

**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 is designed to manage language updates within an application using a delegate and event-based system. Below is an explanation of how each part works, including variables and methods, followed by its usage:  ### \*\*Namespace: `LanguageTools`\*\*  The namespace groups the script into a logical container, making it part of a larger feature set related to language tools within the application.  ### \*\*Class: `LanguageManagerDelegate`\*\*  This is a static class, meaning it cannot be instantiated and only contains static members. The class is responsible for managing and notifying language updates in the application.  ### \*\*Delegate: `LanguageUpdateDelegate`\*\*  A delegate is a type that represents references to methods with a specific signature. Here, `LanguageUpdateDelegate` is declared without parameters and is designed to notify subscribers when a language update happens. It acts as a blueprint for the methods that will handle the language update event.  ### \*\*Event: `OnLanguageUpdate`\*\*  This is a static event of type `LanguageUpdateDelegate`. It allows other parts of the program to subscribe to notifications about language updates. Whenever a language update occurs, all the methods that are subscribed to this event are invoked. The event itself acts as a central hub for managing all the subscribers.  ### \*\*Method: `NotifyLanguageUpdate()`\*\*  This is the main method that triggers the event. When a language update occurs within the application, calling this method checks whether there are any subscribers to the event (through `OnLanguageUpdate?.Invoke()`) and, if so, it invokes the delegate, notifying all registered listeners.  - The `?.Invoke()` syntax is a safe way to call the delegate, ensuring that the delegate is only invoked if there are subscribers. If no one is listening for the event (i.e., `OnLanguageUpdate` is `null`), nothing happens, and no errors are thrown.  ### \*\*Usage:\*\*  - \*\*Subscribing:\*\* Other parts of the application that want to be notified when a language update occurs will subscribe to the `OnLanguageUpdate` event. They will provide a method that matches the `LanguageUpdateDelegate` signature, which gets executed when `NotifyLanguageUpdate()` is called.    - \*\*Notifying:\*\* Whenever the application needs to inform all listeners that a language change has happened (for example, after changing the user interface language), the `NotifyLanguageUpdate()` method is called. This will trigger the `OnLanguageUpdate` event, notifying all subscribers.  In summary, this script provides a centralized way to manage and notify different parts of the application when a language update occurs. By using delegates and events, it ensures a loosely coupled architecture where various parts of the application can react to language changes without directly depending on the `LanguageManagerDelegate` class. |

|  |
| --- |
| **Language Class Group** |
| This script defines data structures for handling language settings and canvas configurations in an application, likely within a Unity-based environment. It leverages Unity's serialization system to store and manage persistent data such as language selections, UI elements, and canvas states. Below is an explanation of each class, variable, method, and their usage:  ### \*\*Class: `LanguageSaveData`\*\*  - \*\*`selectedFile`\*\*: A string that stores the name of the file associated with the chosen language.  - \*\*`selectedLanguage`\*\*: A string that holds the name of the currently selected language.    \*\*Usage\*\*: This class is used to save and load user preferences for the selected language, likely in conjunction with settings or configuration files.  ---  ### \*\*Class: `LanguageOptions`\*\*  - \*\*`text`\*\*: The language name or label displayed in the UI.  - \*\*`sprite`\*\*: An image (sprite) that may represent the language (like a flag).  - \*\*`iD`\*\*: A unique identifier for each language option.  \*\*Usage\*\*: This class is useful for displaying different language options in a dropdown or a selection menu. Each option can have a visual representation and a unique ID for easy reference.  ---  ### \*\*Class: `ScriptText`\*\*  - \*\*`text`\*\*: The actual content of a text element (possibly part of a script).  - \*\*`iD`\*\*: A unique identifier for each piece of text.  - \*\*`targetScripts`\*\*: A Unity event that can be triggered with a string parameter. This allows other parts of the code to subscribe and react when the event is triggered.  \*\*Usage\*\*: This class manages text content that is part of the application's scripts and triggers certain actions (via `targetScripts`) when the text is updated or interacted with.  ---  ### \*\*Class: `LanguageLines`\*\*  - \*\*`text`\*\*: The content of a specific language line (sentence or word).  - \*\*`translateText`\*\*: A boolean flag to determine whether the text should be translated.  - \*\*`iD`\*\*: A unique identifier for the language line.  \*\*Usage\*\*: This is likely used in translation systems where certain lines of text can be marked for translation. The `translateText` flag helps the system know if the text requires translation.  ---  ### \*\*Class: `LanguageFileData`\*\*  - \*\*`fileName`\*\*: The name of the language file.  - \*\*`computerLanguage`\*\*: Refers to the system or programming language used.  - \*\*`firstTime`\*\*: A flag to determine whether this is the first time the language file is used.  - \*\*`idIndex`\*\*: Keeps track of the ID currently in use.  - \*\*`componentSave`\*\*: A list of `LanguageComponentSave` objects that contain language-related component data.  - \*\*`canvasSave`\*\*: A list of `CanvasSave` objects that store the state of canvas elements.  \*\*Usage\*\*: This class manages the language data for components and canvas objects, saving their states to a file.  ---  ### \*\*Class: `LanguageComponentSave`\*\*  - \*\*`iD`\*\*: A unique identifier for each language component.  - \*\*`text`\*\*: The text content of the component.  - \*\*`alignment`, `fontSize`, `fontListIndex`\*\*: Text styling attributes like alignment, font size, and font index.  - \*\*Various `Write` flags\*\*: Flags (`textWrite`, `alignmentWrite`, etc.) that indicate whether specific attributes (e.g., text, alignment) should be written to the language file.  \*\*Usage\*\*: This class stores the text content and formatting options for various UI components in the language file. The `Write` flags allow selective saving of properties.  ---  ### \*\*Class: `ExcelData`\*\*  - \*\*`language`\*\*: The language corresponding to this data set.  - \*\*`lines`\*\*: A list of key-value pairs where the key is the unique ID and the value is the translated text.  \*\*Usage\*\*: This is used for loading or saving data from an Excel file that contains translations. Each entry is mapped by its unique ID.  ---  ### \*\*Class: `ExcelTableData`\*\*  - \*\*`filesData`\*\*: A list of file names related to Excel data.  \*\*Usage\*\*: Manages the list of Excel files containing language or translation data.  ---  ### \*\*Class: `CanvasSave`\*\*  - \*\*`canvasID`\*\*: A unique identifier for each saved canvas.  - \*\*`json`\*\*: A JSON string that represents the serialized state of the canvas.  \*\*Usage\*\*: This class stores the saved state of a canvas in JSON format, enabling the application to restore the UI layout later.  ---  ### \*\*Class: `CanvasDataSave`\*\*  - \*\*`canvasData`\*\*: Contains information about the canvas itself.  - \*\*`savedCanvasData`\*\*: A list of `CanvasDataList` items that store data for each canvas object in the hierarchy.  \*\*Usage\*\*: Manages the overall structure and elements of a canvas, enabling the saving and loading of its state.  ---  ### \*\*Class: `CanvasData`\*\*  - \*\*`canvasName`, `canvasHierarchy`\*\*: The name and hierarchy of the canvas and its elements.  - \*\*`localRotation`, `localPosition`, `localScale`\*\*: The transformation properties (rotation, position, and scale) of the canvas.  - \*\*`anchorMin`, `anchorMax`, `anchoredPosition`, `sizeDelta`, `pivot`\*\*: RectTransform properties, defining how the canvas is anchored and positioned.  - \*\*CanvasScaler properties\*\*: Configuration for how the canvas scales, including `uiScaleMode`, `referencePixelsPerUnit`, and screen size options.  \*\*Usage\*\*: This class stores detailed information about a canvas's layout and configuration, making it easy to save and restore.  ---  ### \*\*Class: `CanvasDataList`\*\*  - \*\*`gameObjectName`\*\*: The name of the GameObject associated with the canvas.  - \*\*`gameObjectEnable`\*\*: Flag to indicate if the GameObject is enabled or not.  - \*\*RectTransform properties\*\*: The transform properties (position, scale, rotation) for the GameObject.  \*\*Usage\*\*: This class handles the transformation and state of individual GameObjects within a canvas, allowing the application to restore their states accurately.  ---  ### \*\*Overall Usage:\*\*  This script provides a system to manage language settings, text, and canvas configurations in a Unity-based application. It allows for saving and loading language options, UI components, and canvas elements across sessions, ensuring that user preferences and interface layouts are preserved and dynamically restored. |

|  |
| --- |
| **Language File Manager** |
| This script manages the loading and saving of language settings for an application. It operates within the `LanguageTools` namespace and is designed to help manage language files, extract specific data from these files, and provide localized content for different parts of the application, such as text lines and canvas configurations. Here’s an explanation of how the script works by discussing its methods and variables:  ### Core Methods:  1. \*\*LoadLanguageSettings()\*\*:  - This method is responsible for loading the `LanguageSettingsData` from the "Resources" folder within the project.  - If it can't find the file, it logs an error and returns `null`.  - \*\*Usage\*\*: The method is called whenever the language settings are required, particularly when constructing file paths for language files or retrieving default settings.  2. \*\*GetSaveFilePath()\*\*:  - This method constructs the save file path for the language settings.  - The path varies depending on whether the game is running in the Unity Editor or as a build:  - In the Unity Editor, it saves in the `Assets` folder.  - In the build, it saves in the `Application`'s data path.  - \*\*Usage\*\*: It is used to determine where to store or retrieve the saved language settings data.  3. \*\*GetLanguageFilesFolderPath()\*\*:  - This method returns the folder path where the language files are stored, which is defined in the `LanguageSettingsData`.  - \*\*Usage\*\*: It helps locate the directory where all the language files are stored, such as ".txt" files that contain translations.  4. \*\*FindDefaultLanguageFilePath()\*\*:  - This method searches for a language file based on the default language set in `LanguageSettingsData`.  - It scans through all the files in the specified folder and matches the second line of the file (which stores the language) with the default language.  - \*\*Usage\*\*: This method is helpful when no specific language is selected by the user, defaulting the application to the pre-configured language.  5. \*\*FindLineByID()\*\*:  - This method searches for a specific line in a language file based on its unique identifier (ID).  - It reads the file line by line and checks for an `id:` prefix, extracting and comparing the ID with the provided one.  - \*\*Usage\*\*: It allows the application to retrieve a specific translation or localized content based on an ID.  6. \*\*FindCanvasByID()\*\*:  - Similar to `FindLineByID()`, this method searches for a specific canvas configuration within a language file based on a `canvasID`.  - \*\*Usage\*\*: It helps locate canvas-related content (like layout or design settings) in the language file for localized UI adjustments.  7. \*\*GetLocalizedLineByID()\*\*:  - This method retrieves a specific localized string by its ID. It checks for a saved language file, and if it exists, it fetches the string from there.  - If no save file exists, it defaults to using the default language file.  - \*\*Usage\*\*: This is the primary method for retrieving translated text, whether from a saved language file or a default file.  8. \*\*GetLocalizedJsonByID()\*\*:  - This method retrieves the JSON content for a canvas by its ID. It works similarly to `GetLocalizedLineByID()`, either pulling from a saved file or the default file.  - \*\*Usage\*\*: Useful for restoring saved canvas configurations specific to a language, allowing UI elements to adapt based on language settings.  9. \*\*ExtractTextBetweenBraces()\*\*:  - This method extracts text that is enclosed within curly braces `{}` from a given input string.  - \*\*Usage\*\*: This could be used for parsing special format strings or placeholders within language files.  10. \*\*RemoveTextBetweenBraces()\*\*:  - This method removes any text that is enclosed within curly braces `{}` from the input string.  - \*\*Usage\*\*: This is useful for cleaning up strings that may contain tags or placeholders that are no longer needed.  11. \*\*ExtractIntValue()\*\*:  - This method extracts an integer value from a line of text, based on a given identifier. The identifier is used to locate where the integer is stored in the text, and the integer is parsed.  - \*\*Usage\*\*: It is likely used for parsing numeric data from language files, such as IDs or other metadata.  12. \*\*GetLanguageTagFromFile()\*\*:  - This method retrieves the language tag (the actual language identifier, such as "English" or "Spanish") from a language file by reading the second line.  - \*\*Usage\*\*: It helps determine what language a file corresponds to, which could be used to display available languages or validate that a file matches the user's preferred language.  ### How it Works:  The script primarily focuses on managing language settings and localizing content for the application. It deals with:  - \*\*Loading language settings\*\*: through resources (`LoadLanguageSettings`).  - \*\*Finding specific files\*\*: through methods like `FindDefaultLanguageFilePath` and `GetSaveFilePath`.  - \*\*Extracting localized text or UI elements\*\*: using methods like `FindLineByID` and `FindCanvasByID`.  This structure allows the application to switch between languages smoothly, load the appropriate localized files, and retrieve specific content based on user settings or defaults. |

|  |
| --- |
| **Language Editor Utilities** |
| This script, titled `LanguageEditorUtilities`, provides utility functions for managing language-related tasks in the Unity Editor. It offers various tools such as handling language files, working with UI components, fonts, alignment, and managing canvas and database elements related to language systems in a Unity project. Here's a breakdown of the key components, methods, and their usage:  ### 1. \*\*Draw Methods:\*\*  These methods handle UI rendering in the Unity Editor:    - \*\*`DrawReadOnlyMonoScriptField()`\*\*: Renders a disabled MonoScript field (readonly) for a MonoBehaviour object in the Unity Editor. This allows the script to be visible in the editor but not editable.    - \*\*`DrawLabeledTextField()` and `DrawLabeledIntField()`\*\*: Draw text and integer fields in the Unity Editor with specified labels and sizes, providing a clean way to edit values in a GUI format.    - \*\*`DrawArrowButton()`\*\*: Draws a button in the Editor for navigating through a list of IDs. It enables or disables the button based on the current index.  - \*\*`DrawColoredBox()`\*\*: Draws a colored background around a content block in the Unity Editor, making it visually distinct.  \*\*Usage\*\*: These methods simplify the process of creating custom UI elements in the Unity Editor. They are useful for drawing various types of input fields or custom styled buttons, aiding developers in managing language settings directly in the editor.  ### 2. \*\*Style Creation Methods:\*\*  These methods define the style (appearance) of UI components:  - \*\*`CreateLabelStyle()`\*\*: Creates a custom style for labels with configurable font size, boldness, and alignment.  - \*\*`CreateCustomButtonStyle()`\*\*: Defines a style for buttons with specific font size, bold text, and color properties (e.g., white text for normal, red text for hover).  \*\*Usage\*\*: These methods allow for consistent and stylized UI components, especially in custom Unity Editor windows.  ### 3. \*\*Language and Canvas ID Management:\*\*    - \*\*`IsIDInLanguageList()`\*\*: Checks if a given language ID exists in the list of available language IDs. Uses `LoadLanguageIDs()` to retrieve language IDs from a JSON file.    - \*\*`IsIDInCanvasList()`\*\*: Similar to the previous method, this checks if a given canvas ID is in the canvas list by reading from a JSON file.    - \*\*`FindDuplicateIDs()`\*\*: Finds and returns duplicate IDs in a list of integers. Useful for ensuring no two components share the same ID.  \*\*Usage\*\*: These methods are vital for managing IDs within the system, ensuring there are no conflicts, such as duplicate IDs, and that the system correctly recognizes IDs from language and canvas data files.  ### 4. \*\*Asset Search Functions:\*\*  - \*\*`FindPrefabByName()`\*\*: Searches for a prefab in the Unity project by name. It uses Unity's `AssetDatabase` to search through the project files.    - \*\*`FindTextureByName()`\*\*: Similar to `FindPrefabByName()`, this method searches for a texture asset by name and returns the corresponding texture.  \*\*Usage\*\*: These methods help developers quickly find specific assets (prefabs, textures) by name, making it easier to load and manage assets in the Unity Editor.  ### 5. \*\*Language File Parsing:\*\*  - \*\*`ParseLanguageComponent()`\*\*: Parses a line of text from a language file and extracts data into a `LanguageComponentSave` object. It identifies component properties like text, alignment, font size, and writes flags.  - \*\*`ParseCanvasSave()`\*\*: Parses a line from a language file and extracts data into a `CanvasSave` object, managing canvas properties like ID and JSON content.  - \*\*`ExtractIntValueOrDefault()`\*\*: Extracts an integer value from a string based on a given prefix (e.g., `Type:`), or returns a default value if not found. This method is used in parsing text from language files.  \*\*Usage\*\*: These methods are responsible for reading data from language files and converting them into usable objects in the system. They handle the extraction of language component properties and canvas settings from serialized language data.  ### 6. \*\*Utility Methods:\*\*  - \*\*`AddDefaultLanguageComponents()`\*\*: Adds a set of predefined language components and canvas save data to the respective lists. This method ensures that default data is present, which may act as fallback or initial content.  \*\*Usage\*\*: These utility methods are useful for initializing or resetting language-related data to a known state, ensuring that the application has a base set of language and UI components to work with.  ---  ### \*\*Overall Usage:\*\*  This script provides a set of editor tools for managing and interacting with language files, UI elements, and asset components in a Unity project. It helps developers:  - Create custom editor GUIs for managing language data.  - Ensure proper handling of IDs in the language and canvas systems.  - Load and manage assets like prefabs and textures.  - Parse language-related data from serialized files into usable Unity objects.  By integrating these tools, the language system becomes more manageable, especially in editor-mode, enabling streamlined processes for localization and user interface management. |

