👋 Welcome

Welcome to LanguageSystem Pro Documentation

LanguageSystem Pro is a powerful and versatile language management solution designed to streamline multilingual support in your Unity projects. Whether you're building games or interactive applications, LanguageSystem Pro enables you to easily manage translations, switch languages dynamically, and integrate with external language files using a variety of formats such as JSON, XML, and CSV.

This documentation will guide you through the features and functionality of LanguageSystem Pro, providing detailed explanations and examples to help you get the most out of this tool. From setup and integration to advanced customization, you'll find everything you need to implement seamless language support in your project.

Key Features:

- **Multi-format Support**: Handle language data in JSON, XML, or CSV formats.
- **Dynamic Language Switching**: Effortlessly switch between languages at runtime.
- **Text and TextMeshPro Integration**: Fully compatible with both Unity's UI system and TextMeshPro.
- **External and Internal Language Files**: Load languages from internal resources or external files located in StreamingAssets.
- **Global Language Support**: Manage global language files to maintain consistency across different parts of your project.
-

🎯 Quickstart

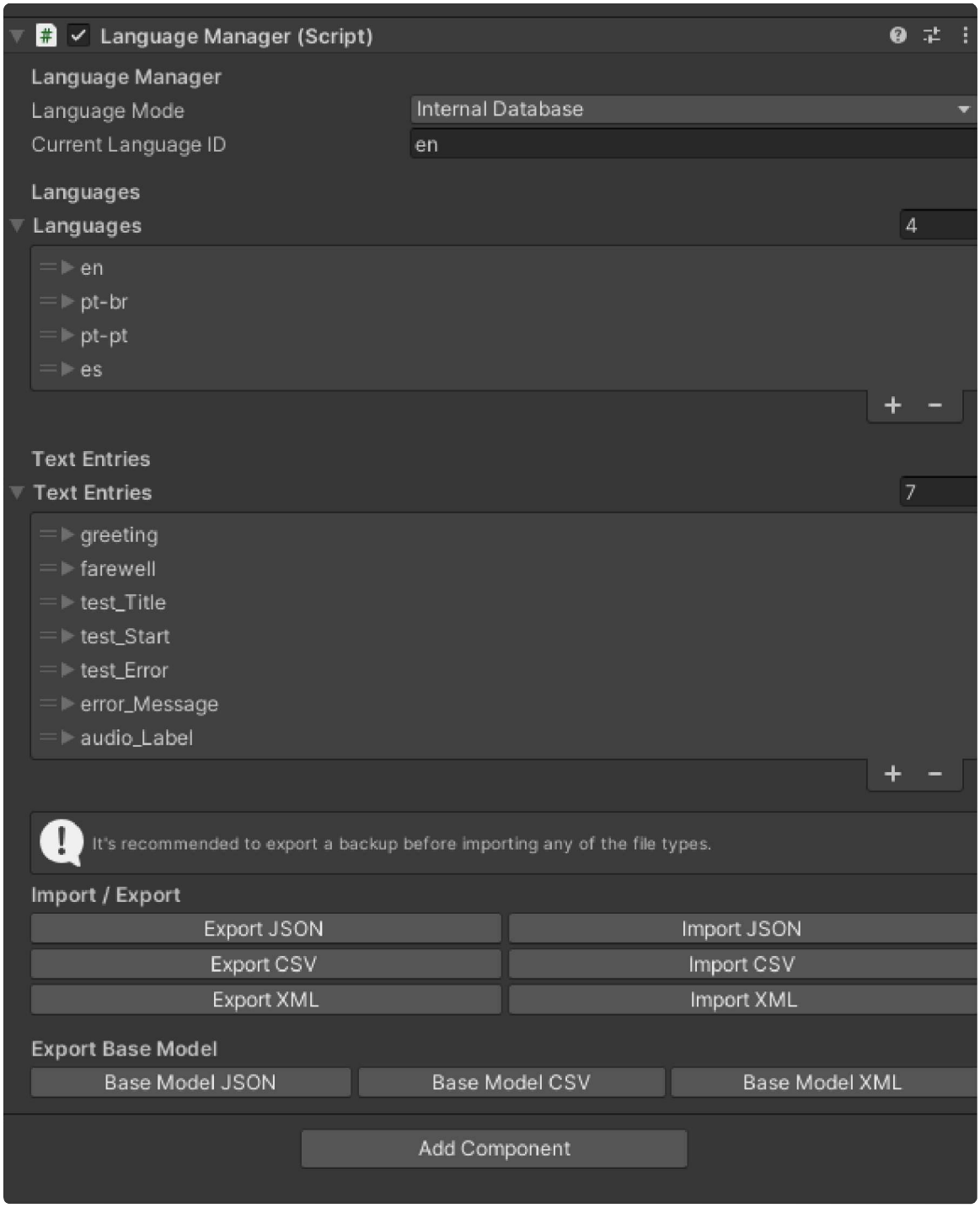
Getting Started: In the following sections, you'll learn how to set up LanguageSystem Pro, integrate it into your project, and configure it to meet the needs of your application.

