set the image file name.
- **UseImage:** Determines whether the Image component should be updated automatically.
- **Image:** Provides access to the Image component for assignment or modification.
- **ImageTexture:** Provides access to the loaded texture for other systems or scripts.
- **SpriteTexture:** Provides access to the sprite created from the texture.

Unity Events

- **OnEnable:**
  - Subscribes to a global language update event so that the image updates automatically whenever the language changes.
  - Immediately triggers an image update in case the language was already set before the component became active.
- **OnDisable:**

- Unsubscribes from the language update event to prevent unnecessary updates when the component is inactive.

Core Methods

- **LanguageUpdate:**
  - Validates that the Image component is assigned if it is required.
  - Loads the current language settings to determine the selected culture.
  - Constructs the full path to the localized image file based on the culture and file name.
  - Skips loading if the same image was already loaded.
  - Checks whether the file exists on disk and logs an error if it does not.
  - Cleans up previously loaded textures and sprites to free memory.
  - Starts asynchronous loading of the new image.
- **LoadFileCoroutine:**
  - Asynchronously loads the texture from the specified file path.
  - Handles errors in loading and logs messages if the file cannot be read.
  - Converts the loaded texture into a sprite.
  - Assigns the sprite to the Image component if useImage is enabled.

Editor Integration

- The script includes a custom inspector for the Unity Editor:
  - Displays a warning to avoid using non-ASCII characters in file names.
  - Draws the default inspector to allow easy assignment of the file name, Image component, and other fields.
  - Supports multiple object selection, though preview functionality is limited in that case.

How It Works Together

1. A developer assigns an Image component and specifies a file name.
2. When the object is enabled or the language changes:
   - The script determines the correct file path for the current language.
   - Prevents unnecessary reloads if the image has already been loaded.
   - Loads the image asynchronously from disk.
   - Converts the texture into a sprite.
   - Updates the Image component to display the new sprite.
3. The process repeats automatically whenever the language changes, allowing dynamic language-based UI updates.

Typical Usage

- Attach the component to a GameObject with a Unity UI Image.
- Set fileName to match the desired localized image.
- Ensure that the image files are organized by language folders.
- The component automatically updates the UI whenever the language changes.

Key Points

- Supports language-specific image updates.
- Loads images asynchronously to avoid blocking the main thread.
- Cleans up previously loaded textures to manage memory efficiently.
- Integrated with a global language update system for real-time updates.
- Editor-friendly, with warnings for proper file naming and easy field assignment.

Language Raw Image

Overview
This component is designed to load and display localized images in a Unity scene using a RawImage component. It automatically selects the image file based on the currently active language or culture and loads it asynchronously. This allows for different images per language, which is useful for user interfaces, in-game signs, or any visual content that requires localization.

Inspector Fields / Settings
- **fileName:** The name of the image file to load, including its extension (e.g., .png or .jpg). This is the main identifier for the image content.
- **useRawImage:** A toggle to determine whether the loaded image should be displayed in a RawImage component. If disabled, the texture can still be loaded but will not be automatically displayed.
- **rawImage:** The RawImage component in the scene where the loaded texture will be displayed. Must be assigned if useRawImage is true.
- **imageTexture:** Stores the currently loaded image as a texture. This is automatically updated when a new image is loaded.

Private Fields
- **languageData:** Holds the current language configuration, including which culture is active. It's used to resolve the correct image path.
- **previousFilePath:** Stores the path of the last loaded image to prevent unnecessary reloads when the language hasn't changed.
- **filePath:** The full path to the image file that corresponds to the current culture and file name.

Properties
- **FileName:** Allows external scripts or editor tools to get or set the image file name.
- **UseRawImage:** Controls whether the RawImage component should display the loaded image.
- **RawImage:** Provides access to the RawImage component for assignment or modification.
- **ImageTexture:** Provides access to the loaded Texture2D for other scripts or components to use.

Unity Events
- **OnEnable:**
  - Subscribes to a global language update event so that the image updates automatically whenever the language changes.
  - Immediately triggers an image update in case the current language was set before this component became active.
- **OnDisable:**
  - Unsubscribes from the language update event to avoid unnecessary processing when the component is inactive.

Core Methods
- **LanguageUpdate:**
  - Ensures the RawImage component is assigned if it is required.
  - Loads the current language settings to determine the active culture.
  - Constructs the file path for the image based on the selected culture and file name.
  - Skips loading if the same image was already loaded.
  - Checks if the file exists on disk, logging an error if it does not.
  - Cleans up any previously loaded texture to free memory.
  - Initiates asynchronous loading of the new image using a coroutine.

- **LoadFileCoroutine:**
  - Performs an asynchronous request to load the image file from the local disk.
  - Waits until the file is fully loaded.
  - Logs an error if the request fails.
  - Extracts the image as a Texture2D.
  - Assigns the texture to the RawImage component if useRawImage is enabled.

Editor Integration

- Includes a custom inspector to improve usability in the Unity Editor:
  - Displays a warning to avoid using non-ASCII characters in the file name.
  - Allows the default inspector to manage fields easily.
  - Supports multiple object selection but does not display per-object preview in this case.

How It Works Together

1. Developers assign a RawImage component and specify the image file name.
2. The component listens for language changes through the global language update system.
3. When the language changes or the object is enabled:
   - The script calculates the correct file path for the active language.
   - Checks if the image has already been loaded.
   - Loads the image asynchronously from disk if it is new or different.
   - Updates the RawImage component to display the newly loaded texture.
4. The component can handle multiple languages dynamically, updating images without restarting the scene.

Typical Usage

- Attach the component to a GameObject that has a RawImage (or leave it unassigned if only the texture is needed).
- Set the fileName to match the desired image for each culture.
- Ensure images are placed in language-specific folders following the project's localization structure.
- The component automatically updates the image when the language changes, simplifying multilingual UI management.

Key Points

- Supports dynamic, language-specific image loading.
- Uses asynchronous loading to avoid blocking the main thread.
- Automatically handles texture cleanup to prevent memory leaks.
- Integrated with a global language update system for seamless runtime updates.
- Editor-friendly with warnings and default inspector support.

Language Text Mesh / Language Text Mesh TMP

General Purpose

These two scripts are responsible for automatically translating and updating **3D text objects** in Unity based on the currently selected language.
They connect with a localization framework (LanguageTools) to pull translated text and apply matching font styles, sizes, and alignments dynamically whenever the language changes in the game.
Both versions serve the same role:

- The **Legacy version** works with the classic TextMesh system.
- The **TMP version** works with the newer TextMeshPro system.

⚙️ Core Concept

Each text object in the scene has an **ID** (a numeric identifier).
This ID is linked to:

1. The text translation (the localized sentence or word).
2. The font style and size metadata for that entry.

When the game language is changed:

- The script detects the change through a **language update event**.
- It looks up the text and metadata for its assigned ID.
- It replaces the on-screen text, font, and font size according to that data.

📦 Main Components

1. Text Component Reference

- **Variable purpose:** Holds the reference to the actual text element on the object.
- For the legacy system, this is a standard 3D TextMesh.
- For TMP, it is a TMP_Text element.
- The script needs this reference to modify what is displayed.

2. Translation Toggle

- **Variable purpose:** Determines if the component should automatically translate the text when the language updates.
- If disabled, the text remains static but can still have font and size adjusted.