|  |
| --- |
| **Font And Alignment Utility / Font And Alignment Utility TMP** |
| This script, `FontAndAlignmentUtility`, is a utility class that handles font management and text alignment in Unity's UI system. It is particularly useful in multilingual applications, as it ensures that the correct fonts and alignments are applied across different languages. The script provides methods to retrieve fonts by index, find the index of a given font, and convert between alignment codes and `TextAnchor` values.  ### Key Components and Methods:  #### 1. \*\*GetFontByIndex(int fontListIndex)\*\*  - \*\*Description\*\*: This method retrieves a font from a list based on a given index.  - \*\*How it works\*\*:  - It loads the language settings using `LanguageFileManager.LoadLanguageSettings()`.  - The `fontListData` is accessed, which contains the list of fonts.  - If the provided index is within a valid range, it returns the font at the specified index (adjusting for zero-based indexing by subtracting 1).  - If the index is out of range, it returns `null`.  - \*\*Usage\*\*: This method is used to retrieve a specific font from the font list, which can then be applied to UI text components based on the selected language or settings.  #### 2. \*\*GetFontIndex(Font font)\*\*  - \*\*Description\*\*: This method finds and returns the index of a specified font in the font list.  - \*\*How it works\*\*:  - Similar to `GetFontByIndex()`, it loads the language settings and accesses the `fontListData`.  - It iterates through the list of fonts and checks if each font matches the provided font.  - If a match is found, it returns the index of the font (plus 1 to adjust for one-based indexing).  - If the font is not found, it returns `0`.  - \*\*Usage\*\*: This method is used when you need to determine the index of a specific font within the list, perhaps for saving the font index or verifying which font is in use.  #### 3. \*\*ConvertToTextAnchor(int alignment)\*\*  - \*\*Description\*\*: This method converts an integer alignment code to a corresponding `TextAnchor` value.  - \*\*How it works\*\*:  - It maps specific integer codes to Unity's `TextAnchor` enum values, which define text alignment in the UI.  - The method handles several predefined alignment codes (1, 2, 3, 7, 8, 9, 13, 14, 15) and converts them to corresponding positions like `UpperLeft`, `MiddleCenter`, `LowerRight`, etc.  - If the alignment code is not recognized, it logs a warning and defaults to `UpperLeft`.  - \*\*Usage\*\*: This method is essential for converting alignment data (stored as integers) into actual `TextAnchor` values that can be applied to Unity's UI components like `Text` or `TMP\_Text`.  #### 4. \*\*ConvertToAlignmentCode(TextAnchor alignment)\*\*  - \*\*Description\*\*: This method converts a `TextAnchor` value to its corresponding integer alignment code.  - \*\*How it works\*\*:  - It performs the reverse mapping from the `ConvertToTextAnchor()` method.  - For each `TextAnchor` value (e.g., `UpperLeft`, `MiddleRight`, etc.), it returns the corresponding integer code.  - If the `TextAnchor` value is not recognized, it logs a warning and returns the default code (`1` for `UpperLeft`).  - \*\*Usage\*\*: This method is useful for serializing or storing alignment values as integers, which can then be saved in configuration files or data structures and later converted back into `TextAnchor` values for UI layout purposes.  ### Overall Usage:  This utility class plays a crucial role in ensuring that fonts and text alignments are managed correctly across different languages in Unity's UI system. It helps:  - Retrieve fonts dynamically based on language settings.  - Convert text alignment settings between serialized integer codes and Unity's `TextAnchor` values, ensuring that text appears correctly aligned regardless of language.  - Assist with multilingual text rendering by ensuring that the appropriate fonts are loaded and aligned as per the requirements of different languages.  By utilizing these methods, developers can efficiently manage fonts and text alignment across different parts of the UI, ensuring a smooth localization process for multilingual support. |

|  |
| --- |
| **Language Settings Data** |
| This script defines a `ScriptableObject` class called `LanguageSettingsData` used for managing language settings in Unity. The object holds data related to localization, such as paths to language files, default language settings, and font information. Here's an explanation of the key elements and how they function:  ### Class: `LanguageSettingsData`  This class is a `ScriptableObject`, which is a special Unity asset type used to store data. The data stored in `LanguageSettingsData` is meant to manage localization configurations in the application. Let's break down each variable and method:  ### Variables:  1. \*\*`saveNameInUnity`\*\*:  - \*\*Description\*\*: This string represents the path to the JSON file that holds language settings when working inside the Unity Editor.  - \*\*Usage\*\*: During development in Unity, this file path is used to save and load the selected language settings, allowing changes without requiring a build.  2. \*\*`saveNameInBuild`\*\*:  - \*\*Description\*\*: The path to the JSON file for saving language settings when the application is built (i.e., when it is no longer running in the Unity Editor).  - \*\*Usage\*\*: During runtime in a built application, this path ensures the language settings are stored in an accessible location.  3. \*\*`folderName`\*\*:  - \*\*Description\*\*: The name of the folder containing all the language files.  - \*\*Usage\*\*: This is the directory where language files (such as `.txt` or `.json` files containing translations) are stored. The system uses this folder to load language-specific content.  4. \*\*`defaultLanguage`\*\*:  - \*\*Description\*\*: This string defines the default language to be used if no specific language is selected.  - \*\*Usage\*\*: When the application first starts or if no saved language preference exists, this default value ensures the application uses a specified language, such as English.  5. \*\*`fontListData`\*\*:  - \*\*Description\*\*: This object stores information about the fonts used in the application.  - \*\*Usage\*\*: The system references this list to select and apply the appropriate fonts for different languages, ensuring that characters are displayed correctly.  6. \*\*`fontListDataTMP`\*\*:  - \*\*Description\*\*: This is similar to `fontListData`, but it specifically holds information about TextMeshPro fonts, which are commonly used for sharper and more customizable text rendering in Unity.  - \*\*Usage\*\*: If the application uses TextMeshPro, this field manages the fonts used for localization.  7. \*\*`errorLanguageTool`\*\*:  - \*\*Description\*\*: A `GameObject` reference that might be used to display or log errors related to the language tool.  - \*\*Usage\*\*: This field provides a reference to a UI element (or another object) that handles errors or displays warnings when there are issues with language settings or loading.  ---  ### Editor-Specific Code (Only runs inside Unity Editor):  The code in this section is designed to interact with Unity’s editor, making it easier to manage the `LanguageSettingsData` from within the Unity Editor interface.  ### \*\*Class: `LanguageSettingsEditor`\*\*  1. \*\*`OpenLanguageSettingsData()`\*\*:  - \*\*Description\*\*: This method is linked to a Unity Editor menu item (`Window/Language/Language Settings`). When selected, it opens the `LanguageSettingsData` asset in Unity's Property Editor.  - \*\*How it works\*\*:  - It retrieves the `LanguageSettingsData` object using `LanguageFileManager.LoadLanguageSettings()`.  - If the data is found, it checks if there is already an open Property Editor window displaying the settings.  - If such a window exists, it brings that window into focus. If not, it opens a new Property Editor window with the settings data.  - If the settings data cannot be found, it logs an error message.  \*\*Usage\*\*: This method provides a way for developers to easily access and modify the `LanguageSettingsData` directly in the Unity Editor, streamlining the localization workflow.  ### Overall Usage:  The `LanguageSettingsData` `ScriptableObject` stores important settings for language management in a Unity project, allowing the application to:  - Easily reference paths for storing and loading language files in both the Unity Editor and the built application.  - Manage default language settings and font lists for localization purposes.  - Provide a clear and centralized place to configure language-related settings for multilingual applications.  The Editor utility, meanwhile, offers a convenient way to access and modify these settings within Unity's interface, enhancing the workflow for developers working on localization. |

|  |
| --- |
| **Language Settings Data Editor** |
| This script is a \*\*custom editor\*\* for the `LanguageSettingsData` `ScriptableObject` in Unity. It provides a user-friendly interface for modifying language settings, including default language selection, font lists, and file path configurations within the Unity Editor. Here's a breakdown of how the script works and what each part does:  ### \*\*Class: `LanguageSettingsDataEditor`\*\*  This class extends `Editor` and customizes how the `LanguageSettingsData` `ScriptableObject` is displayed and edited in the Unity Inspector. By overriding the default editor, it provides additional functionality and an enhanced user interface for managing language settings.  ### \*\*Variables:\*\*  1. \*\*`availableCultureDisplayNames`\*\*:  - \*\*Description\*\*: An array of strings that holds the display names of all available cultures (languages).  - \*\*Usage\*\*: These names are shown in a dropdown menu for selecting the default language of the application.  2. \*\*`availableCultures`\*\*:  - \*\*Description\*\*: An array of `CultureInfo` objects representing all cultures (languages) available in the system.  - \*\*Usage\*\*: This array is used to populate the `availableCultureDisplayNames`, ensuring the user can select any supported language from a standardized list.  3. \*\*`currentSelectedCultureIndex`\*\*:  - \*\*Description\*\*: An integer representing the index of the currently selected culture in the dropdown menu.  - \*\*Usage\*\*: This index keeps track of the user's current language selection and updates it as necessary.  ### \*\*Methods:\*\*  #### 1. \*\*`OnEnable()`\*\*:  - \*\*Description\*\*: This method initializes the available cultures and their display names when the custom editor is loaded.  - \*\*How it works\*\*:  - It retrieves all available cultures using `CultureInfo.GetCultures()`.  - It then populates the `availableCultureDisplayNames` array with the display names of these cultures, which are shown in the dropdown menu for language selection.  \*\*Usage\*\*: This method sets up the list of languages (cultures) for the user to choose from in the custom inspector.  #### 2. \*\*`OnInspectorGUI()`\*\*:  - \*\*Description\*\*: This method overrides the default Inspector interface and defines how the `LanguageSettingsData` object is displayed and edited in the Unity Inspector.  - \*\*Key UI Elements and Features\*\*:    1. \*\*Title and Path Fields\*\*:  - \*\*`saveNameInUnity`\*\*: The file path for saving language settings in the Unity Editor.  - \*\*`saveNameInBuild`\*\*: The file path for saving language settings in a built application.  - \*\*`folderName`\*\*: The folder where language files are stored.  - \*\*Usage\*\*: These fields allow the user to define where language-related files are stored during development and after the game is built.  2. \*\*Default Language Dropdown\*\*:  - \*\*`defaultLanguage`\*\*: The current default language of the application.  - The dropdown menu is populated using `availableCultureDisplayNames`, and the user can select a new default language.  - The selected language updates the `defaultLanguage` field of the `LanguageSettingsData` object.  3. \*\*Font List Data Fields\*\*:  - \*\*`fontListData`\*\*: Allows the user to select a `LanguageFontListData` asset for managing fonts.  - \*\*`fontListDataTMP`\*\*: Similar to `fontListData`, but specifically for TextMeshPro fonts.  - If these fields are populated, the editor also displays and allows editing of the associated font lists.  4. \*\*Error Language Tool\*\*:  - \*\*`errorLanguageTool`\*\*: A `GameObject` reference used for error handling or logging in the language tool.  - The user can assign or modify this reference in the inspector.  - \*\*How it works\*\*:  - The `Undo.RecordObject()` ensures that changes made to the `LanguageSettingsData` object can be undone.  - The `EditorGUILayout` methods are used to draw UI components such as text fields, object fields, and dropdowns.  - If any changes are made in the editor, the script marks the object as dirty (`EditorUtility.SetDirty()`), which triggers Unity to save the changes.  \*\*Usage\*\*: This method provides a customized, visually appealing, and easy-to-use interface for editing the language settings in Unity. It allows the user to configure file paths, font lists, and default languages, as well as to manage error handling objects.  ### \*\*Overall Usage:\*\*  This custom editor script significantly enhances the user experience when managing the `LanguageSettingsData` in Unity. It simplifies the process of selecting languages, configuring font lists, and managing localization settings by providing a structured and intuitive UI. Specifically, it is used for:  - \*\*Setting language paths\*\*: Defining where language files are saved and loaded during development and after the build.  - \*\*Selecting default languages\*\*: Enabling users to easily switch between different default languages via a dropdown menu.  - \*\*Managing font data\*\*: Allowing users to select and modify font lists used for localization, including regular and TextMeshPro fonts.  - \*\*Handling errors\*\*: Providing an option to assign a `GameObject` for handling language-related errors in the UI.  By integrating this custom editor, developers can ensure that localization settings are easily manageable within Unity’s Inspector, streamlining the process of building multilingual applications. |

|  |
| --- |
| **Language Data Creator** |
| This script is designed to streamline the creation and management of a `LanguageSettingsData` asset in a Unity project. It consists of two main components: a manual creation tool and an automatic initializer. These components work together to ensure that the `LanguageSettingsData` asset, which stores language settings for localization, is easily created and maintained.  ### \*\*Class: `LanguageDataCreator`\*\*  This class provides functionality for manually creating a `LanguageSettingsData` asset via Unity's menu system.  #### \*\*Method: `CreateLanguageDataAsset()`\*\*  - \*\*Purpose\*\*: To create a new `LanguageSettingsData` asset in the Unity project, specifically within the `Assets/Resources` folder.    - \*\*Key Steps\*\*:  1. \*\*Define asset path\*\*:  - The path where the asset is saved is defined as `Assets/Resources/Language Data.asset`. The `Resources` folder is special in Unity, allowing assets within it to be loaded at runtime using `Resources.Load()`.  2. \*\*Check for Resources folder\*\*:  - The method checks whether the `Resources` folder exists. If it doesn’t, it creates the folder. This ensures that the language data can be stored in a location accessible during both development and runtime.  3. \*\*Check for existing asset\*\*:  - The script checks if a `LanguageSettingsData` asset already exists at the specified path.  - If it exists, a dialog is shown asking the user if they want to replace the existing asset. This prevents accidental overwriting of important data.  4. \*\*Create and save the asset\*\*:  - If the user agrees (or if no asset exists), the script creates a new instance of `LanguageSettingsData` using `ScriptableObject.CreateInstance<LanguageSettingsData>()` and saves it at the specified path.  - The asset is marked as "dirty" using `EditorUtility.SetDirty()`, which signals Unity that the asset has been modified and should be saved.  5. \*\*Refresh Unity's asset database\*\*:  - The script then calls `AssetDatabase.SaveAssets()` and `AssetDatabase.Refresh()` to ensure the new asset is written to disk and that Unity's asset database is updated with the new file.    6. \*\*Select the created asset\*\*:  - Finally, the script focuses on the Project window and selects the newly created `LanguageSettingsData` asset using `Selection.activeObject`.  - \*\*Usage\*\*:  - This method is exposed to the Unity Editor as a menu item under \*\*Assets > Create > Language > Language Data\*\*. Developers can manually create or replace the `LanguageSettingsData` asset from the menu.  ---  ### \*\*Class: `LanguageDataAutoInitializer`\*\*  This class automatically ensures that a `LanguageSettingsData` asset exists in the project when the project is loaded.  #### \*\*Static Constructor: `LanguageDataAutoInitializer()`\*\*  - \*\*Purpose\*\*: To check whether the `LanguageSettingsData` asset exists when the Unity Editor loads. If the asset doesn't exist, it automatically creates one.  - \*\*Key Steps\*\*:  1. \*\*Register a delayed callback\*\*:  - Using `EditorApplication.delayCall`, a callback is registered to check for the existence of the language data asset. This is done after the Editor finishes loading.    2. \*\*Check if the asset exists\*\*:  - The method checks if the `Language Data.asset` file exists at `Assets/Resources/Language Data.asset` using `File.Exists()`.    3. \*\*Create the asset if missing\*\*:  - If the asset doesn't exist, it calls the `CreateLanguageDataAsset()` method from `LanguageDataCreator` to create the asset automatically.  - \*\*Usage\*\*:  - This auto-initializer ensures that a `LanguageSettingsData` asset is always present in the project. Developers don’t have to manually create the asset every time they set up a project. It simplifies the workflow by automating this repetitive task, particularly useful in team environments or new projects where localization is required.  ---  ### \*\*Overall Workflow and Benefits\*\*:  1. \*\*Manual Creation\*\*:  - Developers can manually create a new `LanguageSettingsData` asset through the Unity Editor's menu if they want explicit control over when and how the asset is created.  2. \*\*Automatic Initialization\*\*:  - The `LanguageDataAutoInitializer` class ensures that a `LanguageSettingsData` asset always exists when a Unity project is loaded. This is useful to avoid missing assets in new or freshly cloned projects.  3. \*\*Asset Management\*\*:  - Both classes contribute to effective management of the language settings asset, ensuring it is always available for localization tools and avoiding errors caused by missing data. By checking for existing assets before overwriting, the script prevents accidental data loss.  This script, by automating and simplifying the process of creating and managing the `LanguageSettingsData` asset, ensures that Unity projects using localization tools always have the necessary data in place. It reduces manual setup and potential errors while providing flexibility for developers to manage language settings as needed. |