Let's begin by exploring the installation process and basic configuration steps to get your language system up and running!

LanguageSystem Pro: Step-by-Step Guide

Welcome to the complete guide for using `LanguageSystem Pro` in your Unity projects. This document will walk you through each mode available in the system and how to import, export, and manage language data efficiently.

LanguageSystem Pro supports multiple formats (JSON, XML, CSV) and offers both internal and external language management options. Additionally, this guide will introduce you to our free desktop tool, `LanguageManagerEditor`, which simplifies the process of managing your language files.



Modes Overview

LanguageSystem Pro provides four modes to handle language data. Each mode serves different project needs, whether you're working with internal resources, external files, or global language data. Below is an explanation of each mode and how to use them effectively:

1. Internal Database Mode

In `Internal Database Mode`, all translations are stored directly inside the Unity project, which is ideal for smaller projects where you want to bundle language data within the game files.

- **How to Use**:
  You can add translations by directly editing the `TextEntries` list in the `LanguageManager` component. This mode does not require external files, making it straightforward for smaller projects or when no external localization system is needed.
- **Advantages**:
  - Quick and easy for small projects.
  - No need to manage external files.
- **Disadvantages**:
  - Not ideal for large projects with frequent updates to translations.
  - Any change to language data requires re-building the project.

2. External Files Mode

In `External Files Mode`, translations are stored as separate files in the `StreamingAssets` folder. You can manage languages individually, using files named after each language code (e.g., `en.json`, `es.xml`, `pt-br.csv`).

- **How to Use**:
  Place language files in the `StreamingAssets/Languages/` folder. Each file should be named according to its language code (`en.json`, `pt-br.xml`, etc.). These files can be in JSON, XML, or CSV format, and the system will load them at runtime.
- **Advantages**:
  - Easy to update translations without needing to re-build the project.
  - Developers can add or remove languages by simply modifying the `StreamingAssets` folder.
- **Disadvantages**:
  - Requires proper file management in the `StreamingAssets` folder.
- **Example Directory Structure**:

```
StreamingAssets/
    Languages/
        en.json
        es.json
        pt-br.json
```

3. Global External Files Mode

This mode allows you to store all translations in a single global file. This is useful for managing translations in one place, avoiding the need for multiple files.

- **How to Use**:
  Place a file named `globalLanguages` in the `StreamingAssets/Languages/` folder. This file can also be in JSON, XML, or CSV format. The system will load all language data from this global file.
- **Advantages**:
  - Centralized management of translations.
  - Easy to maintain consistency across all languages.
- **Disadvantages**:
  - For very large projects, a global file can become cumbersome to edit.
- **Example File**:
  A global file (`globalLanguages.json`, `globalLanguages.xml`, or `globalLanguages.csv`) will contain all the translations for all languages in one place.

4. Component-Based Mode

In `Component-Based Mode`, individual UI components can define their translations. This mode allows you to directly attach translations to objects in the Unity Editor. This is useful for highly customized localization setups.

- **How to use:** Add the LocalizedText script to any Text or TextMeshProUGUI and set your TextID and translations for each language. LanguageSystem Pro will dynamically update the text based on the active language.
- **Advantages**:
  - Fine-grained control over individual UI elements.
  - Easily supports dynamic language switching.
- **Disadvantages**:
  - Requires setting up each component manually.