3. Unique Identifier (ID)

- **Variable purpose:** Connects the object to its corresponding translation and font metadata inside the language database.
- Each piece of localized text is stored with an ID.
- This allows the script to fetch the correct translation and style settings for the specific text element.

4. Language Data Cache

- **Variable purpose:** Temporarily stores the loaded language configuration so it doesn't have to be reloaded repeatedly.
- Contains:
  - The list of translations.
  - Font data for each language.
  - Font size and style metadata.

🧩 Script Behavior

When the Component Is Enabled

1. The script subscribes to a global event that triggers whenever the active language changes.
2. It immediately runs a localization check to update the text, font, and font size with the current settings.

When the Component Is Disabled

- It unsubscribes from the language event to prevent unnecessary updates or memory leaks.

Language Update Logic

Whenever a language update occurs (for example, if the user switches from English to Japanese), the following process happens:

1. **Validation:**
   Checks whether the text component and language data exist.
   If missing, logs an error or warning to the developer console.
2. **Load Settings:**
   Retrieves the full language configuration from the Language Manager (this includes all translation and metadata tables).
3. **Translation:**
   - If translation is enabled, looks up the text that corresponds to the component's unique ID.
   - If found, updates the visible text in the 3D object.
4. **Font and Size Application:**
   - Retrieves the associated metadata (font size and font index).
   - Updates the text element's font size and assigns the correct font asset.
   - For legacy TextMesh objects, it also synchronizes the mesh renderer's texture with the selected font material.

This ensures that the displayed text always uses the correct characters and styles for the current language (for example, switching between Latin, Cyrillic, or Asian scripts).

🧰 Editor Functionality

Both scripts also include **custom editor extensions** to help developers manage localization data inside Unity's Editor.

Import Settings Button

- Appears in the Inspector panel.
- When pressed:
  1. Extracts the current text, font size, and font reference from the selected text component.
  2. Opens the custom language editor tool.
  3. Allows the user to save or replace the corresponding translation entry for the assigned ID.

Duplicate Handling

- If the selected ID already exists in the system, a confirmation dialog appears before replacing the stored data.

Custom Layout

- The editor provides clearer information and prevents accidental edits on multiple objects simultaneously.

🧭 Usage Workflow

1. **Attach** the script to any 3D text object in your scene.
2. **Assign** the text component reference (either TextMesh or TMP).
3. **Enable** or disable automatic translation depending on your needs.
4. **Assign** a unique ID that matches the entry in your localization table.
5. **Optionally** click *"Import Settings"* in the Editor to register or update this entry.
6. When the game starts or the language changes, the text automatically updates.

🧩 Differences Between the Two Versions

Feature	Legacy TextMesh	TMP (TextMeshPro)
Text system	Old Unity 3D text system	Modern, advanced text rendering
Font support	Standard Font assets	TMP_FontAsset (improved Unicode coverage)
Font material handling	Updates mesh renderer's texture	Handled directly by TMP component

Field names	textComponent (TextMesh)	textComponent (TMP_Text)
Font metadata source	FontAndAlignmentUtility	FontAndAlignmentUtilityTMP

Despite these technical differences, their workflow, structure, and purpose are identical.

Why It Matters
Without these scripts, developers would need to manually update text content and fonts for every language.
This system automates that process, ensuring:

- Consistent text rendering across languages.
- Proper font selection for all character sets.
- Automatic synchronization when language settings change.
- Easy management through editor integration.

In short:
Both **LanguageTextMesh** and **LanguageTextMeshTMP** are **automatic localization handlers** that update 3D text objects in Unity based on current language settings, ensuring correct translation, font, and style are applied without manual intervention.

Language Create File

Overview
This component automates the creation of language files that contain either static or translated lines of text. It can be used in both the Unity Editor and at runtime, generating files in the appropriate folder based on the environment. Its purpose is to facilitate multilingual content generation by automatically applying translations based on the current language settings.

Inspector Fields / Settings

- **fileName:** The name of the output file that will be created. Developers can set this in the inspector.
- **fileExtension:** The extension of the output file, such as .txt. This determines the file type.
- **folderInUnity:** Folder path used when running inside the Unity Editor. It specifies where the file will be saved during development.
- **folderInBuild:** Folder path used when running in a built game. The file is saved in the streaming assets folder for runtime access.
- **fileLines:** A list of lines to include in the generated file. Each line can be:
  - A static text string.
  - Marked for translation, in which case the script will fetch the localized version based on the current language settings.
  - Associated with a unique language ID to retrieve translations.

Private Fields

- **languageData:** Stores the current language configuration, including selected language, translations, and IDs. This is used to look up translated text.

Properties
These allow external scripts or editor tools to get or set the main settings:

- **FileName:** Access or modify the file name.
- **FileExtension:** Access or modify the file extension.
- **FolderInUnity:** Access or modify the editor folder path.
- **FolderInBuild:** Access or modify the build folder path.
- **FileLines:** Access or modify the list of lines to be written to the file.

Unity Events

- **OnEnable:**
  - Subscribes to a global language update event.
  - Immediately triggers the file update so the content reflects the current language at the moment the object becomes active.
- **OnDisable:**
  - Unsubscribes from the language update event to avoid unnecessary updates when the object is inactive.

Core Method

- **LanguageUpdate:**
  - Loads the current language settings.
  - Iterates over each line in fileLines.
  - If a line is marked for translation, it fetches the corresponding translated text using the line's language ID.
  - Updates the line with the translation if it exists.
  - Calls the file creation method to write all lines to disk, creating or overwriting the target file.

Helper Methods

1. **GetFolderPath:**
   - Determines which folder to use for file creation depending on whether the script is running in the Editor or a build.
   - Returns the correct path based on the environment.
2. **CreateFile:**
   - Ensures the target folder exists, creating it if necessary.
   - Combines the folder path, file name, and extension to form the full file path.
   - Writes each line from fileLines to the file in order.
   - Logs the location of the created file for developer reference.

Editor Integration

The script includes a custom inspector to enhance usability in the Unity Editor:

- **Import Settings Button:**
  - Allows developers to load or update translations for each line from the language system.
  - Checks if any IDs already exist in the language system and prompts the user to confirm replacement.
  - Opens an editor window for each translatable line to allow manual editing.
- **File Preview:**
  - Displays the content of the file directly in the inspector.
  - Shows the file name and extension as a heading.
  - Lists all lines in the file so developers can quickly verify the output without opening the file externally.

How It Works Together

1. Developers define the file name, extension, folder paths, and content lines.
2. Some lines may be marked for translation with a language ID.
3. When the language changes or the component is enabled:
   - The script loads the current language data.
   - Translations are applied to lines marked for translation.
   - The file is created or updated in the correct folder for the current environment.

4. In the Unity Editor, the custom inspector allows previewing the file and importing translations without leaving the Editor.

Typical Usage

- Add the component to a GameObject.
- Configure the output file name, extension, and target folder.
- Define the lines to include in the file and mark which ones need translation.
- Optionally, use the inspector's import button to automatically fetch translations from the language system.
- At runtime or in the Editor, the file is generated with the appropriate translated content, ready for use in the game or for external processing.