|  |
| --- |
| **Language Font List Data / Language Font List Data TMP** |
| This script defines a `ScriptableObject` named `LanguageFontListData`, which holds a list of legacy font assets for use in localization within Unity. It also includes a custom editor that allows for an easy drag-and-drop interface to add fonts to this list. Here’s a detailed explanation of the key elements and how they work:  ### \*\*Class: `LanguageFontListData`\*\*  This is a `ScriptableObject` that stores a list of font assets. It is used in the Unity Editor to manage and organize fonts for localization purposes, specifically for legacy systems.  #### \*\*Variable:\*\*  - \*\*`fontList`\*\*:  - \*\*Type\*\*: `List<Font>`  - \*\*Description\*\*: This variable holds a list of font assets used for localization. Each element in the list is a `Font` object.  - \*\*Usage\*\*: The fonts in this list are likely referenced elsewhere in the application to dynamically set the font used for various localized languages. By centralizing these fonts in a `ScriptableObject`, the fonts can be managed efficiently and edited directly from the Unity Inspector.  #### \*\*Menu Attribute:\*\*  - \*\*`[CreateAssetMenu]`\*\*:  - \*\*Purpose\*\*: This attribute makes it possible to create a new `LanguageFontListData` asset from the Unity Editor by navigating to \*\*Assets > Create > Language > Language Font List Data (Legacy)\*\*.  - \*\*Usage\*\*: It simplifies the process of creating and organizing font lists by providing a UI menu option to generate new font list assets directly within the Unity Editor.  ---  ### \*\*Custom Inspector Class: `LanguageFontListDataInspector`\*\*  This class extends the default Unity `Editor` functionality to provide a custom user interface for editing the `LanguageFontListData`. It enables the user to drag and drop font assets directly into the font list, making it easier to manage.  #### \*\*Method: `OnInspectorGUI()`\*\*  This method is responsible for rendering the custom inspector UI in the Unity Editor.  - \*\*`serializedObject.Update()` and `ApplyModifiedProperties()`\*\*:  - \*\*Purpose\*\*: These methods synchronize changes between the serialized fields in the inspector and the actual data in the `ScriptableObject`.  - \*\*Usage\*\*: They ensure that any modifications made via the inspector (like adding fonts) are applied to the actual `fontList`.  - \*\*`GUILayoutUtility.GetRect()` and `EditorGUI.DrawRect()`\*\*:  - \*\*Purpose\*\*: These lines create a designated "drop area" in the inspector where users can drag and drop font assets.  - \*\*Usage\*\*: They visually define a space with a label ("Drop Font Assets Here") that indicates to users where to drag their font assets.  - \*\*Drag-and-Drop Handling\*\*:  - \*\*Purpose\*\*: The script listens for drag events (`EventType.DragUpdated` and `EventType.DragPerform`). When a font is dragged over the drop area and released, the event is processed to add the dragged object (if it is a valid `Font`) to the `fontList`.  - \*\*How it works\*\*:  1. \*\*`DragUpdated`\*\*: This event is triggered as the user drags something over the drop area. The script checks whether the drag is happening inside the drop area and changes the visual feedback (i.e., `DragAndDrop.visualMode = DragAndDropVisualMode.Copy`) to indicate that the user can drop the item.  2. \*\*`DragPerform`\*\*: When the user releases the dragged object, this event is triggered. The script accepts the drag-and-drop operation (`DragAndDrop.AcceptDrag()`) and adds any valid `Font` objects that are not already in the list.  3. \*\*`Undo.RecordObject()`\*\*: This records the action of adding fonts so that it can be undone, providing a safeguard in case the user makes an unwanted change.  4. \*\*`EditorUtility.SetDirty()`\*\*: Marks the `ScriptableObject` as "dirty," which signals to Unity that the object has changed and needs to be saved.  - \*\*`DrawDefaultInspector()`\*\*:  - \*\*Purpose\*\*: After handling the custom drag-and-drop functionality, the script still draws the default inspector fields for the `LanguageFontListData`. This includes the default display and controls for the `fontList`, allowing users to manually inspect and edit the list.  - \*\*Usage\*\*: This method ensures that all other properties (if any) of the `LanguageFontListData` are still displayed in the inspector, maintaining standard functionality alongside the custom features.  ---  ### \*\*Overall Usage:\*\*  This script is designed to help manage and organize a list of fonts used for localization in Unity. The `LanguageFontListData` `ScriptableObject` stores the font assets in a centralized way, which can be referenced by other parts of the localization system to dynamically switch fonts based on the selected language.  - \*\*Adding Fonts\*\*:  - With the custom inspector, fonts can easily be added to the `fontList` by dragging and dropping them into the designated area in the inspector. This reduces the manual effort needed to add fonts, making it more user-friendly.  - \*\*Managing Fonts\*\*:  - Users can still view and edit the list of fonts through the default inspector interface, giving them flexibility in how they manage the fonts.  - \*\*Automated Undo/Redo Support\*\*:  - The use of `Undo.RecordObject()` ensures that any changes made to the font list (such as adding or removing fonts) are recorded and can be undone if necessary, improving the robustness of the editing experience.  By combining the flexibility of `ScriptableObjects` with an intuitive drag-and-drop interface, this script simplifies the task of managing legacy fonts for multilingual applications in Unity, ensuring an efficient workflow for developers working on localization. |

|  |
| --- |
| **Language Initialization** |
| This script, `LanguageInitialization`, is responsible for initializing and managing language settings and font loading in a Unity game. It handles tasks such as loading language settings, selecting default fonts based on system language, managing font lists, and working with `AssetBundles` for both regular and TextMeshPro (TMP) fonts. The script ensures that the correct fonts are loaded and available for use in the game, streamlining localization for multilingual applications.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`fontListData`\*\*:  - A static reference to the `LanguageFontListData`, which holds a list of regular fonts for localization.    - \*\*`fontListDataTMP`\*\*:  - A static reference to `LanguageFontListDataTMP`, which holds a list of TextMeshPro (TMP) fonts for localization.  \*\*Usage\*\*: These variables store references to the font lists that are loaded during game initialization. They are populated with font data, which is later used for rendering text in different languages.  ---  ### 2. \*\*Method: `InitializeLanguageSettings()`\*\*  - \*\*Description\*\*: This method is automatically executed when the game starts, thanks to the `[RuntimeInitializeOnLoadMethod]` attribute.    - \*\*Functionality\*\*:  - It logs the start of the language initialization process.  - Calls `LoadLanguageSettings()` to load the necessary language settings.  - Calls `SetupDefaultLanguage()` to initialize language settings based on the system language and available language files.    - \*\*Conditional Blocks\*\*:  - Outside of the Unity Editor, additional methods are invoked to:  - Save the list of fonts to a file (`SaveFontListsToFile()`).  - Load regular and TMP fonts from `AssetBundles`.  - Reload fonts from saved font lists using `LoadFontListsFromFile()`.  \*\*Usage\*\*: This method ensures that all necessary language settings and fonts are initialized when the game starts, setting the stage for proper localization and font management.  ---  ### 3. \*\*Method: `LoadLanguageSettings()`\*\*  - \*\*Description\*\*: This method loads the language settings stored in `LanguageSettingsData`.  - \*\*Functionality\*\*:  - It retrieves the `LanguageSettingsData` using `LanguageFileManager.LoadLanguageSettings()`.  - Assigns the loaded `fontListData` and `fontListDataTMP` to the static variables for use throughout the game.  - \*\*Error Handling\*\*:  - If the `LanguageSettingsData` cannot be loaded, an error message is logged.  \*\*Usage\*\*: This method is crucial for ensuring that the language configuration, including font lists, is loaded at the start of the game.  ---  ### 4. \*\*Method: `SetupDefaultLanguage()`\*\*  - \*\*Description\*\*: This method sets up the default language for the game based on the system's language and available language files.  - \*\*Functionality\*\*:  - It retrieves the system language using `CultureInfo.InstalledUICulture.DisplayName`.  - It checks if a save file for language settings exists. If not, it searches through available language files and creates a new save file that matches the system language.  - The method reads and processes language files to match the language settings with the system's language.  \*\*Usage\*\*: This method ensures that the game defaults to the system language if no language preference is saved. It reads language files and saves the corresponding language settings for future use.  ---  ### 5. \*\*Method: `LoadRegularFontsFromAssetBundle()`\*\*  - \*\*Description\*\*: Loads regular fonts from an `AssetBundle` and adds them to the `fontListData`.  - \*\*Functionality\*\*:  - It searches for font asset bundles in the `Font` folder and loads them.  - Extracts and adds the fonts to `fontListData.fontList`.    - \*\*Error Handling\*\*:  - It logs an error if the asset bundle cannot be loaded or if no `LanguageFontListData` is found.  \*\*Usage\*\*: This method is responsible for dynamically loading regular fonts from asset bundles, which can be useful for games with a large number of fonts or for modular updates.  ---  ### 6. \*\*Method: `LoadTMPFontsFromAssetBundle()`\*\*  - \*\*Description\*\*: Similar to `LoadRegularFontsFromAssetBundle()`, but specifically for loading TextMeshPro fonts from `AssetBundles`.  - \*\*Functionality\*\*:  - It searches for TMP font asset bundles in the `Font` folder and loads them.  - Extracts and adds the fonts to `fontListDataTMP.TMPFontList`.    - \*\*Error Handling\*\*:  - Logs an error if the TMP asset bundle cannot be loaded or if no `LanguageFontListDataTMP` is found.  \*\*Usage\*\*: This method ensures that TMP fonts are loaded dynamically, allowing the game to use advanced text rendering with TextMeshPro.  ---  ### 7. \*\*Method: `SaveFontListsToFile()`\*\*  - \*\*Description\*\*: Saves the current font lists (regular and TMP) to text files in the `Font` folder.  - \*\*Functionality\*\*:  - It checks if the `Font` folder exists and creates it if necessary.  - Saves the names of fonts in `fontListData.fontList` and `fontListDataTMP.TMPFontList` to separate text files.  \*\*Usage\*\*: This method ensures that the list of fonts is stored persistently, making it easier to reload them later without having to reinitialize the entire font loading process.  ---  ### 8. \*\*Method: `LoadFontListsFromFile()`\*\*  - \*\*Description\*\*: Loads the font lists from previously saved text files and updates the font lists accordingly.  - \*\*Functionality\*\*:  - It reads the saved font names from the text files.  - Matches the loaded font names with existing fonts in `fontListData` and `fontListDataTMP`.  - Updates the font lists with the matched fonts.  \*\*Usage\*\*: This method is used to reload font data from files, minimizing the need for reloading fonts from asset bundles every time the game starts.  ---  ### \*\*Overall Usage and Purpose:\*\*  This script manages the entire initialization process for language settings and font loading in a Unity game. It ensures that:  1. \*\*Language settings\*\* are loaded and saved based on system language and available files.  2. \*\*Fonts\*\* (both regular and TMP) are dynamically loaded from asset bundles, reducing memory usage and allowing flexibility in updating fonts.  3. \*\*Font lists\*\* are saved and loaded efficiently to minimize the need for repeated loading operations.  By handling both language and font management in a centralized manner, this script plays a key role in ensuring that the game is properly localized and that all required fonts are available, especially in multilingual environments. |

|  |
| --- |
| **Access Permission Checker** |
| This script, `AccessPermissionChecker`, is designed to verify if the game has the necessary access permissions for critical folders in a Unity project, such as the `Assets` and `StreamingAssets` directories. If the game lacks access to these folders, the script will display a warning and log an error message. The check occurs automatically when the game starts. Here’s a detailed explanation of how the script works, including each variable, method, and its usage:  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Method: `InitializeCheckSettings()`\*\*  - \*\*Description\*\*: This method is automatically called when the game starts because it is tagged with `[RuntimeInitializeOnLoadMethod]`.    - \*\*Functionality\*\*:  - It invokes the `CheckAccessPermissions()` method to start the process of verifying folder access permissions.    - \*\*Usage\*\*: This method ensures that the access check runs immediately at the beginning of the game, providing an early detection mechanism for folder access issues.  ---  ### 2. \*\*Method: `CheckAccessPermissions()`\*\*  - \*\*Description\*\*: This method performs the actual check for access permissions to essential directories, specifically the `Assets` and `StreamingAssets` folders.  - \*\*Functionality\*\*:  - It calls `CheckFolderAccess()` twice:  - Once for `Application.dataPath` (the `Assets` folder).  - Once for `Application.streamingAssetsPath` (the `StreamingAssets` folder).  - If either of these folders cannot be accessed, it triggers `ShowWarning()`, which displays a warning to the user.  - It logs an error message if access is restricted or confirms success if access is granted.  - \*\*Usage\*\*: This method centralizes the logic for verifying directory access. It checks the key directories for permission to write and delete files and handles logging the results or showing a warning.  ---  ### 3. \*\*Method: `CheckFolderAccess(string path)`\*\*  - \*\*Description\*\*: This method tests whether the application has read and write access to a specified folder. It does so by attempting to write and then delete a temporary file.  - \*\*Parameters\*\*:  - \*\*`path`\*\*: The file path of the folder to check (e.g., `Application.dataPath` or `Application.streamingAssetsPath`).  - \*\*Functionality\*\*:  - It first checks if the directory exists. If not, it returns `false`, indicating no access.  - It creates a temporary file (`TestAccessFile.tmp`) in the specified folder and writes a simple text string to it.  - After writing, the script attempts to delete the file.  - If both file operations (write and delete) succeed, the method returns `true`. If an error occurs (such as `IOException` or `UnauthorizedAccessException`), it catches the exception, logs the error, and returns `false`.  - \*\*Error Handling\*\*:  - \*\*`IOException`\*\*: Catches general input/output errors (e.g., file system issues) and logs the error message.  - \*\*`UnauthorizedAccessException`\*\*: Catches permission-related errors (e.g., the application doesn't have the necessary permissions) and logs the error message.  - \*\*Usage\*\*: This method provides a detailed and robust way to check if the game has the necessary file access to perform read/write operations. It is the core utility function used to verify directory permissions.  ---  ### 4. \*\*Method: `ShowWarning()`\*\*  - \*\*Description\*\*: This method is called if folder access is restricted. It displays a visual warning to the user via a `GameObject` and logs the restriction in the console.  - \*\*Functionality\*\*:  - It loads the `LanguageSettingsData` via `LanguageFileManager.LoadLanguageSettings()`. This data contains a reference to a `GameObject` (likely a UI warning element) that will be displayed if access is restricted.  - If the `LanguageSettingsData` is missing, it logs an error message and returns.  - If the `errorLanguageTool` object exists, it instantiates it as a warning instance and marks it with `DontDestroyOnLoad()`, meaning it persists across scene transitions.  - If the `errorLanguageTool` cannot be instantiated, it logs an error indicating failure to show the warning.  - \*\*Usage\*\*: This method is responsible for alerting the user to folder access issues. It loads a predefined UI element (likely a pop-up or warning message) that informs the player or developer of the problem, ensuring they can address it.  ---  ### \*\*Overall Usage and Purpose\*\*:  This script ensures that the game can access important directories like `Assets` and `StreamingAssets`. If the game cannot access these folders, it:  1. Logs an error message.  2. Displays a warning UI to inform the user or developer of the issue.  This mechanism helps developers detect file permission problems early, especially in environments where folder access might be restricted (e.g., certain platforms or permissions settings). It can prevent the game from running into more significant issues later if it needs to load or save files but lacks permission to do so. |

|  |
| --- |
| **Language Text / Language Text TMP** |
| This script, `LanguageText`, is responsible for managing the localization of legacy UI `Text` components in Unity. It updates the text content, alignment, font size, and other properties based on the currently selected language. The script listens for language updates and applies the corresponding localized settings to the assigned UI text component. Additionally, the script provides a custom Unity Editor interface for modifying the text settings, making it easier to manage localization directly within the Editor.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`Text text`\*\*:  - \*\*Description\*\*: A reference to the UI `Text` component that will display the localized text.  - \*\*Usage\*\*: The script updates this `Text` component with localized content, changing its text, alignment, and font settings based on the selected language.  - \*\*`int iD`\*\*:  - \*\*Description\*\*: The ID that corresponds to a specific line in the language file. This ID is used to fetch the correct localized text.  - \*\*Usage\*\*: This ID is used to look up the corresponding line in the localization data. It allows the script to identify which string from the language file should be applied to the text component.  - \*\*`bool translateText`\*\*:  - \*\*Description\*\*: A flag that indicates whether or not the text should be translated.  - \*\*Usage\*\*: If `translateText` is `false`, the text component will not be updated with localized content. This allows certain UI elements to remain untranslated if necessary.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Stores the path of the selected language file from which the text is being pulled.  - \*\*Usage\*\*: This variable holds the path to the current language file used to retrieve localized content.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the file where user preferences, such as the selected language, are saved.  - \*\*Usage\*\*: This is used to load user settings, including language preferences, and ensure the correct language file is used for localization.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: This method subscribes the `LanguageText` component to the `OnLanguageUpdate` event when the object is enabled.    - \*\*Functionality\*\*:  - Registers the `LanguageUpdate()` method to be called whenever the `OnLanguageUpdate` event is triggered.  - Calls `LanguageUpdate()` immediately to apply the correct language settings when the object becomes active.  - \*\*Usage\*\*: This ensures that the UI text is updated with localized content as soon as the object is enabled or when the language is changed globally.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: This method unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Removes the `LanguageUpdate()` method from the event listener, preventing further updates when the object is no longer active.  - \*\*Usage\*\*: This method ensures that the object stops receiving language updates when it is disabled, conserving resources and preventing unnecessary updates.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: This method is responsible for retrieving the localized text and applying it to the `Text` component.  - \*\*Functionality\*\*:  - Retrieves the path to the save file using `LanguageFileManager.GetSaveFilePath()`.  - Fetches the localized text corresponding to the `iD` from the selected language file using `LanguageFileManager.GetLocalizedLineByID()`.  - Calls `ProcessLine()` to apply the localized text and any associated settings (alignment, font size, etc.) to the `Text` component.  - \*\*Usage\*\*: This method is invoked whenever the language changes or when the component is first enabled, ensuring that the `Text` component always displays the correct localized content.  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: This method processes the localized line of text and updates the `Text` component's properties accordingly.  - \*\*Functionality\*\*:  - If `translateText` is enabled, it extracts the text between curly braces from the localized line and updates the `Text` component with that content.  - It removes any curly braces from the localized line and extracts other properties such as:  - \*\*`alignment`\*\*: The text alignment is extracted and converted using `FontAndAlignmentUtility.ConvertToTextAnchor()`.  - \*\*`fontSize`\*\*: The font size is extracted from the localized line.  - \*\*`fontListIndex`\*\*: The font index is extracted, and the corresponding font is applied using `FontAndAlignmentUtility.GetFontByIndex()`.  - \*\*Usage\*\*: This method applies both the localized text and its associated properties (like alignment and font) to the `Text` component, ensuring that the component adheres to the language-specific settings.  ---  ### \*\*Custom Editor (`LanguageTextEditor`)\*\*  The custom editor allows the developer to interact with the `LanguageText` component in a more user-friendly way through Unity's Inspector. It adds custom buttons and fields for easily importing settings and handling ID conflicts.  - \*\*Method: `OnInspectorGUI()`\*\*:  - \*\*Description\*\*: This method overrides the default Unity Inspector for `LanguageText`, adding custom fields and buttons for managing the localization process.  - \*\*Key Features\*\*:  - Adds a button labeled \*\*"Import Settings"\*\*. When clicked, it allows the developer to import the settings for text, alignment, font size, and font index from the current `Text` component.  - Provides visual feedback (highlighting fields in red or yellow) when critical properties like the `Text` component or the `iD` are missing or conflicting.  - Displays warnings when the current `iD` is already in use, helping the developer avoid conflicts with other localized text entries.  - \*\*Usage\*\*: This custom editor simplifies the process of setting up localized text in the Inspector, offering helpful tools to avoid common pitfalls like conflicting IDs or unassigned `Text` components.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageText` script is designed to handle the localization of legacy Unity UI `Text` components. It:  1. \*\*Updates localized text\*\*: Based on the selected language, it retrieves and applies the correct text to the UI component.  2. \*\*Manages text properties\*\*: Updates alignment, font size, and font based on the language settings in the localization files.  3. \*\*Responds to language changes\*\*: Automatically updates the UI text when the global language setting changes.  4. \*\*Custom Inspector\*\*: Provides an easy-to-use interface in Unity's Inspector for managing the localization process and ensuring correct setup.  This script is ideal for games or applications that require legacy UI support with dynamic language switching and text localization. |