Importing and Exporting

Importing and Exporting Language Files

Importing Language Files

LanguageSystem Pro supports importing language data from `StreamingAssets`. When using external files (either global or per-language files), ensure your files are properly formatted and placed in the `Languages` folder.

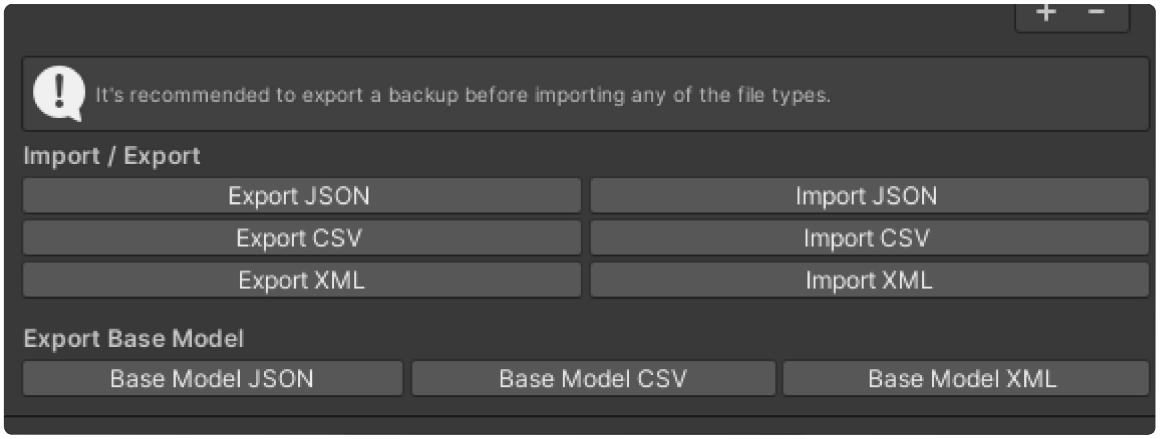
- **Supported Formats**:
  - JSON: Structured with `TextID` and corresponding `TranslatedText`.
  - XML: Using a serialized object structure.
  - CSV: CSV files should have the first row as the language IDs, and each following row should map a `TextID` to its translations.
- **Automatic Detection**:
  The system will automatically detect whether a JSON, XML, or CSV file exists in the `StreamingAssets/Languages/` folder and load the appropriate one.

If you choose to use InternalDatabase mode, you can also import files with translations already edited in an external application into Unity using the Import(JSON,XML,CSV) buttons in the LanguageManager component.

Exporting Language Files

You can also export your current language data to the StreamingAssets folder in the format of your choice (JSON, XML, or CSV). This can be done directly from the Unity Editor using the Import(JSON,XML,CSV) button in the LanguageManager component.

How to export: Just click the Export(JSON,XML,CSV) button in the LanguageManager component, specifying the path of the desired file. You can use this to backup your data or prepare files with translations.



Previous
Quickstart

Next
InternalDatabase Mode

Last updated 1 day ago

InternalDatabase Mode

InternalDatabase Mode

When opting to use the **InternalDatabase Mode**, you can manually add languages to the `LanguageManager` component, and below that, you can define a list of `TextEntries` by adding the translations for each `TextID` along with the corresponding `LanguageID`.

You also have the flexibility to edit these files externally by exporting a base model in your desired format. You can modify the file in external applications like Google Sheets or Excel (in the case of CSV files).

We also offer a **free editing tool** specifically built for the LanguageSystem Pro format, featuring a user-friendly layout that supports CSV, JSON, and XML formats. The tool makes it easy to edit your language files, and you can export them as a global file for use in **InternalDatabase** and **GlobalExternal** modes, or as separate files per language for **ExternalFiles** mode. For more details, check the LanguageManagerEditor section.

Adding a Language:

1. Open the `Languages` list in the `LanguageManager` component.
2. Click the **Add** button.
3. Define the **LanguageID** (e.g., `es`, `en`, `pt-br`, `pt-pt`, etc.), the **LanguageTitle**, and optionally set a **LanguageIcon**.