Key Points

- Automatically applies translations to file lines based on current language settings.
- Generates files in a path appropriate to Editor or build environments.
- Supports multilingual content creation for localization workflows.
- Editor preview allows developers to check file content quickly.
- Ensures safe handling of existing IDs, prompting confirmation before overwriting translations.

Warning this component can be used for malicious purposes depending on the file format you define in its settings. It is your responsibility not to leave formats that can be used in malicious ways.

Language Script

Overview

This component is responsible for managing localized text content in a Unity scene. It updates text elements—whether in UI or 3D objects—based on the active language. The updates are applied using a list of language IDs, which link the text entries to their translations in the language settings. UnityEvents are used to dynamically apply these localized texts to various objects.

Main Component (LanguageScript)

- **Purpose:** Synchronizes scene elements with translated content by listening to language updates and applying localized texts automatically.

Inspector Fields / Settings

- **debug:** A boolean flag to enable or disable debug messages. When active, the component logs each text update to help developers verify which translations are applied.
- **scriptTexts:** A list of text entries that store:
  - **iD:** The unique identifier used to fetch the corresponding translation from the language data.
  - **text:** The actual localized string, updated at runtime.
  - **targetScripts:** UnityEvents associated with each text entry that apply the translated text to relevant objects.

Private Fields

- **languageData:** Holds the current language configuration, which contains all translations and IDs for the active language.

Properties

- **DebugLog:** Allows external scripts or editor tools to enable or disable debug logging.

- **ScriptTexts:** Exposes the list of text entries so other components can read or modify it if needed.

Unity Lifecycle Methods

1. **OnEnable:**
    - Subscribes to a global language update event.
    - Immediately triggers the text update to ensure the scene reflects the current language settings.
2. **OnDisable:**
    - Unsubscribes from the language update event to prevent unnecessary updates when the object is inactive.

Core Method

- **LanguageUpdate:**
    - Loads the current language settings.
    - Iterates through each text entry in scriptTexts.
    - Fetches the translated text corresponding to the entry's ID from the language data.
    - Updates the entry's text field with the localized content.
    - Logs the applied translation if debugging is enabled.
    - Invokes any UnityEvent callbacks associated with the text entry, applying the translation to the intended objects.

Utility Method

- **ApplyUnityEvent:**
    - Receives a UnityEvent and a string value (the localized text).
    - Iterates over all listeners of the UnityEvent.
    - Validates that each listener method exists and can accept a single string parameter.
    - Invokes the listener method, passing the localized text.
    - Ensures that translations are applied dynamically to all bound objects, whether UI labels, 3D text, or other components.

Editor Component (LanguageScriptEditor)

- **Purpose:** Provides a custom interface in the Unity Inspector for managing language entries and importing translations.
- **Features:**
    - Displays an "Import Settings" button to load or update translations based on existing IDs.
    - Checks whether any IDs are already saved in the language system and asks for confirmation before overwriting them.
    - Opens an editor window for each entry, allowing developers to manually edit or verify translations.
    - Maintains default inspector fields for easy access to all component settings.

How It Works Together

1. **At Runtime:**
    - The component listens for language change events.
    - When a language change occurs, it loads the current language data and updates all text entries.
    - Each text entry updates its value and applies it through UnityEvents, ensuring that the scene's UI and 3D objects reflect the current language.
2. **In the Editor:**
    - Developers can import or update translations for each text entry.

- The custom inspector ensures IDs are unique or prompts confirmation if they are already in use.
- Debug logging helps verify which translations are being applied.

Typical Usage Scenario

1. Attach the component to a UI or 3D object in the scene.
2. Add text entries to scriptTexts, each with a unique ID and optional UnityEvent bindings.
3. Enable debug logging during development to see which translations are applied.
4. At runtime, the component automatically updates all text entries when the language changes.
5. Developers can manage translations through the custom inspector, importing or editing entries as needed.

Key Points

- Automatically updates localized text in the scene.
- Works with both UI and 3D text elements.
- UnityEvents provide flexibility to apply translations dynamically.
- Debug mode helps verify applied translations.
- Editor integration simplifies importing and managing text IDs.

Language Audio Player

Overview

This script dynamically loads and plays audio files that are specific to a user's selected language. It works with multiple audio formats (such as WAV, MP3, or OGG) and can optionally play the loaded audio through a Unity AudioSource component. The script automatically updates the audio whenever the language changes and provides editor support to ensure proper configuration.

Main Component (LanguageAudioPlayer)

- **Purpose:** Handles dynamic playback of language-specific audio files and manages audio updates based on the active language.
- **Inspector Fields / Settings:**
  - **fileName:** The name of the audio file to be loaded. This file should exist in a folder corresponding to the selected language.
  - **audioType:** The format of the audio file, such as WAV, MP3, or OGG. This ensures the audio is correctly interpreted during loading.
  - **useAudioSource:** Determines whether the audio will be played using a dedicated AudioSource component.
  - **audioSource:** Optional component used to play the audio if the previous setting is enabled.
  - **audioClip:** Reference to the currently loaded audio clip, updated automatically when a file is loaded.
- **Private Fields:**
  - **languageData:** Cached reference to the current language settings.
  - **previousFilePath:** Stores the last loaded file path to avoid redundant loading.
  - **filePath:** Current resolved path of the audio file for the selected language.
- **Public Properties:** Provides external access to the key settings and loaded audio, including the file name, audio type, AudioSource, and loaded AudioClip.

Key Methods

1. **OnEnable:**
   - Registers the component to respond to language change events.

    - Immediately triggers an audio update to apply the correct audio if the language is already set.
2. **OnDisable:**
    - Unregisters from language change events to avoid unnecessary updates when the object is inactive.
3. **LanguageUpdate:**
    - Loads the current language configuration.
    - Resolves the full path to the audio file based on the selected culture/language.
    - Prevents redundant loading if the same audio file was previously loaded.
    - Verifies that the file exists; if not, it logs an error.
    - Cleans up any previously loaded audio clip to free memory.
    - Starts the coroutine to load the file asynchronously.
4. **LoadFileCoroutine:**
    - Loads the audio file asynchronously using Unity's web request system (even for local files).
    - Checks for errors during loading and reports them.
    - Extracts the audio clip from the loaded data.
    - Plays the audio using the AudioSource if enabled.

Editor Component (LanguageAudioPlayerEditor)

- **Purpose:** Provides a custom interface in the Unity Inspector to improve usability and prevent common mistakes.
- **Key Features:**
    - Shows a warning about using only basic ASCII characters in the file name to avoid path or loading issues.
    - Displays the default inspector fields for editing all the serialized settings (file name, audio type, AudioSource, etc.).
- **Usage:** When inspecting the component in Unity, the developer can easily assign the audio file, set the format, choose whether to use an AudioSource, and see warnings about potential issues.

How It Works Together

1. **At Runtime:**
    - The script listens for language changes globally.
    - When the language changes, it constructs the file path for the audio based on the selected culture.
    - Loads the audio clip asynchronously from the resolved path.
    - Plays the clip through an AudioSource if configured.
2. **Editor Workflow:**
    - Developers configure the audio file name, format, and optional AudioSource.
    - The custom inspector displays warnings to ensure proper setup.
    - This setup ensures that, at runtime, the correct audio plays automatically for each language without manual intervention.