|  |
| --- |
| **Language Text Input Field / Language Text Input Field TMP** |
| This script, `LanguageTextInputField`, manages the localization of the `InputField` UI component and its placeholder in Unity's legacy UI system. It ensures that both the input text and placeholder text are updated according to the selected language, including properties like text alignment, font size, and font type, based on the localization settings.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`InputField inputField`\*\*:  - \*\*Description\*\*: A reference to the `InputField` UI component that will display localized text and a localized placeholder.  - \*\*Usage\*\*: This component is updated with localized text for both the input field itself and its placeholder, reflecting the active language settings.  - \*\*`int iD`\*\*:  - \*\*Description\*\*: An ID used to reference the specific line of text in the language file.  - \*\*Usage\*\*: This ID is used to fetch the corresponding localized text from the language files, ensuring the correct text is applied to the `InputField`.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: The path to the selected language file.  - \*\*Usage\*\*: Stores the path to the current language file being used for localization.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the save file that contains user preferences, such as selected language.  - \*\*Usage\*\*: This file is used to retrieve the selected language and other localization settings that are saved between game sessions.  - \*\*`Text text`\*\*:  - \*\*Description\*\*: A reference to the main text component of the `InputField`.  - \*\*Usage\*\*: This is the text that the user inputs. The script updates this field with the appropriate localized content, adjusting properties such as font and alignment.  - \*\*`Text placeholder`\*\*:  - \*\*Description\*\*: A reference to the placeholder text component of the `InputField`.  - \*\*Usage\*\*: This is the default text displayed when the `InputField` is empty. The script updates it based on the localized text.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event when the `InputField` becomes active.  - \*\*Functionality\*\*:  - Subscribes the `LanguageUpdate()` method to the `OnLanguageUpdate` event, ensuring that the `InputField` updates its content when the language changes.  - Calls `LanguageUpdate()` immediately to apply localization when the component is enabled.  - \*\*Usage\*\*: Ensures that the `InputField` updates its localized text when the language is changed globally or when the component is first enabled.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the `InputField` is disabled.  - \*\*Functionality\*\*:  - Unsubscribes the `LanguageUpdate()` method from the event, preventing updates when the component is inactive.  - \*\*Usage\*\*: Stops unnecessary updates when the component is disabled, optimizing performance by preventing event triggers.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: Updates the text and placeholder content of the `InputField` based on the selected language.  - \*\*Functionality\*\*:  - Retrieves the localized text from the language file using the `iD`.  - Applies the localized text to both the `InputField` and its placeholder by calling `ProcessLine()`.  - \*\*Error Handling\*\*:  - If the `InputField` or its `placeholder` component is missing, it logs an error and halts the process.  - \*\*Usage\*\*: This method is triggered whenever the language changes, ensuring that the `InputField` and its placeholder are updated with the correct localized content.  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: Processes the localized line of text and updates the `InputField` and its placeholder with the appropriate content and properties.  - \*\*Functionality\*\*:  - Extracts the localized text from the string (within curly braces) and assigns it to the `placeholder`.  - Strips the curly braces and extracts additional properties (alignment, font size, font index) from the remaining string.  - Updates the alignment, font size, and font based on the extracted values for both the `InputField`'s text and placeholder.  - \*\*Usage\*\*: This method ensures that the visual properties of the `InputField` (like text alignment and font) are updated alongside the localized text. It ensures that the UI behaves consistently with the selected language's settings.  ---  ### \*\*Custom Editor (`LanguageTextInputFieldEditor`)\*\*  This custom editor provides a more intuitive way to interact with the `LanguageTextInputField` component through Unity's Inspector. It includes custom buttons and fields to streamline the process of importing and editing localized settings.  - \*\*Method: `OnInspectorGUI()`\*\*:  - \*\*Description\*\*: Customizes the Unity Inspector for the `LanguageTextInputField` component by adding new controls for managing localization.  - \*\*Key Features\*\*:  - Provides a button to \*\*"Import Settings"\*\*, which allows developers to pull existing settings (text, alignment, font size, etc.) from the `InputField`'s placeholder and apply them to the localization system.  - Highlights required fields like the `inputField` in red if they are missing, and shows warnings for potential issues (like conflicting IDs).  - Displays fields for editing the `iD`, `selectedFile`, and `saveFile` properties directly in the Inspector, along with visual feedback if the `iD` is already used.  - \*\*Usage\*\*: This custom editor simplifies the process of configuring the `LanguageTextInputField` component, making it easier to manage and modify localization settings directly in Unity’s Inspector. It provides helpful feedback and visual cues to avoid errors like missing components or conflicting IDs.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageTextInputField` script is designed to handle the localization of legacy `InputField` components in Unity, ensuring that:  1. \*\*Localized content\*\*: The input field and its placeholder display the correct text based on the selected language.  2. \*\*Dynamic updates\*\*: The component updates automatically when the language changes or when the component is enabled.  3. \*\*Text properties\*\*: It adjusts the alignment, font size, and font type to match the language settings.  4. \*\*User-friendly customization\*\*: The custom editor provides an easy way to configure localization settings in Unity's Inspector.  This script is ideal for projects that require localized text input fields, ensuring that text content and UI properties are appropriately updated based on the active language settings. |

|  |
| --- |
| **Language Dropdown / Language Dropdown TMP** |
| This script, `LanguageDropdown`, handles the localization of a legacy `Dropdown` UI component in Unity. It dynamically updates the dropdown options, font, alignment, and font size based on the selected language settings, ensuring that the UI adapts to different languages. The script listens for language updates and applies the appropriate localized settings to the dropdown’s caption and options.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`Dropdown dropdown`\*\*:  - \*\*Description\*\*: A reference to the `Dropdown` UI component that will be localized.  - \*\*Usage\*\*: The script updates this dropdown with localized text and properties, such as alignment and font, for the caption and options.  - \*\*`List<LanguageOptions> options`\*\*:  - \*\*Description\*\*: A list of predefined options for the dropdown. Each `LanguageOptions` object contains text, a sprite, and an ID.  - \*\*Usage\*\*: This list holds the initial options for the dropdown, and each option is localized based on the ID.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Stores the path to the selected language file.  - \*\*Usage\*\*: The path is used to retrieve the correct language file for fetching localized text.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the saved language settings file containing user preferences.  - \*\*Usage\*\*: This file is used to load the saved language and other localization settings.  - \*\*`Text captionText`\*\*:  - \*\*Description\*\*: A reference to the dropdown’s caption text.  - \*\*Usage\*\*: The script updates this component with the localized text for the currently selected dropdown option.  - \*\*`Text itemText`\*\*:  - \*\*Description\*\*: A reference to the text component of each dropdown item.  - \*\*Usage\*\*: This is the text for the individual options in the dropdown, and the script updates it with the localized content.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event when the object is enabled.  - \*\*Functionality\*\*:  - Registers the `LanguageUpdate()` method to be called when the language changes.  - Calls `LanguageUpdate()` immediately to apply the correct localized settings upon enabling.  - \*\*Usage\*\*: Ensures that the dropdown is updated with localized text as soon as it is enabled and responds to any global language changes.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Removes the `LanguageUpdate()` method from the event listener to prevent updates when the object is no longer active.  - \*\*Usage\*\*: This prevents unnecessary updates when the component is disabled, optimizing performance by halting event triggers.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: Updates the dropdown options and caption based on the selected language settings.  - \*\*Functionality\*\*:  - Retrieves references to the dropdown's `captionText` and `itemText` components. If they are missing, it logs an error.  - Loads the save file path using `LanguageFileManager.GetSaveFilePath()`.  - Updates the first option in the dropdown by fetching its localized text using `LanguageFileManager.GetLocalizedLineByID()`.  - Calls `ProcessLine()` to apply font, alignment, and font size settings to the dropdown.  - Calls `ProcessOption()` to localize all dropdown options.  - If no option is selected, it sets the dropdown's caption text to the first option’s localized text.  - \*\*Usage\*\*: This method ensures that the dropdown adapts to the selected language by localizing the options and updating the visual properties (alignment, font size, etc.).  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: Applies formatting properties such as alignment, font size, and font to the dropdown’s caption and options.  - \*\*Functionality\*\*:  - Extracts text properties like alignment, font size, and font index from the localized string using `LanguageFileManager.ExtractIntValue()`.  - Converts the extracted alignment code into a `TextAnchor` value using `FontAndAlignmentUtility.ConvertToTextAnchor()`.  - Updates the alignment, font size, and font of both the `captionText` and `itemText`.  - \*\*Usage\*\*: This method ensures that the visual properties of the dropdown, such as font and alignment, are consistent with the language settings.  ---  ### 6. \*\*Method: `ProcessOption()`\*\*  - \*\*Description\*\*: Updates the dropdown options with localized text.  - \*\*Functionality\*\*:  - Iterates through each option in the `options` list, retrieving its localized text using `LanguageFileManager.FindLineByID()`.  - Clears the existing options in the dropdown and replaces them with the newly localized options.  - Restores the previously selected option after updating the dropdown.  - \*\*Usage\*\*: This method handles the localization of all dropdown options, ensuring that the correct text is displayed for each option based on the selected language.  ---  ### \*\*Custom Editor (`LanguageDropdownEditor`)\*\*  This custom editor enhances the Unity Inspector for the `LanguageDropdown` component, allowing for easier configuration and management of dropdown localization.  - \*\*Method: `OnInspectorGUI()`\*\*:  - \*\*Description\*\*: Customizes the Inspector interface for the `LanguageDropdown` component.  - \*\*Key Features\*\*:  - A button labeled \*\*"Import Settings"\*\* allows users to import the current dropdown’s alignment, font size, and font settings from the language files.  - Displays warnings if any dropdown option ID is already saved in the language list, helping avoid conflicts.  - Provides fields for editing the dropdown, options, selected file, and save file.    - \*\*Usage\*\*: The custom editor simplifies the process of setting up and configuring localized dropdowns, giving users a clear interface for managing text, fonts, and settings in the Inspector.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageDropdown` script integrates localization into Unity's legacy `Dropdown` UI component by dynamically updating the options and visual properties based on the selected language. The script:  1. \*\*Localizes dropdown text\*\*: It retrieves and applies localized content to both the dropdown caption and options.  2. \*\*Manages dropdown appearance\*\*: It adjusts the font, alignment, and font size to match the language settings.  3. \*\*Responds to language changes\*\*: It listens for global language updates and adapts the dropdown accordingly.  4. \*\*Custom Editor\*\*: Provides a user-friendly interface in the Unity Inspector for managing localization settings and resolving conflicts, ensuring an intuitive setup for developers.  This script is ideal for projects that require dropdown UI elements to be localized and adjusted dynamically based on the active language settings. |

|  |
| --- |
| **Language Manager / Language Manager TMP** |
| This script, `LanguageManager`, manages the language selection for a Unity application using a legacy `Dropdown` UI component. It automatically detects the system language, loads available language files, allows users to select a language, and saves the chosen language for future sessions. The script dynamically updates the application language based on the user’s selection.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`Dropdown dropdown`\*\*:  - \*\*Description\*\*: A reference to the `Dropdown` UI component where the user selects the desired language.  - \*\*Usage\*\*: The script updates this dropdown with available language options and listens for changes in the selected language.  - \*\*`string systemLanguage`\*\*:  - \*\*Description\*\*: Stores the system’s current UI language.  - \*\*Usage\*\*: Used to detect the default language of the system and match it with available language files.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Holds the file path of the selected language file.  - \*\*Usage\*\*: This path points to the specific language file that corresponds to the user's selected language.  - \*\*`string selectedLanguage`\*\*:  - \*\*Description\*\*: Stores the name of the selected language.  - \*\*Usage\*\*: This variable keeps track of the user’s selected language, which is displayed in the dropdown and saved for future use.  - \*\*`string pathFolder`\*\*:  - \*\*Description\*\*: The folder path where language files are stored.  - \*\*Usage\*\*: This folder contains the language files, which are loaded and used for language selection.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: Path to the file that stores the user’s language selection.  - \*\*Usage\*\*: This file is used to save and load the selected language to ensure the same language is applied in future sessions.  ---  ### 2. \*\*Method: `Start()`\*\*  - \*\*Description\*\*: Initializes the language dropdown by loading available language files, detecting the system’s language, and setting the dropdown to the correct option. Also saves the selected language for future sessions.  - \*\*Functionality\*\*:  - Ensures the `Dropdown` is assigned and clears any existing options.  - Detects the system’s language and loads available language files from the `pathFolder`.  - If no saved language file exists, it looks for a language file that matches the system language and saves it.  - Populates the dropdown with the available languages by reading the names from the language files.  - Loads the saved language file (if it exists) and sets the dropdown to the saved language.  - If no language is saved, the dropdown is set to the default language.  - Listens for changes in the dropdown selection.  - \*\*Usage\*\*: This method sets up the dropdown with all available language options and ensures that the appropriate language is applied when the application starts, either by detecting the system language or loading the saved preference.  ---  ### 3. \*\*Method: `OnLanguageChanged(int index)`\*\*  - \*\*Description\*\*: Handles the logic for updating the application when the user selects a new language from the dropdown.  - \*\*Functionality\*\*:  - Updates the `selectedLanguage` variable with the new language selected from the dropdown.  - Searches for the corresponding language file based on the selected language.  - Saves the selected language and file path to the `saveFile`.  - Notifies other parts of the application that the language has changed by calling `LanguageManagerDelegate.NotifyLanguageUpdate()`.  - \*\*Usage\*\*: This method is triggered whenever the user selects a different language from the dropdown, ensuring the application updates to reflect the new language and saving the selection for future sessions.  ---  ### \*\*Additional Functionalities\*\*  ---  ### \*\*Saving and Loading Language Preferences\*\*  - \*\*Saving the selected language\*\*:  - When the user selects a language, the `OnLanguageChanged()` method creates a `LanguageSaveData` object containing the selected language name and the corresponding file path. This data is serialized to JSON format using `JsonUtility.ToJson()` and written to the `saveFile`.    - \*\*Loading saved language\*\*:  - If a saved language file exists, it is read using `File.ReadAllText()`, and the language data is deserialized from JSON format using `JsonUtility.FromJson<LanguageSaveData>()`. The saved language is then set as the current selection in the dropdown.  ### \*\*Language File Detection\*\*  - \*\*File structure\*\*:  - The script expects the first line of each language file to contain the language name in the format `"Linguagem - [LanguageName]"` and the second line to contain the system language for matching purposes.    - \*\*Matching system language\*\*:  - When no saved language exists, the script attempts to match the system’s language (detected using `CultureInfo.InstalledUICulture.DisplayName`) with the second line of the language files.  ### \*\*Dropdown Initialization\*\*  - \*\*Populating the dropdown\*\*:  - The script reads all `.txt` files in the `pathFolder`. Each file’s first line (formatted as `"Linguagem - [LanguageName]"`) is used to add an option to the dropdown, representing each available language.    - \*\*Setting the default language\*\*:  - The default language is retrieved from `LanguageFileManager.LoadLanguageSettings()`. If no language is saved, the dropdown is set to the default language.  ### \*\*Event Handling\*\*  - \*\*Language updates\*\*:  - The script subscribes to the `LanguageManagerDelegate.OnLanguageUpdate` event. This ensures that whenever the language is changed, all parts of the application that depend on the current language can be updated accordingly.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageManager` script integrates language selection into Unity’s legacy UI system using a dropdown. It:  1. \*\*Automatically detects the system language\*\*: Tries to match the system's language with the available language files and sets it as the default.  2. \*\*Populates the dropdown with available languages\*\*: It reads from the language files and populates the dropdown with language options.  3. \*\*Saves and loads language preferences\*\*: The selected language is saved to a file and reloaded for future sessions to maintain user preference.  4. \*\*Dynamically updates the application language\*\*: When the user selects a different language, the application updates to reflect the new language, and the selection is saved for future use.  This script is essential for managing user language preferences in applications that need to support multiple languages, ensuring a smooth and persistent localization experience. |