Adding a Text Entry:

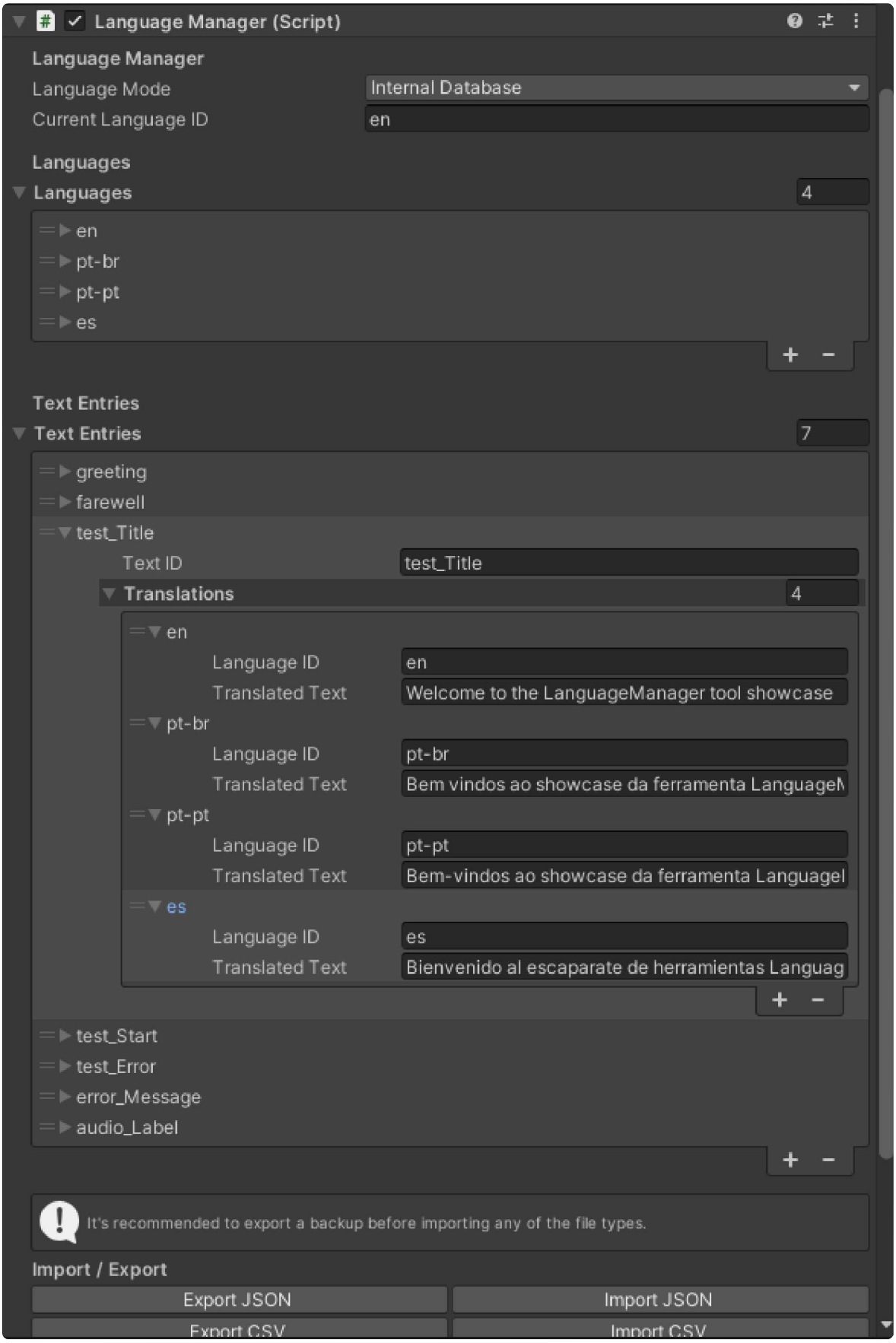
1. Open the `TextEntries` list and click **Add** to create a new text entry.
2. For each `TextEntry`, define a unique **TextID** to identify the text.
3. In the **Translations** list, add the **LanguageID** and the corresponding translation for each language.