Typical Usage Scenario

1. Attach the component to a 3D object or manager in the scene.
2. Assign the name of the audio file you want to play.
3. Set the file format to match the actual audio.
4. Optionally assign an AudioSource component to play the audio.
5. At runtime, whenever the language changes, the script automatically loads and plays the corresponding audio file.
6. Developers can configure multiple instances for different sounds or objects, each reacting to the current language settings.

Key Points

- Dynamically updates audio based on the current language.
- Supports multiple audio formats and optional AudioSource playback.
- Loads audio asynchronously to prevent frame drops or freezes.
- Avoids redundant loading for performance optimization.
- Provides editor support for proper configuration and warnings.

Language Canvas

Overview

This script is a Unity tool that manages language-specific user interface (UI) layouts for a canvas. It allows a canvas to automatically load and update its layout based on the currently active language. During development, it also provides editor tools to capture, save, and manage these localized layouts. Essentially, it connects canvas elements in the scene with external language-specific configuration data.

Main Component (LanguageCanvas)

- **Purpose:** Handles the application of localized canvas layouts at runtime and provides editor support for capturing canvas layouts.
- **Variables / Fields:**
  - **canvasID:** A unique identifier linking this canvas to its corresponding localized data. This ensures that the correct layout is loaded for the canvas.
  - **canvasStructure:** Holds the actual layout and hierarchy information of the canvas. This includes positions, scales, and other configuration details for all UI elements.
- **Properties:**
  - **CanvasID:** Allows getting or setting the canvas ID for external access.
  - **CanvasStructure:** Allows getting or setting the canvas layout data externally.
- **Key Methods / Events:**
  - **OnEnable:** Registers the canvas to respond to language changes and immediately applies the correct layout for the current language.
  - **OnDisable:** Unregisters the canvas from the language update event to avoid unnecessary updates when the canvas is inactive.
  - **LanguageUpdate:** Retrieves the active language settings and updates the canvas layout based on the canvas ID.
  - **LoadCanvasData:** Takes a JSON string representing the layout data, validates it, parses it into a structured format, stores it internally, and applies it to the canvas in the scene.
  - **SaveCanvasData (editor only):** Captures the current layout of the canvas and converts it into a JSON string. This allows developers to save the current state for editing or localization purposes.

Editor Component (LanguageCanvasEditor)

- **Purpose:** Provides a custom interface in the Unity Inspector to help developers import and manage localized canvas layouts without manually editing JSON.
- **Key Features:**
  - Displays a message reminding developers to activate all objects in the canvas before capturing the layout.
  - Provides an "Import Settings" button:
    - Checks if the canvas ID already exists in saved data.
    - Prompts the developer to confirm replacement if needed.
    - Captures the current canvas layout and opens a dedicated editor window for managing the canvas data.

- Draws default inspector fields for editing other component settings.
- **Usage:** When inspecting a canvas with this component, developers can use the editor interface to import, save, or replace canvas layouts for specific languages efficiently.

How It Works Together

1. **Runtime Behavior:**
   - The canvas registers itself to listen for language changes.
   - Whenever the language changes (or when the canvas is enabled), it retrieves the correct layout for its canvas ID.
   - The layout is applied automatically, adjusting positions, scales, and hierarchy of all UI elements.
2. **Editor Workflow:**
   - Developers can activate the canvas and capture its current layout.
   - The layout is stored as structured data, serialized into JSON.
   - This JSON can be reused or modified for localization purposes.
   - Editor tools ensure proper management of multiple canvas layouts and prevent accidental overwrites.

Typical Usage Scenario

1. Add the LanguageCanvas component to a Unity Canvas.
2. Assign a unique canvas ID that corresponds to a saved layout for the current language.
3. During runtime, the script automatically loads and applies the correct layout.
4. In the editor, activate all objects in the canvas and use "Import Settings" to capture the current layout.
5. The captured layout can then be saved, edited, or reused for other languages.

Key Points

- Connects a canvas to language-specific data for dynamic localization.
- Updates the canvas layout automatically when the language changes.
- Supports development workflow with editor tools for capturing and saving layouts.
- Uses structured data (JSON) for portable, reusable canvas configurations.
- Ensures safe editing with confirmation prompts and undo support in the editor.

Rebuild Canvas

Overview

This script is a Unity Editor tool designed to help developers rebuild and manage user interface (UI) canvas hierarchies based on language-specific data. The canvas hierarchy represents UI elements and their layout, and this script allows creating, saving, and replacing these layouts in a Unity scene using JSON data. It also includes a custom inspector for a user-friendly interface within Unity's Editor.

Core Components

Main Component (RebuildCanvas)

- **Purpose:** This component is attached to a GameObject and serves as the bridge between the canvas hierarchy in the scene and the JSON data stored externally.
- **Variables / Fields:**
  - canvasObject: The GameObject that will hold the generated canvas hierarchy in the scene.
  - canvasStructure: A structured representation of the canvas layout, which can be loaded from or saved to JSON.
  - canvasID: An internal identifier used to track which canvas data entry is currently selected. This helps map the in-scene canvas to its stored data.

- **Key Methods:**
  - **CreateCanvasData(json):**
    - Takes a JSON string describing a canvas layout.
    - Validates the JSON.
    - Parses it into a structured canvas representation.
    - Deletes any existing canvas in the scene to avoid duplicates.
    - Builds a new canvas hierarchy in the scene according to the data.
    - Uses Unity's Undo system to allow reverting the creation if needed.
  - **SaveCanvasData():**
    - Captures the current layout of the canvas in the scene.
    - Updates the internal canvasStructure variable with this data.
    - Converts the structure to a JSON string.
    - Returns the JSON, which can later be saved externally or reused.
- **Usage:** Attach this component to an empty GameObject, then either create a canvas from stored JSON data or save the current canvas to update the JSON.

Editor Component (RebuildCanvasEditor)

- **Purpose:** Provides a custom interface in the Unity Inspector for working with the RebuildCanvas component. It allows users to select canvas entries from previously saved data and generate or save hierarchies in the scene.
- **Variables / Fields:**
  - fileData: Path to the JSON file storing canvas configuration data (ProjectSettings/LanguageFileData.json).
  - canvasSave: A list of all canvas structures loaded from the JSON file. Each entry contains its canvas ID, layout data, and description.
  - script: Reference to the RebuildCanvas component currently being edited in the Inspector.
- **Key Methods:**
  - **OnInspectorGUI():**
    - Draws a custom interface in the Unity Inspector.
    - Shows warnings and instructions to guide the user.
    - Loads available canvas data from the JSON file.
    - Displays a dropdown to select a canvas ID if only one object is being edited.
    - Provides a button to generate a canvas hierarchy from the selected data.
    - Provides a button to save the current canvas hierarchy back to the JSON data, replacing the stored version.
    - Handles undo operations and disables buttons when necessary.
  - **LoadDataJson():**
    - Reads the JSON file from the project settings.
    - Deserializes the JSON into structured canvas data.
    - Populates the canvasSave list with all saved canvas entries for use in the dropdown.
- **Usage:** Select the GameObject with the RebuildCanvas component. In the Inspector, choose the canvas ID you want to recreate and click "Create Canvas Hierarchy" to generate it in the scene. After editing, click the "Replace canvasID" button to save changes.