|  |
| --- |
| **Adjust Size To Dropdown / Adjust Size To Dropdown TMP** |
| This script, `AdjustSizeToDropdown`, dynamically adjusts the size of a UI element in Unity based on the dimensions of a dropdown menu and the text it contains. It ensures that the UI element fits well within the boundaries of the canvas, handling line breaks and allowing for manual size adjustments. The script supports resizing in two directions (Left or Right), depending on the layout needs.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`float manualSizeAdjustment`\*\*:  - \*\*Description\*\*: A manual adjustment value for tweaking the size of the UI element.  - \*\*Usage\*\*: Allows the user to manually fine-tune the size of the UI element beyond the automated adjustments.  - \*\*`int sizeMultiplier`\*\*:  - \*\*Description\*\*: A multiplier used to adjust the size when text breaks into multiple lines.  - \*\*Usage\*\*: Increases the UI element size proportionally when text wraps into more than one line, providing enough space for the text.  - \*\*`float margin`\*\*:  - \*\*Description\*\*: A margin value that adds extra spacing between the UI element and the canvas boundaries.  - \*\*Usage\*\*: Ensures that the UI element does not touch or overflow past the canvas boundaries.  - \*\*`FitDirection fitDirection`\*\*:  - \*\*Description\*\*: An enum that determines whether the size adjustment should occur to the Left or Right.  - \*\*Usage\*\*: Controls the direction in which the UI element will expand or contract based on the text size and position relative to the canvas.  - \*\*`RectTransform parentRect`\*\*:  - \*\*Description\*\*: A reference to the `RectTransform` of the UI element being resized.  - \*\*Usage\*\*: This is the main UI element that will be resized based on text content and the dropdown’s size.  - \*\*`RectTransform canvasRectTransform`\*\*:  - \*\*Description\*\*: A reference to the `RectTransform` of the parent canvas.  - \*\*Usage\*\*: This is used to calculate the position and boundaries of the UI element relative to the canvas to ensure it fits within the canvas.  - \*\*`float canvasWidth`\*\*:  - \*\*Description\*\*: The width of the canvas.  - \*\*Usage\*\*: Helps determine if the UI element is fitting within the canvas boundaries.  - \*\*`float objectWidth`\*\*:  - \*\*Description\*\*: The current width of the UI element.  - \*\*Usage\*\*: Used in calculations to resize the element based on its content and position.  - \*\*`List<BrokenTexts> textList`\*\*:  - \*\*Description\*\*: A list of `Text` components within the UI element and their line break status.  - \*\*Usage\*\*: Keeps track of the `Text` components inside the UI element, checking if any text breaks into multiple lines.  - \*\*`bool textIsBroken`\*\*:  - \*\*Description\*\*: A flag indicating if any text in the UI element breaks into multiple lines.  - \*\*Usage\*\*: If true, it triggers additional size adjustments to accommodate the text.  - \*\*`float sizeAdjustment`\*\*:  - \*\*Description\*\*: Stores the final adjustment value for resizing the UI element.  - \*\*Usage\*\*: This value is calculated dynamically and applied to the UI element based on the size of the text and other factors.  ---  ### 2. \*\*Method: `Start()`\*\*  - \*\*Description\*\*: Initializes the component, setting up references to necessary UI elements, including the canvas and text components.  - \*\*Functionality\*\*:  - Retrieves the `RectTransform` of the UI element (`parentRect`) and the `RectTransform` of the canvas containing the element (`canvasRectTransform`).  - Collects all `Text` components inside the UI element and adds them to the `textList` for monitoring line breaks.  - Ensures that all text components allow overflow vertically, preventing unexpected clipping of text.  - \*\*Usage\*\*: This method prepares the component for dynamic resizing by gathering all necessary references and initializing the text list for line break detection.  ---  ### 3. \*\*Method: `Update()`\*\*  - \*\*Description\*\*: Dynamically adjusts the size of the UI element in each frame based on the current text content, its position relative to the canvas, and line breaks.  - \*\*Functionality\*\*:  - Updates the `canvasWidth` and `objectWidth` to reflect the current sizes of the canvas and UI element.  - Calls `CheckTheLineBreak()` to see if any text is breaking into multiple lines, setting the `textIsBroken` flag.  - Increases `sizeAdjustment` if the text is breaking into multiple lines, ensuring the UI element grows to accommodate.  - Depending on the `fitDirection` (Left or Right), calculates adjustments and applies them to the UI element using the `offsetMin` and `offsetMax` properties.  - Checks if the UI element is exceeding the canvas boundaries and switches the `fitDirection` if necessary to ensure the element remains within the canvas.  - \*\*Usage\*\*: This method handles the real-time resizing of the UI element, adjusting its size based on the text inside it and ensuring it fits within the canvas. It runs continuously, checking if resizing is necessary.  ---  ### 4. \*\*Method: `CheckTheLineBreak()`\*\*  - \*\*Description\*\*: Checks if any `Text` component inside the UI element breaks into multiple lines.  - \*\*Functionality\*\*:  - Iterates through each `Text` component in `textList` and checks its line count using `cachedTextGenerator.lineCount`.  - Sets the `brokenText` flag for each `Text` object accordingly, logging a warning if any text breaks into multiple lines.  - \*\*Usage\*\*: This method detects when text wraps into multiple lines and ensures that the UI element adjusts its size to accommodate the increased height of the text.  ---  ### \*\*Helper Class: `BrokenTexts`\*\*  - \*\*Description\*\*: A small class used to track individual `Text` components and whether they break into multiple lines.    - \*\*Variables\*\*:  - \*\*`Text itemLabel`\*\*: Reference to the `Text` component.  - \*\*`bool brokenText`\*\*: A flag indicating if the text breaks into multiple lines.  - \*\*Usage\*\*: This class allows the script to monitor each `Text` component and adjust the size of the UI element if any of them break into multiple lines.  ---  ### \*\*Overall Purpose and Usage\*\*  The `AdjustSizeToDropdown` script dynamically adjusts the size of a UI element in Unity based on the content of a dropdown menu and its text components. It ensures that:  1. \*\*Text accommodation\*\*: The UI element resizes to fit its content, accounting for line breaks and providing extra space using a configurable margin.  2. \*\*Canvas boundary adherence\*\*: The UI element stays within the boundaries of the canvas, adjusting its size and position (either Left or Right) as necessary.  3. \*\*Real-time adjustments\*\*: The script continuously checks and adjusts the size of the UI element, responding to changes in content or screen size in real time.  This script is ideal for applications where dynamic text content, such as in dropdowns or menus, needs to adjust its UI layout while ensuring a clean and flexible fit within the overall UI structure. |

|  |
| --- |
| **Language Image** |
| This script, `LanguageImage`, dynamically updates a UI `Image` component based on the selected language. It loads an appropriate image from the file system corresponding to the current language settings, using asynchronous file loading with a coroutine. The script ensures that the correct image is displayed based on the user's language selection, and it updates automatically when the language changes.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`Image image`\*\*:  - \*\*Description\*\*: A reference to the `Image` UI component that will display the localized image.  - \*\*Usage\*\*: The script updates this `Image` component by loading and applying an image file based on the selected language.  - \*\*`string fileName`\*\*:  - \*\*Description\*\*: The name of the image file to be loaded, specified in the Inspector.  - \*\*Usage\*\*: This filename is used to construct the full path to the image based on the current language settings.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: The path to the currently selected language file.  - \*\*Usage\*\*: This is used to find the corresponding image file for the selected language.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the file that stores the user's selected language settings.  - \*\*Usage\*\*: This file is used to load and save the user’s language preferences, ensuring the correct language and associated image are loaded.  - \*\*`string filePath`\*\*:  - \*\*Description\*\*: The full path to the image file to be loaded.  - \*\*Usage\*\*: This path is constructed dynamically based on the selected language and the `fileName`. It is used to locate and load the image file.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: This method subscribes to the `OnLanguageUpdate` event when the object is enabled and immediately calls `LanguageUpdate()` to ensure the image is updated based on the current language.  - \*\*Functionality\*\*:  - Subscribes to the `LanguageManagerDelegate.OnLanguageUpdate` event so the `LanguageUpdate()` method is triggered when the language changes.  - Immediately calls `LanguageUpdate()` to update the image when the component becomes active.  - \*\*Usage\*\*: This ensures that the image is updated when the script is first enabled or when the global language changes.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: This method unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Unsubscribes from the `OnLanguageUpdate` event to prevent unnecessary updates when the component is not active.  - \*\*Usage\*\*: This stops the image from being updated when the component is disabled, optimizing performance by avoiding unnecessary event triggers.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: This is the core method that handles updating the image based on the current language settings.  - \*\*Functionality\*\*:  - Checks if the `Image` component is assigned. If not, logs an error.  - Loads the path to the save file where user preferences (like selected language) are stored.  - If the save file exists, it reads the user's selected language. If not, it loads the default language file.  - Constructs the full path to the image file based on the selected language and the `fileName`.  - If the image file is found, it starts the `LoadFile()` coroutine to load the image asynchronously.  - If the image file is not found, it logs an error.  - \*\*Usage\*\*: This method ensures the correct image is displayed by reading the user’s selected language from a file and constructing the appropriate file path. It also handles loading the image asynchronously to prevent performance issues.  ---  ### 5. \*\*Coroutine: `LoadFile()`\*\*  - \*\*Description\*\*: This coroutine asynchronously loads the image file from disk using `UnityWebRequestTexture`.  - \*\*Functionality\*\*:  - Constructs a `UnityWebRequest` to load the image file located at `filePath`.  - Waits for the web request to complete (`request.SendWebRequest()`).  - If successful, retrieves the texture from the response and creates a sprite from it, which is then assigned to the `Image` component.  - If the web request fails, it logs an error with the failure reason.  - \*\*Usage\*\*: This coroutine ensures that the image is loaded asynchronously from disk, which prevents the application from freezing or experiencing performance drops while the image is being loaded. It uses `UnityWebRequestTexture` to handle file loading, which is efficient and non-blocking.  ---  ### \*\*Additional Functionalities\*\*  ---  ### \*\*Event Handling\*\*  - \*\*Language Updates\*\*:  - The script listens for language changes via the `LanguageManagerDelegate.OnLanguageUpdate` event. When the language is updated, the script automatically updates the image to reflect the selected language.    - \*\*Error Handling\*\*:  - The script includes several error-checking steps:  - If the `Image` component is not assigned, it logs an error to remind the user to assign it.  - If the image file cannot be found, it logs an error indicating the file path where the missing image was expected.  - If the web request fails, it logs the error returned by the `UnityWebRequest`.  ### \*\*Asynchronous Image Loading\*\*  - \*\*UnityWebRequest\*\*:  - The script uses `UnityWebRequest` to load the image file asynchronously. This ensures that the file loading does not block the main thread, allowing the application to remain responsive even during file loading.  - \*\*Sprite Creation\*\*:  - Once the image file is loaded, a new `Sprite` is created from the loaded texture using `Sprite.Create()`, and the sprite is applied to the `Image` component.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageImage` script dynamically updates a UI `Image` component in Unity based on the current language settings. It:  1. \*\*Loads images based on language\*\*: It reads the user’s language settings and loads the corresponding image for that language from the file system.  2. \*\*Handles asynchronous file loading\*\*: It loads images from disk asynchronously using a coroutine, ensuring the application remains responsive during the loading process.  3. \*\*Listens for language updates\*\*: The script automatically updates the image when the language changes, ensuring the UI is always in sync with the selected language.  4. \*\*Error handling\*\*: Provides error feedback for missing components, files, or failed network requests.  This script is ideal for projects that need to dynamically update images in the UI based on the current language, such as for localized branding or specific images tied to different regions or languages. |

|  |
| --- |
| **Language Raw Image** |
| This script, `LanguageRawImage`, dynamically loads and assigns localized textures to a `RawImage` UI component based on the current language setting. It retrieves the appropriate image file path from language settings and loads the texture asynchronously using `UnityWebRequest`. If the specified image file is missing or the loading process fails, an error is logged.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`RawImage rawImage`\*\*:  - \*\*Description\*\*: A reference to the `RawImage` component that will display the localized image.  - \*\*Usage\*\*: This is the target UI component that the script updates by loading and assigning the correct localized texture.  - \*\*`string fileName`\*\*:  - \*\*Description\*\*: The name of the image file to be loaded.  - \*\*Usage\*\*: This variable is used to construct the full path to the image file based on the selected language.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: The path to the currently selected language file.  - \*\*Usage\*\*: The script uses this file path to identify the language and retrieve the corresponding image.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the file that contains the saved user preferences, including the selected language.  - \*\*Usage\*\*: This file is read to load the user’s saved language preferences, ensuring the correct image is selected based on the language setting.  - \*\*`string filePath`\*\*:  - \*\*Description\*\*: The full path to the image file that needs to be loaded.  - \*\*Usage\*\*: This path is constructed dynamically based on the selected language and `fileName`, and is used to load the appropriate image.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event when the object is enabled and immediately updates the image by calling `LanguageUpdate()`.  - \*\*Functionality\*\*:  - Subscribes to the `LanguageManagerDelegate.OnLanguageUpdate` event so the `LanguageUpdate()` method is triggered when the language changes.  - Calls `LanguageUpdate()` immediately to ensure the correct image is loaded when the component is enabled.  - \*\*Usage\*\*: This ensures the image is updated whenever the script is first enabled or when the global language changes.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Removes the subscription to the `LanguageUpdate` event to stop unnecessary updates when the component is disabled.  - \*\*Usage\*\*: This method ensures that no updates are triggered when the component is inactive, preventing unnecessary processing.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: The main method that handles loading the correct image based on the selected language.  - \*\*Functionality\*\*:  - Checks if the `RawImage` component is assigned. If not, it logs an error and stops further processing.  - Loads the save file path using `LanguageFileManager.GetSaveFilePath()` and checks if a saved language exists.  - If the save file exists, it reads the selected language from the file. If not, it loads the default language file.  - Constructs the full file path to the image using the selected language and the specified `fileName`.  - If the image file exists at the constructed file path, it starts the `LoadFile()` coroutine to load the image asynchronously. If not, it logs an error indicating the file was not found.  - \*\*Usage\*\*: This method ensures the correct image is displayed by checking the selected language and dynamically loading the corresponding image file.  ---  ### 5. \*\*Coroutine: `LoadFile()`\*\*  - \*\*Description\*\*: Asynchronously loads the image file from disk using `UnityWebRequest`.  - \*\*Functionality\*\*:  - Creates a `UnityWebRequest` to load the image file as a texture using `UnityWebRequestTexture.GetTexture()`.  - Sends the request and waits for the web request to complete.  - If the request is successful, it retrieves the texture from the response and assigns it to the `RawImage` component.  - If the request fails, it logs an error with the failure message.  - \*\*Usage\*\*: This coroutine ensures the image is loaded from disk in an asynchronous, non-blocking manner, allowing the application to remain responsive while the image is being loaded.  ---  ### \*\*Additional Functionalities\*\*  ---  ### \*\*Event Handling\*\*  - \*\*Language Update\*\*:  - The script listens for the `LanguageManagerDelegate.OnLanguageUpdate` event, ensuring that whenever the language is changed, the image is updated to reflect the current language setting.  ### \*\*Asynchronous Image Loading\*\*  - \*\*UnityWebRequest\*\*:  - The script uses `UnityWebRequest` to asynchronously load the image file. This prevents blocking the main thread during file loading, allowing the application to remain responsive.  - \*\*Error Handling\*\*:  - If the `RawImage` component is missing or the image file cannot be found, the script logs errors to notify the developer of the issue.  - If the web request to load the image fails, the script logs the error message returned by `UnityWebRequest`.  ### \*\*File Path Construction\*\*  - \*\*Dynamic File Paths\*\*:  - The script dynamically constructs the file path to the image based on the selected language and the `fileName`. This allows for easy localization, where different images can be displayed depending on the active language setting.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageRawImage` script is designed to dynamically load and assign localized textures to a `RawImage` UI component in Unity based on the current language settings. It:  1. \*\*Loads localized textures\*\*: Based on the current language, it retrieves the appropriate image file path and loads the texture asynchronously.  2. \*\*Handles asynchronous loading\*\*: The script uses `UnityWebRequest` to load images asynchronously, ensuring that the main thread is not blocked while loading.  3. \*\*Responds to language updates\*\*: It listens for global language changes and updates the `RawImage` component to display the correct texture based on the new language.  4. \*\*Error logging\*\*: It logs errors if the `RawImage` component is not assigned, if the image file is missing, or if the web request fails.  This script is ideal for applications that require localized textures or images in the UI, where the content changes based on the user's language preferences. It provides efficient, real-time updates to the UI based on language changes while ensuring non-blocking, responsive performance. |