How it Works Together

1. **Loading Canvas Data:**
   - The editor reads JSON data from a file containing all previously saved canvas layouts.
   - Each canvas entry has an ID and description to help identify it.
2. **Creating a Canvas in the Scene:**

    - When a canvas ID is selected, its JSON is parsed.
    - Any existing canvas is removed.
    - A new canvas hierarchy is generated according to the parsed structure.
    - Unity's Undo system allows reverting this creation.
3. **Saving a Canvas Layout:**
    - The current state of the canvas in the scene is captured into a structured representation.
    - This structure is converted into JSON.
    - The JSON can replace the previously stored entry, updating the external file.
4. **Editor Interface:**
    - Provides a dropdown to select which canvas to work on.
    - Provides buttons for creating and saving the canvas hierarchy.
    - Gives feedback and warnings to the user.

Typical Usage Scenario

1. Open Unity and attach RebuildCanvas to a GameObject.
2. Open the Inspector; the custom interface shows a dropdown with canvas IDs loaded from JSON.
3. Select a canvas ID and click "Create Canvas Hierarchy" to generate the layout in the scene.
4. Make changes to the canvas in the scene (moving UI elements, adding buttons, etc.).
5. Click "Replace canvasID" to save the current canvas back to JSON, updating the stored configuration.

Key Points

- Uses JSON to store canvas hierarchy data, making it portable and language-independent.
- Supports Undo operations for safety during editing.
- Ensures a smooth workflow for editors to rebuild complex UI layouts without manually recreating elements.
- Editor customization enhances usability and minimizes mistakes.

Language File Manager Window

Purpose of the Script

This script is a custom editor window for Unity designed to manage language files and UI components. It allows a developer to:

- Load, edit, and save language localization data.
- Organize UI text and canvas data.
- Detect duplicate IDs and maintain consistency across language files.
- Handle metadata like font, alignment, and component types.

The interface is scrollable and includes toggles, buttons, and dropdowns to control and view different aspects of language data.

Key Variables

- **languageForEditing**: Holds the language currently being edited (like "en" for English).
- **componentSave**: Stores all localized text components. Each entry contains ID, text, context, alignment, font size, and component type.
- **canvasSave**: Stores canvas-related UI components, including ID, JSON representation, and context.
- **availableLanguages**: List of languages detected on disk, including culture code, display name, and availability.
- **scrollPosition**: Keeps track of scroll position in the editor window.

- **firstTime**: Boolean flag indicating if the window is being opened for the first time to initialize data.
- **idIndex**: The currently selected ID for navigation and editing.
- **showTextData / showCanvasData**: Toggles to show or hide text and canvas component data.
- **fileIsSaved**: Tracks whether the current data state has been saved to disk.
- **assetsPath, canvasDataFile, languageDataFile, metaDataFile**: Paths for storing and reading TSV and JSON data files.
- **serializedObject**: Unity representation for serializing and editing fields in the inspector.
- **duplicateID / canvasID**: Lists of IDs that are duplicated for validation purposes.
- **cultures / cultureNames**: Arrays representing all supported cultures and their display names.
- **selectedCultureIndex**: Tracks which culture is currently selected in the dropdown.
- **trashImage**: Cached image for the delete button in the editor.

Core Methods and Their Functions
Editor Lifecycle Methods

- **ShowEditorWindow**: Opens the editor window and sets its title and icon.
- **OnEnable**: Initializes paths, loads cultures, ensures files exist, and loads a trash icon. This sets up the editor environment.
- **OnDestroy**: Saves editor data to JSON and unsubscribes from undo/redo events.
- **OnUndoRedo**: Repaints the window when an undo or redo action occurs.

GUI Drawing Methods

- **OnGUI**: Main method that draws the editor interface. It handles initialization, scroll view, language dropdown, ID display, toggles for text and canvas data, and property fields.
- **DrawActionButtons**: Renders buttons like “Open Folder,” “Save File,” “Load File,” “Rebuild Canvas,” “Organize Items,” and “Search by ID.” Each button triggers a corresponding action.
- **DrawLanguageDropdown**: Displays a dropdown with all available cultures, allowing the user to select the editing language and optionally load its data.
- **DrawIdDisplayPanel**: Shows the ID navigation panel, including duplicate warnings and controls to navigate or remove IDs.
- **RenderComponentDisplay**: Shows editable fields for a single text component, including ID, context, text content, alignment, font size, and font index. It adjusts fields depending on the component type.
- **DrawButton**: Utility to render a custom-styled button and execute an action when clicked.

Data Management Methods

- **CompareID**: Sorts text and canvas components by their ID to maintain order.
- **FindDuplicateIds**: Detects duplicate IDs in both text and canvas components to prevent conflicts.
- **RemoveID**: Removes a component with the current ID, optionally confirming with the user. Updates the current ID index after deletion.
- **SaveLanguageFile**: Saves the language and canvas data to TSV files. Steps include:
  - Sorting components and detecting duplicates.
  - Confirming overwrites with the user.
  - Syncing internal data with TSV tables.
  - Saving the final tables to disk.
- **LoadLanguageFile**: Loads TSV files, parses metadata, and updates internal data structures. Handles invalid files and exceptions.
- **GetAllavailableLanguages**: Extracts all available languages from a language data table and updates the internal list.

- **SaveDataJson / LoadDataJson**: Saves or loads editor state to a JSON file to preserve window state between sessions.
- **AddComponent / AddCanvas**: Adds or updates text or canvas components in their respective lists, sorts them, and saves the state.

Usage Workflow

1. **Open the Window**: Through Unity's menu under "Window → Language → Language File Manager."
2. **Select Language**: Use the dropdown to choose the language for editing.
3. **View Components**: Toggle text or canvas components to display in the editor.
4. **Navigate IDs**: Use the navigation panel to select, increment, or jump to IDs.
5. **Edit Components**: Modify text, context, alignment, font size, or font index as required.
6. **Add or Remove Items**: Use buttons or add new components to the list.
7. **Save Changes**: Click "Save File" to export all changes to TSV files and JSON for metadata.
8. **Load Existing Data**: Click "Load File" to import saved language and canvas data.
9. **Manage Canvas**: Rebuild canvas structures or organize items using dedicated buttons.
10. **Duplicate ID Handling**: The window highlights duplicate IDs and prevents saving until resolved.

This system is designed for Unity developers to maintain multilingual support efficiently, ensuring that all text and UI elements are organized, versioned, and easily editable in one centralized editor window.

Language ID Scanner Window

Purpose Overview

This script is an **Editor utility for Unity** called the **Language ID Scanner**. Its purpose is to help developers:

- Scan all scenes included in the Build Settings for components that have an integer field named **iD**.
- Detect duplicate IDs to avoid conflicts in language systems.
- Quickly locate and select GameObjects by their ID.
- Provide a user-friendly interface to browse IDs across multiple scenes.

It is intended for use **only in the Unity Editor**.

Key Variables

Variable	Purpose
scrollPos	Tracks the scroll position for the results list in the window.
sceneLanguageIDs	Stores a dictionary where the key is a scene path and the value is a list of found IDs in that scene.
currentScenePath	Saves the path of the currently open scene so it can be restored after scanning other scenes.
sceneFoldoutStates	Keeps track of which scene entries in the window are expanded or collapsed.
searchID	Holds the ID entered by the user to search for a specific component.

Core Methods and Their Functionality

1. ShowWindow

- Opens the scanner window from the Unity menu.
- Sets the window title and icon.
- Usage: The developer clicks Window → Language → Language ID Scanner to open the tool.

2. ShowWindowID

- Opens the scanner window and pre-fills it with a specific ID for immediate search.
- Usage: Useful for quickly finding a known ID without manually typing it.

3. OnGUI

- Draws the window interface.
- Displays buttons, input fields, and the scrollable list of results.
- Key functionalities in the interface:
    1. **Scan All Scenes Button**: Scans every scene in the Build Settings for all IDs.
    2. **Search Specific ID Field**: Lets the user enter an ID to search for across all scenes. Invalid IDs are highlighted.
    3. **Search ID Button**: Performs a scan for a single ID.
    4. **Clear Results Button**: Clears all previous scan results.
    5. **Results Display**: Lists scenes with foldouts, showing each ID found, the component type, object name, and a button to select the object in the hierarchy.
- Allows selection of objects directly from the results, highlighting them in the hierarchy.

4. ScanAllScenesInBuild

- Scans all enabled scenes in the Build Settings.
- Steps:
    1. Clears previous results.
    2. Opens each scene one by one.
    3. Calls ScanSceneForLanguageIDs to find all components with an iD field.
    4. Keeps a count of each ID to detect duplicates.
    5. Marks duplicate IDs in the results.
- Usage: Ensures the developer can see all IDs across all build scenes, including duplicates.

5. ScanSpecificID

- Scans for a single user-specified ID across all Build Settings scenes.
- Steps:
    1. Clears previous results.
    2. Opens each scene.
    3. Filters results to include only components with the target ID.
    4. Alerts the user about the number of matches or if none are found.
- Usage: Quickly locate a specific language ID in the project.

6. ScanSceneForLanguageIDs

- Scans a single scene for components with an integer field named iD.
- Steps:
    1. Iterates through all root GameObjects and their children.
    2. For each component, checks for a field named iD of type integer.
    3. Also checks for lists of objects that may contain an iD field.
    4. Records metadata: ID value, component type, object name, hierarchy path, and whether it is duplicate.
- Usage: Used internally by full-scene scans and single-ID scans to detect all relevant components.

7. GetHierarchyPath

- Builds a full path from the root of the scene hierarchy down to a specific GameObject.
- Helps identify objects uniquely in the hierarchy.
- Usage: Shown in the results to allow selection of the correct object.

Helper Class: LanguageIDInfo

- Represents metadata for a component found during scanning:
    - id: The integer ID of the component.
    - componentType: Type of component containing the ID.
    - objectName: Name of the GameObject.

    - hierarchyPath: Full path in the scene hierarchy.
    - isDuplicate: Whether this ID appears more than once in the scanned scenes.
- Usage: Each scanned ID is stored as a LanguageIDInfo object for display and reference.

How the Script Works in Practice

1. Developer opens the **Language ID Scanner** window in Unity.
2. They can either:
    - Scan **all scenes** in the Build Settings to gather every ID.
    - Enter a **specific ID** to search for a particular component.
3. The script opens scenes one by one and examines all GameObjects for components with an iD field.
4. Each result is stored with metadata including hierarchy, component type, and duplicate status.
5. Results are displayed in the editor window with foldouts for each scene.
6. Developers can click **Select** to jump directly to the object in the hierarchy.
7. Duplicate IDs are highlighted in red, making it easy to identify conflicts.

Summary

The **Language ID Scanner** is a developer tool that provides:

- Complete visibility of all language-related IDs across scenes.
- Duplicate detection to prevent conflicts in language systems.
- Quick navigation to objects by ID.
- Seamless integration with Unity Editor workflows, preserving the current scene and supporting undo/redo operations.

It's essentially a **comprehensive ID auditing tool** for Unity projects that rely on integer-based language identifiers.

Language Table Editor Window

Purpose of the Script

This script creates a custom editor window in Unity that allows developers to manage **language tables**. These tables are stored in TSV (tab-separated values) files and can be loaded, edited, saved, and resized. The editor also supports:

- Selecting specific rows and columns.
- Showing only selected rows/columns.
- Undo and redo operations.
- Visual cues for different types of cells, such as booleans, headers, IDs, and numeric values.

Core Components

1. Table Data and State

- **tableData**: Holds the complete set of language table rows. Each row contains an array of cell values.
- **currentFilePath**: Path of the currently loaded table file.
- **selectedRows / selectedColumns**: Lists containing the indices of selected rows and columns.
- **showOnlySelected**: When active, only selected rows and columns are displayed.
- **rowHeights / columnWidths**: Arrays storing the height of each row and the width of each column.
- **currentChargingMode**: Determines the type of table being loaded, e.g., a custom file or predefined LanguageData.

2. Interaction State

- **isResizingRow / isResizingColumn**: Flags indicating if a user is resizing a row or column.
- **initialResizePos**: Mouse position when resizing begins.
- **originalSize**: Original size of the row or column being resized.
- **resizeIndex**: Index of the row or column currently being resized.
- **mainScroll / tableScroll**: Positions for scrolling the editor and the table view.
- **isDirty**: Indicates if there are unsaved changes.
- **lastScroll**: Tracks the last scroll position to detect changes.

3. File Paths

- Paths to specific TSV files used by the editor: LanguageData, MetaData, and CanvasData.
- **assetsPath**: Root path for language-related files.

4. Appearance Constants

- Minimum sizes for rows and columns to prevent them from being too small.
- Colors for different types of cells:
    - **COLOR_LABEL_BACKGROUND** for headers.
    - **COLOR_TRUE / COLOR_LIGHT / COLOR_NUMBER** for boolean, light-themed, or numeric cells.
    - **COLOR_EDITOR_BACKGROUND / COLOR_BACKGROUND** for general backgrounds.
    - And other colors for highlighting errors, info, or selected cells.

5. Styles

Defines how text and labels are displayed:

- Editable text, header labels, centered or left-aligned labels, and boolean popups.

Key Methods and Their Roles

Opening and Managing the Editor

- **OpenWindow**: Opens the editor window from the Unity menu.
- **OnEnable**: Initializes the editor, sets file paths, and subscribes to undo/redo callbacks.
- **OnDestroy**: Cleans up the editor state, saves temporary data, and unsubscribes from undo/redo.
- **OnUndoRedo**: Repaints the window when undo or redo actions occur.
- **OnGUI**: Main method responsible for rendering the GUI. Displays scroll views, buttons, visibility toggles, and the table itself.

Styles Initialization

- **InitializeStyles**: Sets up text styles, header styles, popup styles, and ensures colors and alignment are correct. Prevents re-initialization if already set.

File Handling

- **RenderOpenFileButtons**: Shows buttons to load tables from different sources (custom file, LanguageData, MetaData, CanvasData).
- **DrawOpenButton**: Draws a single button and executes an action when clicked.
- **RenderSaveButtons**: Draws buttons for saving the table, showing unsaved changes as an asterisk, and handles both custom and predefined file paths.
- **AllowOverwrite**: Confirms with the user if an existing file should be overwritten.
- **DrawFixedSave**: Saves a predefined TSV file.
- **DrawResetButton**: Resets the editor to its initial state after confirmation.
- **ResetEditorState**: Clears all selections, table data, scrolls, and resizing states.