|  |
| --- |
| **Language Text Mesh / Language Text Mesh TMP** |
| This script, `LanguageTextMesh`, is designed to manage the translation and display of localized text for legacy `TextMesh` objects in Unity. It dynamically updates the `TextMesh` based on the current language settings, ensuring that localized text is rendered appropriately. The script also handles font and size changes, making it versatile for multilingual applications.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`TextMesh text`\*\*:  - \*\*Description\*\*: A reference to the `TextMesh` component that displays the localized text.  - \*\*Usage\*\*: This is the target 3D text object that the script updates with localized content. The text, font, and size of this `TextMesh` are changed based on language settings.  - \*\*`int iD`\*\*:  - \*\*Description\*\*: The ID used to identify and retrieve the corresponding localized text from the language files.  - \*\*Usage\*\*: This unique ID links the `TextMesh` object to a specific line in the language files, allowing the script to fetch the appropriate text for rendering.  - \*\*`bool translateText`\*\*:  - \*\*Description\*\*: A flag indicating whether the `TextMesh` should be translated or not.  - \*\*Usage\*\*: If set to `true`, the script will update the `TextMesh` with the localized text. If `false`, the translation is skipped.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Stores the path of the selected language file.  - \*\*Usage\*\*: This variable holds the current language file path, which is used to retrieve localized text and settings.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the save file where user language preferences are stored.  - \*\*Usage\*\*: This file is checked to determine which language is selected and which language file to load for localization.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event when the object is enabled, ensuring that the `TextMesh` is updated whenever the language changes.  - \*\*Functionality\*\*:  - Registers the `LanguageUpdate()` method to be called whenever a language update occurs.  - Immediately calls `LanguageUpdate()` to apply the correct localized text when the object becomes active.  - \*\*Usage\*\*: This method ensures that the `TextMesh` content is kept up-to-date with the current language settings, even if the language changes dynamically during runtime.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Removes the event subscription to prevent the script from continuing to update the `TextMesh` while the object is disabled.  - \*\*Usage\*\*: This method ensures that no unnecessary processing occurs when the `TextMesh` is inactive, improving performance.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: The main method that handles updating the `TextMesh` based on the current language.  - \*\*Functionality\*\*:  - Retrieves the save file path using `LanguageFileManager.GetSaveFilePath()`.  - Uses the `iD` to fetch the corresponding localized text from the selected language file using `LanguageFileManager.GetLocalizedLineByID()`.  - Passes the fetched text to `ProcessLine()` for further processing and display.  - \*\*Usage\*\*: This method is responsible for ensuring that the `TextMesh` displays the correct localized text, based on the user’s language settings. It serves as the core update function.  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: Processes the localized text retrieved from the language file and applies it to the `TextMesh` component.  - \*\*Functionality\*\*:  - If `translateText` is enabled, it updates the `text.text` with the localized content extracted from the language file using `LanguageFileManager.ExtractTextBetweenBraces()`.  - Extracts additional information like font size and font type from the localized line using `LanguageFileManager.ExtractIntValue()`.  - If valid values for font size and font index are found, it updates the `TextMesh`’s font size and font using `FontAndAlignmentUtility.GetFontByIndex()`.  - \*\*Usage\*\*: This method ensures that the localized text, font, and size are applied to the `TextMesh` component. It is responsible for formatting and rendering the text based on the selected language.  ---  ### \*\*Custom Editor (`LanguageTextMeshEditor`)\*\*  ---  The custom editor extends Unity’s Inspector for the `LanguageTextMesh` component, providing additional functionality and visual enhancements for working with the script in the Unity Editor.  - \*\*Method: `OnInspectorGUI()`\*\*:  - \*\*Description\*\*: Customizes the Unity Inspector, adding buttons and warnings for user convenience.    - \*\*Key Features\*\*:  - \*\*Import Settings Button\*\*: Allows users to import the current text and font settings from the `TextMesh` object into the language files, making it easy to store and edit settings.  - \*\*ID Warnings\*\*: Displays a warning if the `iD` is already saved in the language list, preventing conflicts with existing language entries.  - \*\*TextMesh Reference Check\*\*: Highlights the `TextMesh` field in red if it is not assigned, ensuring that the user knows the reference is missing.    - \*\*Usage\*\*: The custom editor simplifies the workflow for configuring the `LanguageTextMesh` component. It provides easy access to import settings, helps manage localization IDs, and ensures the `TextMesh` reference is correctly assigned.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageTextMesh` script handles the localization and dynamic rendering of text for legacy 3D `TextMesh` components in Unity. It:  1. \*\*Translates and updates text\*\*: It fetches localized text based on an `iD` and updates the `TextMesh` component with the appropriate translation.  2. \*\*Handles font and size changes\*\*: Along with the text, the script can also update the font type and size dynamically based on the language settings.  3. \*\*Responds to language changes\*\*: Listens for global language updates and adjusts the text content, font, and size accordingly.  4. \*\*Custom Editor\*\*: Includes a custom Inspector in Unity, which simplifies the management of the `TextMesh` localization settings and provides useful warnings and tools for developers.  This script is ideal for applications that need to localize 3D text objects (using `TextMesh`) and dynamically update their content and appearance based on language settings, making it a valuable tool for multilingual Unity projects. |

|  |
| --- |
| **Language Create File** |
| The `LanguageCreateFile` script is responsible for generating language files with predefined lines and handling language updates in Unity, supporting multilingual functionality. It integrates with the language manager system, creating language files that can be localized based on user preferences or default settings. The script works both in the Unity Editor and in builds, allowing the creation of language files in different directories.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`string folderInUnity`\*\*:  - \*\*Description\*\*: Specifies the folder path where the language file will be created when running in the Unity Editor.  - \*\*Usage\*\*: Used to store the output file during development, ensuring it's easily accessible within the Unity project’s structure.  - \*\*`string folderInBuild`\*\*:  - \*\*Description\*\*: Specifies the folder path where the language file will be created in builds (runtime).  - \*\*Usage\*\*: Used to store the output file when the application is built and deployed, typically within the `StreamingAssets` folder for access at runtime.  - \*\*`string fileName`\*\*:  - \*\*Description\*\*: The name of the file that will be created.  - \*\*Usage\*\*: This is combined with the `fileExtension` to form the full file name when writing the language file to disk.  - \*\*`string fileExtension`\*\*:  - \*\*Description\*\*: The extension of the file to be created (e.g., `.txt`).  - \*\*Usage\*\*: Combined with the `fileName` to specify the type of file being written, typically a text file for language files.  - \*\*`List<LanguageLines> fileLines`\*\*:  - \*\*Description\*\*: A list of language lines, where each entry contains text, an ID, and a flag for translation.  - \*\*Usage\*\*: This list is processed and written to the language file. Each line can be marked for translation or simply used as-is.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Holds the path of the currently selected language file.  - \*\*Usage\*\*: This is used to retrieve the appropriate language file for processing and translation.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the save file that stores user preferences, such as the selected language.  - \*\*Usage\*\*: The save file is read to determine the user's selected language, which influences the creation of the language file.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event, which ensures that the script updates whenever the language settings change.  - \*\*Functionality\*\*:  - Registers the `LanguageUpdate()` method to be called when the language manager triggers an update.  - Immediately calls `LanguageUpdate()` to ensure the language file is created or updated when the component is enabled.  - \*\*Usage\*\*: This ensures that the language file is dynamically updated based on the current language settings whenever the component becomes active.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the object is disabled.  - \*\*Functionality\*\*:  - Removes the event listener to stop further updates when the component is disabled.  - \*\*Usage\*\*: This method ensures that unnecessary processing does not occur when the object is inactive, improving performance.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: The core method that handles updating the language file based on the current language settings.  - \*\*Functionality\*\*:  - Retrieves the save file path using `LanguageFileManager.GetSaveFilePath()`.  - If the save file exists, it reads the selected language file from it. If not, it defaults to the standard language file.  - Calls `ProcessLine()` to handle the text lines for the language file.  - Finally, it calls `CreateFile()` to write the processed lines to a new file.  - \*\*Usage\*\*: This method is responsible for generating or updating the language file whenever the language settings are modified. It handles both reading from and writing to files.  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: Processes each line in the `fileLines` list, updating the text with its localized version if translation is enabled.  - \*\*Functionality\*\*:  - Iterates over the `fileLines` list.  - For each line where translation is enabled, it retrieves the localized text using `LanguageFileManager.FindLineByID()` and updates the line’s text.  - \*\*Usage\*\*: This method ensures that each line in the language file is correctly translated before it is written to the output file. It handles the translation process by matching IDs and extracting the appropriate localized text.  ---  ### 6. \*\*Method: `GetFolderPath()`\*\*  - \*\*Description\*\*: Determines the folder path for where the language file should be saved, depending on whether the application is running in the Unity Editor or in a build.  - \*\*Functionality\*\*:  - If running in the Unity Editor, it returns a path based on `folderInUnity`.  - If running in a build, it returns a path based on `folderInBuild`.  - \*\*Usage\*\*: This method ensures that the language file is saved in the appropriate directory based on the runtime environment (Editor vs Build).  ---  ### 7. \*\*Method: `CreateFile()`\*\*  - \*\*Description\*\*: Writes the processed language lines to a file at the specified location.  - \*\*Functionality\*\*:  - Calls `GetFolderPath()` to determine the correct directory for the file.  - Ensures the directory exists by creating it if necessary.  - Constructs the full file path by combining the folder path, file name, and extension.  - Writes each processed line from `fileLines` to the file.  - \*\*Usage\*\*: This method creates the actual language file that will be used in the application. It writes the processed content to disk, ensuring the file is available for localization purposes.  ---  ### \*\*Custom Editor (`LanguageCreateFileEditor`)\*\*  ---  The custom editor provides a user-friendly interface for managing the `LanguageCreateFile` component within the Unity Editor. It offers features such as:  - \*\*Import Settings Button\*\*: Imports settings from the language manager for the lines in `fileLines`. It checks if any lines' IDs are already saved and prompts the user to confirm replacement.  - \*\*File Preview\*\*: Displays a preview of the generated language file content within the Unity Editor. This helps developers see the result before actually creating the file.  - \*\*Usage\*\*: The custom editor simplifies the workflow by providing a visual interface to manage language lines, ensuring that the user can interact with and preview the localization data before generating the final file.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageCreateFile` script generates language files based on the current language settings and predefined text lines. It:  1. \*\*Creates language files\*\*: Dynamically generates language files with localized content, writing the processed lines to disk.  2. \*\*Responds to language updates\*\*: Listens for changes in language settings and updates the language file accordingly.  3. \*\*Supports both Editor and Build environments\*\*: The file paths adapt depending on whether the application is running in the Unity Editor or a built version of the game.  4. \*\*Custom Editor integration\*\*: Provides an intuitive interface for managing language file creation directly within the Unity Editor, offering tools like line previews and import functionality.  This script is ideal for projects that require dynamic generation of language files, allowing developers to create and manage language content efficiently within Unity’s ecosystem. |
| **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** |
| The `LanguageScript` component manages the loading and updating of localized text for language scripts in a Unity scene. It supports multilingual functionality by fetching localized text from language files and applying it to target scripts or components using `UnityEvent`. This script dynamically adjusts text based on the current language settings, ensuring the scene's content is properly localized.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`List<ScriptText> scriptTexts`\*\*:  - \*\*Description\*\*: A list of `ScriptText` objects, each containing localized text, its associated ID, and an event that can be invoked when the text is updated.  - \*\*Usage\*\*: These `ScriptText` objects store the text content that will be localized. Each entry is linked to a specific language line in the language files and is updated with localized text when the language is changed.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: The path to the selected language file used for retrieving localized text.  - \*\*Usage\*\*: This file is determined based on user preferences or system settings. It stores the language-specific text and settings.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the save file where the selected language and preferences are stored.  - \*\*Usage\*\*: This file holds the user’s language preferences and is read to determine which language file should be used for localization.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event, ensuring that the script updates the localized text whenever the language changes.    - \*\*Functionality\*\*:  - Registers the `LanguageUpdate()` method to be invoked when the language is updated globally.  - Immediately calls `LanguageUpdate()` to update the text when the component is enabled.  - \*\*Usage\*\*: This ensures that the `LanguageScript` is initialized with the current language and remains updated whenever the language settings change.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event to stop the script from responding to language updates when it is disabled.  - \*\*Usage\*\*: This improves performance by ensuring that the script no longer processes updates when it is inactive.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: Handles updating the localized text for all script texts based on the current language settings.  - \*\*Functionality\*\*:  - Retrieves the save file path using `LanguageFileManager.GetSaveFilePath()`.  - If the save file exists, it reads the selected language file path from the save file. If no save file is found, it defaults to the system’s language settings.  - Calls `ProcessLine()` to update the text for all `ScriptText` entries in the `scriptTexts` list.  - \*\*Usage\*\*: This method is responsible for loading the appropriate language file and updating all text elements in the scene. It ensures that the text displayed matches the currently selected language.  ---  ### 5. \*\*Method: `ProcessLine()`\*\*  - \*\*Description\*\*: Processes each `ScriptText` object in the `scriptTexts` list, fetching the localized text and applying it to the target scripts via `UnityEvent`.  - \*\*Functionality\*\*:  - Iterates through each `ScriptText` entry in the list.  - For each entry, it retrieves the localized text from the selected language file using `LanguageFileManager.FindLineByID()` based on the associated ID.  - Extracts the localized text inside curly braces using `LanguageFileManager.ExtractTextBetweenBraces()`.  - Invokes the `UnityEvent` connected to the `ScriptText`, passing the localized text as a parameter.  - \*\*Usage\*\*: This method ensures that the correct localized text is applied to each script or UI element connected to the `UnityEvent`. It dynamically updates the content based on the current language settings.  ---  ### 6. \*\*Method: `ApplyUnityEvent()`\*\*  - \*\*Description\*\*: Invokes the `UnityEvent` associated with a `ScriptText` object, passing the localized text as an argument.  - \*\*Functionality\*\*:  - Retrieves the persistent calls (methods) registered with the `UnityEvent`.  - Loops through each method, checking if it accepts a single `string` parameter, and then invokes the method, passing the localized text.    - \*\*Usage\*\*: This method applies the localized text to the relevant script or UI component by invoking the `UnityEvent`. It allows the localized text to be processed by external components or scripts.  ---  ### \*\*Custom Editor (`LanguageScriptEditor`)\*\*  ---  The custom editor provides a more intuitive user interface in the Unity Editor for managing the `LanguageScript` component.  - \*\*Method: `OnInspectorGUI()`\*\*:  - \*\*Description\*\*: Overrides the default Inspector GUI, adding custom buttons and fields for easier management of the `LanguageScript` component.    - \*\*Key Features\*\*:  - \*\*Import Settings Button\*\*: Imports settings for each `ScriptText` in the `scriptTexts` list, allowing developers to easily configure the text content and its ID for localization.  - \*\*Warnings\*\*: Displays a warning if any of the `ScriptText` IDs are already saved, preventing conflicts with existing localized content.  - \*\*Property Fields\*\*: Provides fields for editing `scriptTexts`, and viewing the selected language file and save file paths.  - \*\*Usage\*\*: This custom editor improves the workflow for developers working with the `LanguageScript` component, offering tools to easily manage the localized text and ensure there are no conflicts with existing content.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageScript` component facilitates the localization of script texts in Unity by:  1. \*\*Dynamically loading localized text\*\*: Based on the selected language, the script retrieves the appropriate localized text and applies it to various elements in the scene.  2. \*\*Supporting multilingual updates\*\*: It listens for global language changes and automatically updates the relevant text components to reflect the new language settings.  3. \*\*Connecting with external components\*\*: Through the use of `UnityEvent`, it allows external scripts or UI elements to receive and display the localized text, making the system highly flexible for different use cases.  4. \*\*Custom Editor\*\*: A custom Inspector provides a streamlined interface for managing the script’s settings and localized text, helping developers to configure and maintain the localization process more easily.  This script is ideal for projects that require dynamic localization in Unity, especially for managing multilingual content across various scripts or components connected via `UnityEvent`. |

|  |
| --- |
| **Language Audio Player** |
| The `LanguageAudioPlayer` script is designed to play localized audio files based on the current language settings in a Unity project. It dynamically loads audio files from language-specific folders and plays them using the `AudioSource` component. This setup ensures that the appropriate audio clip corresponding to the selected language is played for the user.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`AudioSource audioSource`\*\*:  - \*\*Description\*\*: The `AudioSource` component responsible for playing the loaded audio clip.  - \*\*Usage\*\*: The audio clip loaded based on the language settings will be played through this component.  - \*\*`string fileName`\*\*:  - \*\*Description\*\*: The name of the audio file that will be played.  - \*\*Usage\*\*: This is the specific audio file to be played, located within a language-specific folder.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: Stores the path to the currently selected language file.  - \*\*Usage\*\*: This file is determined from user preferences or system defaults and is used to find language-specific resources like audio files.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The path to the save file that stores the user's language preferences.  - \*\*Usage\*\*: The script reads this file to determine the language-specific audio file to load.  - \*\*`string filePath`\*\*:  - \*\*Description\*\*: The full path of the localized audio file to be played.  - \*\*Usage\*\*: The script constructs this path based on the selected language and the `fileName`.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event to trigger updates when the language changes. It also immediately triggers `LanguageUpdate()` to load and play the correct audio when the component is enabled.  - \*\*Usage\*\*: This ensures that the script is always up-to-date with the latest language settings and plays the correct audio based on user preferences.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event to prevent the script from processing unnecessary updates when disabled.  - \*\*Usage\*\*: This method optimizes performance by stopping the script from processing language changes when it’s inactive.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: This method updates the language settings and attempts to load the appropriate audio file based on the selected language.  - \*\*Functionality\*\*:  - Retrieves the path to the save file using `LanguageFileManager.GetSaveFilePath()`.  - If a save file exists, it reads the selected language file from the save file.  - If no save file exists, it defaults to the system's language settings by loading the default language file.  - Constructs the path to the audio file in the language-specific folder.  - Starts loading the audio file asynchronously by calling the `LoadFile()` coroutine.  - \*\*Usage\*\*: This method ensures that the correct audio file is loaded and prepared for playback according to the current language settings. It triggers an update whenever the language changes or when the component is activated.  ---  ### 5. \*\*Method: `LoadFile()`\*\*  - \*\*Description\*\*: A coroutine that loads the audio file asynchronously and plays it through the `AudioSource` component.  - \*\*Functionality\*\*:  - Creates a `UnityWebRequest` to load the audio file from the file system based on the constructed `filePath`.  - Waits for the audio file to load asynchronously.  - If the audio file is successfully loaded, it sets the `AudioSource`'s clip to the loaded audio and starts playback.  - If the request fails, it logs an error.  - \*\*Usage\*\*: This coroutine ensures that the audio file is loaded without interrupting the main thread, allowing for smooth performance even when dealing with large files. The loaded audio is automatically played when the file is ready.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageAudioPlayer` component is used to:  1. \*\*Dynamically Load Localized Audio\*\*: It automatically selects and loads the appropriate audio file based on the current language settings, ensuring that users hear the correct language.    2. \*\*Asynchronous File Loading\*\*: The script asynchronously loads the audio file from disk, ensuring that the user interface remains responsive while the audio is being prepared.  3. \*\*Automatic Playback\*\*: Once the file is loaded, it is automatically played via the assigned `AudioSource` component.  4. \*\*Support for Multilingual Applications\*\*: This component allows developers to easily manage and play localized audio files without manually adjusting the audio file paths for different languages.  This script is ideal for Unity projects that require localized audio, such as voiceovers, sound effects, or ambient sounds that change based on the user’s language preferences. |