Visibility Controls

- **RenderVisibilityToggles**: Allows selecting specific rows or columns, toggling the visibility of only selected cells, and selecting all rows or columns via buttons. Updates the editor state whenever changes occur.

Table Loading and Saving

- **LoadTableFrom**: Loads a TSV file, initializes the table structure, and sets default row heights and column widths.
- **SaveTableToFile**: Saves the current table to a file and resets the unsaved changes indicator.

Table Rendering

- **RenderTable**: Draws the full table inside a scrollable area, including headers, rows, and cells.
- **RenderColumnHeaders**: Draws the top row with column numbers, handles resizing, and highlights selected columns.
- **RenderRowsAndCells**: Iterates through each row and draws row labels, all cells, and row resize handles.
- **RenderRowLabel / RenderRowCells**: Render the row index label and all cells in a row.
- **IsRectVisible**: Checks whether a cell or row is inside the visible scroll view.

Cell Rendering

- **RenderCell**: Decides how to render each cell based on its type (editable, boolean, header) and color.
- **RenderBoolCell**: Displays a dropdown for boolean values ("True" or "False") and updates the table when changed.
- **RenderEditableTextCell**: Allows inline editing of a cell's text with alignment and coloring.
- **GetCellColor**: Determines the background color of a cell based on row, column, and value.
- **ShouldCenterText**: Returns true if a cell's text should be centered based on its type.

Resizing

- **HandleColumnResize / RenderColumnResizeHandle**: Detects mouse interaction on column edges and allows horizontal resizing.
- **RenderRowResizeHandle**: Detects mouse interaction at the bottom of rows to allow vertical resizing.
- **HandleResize**: Updates the sizes of rows or columns during mouse drag and finalizes on mouse release.

Usage Summary

1. Open the editor from Unity's menu.
2. Load a TSV table (custom or predefined) using the buttons.
3. Edit cell values directly, toggle booleans, or change row/column sizes with drag handles.
4. Use visibility controls to focus on specific rows/columns.
5. Save changes with the save buttons. Unsaved changes are marked with an asterisk.
6. Reset the editor if needed, after confirmation.
7. All interactions are undoable via Unity's undo system.

In short, this script provides a **full-featured visual editor** for managing language tables in Unity, with support for selection, editing, resizing, coloring, and saving data in TSV format, while maintaining a clean and interactive user experience.

Language Prefab Creator

Purpose Overview

This script is a **Unity Editor utility** for **instantiating and configuring language-related prefabs**. It is designed to streamline the creation of localized UI and 3D objects in a Unity project. Key functionalities include:

- Automatic setup of UI Canvases and Event Systems when needed.
- Instantiation of prefabs such as language managers, buttons, toggles, input fields, text objects, audio sources, and more.
- Integration with Unity Editor features like Undo registration and prefab unpacking for seamless editing.
- Support for both modern TMP components and legacy Unity UI components.

Key Concepts and Variables

Concept / Variable	Purpose
Canvas creation	Ensures UI prefabs have a parent Canvas and an EventSystem to function correctly.
Prefab instantiation	Handles creating a prefab instance and placing it in the hierarchy under a selected object or a default Canvas.
Undo registration	Allows the user to undo the creation of prefabs within the Unity Editor.
Prefab unpacking	Converts prefab instances into editable GameObjects after instantiation, enabling full customization.
fileName	Name of the prefab to locate and instantiate.
selectedGameObject	The parent object under which the prefab will be placed. If none is selected, the prefab may be placed under a new Canvas for UI prefabs.
isUI	Indicates whether the prefab is a UI element requiring a Canvas and EventSystem.

Core Methods and Their Functionality

1. CreateUICanvas

- **Purpose:** Creates a new UI Canvas with all necessary components and an EventSystem.
- **Functionality:**
  1. Generates a new GameObject called "Canvas."
  2. Adds Canvas, CanvasScaler, and GraphicRaycaster components.
  3. Configures rendering, layer, sorting, and display settings.
  4. Creates a new EventSystem with input handling.
  5. Registers both the Canvas and EventSystem for Undo actions in the editor.
  6. Returns the Canvas component.
- **Usage:** Automatically invoked when a UI prefab is instantiated without an existing Canvas.

2. CreateAndConfigurePrefab

- **Purpose:** Handles the instantiation and configuration of any prefab, either UI or 3D.
- **Functionality:**
  1. Checks if a Canvas exists when the prefab is a UI element; creates one if necessary.
  2. Finds the requested prefab in the project. Logs an error if it does not exist.
  3. Determines the parent object for the new instance: either the selected GameObject, a Canvas for UI prefabs, or null for standalone 3D prefabs.
  4. Instantiates the prefab as a child of the chosen parent.
  5. Calls FinalizePrefabSetup to complete the setup.

- **Usage:** This method is used internally by all menu commands to create prefabs consistently.

3. FinalizePrefabSetup

- **Purpose:** Completes the setup for a newly created prefab instance.
- **Functionality:**
  1. Registers the creation for Undo, allowing users to revert the action.
  2. Unpacks the prefab so it can be edited like a normal GameObject.
  3. Selects the new instance in the hierarchy.
  4. Automatically triggers rename mode so the developer can assign a new name immediately.
- **Usage:** Ensures newly created prefabs are editable and ready for immediate use.

4. Menu Commands for Prefab Creation

The script provides multiple menu options in Unity under **GameObject → Language**, grouped into 3 categories:

3D Prefabs

- Create language-related 3D objects such as:
  - Language files, scripts, audio sources, legacy text, TMP text.
- These objects are instantiated in the scene hierarchy and can be edited directly.

UI Prefabs (TMP)

- Create UI components compatible with the TMP system, including:
  - Text, buttons, toggles, input fields, dropdowns, images, raw images.
- Automatically placed under a Canvas with an EventSystem if needed.

UI Prefabs (Legacy)

- Create UI components compatible with Unity's legacy system, including:
  - Text, buttons, toggles, input fields, dropdowns.
- Also ensures proper placement under a Canvas when instantiated.
- **Usage:** Developers select a parent object in the hierarchy (optional) and choose the menu command corresponding to the desired prefab.

How It Works in Practice

1. **Developer selects a parent object** in the hierarchy (optional).
2. **Triggers a menu command** to create a prefab.
3. **The tool determines the prefab type** (UI or 3D) and ensures a Canvas exists for UI elements.
4. **The prefab is instantiated** and parented to the selected object or Canvas.
5. **The prefab instance is unpacked**, making it editable.
6. **Undo actions are registered** to allow safe removal or changes.
7. **The object is selected and ready for renaming**, making it intuitive to organize.

Summary

This script **streamlines the process of adding language-ready prefabs** to a Unity project. It automates:

- Canvas and EventSystem setup for UI prefabs.
- Prefab instantiation and parenting.
- Prefab unpacking and Undo registration.
- Menu-based access to create both TMP and legacy UI components, as well as 3D language-related objects.

Essentially, it **makes it faster and safer for developers to create localizable UI and 3D objects** without manual setup or repetitive tasks.

Component Converter

Purpose Overview

This script is a **Unity Editor tool** designed to **convert standard UI, 3D, and audio components into LanguageTool-compatible components**. LanguageTool is a system used for localization, allowing text, images, audio, and interactive elements to automatically adapt to different languages. The script ensures:

- Components are converted only once.
- Child elements are handled properly.
- Undo actions are registered, allowing changes to be reverted.
- Feedback is logged to the developer.

Key Variables

Variable	Purpose
callCount	Tracks how many times the conversion method has been called during a session. This prevents multiple conversions when multiple objects are selected.
expectedCalls	Stores the number of objects expected to be processed, ensuring the conversion only triggers once all objects are accounted for.

Core Methods and Their Functionality

1. ConvertComponents

- **Purpose:** Entry point triggered from the Unity Editor menu. Converts all selected objects in the scene.
- **Functionality:**
  1. Checks how many objects are selected.
  2. Defers execution until all objects in a multi-selection are accounted for.
  3. If no objects are selected, it logs an error.
  4. Calls ObjectAnalysis for each selected object.
- **Usage:** Developers select objects in the hierarchy, then choose the menu command to convert all compatible components to LanguageTool types.

2. ObjectAnalysis

- **Purpose:** Determines which components on a GameObject are eligible for conversion.
- **Functionality:**
  - Checks for common UI components (Text, Button, Toggle, Image, RawImage, Dropdown, InputField, TMP variants).
  - Checks 3D text and meshes (TextMesh, MeshRenderer with TMP_Text).
  - Checks audio sources.
  - Calls the corresponding conversion method for each component type.
- **Usage:** Ensures every supported component on a GameObject is converted while avoiding duplicates.

3. UI Component Converters

- **ConvertButton / ConvertToggle:**
  Converts the text inside a Button or Toggle to a LanguageTool-compatible text component.
- **ConvertTextComponent:**
  Helper method for Buttons and Toggles. It searches for a child text component (regular or TMP) and converts it.
- **ConvertText / ConvertTMPText:**
  Converts standard Text or TMP_Text components into LanguageTool text components. Registers the conversion with Undo for safety.

- **ConvertImage / ConvertRawImage:**
  Converts Image or RawImage components into LanguageTool-compatible image components.
- **ConvertDropdown / ConvertTMPDropdown:**
  Converts Dropdown or TMP_Dropdown components and their options into LanguageTool dropdowns. It also adds a helper for adjusting sizes dynamically.
- **ConvertInputField / ConvertTMPInputField:**
  Converts input fields to localized input fields compatible with LanguageTool.
- **Usage:** These methods allow the localization system to automatically handle UI text, images, and interactive elements.

4. 3D and Audio Converters

- **ConvertTextMesh / ConvertMeshRenderer:**
  Converts 3D text objects and TMP meshes into LanguageTool-compatible 3D text components.
- **ConvertAudioSource:**
  Converts audio sources to LanguageTool audio players, allowing localized audio playback.
- **Usage:** Ensures that 3D objects and audio can also participate in the localization system.

How It Works in Practice

1. **Developer selects objects** in the Unity hierarchy.
2. **Triggers the menu command** to convert components.
3. **The tool analyzes each object**:
   - Converts UI components like buttons, text fields, and images.
   - Converts 3D text and meshes.
   - Converts audio sources.
4. **Checks for duplicates**: If a LanguageTool component already exists, it logs an error instead of adding another.
5. **Registers all additions with Undo**, so any change can be reverted.
6. **Logs every conversion** for developer feedback.

Summary

This script **automates the process of converting existing Unity components into LanguageTool-compatible components**, making it easier to localize a project. Key benefits include:

- Supports UI, TMP, 3D text, and audio.
- Avoids duplicate conversions.
- Handles child components automatically.
- Registers all changes with Undo.
- Provides logs for feedback and debugging.

Essentially, it **prepares a project's components for localization with minimal manual work**, ensuring consistency and reliability.

Font Asset Bundle Builder

Purpose Overview

This script is a **utility tool for Unity** designed to automate the creation of **AssetBundles** specifically for **Font** and **TextMeshPro (TMP) font assets**. AssetBundles are Unity's way of packaging assets so they can be loaded dynamically at runtime. This tool ensures that each font is packaged individually, with proper naming, compression, and error handling.

Key Variables

Variable	Purpose
guids	Stores the identifiers of assets or folders found in the project. Used to locate the source folder and font assets.
folderPath	Represents the path to the folder containing fonts to bundle. Typically this is the "AssetBundles" folder in the project.
assetPath	The full path to a specific font asset derived from its identifier.
extension	The file extension of a font asset, used to validate that it's a supported font type.
bundleName	The final name of the AssetBundle, derived from the font asset name, with a special extension indicating whether it's a TMP or regular font.
build	Holds the configuration for the AssetBundle build, including which asset to include and the bundle's name.

Core Methods and Their Functionality

1. BuildFontAssetBundles

- **Purpose:** Entry point triggered from the Unity Editor menu. It scans a designated folder and builds AssetBundles for every font found.
- **Functionality:**
  1. Searches for a folder named "AssetBundles" in the project.
  2. Converts the folder identifier to a usable folder path.
  3. Finds all regular font assets in that folder and builds an individual AssetBundle for each.
  4. Finds all TMP font assets in that folder and builds an individual AssetBundle for each.
  5. Refreshes the Unity asset database so the newly created bundles appear immediately.
- **Usage:** Allows developers to quickly package fonts into AssetBundles directly from the editor without manually building each one.

2. BuildFontAssetBundle

- **Purpose:** Handles the actual creation of a single AssetBundle for either a regular font or a TMP font.
- **Functionality:**
  1. Converts a font identifier to its actual asset path.
  2. Checks if the asset is a valid font file (for regular fonts). TMP fonts are handled separately.
  3. Constructs a bundle name based on the asset name, adding a special extension: .ltbundle for legacy fonts and .tmpltbundle for TMP fonts.
  4. Sets up a build configuration that tells Unity which asset to include in the bundle and what the bundle's name should be.
  5. Uses chunk-based compression to build the bundle for Windows standalone.
  6. Logs success or error messages to help the developer understand if the build succeeded.
- **Usage:** Ensures that each font asset is packaged correctly, with proper compression and error handling, ready for runtime use.

3. IsValidFontFile

- **Purpose:** Validates whether a given file extension corresponds to a supported font type.
- **Functionality:**
  - Checks the extension against common font file formats: .ttf, .otf, and .ttc.
  - Returns true if the extension matches a known font type; otherwise, false.
- **Usage:** Prevents invalid or unsupported files from being processed into AssetBundles, reducing build errors.

How the System Works in Practice

1. **Developer triggers the menu option** from Unity Editor.
2. **The tool searches for the "AssetBundles" folder**. If the folder does not exist, it logs an error and stops.
3. **It locates all regular and TMP fonts** inside the folder.
4. **For each font found**:
    - It validates the file type.
    - Constructs a unique bundle name based on the font type.
    - Packages the font into an AssetBundle using chunk-based compression.
    - Logs success or failure messages.
5. **Unity's asset database is refreshed**, making the new bundles immediately available for runtime loading.

Summary

This script streamlines the process of creating AssetBundles for fonts in Unity:

- Handles **both legacy fonts and TMP fonts**.
- Packages **each font individually** for dynamic runtime loading.
- Applies **chunk-based compression** for optimized performance.
- Performs **validation** to prevent invalid files from being bundled.
- Logs informative messages to the developer for monitoring success or failure.

Essentially, it **automates a repetitive task**, ensuring fonts are bundled correctly and consistently, which is especially useful in projects with multiple languages and localized fonts.