|  |
| --- |
| **Language Canvas** |
| This script, `LanguageCanvas`, manages language-specific settings for a Unity `Canvas`. It updates the `Canvas` and its child elements' properties (like position, scale, etc.) based on the selected language configuration. It also handles saving and loading these configurations, ensuring the user interface adapts according to the chosen language.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`int canvasID`\*\*:  - \*\*Description\*\*: A unique identifier for the `Canvas` object.  - \*\*Usage\*\*: This ID is used to reference the canvas data, enabling the script to locate and load the appropriate settings for this particular canvas.  - \*\*`string selectedFile`\*\*:  - \*\*Description\*\*: The path to the file storing the current language configuration.  - \*\*Usage\*\*: The selected language file is used to load the appropriate settings for the canvas and its child elements.  - \*\*`string saveFile`\*\*:  - \*\*Description\*\*: The file path where the canvas's state is saved.  - \*\*Usage\*\*: The file stores canvas-specific information, which can be reloaded later to restore the UI configuration.  - \*\*`CanvasData canvasData`\*\*:  - \*\*Description\*\*: An object storing the primary data for the `Canvas`, including position, scale, rotation, and UI scaling information.  - \*\*Usage\*\*: It holds key properties of the main canvas, like the `RectTransform` settings and the `CanvasScaler` properties.  - \*\*`List<CanvasDataList> savedCanvasData`\*\*:  - \*\*Description\*\*: A list of data representing the child elements of the `Canvas`.  - \*\*Usage\*\*: This list holds information about the child objects of the canvas, such as their position, scale, and anchor settings.  ---  ### 2. \*\*Method: `OnEnable()`\*\*  - \*\*Description\*\*: Subscribes to the `OnLanguageUpdate` event, which triggers the `LanguageUpdate()` method whenever the language settings change. It also performs an initial update to apply the correct language settings when the canvas is enabled.  - \*\*Usage\*\*: This method ensures that the canvas is updated according to the selected language whenever the component is enabled or when the language is updated.  ---  ### 3. \*\*Method: `OnDisable()`\*\*  - \*\*Description\*\*: Unsubscribes from the `OnLanguageUpdate` event when the component is disabled, preventing unnecessary updates.  - \*\*Usage\*\*: Ensures that the canvas is not updated when it's inactive, optimizing performance by preventing unnecessary language updates when the component is not in use.  ---  ### 4. \*\*Method: `LanguageUpdate()`\*\*  - \*\*Description\*\*: This method is triggered when the language changes or when the script is enabled. It loads the canvas settings from the saved file and applies the appropriate configurations based on the current language settings.  - \*\*Functionality\*\*:  - Retrieves the path of the save file.  - Loads the localized JSON data corresponding to the `canvasID` from the selected language file.  - Calls `LoadAllCanvasDataList()` to apply the canvas settings.  - \*\*Usage\*\*: This method ensures that the canvas and its child elements are updated with the correct language-specific settings whenever the language changes or the component is initialized.  ---  ### 5. \*\*Method: `LoadAllCanvasDataList(string json)`\*\*  - \*\*Description\*\*: Loads the canvas data from the given JSON string and applies the properties (like position, rotation, scale) to the `RectTransform` of the canvas and its child elements.  - \*\*Functionality\*\*:  - Deserializes the JSON data into a `CanvasDataSave` object.  - Updates the `RectTransform` settings for the main canvas and its child elements.  - If the number of elements in the saved data does not match the current elements, it logs a warning.  - Applies the saved properties (position, rotation, scale, anchors, size) to the child elements.  - \*\*Usage\*\*: This method ensures that the UI elements within the canvas adapt to the language-specific settings by updating their `RectTransform` properties.  ---  ### 6. \*\*Method: `SaveCanvasHierarchy()` (Editor Only)\*\*  - \*\*Description\*\*: This method records the current state of the canvas and its child objects, saving their properties (position, scale, rotation, etc.) in a structured format that can be serialized into a JSON string.  - \*\*Functionality\*\*:  - Clears the existing saved data.  - Iterates through all child objects of the canvas, recording their `RectTransform` properties and adding them to the `savedCanvasData` list.  - Records the `CanvasScaler` settings for the main canvas.  - Returns the saved data as a JSON string.  - \*\*Usage\*\*: This method is used to capture the current state of the canvas and its child elements, saving this information for future use. It is particularly useful in the Unity Editor for saving the canvas configuration after it has been fully set up.  ---  ### \*\*Overall Purpose and Usage\*\*  The `LanguageCanvas` script is designed to:  1. \*\*Adapt the UI to Language-Specific Settings\*\*: It allows the canvas and its child elements to be updated based on the selected language, ensuring the user interface fits the cultural and linguistic preferences of the user.  2. \*\*Save and Load Canvas Configuration\*\*: The script saves and loads data about the position, scale, rotation, and other properties of UI elements, allowing developers to configure the canvas once and reuse those configurations based on the current language.  3. \*\*Real-Time Language Updates\*\*: By subscribing to the `OnLanguageUpdate` event, the script ensures that the UI is updated in real-time whenever the language settings are changed by the user.  4. \*\*Editor Functionality\*\*: The custom editor interface allows for importing and saving canvas settings during development, streamlining the workflow for handling multilingual UI setups.  This script is especially useful in multilingual projects where the UI layout needs to dynamically adjust based on the language settings. It ensures a consistent experience across languages while maintaining the flexibility to modify and save UI configurations. |

|  |
| --- |
| **Rebuild Canvas** |
| The `RebuildCanvas` script is a Unity editor tool designed to help developers manage and create canvas hierarchies using JSON data for multilingual or language-based user interfaces. It integrates with Unity's undo system and provides an interface to create, save, and load canvas configurations based on language settings.  ### \*\*Key Components and Methods\*\*  ---  ### 1. \*\*Variables\*\*  - \*\*`GameObject canvasGameObject`\*\*:  - \*\*Description\*\*: The reference to the main canvas GameObject.  - \*\*Usage\*\*: This object represents the canvas that will be manipulated or created when loading or saving canvas data. It serves as the container for the UI hierarchy.  - \*\*`CanvasData canvasData`\*\*:  - \*\*Description\*\*: Holds various properties related to the canvas, such as position, scale, and UI scaling settings.  - \*\*Usage\*\*: This object stores the settings for the main canvas, including `RectTransform` properties and scaling settings for different screen resolutions.  - \*\*`List<CanvasDataList> savedCanvasData`\*\*:  - \*\*Description\*\*: A list that holds information about each UI element under the main canvas.  - \*\*Usage\*\*: This list contains the properties (such as `RectTransform` settings) of all child objects of the canvas, allowing the system to rebuild the hierarchy and restore the positions and properties of the UI elements.  - \*\*`string saveFilePath`\*\*:  - \*\*Description\*\*: The path where canvas configuration files are saved.  - \*\*Usage\*\*: This path is used to store or load canvas-related data. It's generally set to the language files folder.  - \*\*`int selectedCanvasIndex`\*\*:  - \*\*Description\*\*: Holds the index of the currently selected canvas from a dropdown in the editor interface.  - \*\*Usage\*\*: This index is used to select which canvas configuration to load or modify from the list of saved canvas data.  - \*\*`List<CanvasSave> canvasSave`\*\*:  - \*\*Description\*\*: A list of saved canvas configurations.  - \*\*Usage\*\*: This holds all the canvas configurations read from the language files, allowing users to choose which one to load and manipulate.  ---  ### 2. \*\*Method: `CreateCanvasHierarchy(string json)`\*\*  - \*\*Description\*\*: This method takes a JSON string that contains canvas configuration data and creates a `Canvas` hierarchy in the scene based on that data.    - \*\*Functionality\*\*:  - It first parses the JSON string into `CanvasDataSave` objects (which contain both canvas and child element data).  - If a canvas already exists in the scene, it destroys the existing one before creating a new canvas.  - It creates a new `Canvas` GameObject with the necessary components (`RectTransform`, `Canvas`, `CanvasScaler`, etc.) and configures them based on the data in the JSON.  - It then recursively creates the child objects for the canvas, restoring their `RectTransform` properties and appearance (e.g., color, size).  - \*\*Usage\*\*: This method is used when you want to load a canvas configuration from a language file and build the entire UI hierarchy, complete with the proper positions, sizes, and other properties.  ---  ### 3. \*\*Method: `CreateGameObjectFromHierarchy(string[] hierarchy, Transform parent)`\*\*  - \*\*Description\*\*: This method is responsible for creating the actual GameObjects that make up the UI hierarchy of the canvas.    - \*\*Functionality\*\*:  - It iterates through a hierarchy array, where each string represents a path to a GameObject in the hierarchy (e.g., `"Canvas/Button1"`).  - For each GameObject path, it splits the path into its components (like `"Canvas"`, `"Button1"`) and either finds an existing child object or creates a new one.  - Each created GameObject is assigned its `RectTransform` properties and visual appearance, such as an `Image` and an `Outline` component with randomized colors for depth levels.    - \*\*Usage\*\*: This method is used as part of the process to recreate the full UI hierarchy when loading a canvas configuration. It ensures that all child objects are restored with their correct properties.  ---  ### 4. \*\*Method: `SaveCanvasHierarchy()`\*\*  - \*\*Description\*\*: Saves the current state of the canvas and its child objects, including `RectTransform` properties, to a JSON string.  - \*\*Functionality\*\*:  - It iterates through the list of saved canvas data (`savedCanvasData`) and updates each object's properties based on the current state of the UI elements in the scene.  - It collects all this data into a `CanvasDataSave` object, which is then serialized into a JSON string.    - \*\*Usage\*\*: This method allows users to save the current canvas hierarchy and its properties to a file, which can later be loaded and applied to restore the UI configuration. It’s commonly used after the canvas has been manually modified or designed in the Unity editor.  ---  ### 5. \*\*Editor Interface\*\*  - \*\*Description\*\*: The script includes a custom editor (`RebuildCanvasEditor`) that provides buttons and controls in the Unity editor for working with the canvas hierarchy.    - \*\*Functionality\*\*:  - It includes buttons like "Load Language File", "Create Canvas Hierarchy", and "Replace canvasID", which allow users to interactively load, create, or modify canvas hierarchies.  - It also shows dropdowns to select between different canvases and provides warnings if no valid canvas is selected.    - \*\*Usage\*\*: This editor script simplifies the process of loading and saving canvas hierarchies by offering a user-friendly interface directly within the Unity editor. Developers can quickly load canvas configurations from language files, create or replace existing canvases, and preview changes without writing additional code.  ---  ### \*\*Overall Purpose and Usage\*\*  - The \*\*`RebuildCanvas`\*\* script is primarily used to \*\*load and manage canvas configurations\*\* in a Unity scene. These configurations are often tied to \*\*language settings\*\*, making this tool essential for handling multilingual user interfaces.  - It allows developers to \*\*save and restore UI layouts\*\* dynamically, ensuring that different language configurations can be applied easily during development.  - The \*\*custom editor interface\*\* streamlines the workflow, allowing developers to manage multiple canvas configurations efficiently and test different setups directly in the Unity editor.  This tool is especially useful in projects where the UI layout needs to adjust based on different languages or localization settings. It automates much of the process of saving and loading complex canvas hierarchies. |

|  |
| --- |
| **Language File Manager Window** |
| The script `LanguageFileManagerWindow` is an editor tool for Unity that provides a custom window for managing language-related components and canvas elements in the game. It enables users to manage language files, save and load settings, organize IDs, and handle UI components for localization purposes. Here's an explanation of the key elements, variables, methods, and their functionality:  ### Key Variables  1. \*\*`fileName`\*\*: Stores the name of the current language file being worked on (e.g., "ENGLISH").  2. \*\*`computerLanguage`\*\*: Stores the currently selected language on the computer.  3. \*\*`componentSave`\*\*: A list holding language components (UI text, alignment, font sizes) that will be saved to or loaded from a file.  4. \*\*`canvasSave`\*\*: A list containing canvas elements (UI layout) that will also be saved or loaded.  5. \*\*`scrollPosition`\*\*: Keeps track of the scroll position in the editor window's UI for smooth interaction with large datasets.  6. \*\*`firstTime`\*\*: A flag to determine if the window is being opened for the first time, to initialize default values.  7. \*\*`idIndex`\*\*: Holds the ID currently selected in the GUI for navigation between components.  8. \*\*`saveFilePath`\*\*: The folder path where language files are stored.  9. \*\*`duplicateID`\*\*: A list tracking any duplicate component IDs, used to warn the user about conflicts.  ### Methods and Functionality  #### 1. \*\*OnEnable()\*\*  - This method is triggered when the editor window is opened. It subscribes to Unity's `undoRedoPerformed` event and initializes the editor with culture names and other data. The `saveFilePath` is set by fetching the folder path where language files are stored.  #### 2. \*\*OnGUI()\*\*  - This is the main method that renders the window’s graphical interface. It updates the serialized objects, handles user inputs, displays action buttons, draws fields for editing language data, and provides tools for saving and loading language files. It consists of multiple helper functions:  - \*\*`DrawActionButtons()`\*\*: Renders buttons for performing various actions such as saving files, opening folders, loading language files, and organizing items.  - \*\*`DrawLanguageDropdown()`\*\*: Displays a dropdown to allow users to select the computer's language.  - \*\*`DrawIdDisplayPanel()`\*\*: Manages ID selection, navigation, and duplicate checking.  #### 3. \*\*FindDuplicateIds()\*\*  - This method checks for duplicate IDs within `componentSave` and `canvasSave`. It highlights components with duplicate IDs and warns the user to prevent conflicts.  #### 4. \*\*SaveLanguageFile()\*\*  - Saves the current language file. It collects all component and canvas data and writes it to a `.txt` file in the specified directory. It first checks for duplicates and asks the user to confirm overwriting existing files.  #### 5. \*\*LoadLanguageFile()\*\*  - Allows users to load an existing language file from disk. It parses the file, extracts the components and canvas data, and populates the editor window with this data.  #### 6. \*\*SaveDataJson()\*\*  - Saves the current editor window state to a JSON file (`LanguageFileData.json`) so that settings (e.g., language, components, IDs) can be retained between sessions.  #### 7. \*\*LoadDataJson()\*\*  - Loads the window state from a previously saved JSON file. It restores the editor’s settings, components, and canvas list.  #### 8. \*\*AddComponent(LanguageComponentSave data)\*\* / \*\*AddCanvas(CanvasSave data)\*\*  - These methods allow adding or updating individual components and canvas elements in the `componentSave` or `canvasSave` lists. They sort the lists based on IDs to maintain order and update the editor.  #### 9. \*\*CompareID()\*\*  - This method sorts both `componentSave` and `canvasSave` lists by their IDs, ensuring that items are organized numerically in the editor.  ### Usage  - The script creates a dedicated Unity editor window that simplifies managing language localization by handling text, fonts, alignments, and other UI properties. It allows users to:  1. Load, edit, and save language files.  2. Organize language components and canvas elements.  3. Create JSON representations of canvas hierarchies.  4. Ensure that IDs are unique to avoid conflicts in UI translations.  Overall, the `LanguageFileManagerWindow` provides a powerful interface to manage multilingual UI components and canvas structures within Unity projects. |

|  |
| --- |
| **Language Excel Converter Window** |
| This script, `LanguageExcelConverterWindow`, is a Unity editor tool that converts language files into a CSV format, which can be used with Excel. The script provides an editor window that lists language files from a specified folder and allows users to convert these files into an Excel-compatible format. Here's an explanation of how it works, along with a breakdown of the main variables and methods:  ### Key Variables  1. \*\*`folderPath`\*\*: This variable holds the path to the folder containing the language files. The folder is fetched from the `LanguageFileManager`.  2. \*\*`fileNames`\*\*: A list of strings that stores the names of language files found in the folder.  3. \*\*`jsonPath`\*\*: The path where the list of language files and related data are saved as a JSON file. The script uses this to persist data between editor sessions.  4. \*\*`serializedObject`\*\*: A serialized object that tracks changes in the editor window to ensure data integrity and allow for undo/redo functionality.  5. \*\*`scrollPosition`\*\*: This variable stores the position of the scroll bar in the window to handle long lists of files.  6. \*\*`firstTime`\*\*: A boolean flag used to check if this is the first time the editor window is being opened.  7. \*\*`iDs`\*\*: A list of component IDs found in the language files. These IDs are used to align the translations between different language files.  8. \*\*`excelDataList`\*\*: A list of `ExcelData` objects that contain the text translations for each language. This is the data that will be written to the CSV file.  ### Methods and Their Purpose  #### 1. \*\*ShowEditorWindow()\*\*  - This method adds an item to the Unity editor’s "Window" menu under the "Language" category. When selected, it opens the `LanguageExcelConverterWindow`.  #### 2. \*\*OnEnable()\*\*  - Called when the editor window is first opened. This method initializes the editor by subscribing to the `Undo` event, creating a serialized object for tracking changes, loading saved data from a JSON file, and refreshing the list of language files.  #### 3. \*\*OnDestroy()\*\*  - This method is triggered when the window is closed. It unsubscribes from the undo/redo events and saves the current state of the window (including the file list) to a JSON file.  #### 4. \*\*OnUndoRedo()\*\*  - This method is called when an undo or redo action is performed in Unity. It simply repaints the window to reflect any changes made by the undo/redo action.  #### 5. \*\*OnGUI()\*\*  - The main method responsible for drawing the user interface (UI) of the editor window. This method:  1. Displays a label for the window.  2. Draws a scrollable area containing the list of language files.  3. Provides buttons for converting the files to CSV format and updating the file list.  4. Updates any changes made to the serialized data in the window.    #### 6. \*\*RefreshFileList()\*\*  - This method scans the `folderPath` for text files (`.txt`), extracts their names, and updates the `fileNames` list. If the folder path is invalid, it logs a warning message.  #### 7. \*\*ConvertToExcel()\*\*  - This is the core functionality of the script. It:  1. Extracts the component IDs from the first language file.  2. Iterates through each language file, matching IDs with the corresponding translation text.  3. Calls `ExportToCsv()` to write the data into a CSV file that can be opened with Excel.  4. Opens a save dialog so the user can choose where to save the CSV file.  5. Handles any exceptions that may occur during the process and logs errors if any.  #### 8. \*\*ExportToCsv(List<ExcelData> dataList, string[] languages, string filePath)\*\*  - This method writes the extracted language data to a CSV file. It:  1. Writes the header row with the language names.  2. Iterates through the dictionary of IDs and their translations, writing each row to the CSV.  3. Escapes semicolons in the translations by replacing them with a different character (`¦`).  #### 9. \*\*SaveDataJson()\*\*  - Saves the list of language files (`fileNames`) to a JSON file at `jsonPath`. This allows the window to remember which files were previously loaded when reopened.  #### 10. \*\*LoadDataJson()\*\*  - Loads the previously saved list of language files from the JSON file when the window is first opened. This ensures continuity between editor sessions.  ### Usage  1. \*\*Opening the Tool\*\*: The script provides a custom window accessible from the Unity editor under the "Window > Language > Language Excel Converter" menu. This opens the editor window where the user can manage language files.    2. \*\*Loading Language Files\*\*: The window scans the `folderPath` for text files containing translations and displays them in the list.  3. \*\*Converting to CSV\*\*: By pressing the "Convert to Excel" button, the user can export the language data to a CSV file, which can then be opened in Excel or similar software.  4. \*\*Managing the File List\*\*: The "Update List" button refreshes the list of files in case new files are added or removed from the folder.  In summary, this script provides an efficient workflow for managing language localization files in Unity and exporting them to Excel-compatible formats for easier translation management. It allows users to load, edit, and save localization files, and ensures that these files remain organized and accessible in both Unity and external tools like Excel. |

|  |
| --- |
| **Language Prefab Creator** |
| The `LanguagePrefabCreator` script is a static utility class designed for Unity's editor environment. It provides methods to generate and configure various prefabs related to language tools, including UI and 3D elements. The script ensures proper setup of UI elements like buttons, dropdowns, and text fields, and allows for the creation of audio sources and language-specific components like language managers and scripts. Additionally, the script registers undo actions to allow for easy modification tracking within Unity's editor.  ### Key Variables and Methods  #### \*\*`CreateUICanvas()`\*\*  This method creates a new UI Canvas GameObject if no canvas exists in the scene. It adds essential components like:  - \*\*`CanvasScaler`\*\*: Ensures the UI scales properly based on the screen resolution.  - \*\*`GraphicRaycaster`\*\*: Handles input events for UI elements.  - \*\*`EventSystem`\*\*: Manages user input events, such as mouse clicks and keyboard input.  The method also registers these creations with Unity's undo system for tracking changes in the editor.  #### \*\*`CreateAndConfigurePrefab(string fileName, GameObject selectedGameObject, bool isUI = false)`\*\*  This method is the core of the script, responsible for creating and configuring prefabs based on a provided `fileName`. If it's a UI prefab, it ensures a canvas exists by either finding one or creating a new one via `CreateUICanvas()`.  - \*\*`selectedGameObject`\*\*: The GameObject that will serve as the parent for the new prefab.  - \*\*`isUI`\*\*: A flag indicating if the prefab is a UI element. If true, the prefab will be parented to a canvas.  The method locates the prefab by its name, instantiates it, and calls `FinalizePrefabSetup()` to finish setting up the newly created prefab.  #### \*\*`FinalizePrefabSetup(string fileName, GameObject newGameObject)`\*\*  This method finalizes the setup for the newly created prefab:  - Registers the prefab with Unity’s undo system.  - Unpacks the prefab so that it can be edited in the scene.  - Automatically selects the new GameObject in the hierarchy.  - Sets up a delayed call to focus on renaming the new object, making it easier for the user to adjust its name.  #### \*\*Menu Item Methods\*\*  The script includes a number of `MenuItem` methods, which integrate with Unity’s editor and add menu items under the “GameObject/Language” path. Each of these methods corresponds to a specific type of prefab, such as `Audio Source (LT)` or `Dropdown (LT) [TMP]`.  For example:  - \*\*`CreateAudioSourcePrefab()`\*\*: This method creates and configures an audio source prefab.  - \*\*`CreateTMPTextPrefab()`\*\*: This method creates a TextMeshPro text field.  Each of these methods calls `CreateAndConfigurePrefab()` with the appropriate prefab name and parent GameObject.  ### Usage  1. \*\*Accessing the Prefab Creator\*\*: In the Unity editor, you can access the prefabs under `GameObject > Language`. This provides a submenu with various options for creating 3D and UI objects like audio sources, text fields, and language managers.  2. \*\*Creating a UI Prefab\*\*: If a UI element (e.g., a button or dropdown) is selected, the script ensures that a canvas exists, creating one if necessary. The UI prefab is then instantiated and added as a child of the canvas.  3. \*\*Tracking Changes\*\*: Each action performed by the script is registered with Unity's undo system, allowing developers to undo changes (like creating a prefab) easily within the editor.  4. \*\*Prefab Finalization\*\*: After creating a prefab, the script unpacks it (so it can be edited) and selects the newly created GameObject in the hierarchy. This streamlines the workflow for customizing newly instantiated prefabs.  ### Summary  This script streamlines the process of generating and configuring various language-related prefabs in Unity, ensuring that UI elements have proper setup (e.g., Canvas, EventSystem) and providing undo functionality. The prefabs include standard UI elements like buttons, dropdowns, and text fields, as well as language-specific components such as language managers and audio sources. The integration with Unity’s editor makes it easy to create, modify, and manage these prefabs directly from the editor’s menus. |

|  |
| --- |
| **Component Converter** |
| The `ComponentConverter` script is a Unity Editor utility that scans the selected GameObject for specific components, such as UI elements or 3D components, and converts them to their `LanguageTool`-compatible versions. It ensures that the GameObject maintains its original functionality while adding language-related components to support multilingual features. The script handles components like buttons, text fields, dropdowns, images, and audio sources, converting them and adding new language-handling components, such as `LanguageText`, `LanguageDropdown`, and `LanguageAudioPlayer`. It also ensures that components are not converted multiple times by checking for existing `LanguageTool` components before attempting conversion.  ### Key Variables and Methods  #### \*\*`ConvertComponents()`\*\*  - \*\*Purpose\*\*: This is the main method that gets triggered when the user selects the menu item "Language/Converter to Language Tool." It scans the currently selected GameObject for supported components and attempts to convert them to `LanguageTool` compatible versions.  - \*\*How It Works\*\*:  - Retrieves the selected GameObject from the hierarchy.  - Checks for UI components like `Button`, `Toggle`, `Dropdown`, `InputField`, and `Text`, as well as TMP (TextMeshPro) variants such as `TMP\_Dropdown`, `TMP\_InputField`, and `TMP\_Text`.  - Also checks for 3D components like `TextMesh`, `MeshRenderer`, and `AudioSource`.  - For each component found, the corresponding conversion method is called (e.g., `ConvertButton`, `ConvertDropdown`).  #### \*\*Conversion Methods\*\*:  Each conversion method converts a specific type of component by adding its `LanguageTool` equivalent and setting up relevant properties.  - \*\*`ConvertButton(Button button)`\*\*:  - Converts a standard `Button` component.  - Uses `ConvertTextComponent()` to handle any text child elements (e.g., `Text` or `TMP\_Text`).  - Logs an error if no text components are found in the button.  - \*\*`ConvertDropdown(Dropdown dropdown)`\*\*:  - Converts a `Dropdown` component to a `LanguageDropdown` component.  - Ensures that the dropdown is not already converted (by checking for `LanguageDropdown`).  - Adds an `AdjustSizeToDropdown` component for automatic sizing.  - Converts dropdown options into `LanguageOptions`.  - \*\*`ConvertInputField(InputField inputField)`\*\*:  - Converts an `InputField` component to a `LanguageTextInputField`.  - Ensures no previous conversion has been applied.  - \*\*`ConvertText(Text text)`\*\*:  - Converts a `Text` component to a `LanguageText` component.  - Ensures that `LanguageText` is not already present.  - \*\*`ConvertTMPDropdown(TMP\_Dropdown tmpDropdown)`\*\*:  - Converts `TMP\_Dropdown` components to `LanguageDropdownTMP` components.  - Adds a sizing adjustment component (`AdjustSizeToDropdownTMP`) and converts options into `LanguageOptions`.  - \*\*`ConvertTMPInputField(TMP\_InputField tmpInputField)`\*\*:  - Converts a `TMP\_InputField` into `LanguageTextInputFieldTMP`, ensuring no duplicate conversion.  - \*\*`ConvertTMPText(TMP\_Text tmpText)`\*\*:  - Converts `TMP\_Text` to a `LanguageTextTMP` component.  - \*\*`ConvertImage(Image image)`\*\*:  - Converts an `Image` to a `LanguageImage`.  - Ensures that no other UI components like buttons or input fields are present before converting the image, to prevent unnecessary conversion.  - \*\*`ConvertRawImage(RawImage rawImage)`\*\*:  - Converts a `RawImage` to a `LanguageRawImage`.  - \*\*`ConvertTextMesh(TextMesh textMesh)`\*\*:  - Converts a 3D `TextMesh` component to a `LanguageTextMesh`.  - \*\*`ConvertMeshRenderer(MeshRenderer meshRenderer, TMP\_Text textComponent)`\*\*:  - Converts a `MeshRenderer` associated with a `TMP\_Text` component into `LanguageTextMeshTMP`.  - \*\*`ConvertAudioSource(AudioSource audioSource)`\*\*:  - Converts an `AudioSource` to a `LanguageAudioPlayer` component.  #### \*\*`ConvertTextComponent<T>(T uiComponent, System.Action logError)`\*\*  - \*\*Purpose\*\*: This generic method handles the conversion of any UI component that has a text representation, such as `Button` or `Toggle`. It checks if the component has either a `Text` or `TMP\_Text` child component and converts them accordingly.  - \*\*How It Works\*\*:  - Looks for a `Text` or `TMP\_Text` component in the children of the UI element.  - Calls `ConvertText()` or `ConvertTMPText()` based on what it finds.  - If no text component is found, it logs an error using the provided error handling action.  #### \*\*Undo and Logging\*\*  Each conversion is registered with Unity’s undo system using `Undo.RegisterCreatedObjectUndo()`, which allows users to undo the conversion action in the editor. Success and error messages are logged to indicate whether the conversion was successful or if the component had already been converted.  ### Usage  1. \*\*Menu Access\*\*: Users can access the script’s functionality by selecting "Language/Converter to Language Tool" from Unity's GameObject menu.  2. \*\*Conversion Process\*\*: When triggered, the script scans the selected GameObject for supported components, converts them to `LanguageTool` compatible versions, and logs the process.  3. \*\*Undo Support\*\*: Users can undo each conversion step through Unity's standard undo feature.  4. \*\*Error Handling\*\*: The script prevents duplicate conversions by checking if the GameObject already contains a `LanguageTool` component. If a component has already been converted, an error message is logged, and the conversion is skipped. |

|  |
| --- |
| **Font Asset Bundle Builder** |
| This script, `FontAssetBundleBuilder`, is a Unity editor utility that provides functionality for building asset bundles specifically containing fonts and TMP (TextMesh Pro) fonts. Asset bundles are packages of assets that can be loaded dynamically at runtime in Unity. This script simplifies the process of finding font files and packaging them into asset bundles, ensuring that font files are properly managed and compressed. Here's a breakdown of how the script works, focusing on each method, variable, and its usage:  ### Key Methods and Variables  #### \*\*`BuildFontAssetBundles()`\*\*  - \*\*Purpose\*\*: This method initiates the process of building asset bundles for all font and TMP\_FontAsset files found within a specific folder labeled as "AssetBundles."  - \*\*How It Works\*\*:  - The method is linked to a menu item (`[MenuItem]`), making it accessible via the Unity editor's "Assets" menu.  - \*\*`AssetDatabase.FindAssets()`\*\*: Searches for folders named "AssetBundles" in the project and retrieves their GUIDs (Globally Unique Identifiers). If no such folder is found, an error message is logged.  - \*\*`AssetDatabase.GUIDToAssetPath()`\*\*: Converts the GUID of the first found folder into its actual file path, which is used to locate all font and TMP font files within that folder.  - \*\*`AssetDatabase.FindAssets()`\*\*: Searches for all `Font` and `TMP\_FontAsset` files within the folder.  - \*\*`BuildFontAssetBundle()`\*\*: For each found font and TMP font file, it calls this method to build the corresponding asset bundle.  - \*\*`AssetDatabase.Refresh()`\*\*: Refreshes Unity's asset database to reflect changes after building the asset bundles.  #### \*\*`BuildFontAssetBundle(string fontPath, string fontFolderPath, bool isTMP)`\*\*  - \*\*Purpose\*\*: This method handles the actual building of asset bundles for individual font or TMP font assets.  - \*\*How It Works\*\*:  - \*\*Parameters\*\*:  - `fontPath`: The path to the font asset.  - `fontFolderPath`: The path to the folder where the asset bundles will be saved.  - `isTMP`: A boolean indicating whether the font is a TMP font (`true` for TMP\_FontAsset, `false` for standard fonts).  - \*\*`AssetDatabase.GUIDToAssetPath()`\*\*: Converts the font asset's GUID into a path.  - \*\*`IsValidFontFile()`\*\*: This method is used to check if the font file has a valid file extension (`.ttf`, `.otf`, `.ttc`). If the file is not valid, the method returns early without building an asset bundle.  - \*\*`AssetBundleBuild`\*\*: A struct used to configure the asset bundle. It sets the bundle's name (lowercased to ensure consistency) and assigns the font asset to it.  - \*\*`BuildPipeline.BuildAssetBundles()`\*\*: This Unity method is responsible for actually building the asset bundle, using chunk-based compression for efficient storage and optimized loading at runtime.  - The method logs a success message once the asset bundle is created, and in case of failure, it catches any exceptions and logs an error message.  #### \*\*`IsValidFontFile(string extension)`\*\*  - \*\*Purpose\*\*: This helper method checks whether the file extension of a font is valid, preventing the script from attempting to process unsupported files.  - \*\*How It Works\*\*:  - It defines an array of valid font file extensions (`.ttf`, `.otf`, `.ttc`).  - \*\*`Any()`\*\*: The method checks if the provided extension matches any of the valid extensions. If so, it returns `true`; otherwise, it returns `false`.  ### Usage  1. \*\*Menu Access\*\*: The script can be accessed from the Unity editor via "Assets > Build Font Asset Bundles [Language Tool Unity]". This triggers the `BuildFontAssetBundles()` method, initiating the process of building asset bundles for fonts.  2. \*\*Asset Folder\*\*: The script looks for folders labeled as "AssetBundles" in the project and processes any fonts (both standard and TMP fonts) within that folder.  3. \*\*File Validation\*\*: It ensures that only valid font files are processed, checking file extensions to avoid errors.  4. \*\*Building Asset Bundles\*\*: For each valid font and TMP font asset, the script builds asset bundles and saves them in the designated folder. These bundles are then compressed and made ready for use in the project.  ### Logging and Error Handling  - The script logs success messages after successfully creating asset bundles and logs detailed error messages if any issues arise (such as missing files or invalid formats). This provides valuable feedback during the asset-building process, making it easier for developers to track progress and resolve issues. |