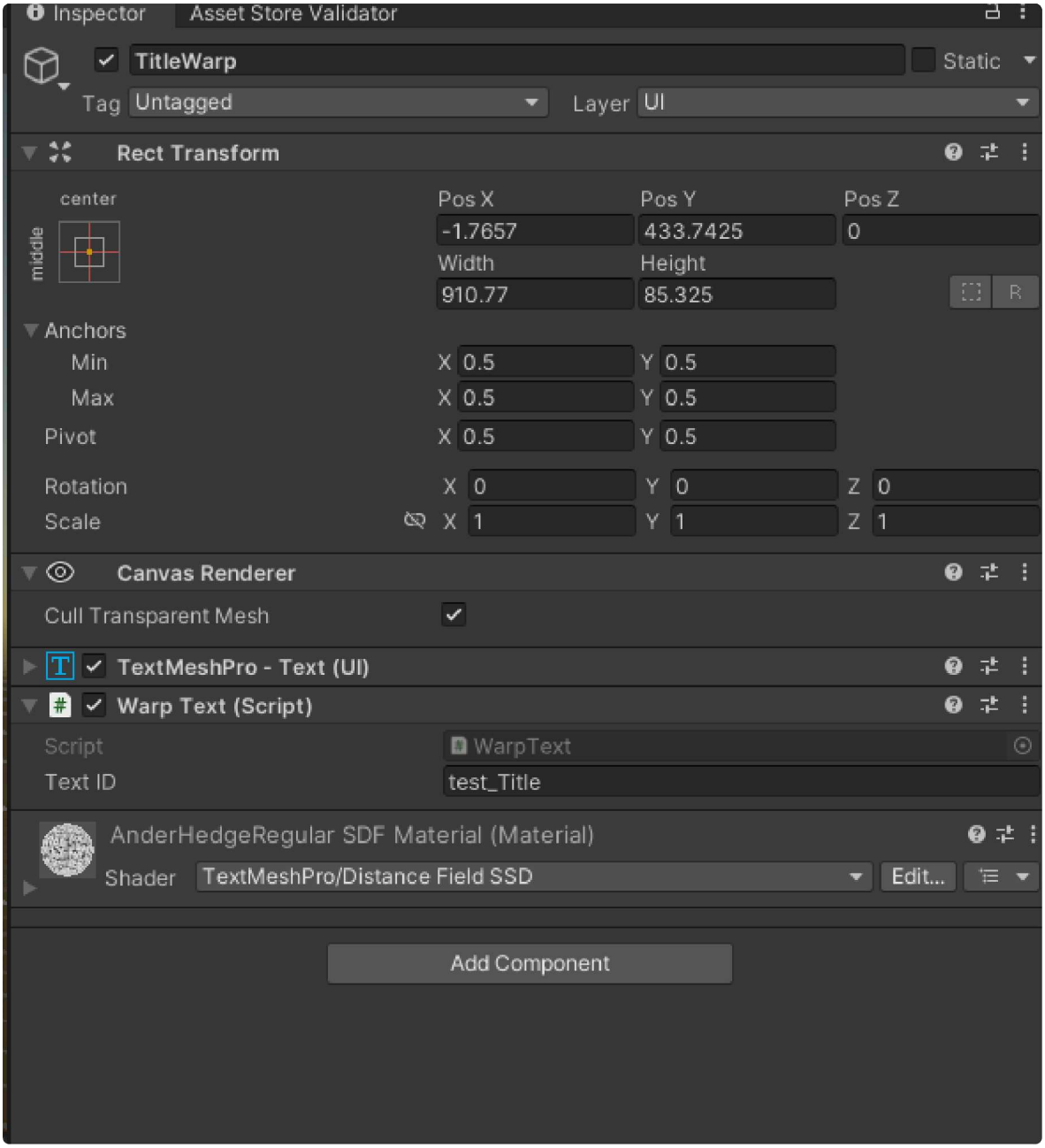
Once you have configured the language and `TextID`, you can:

1. Add the `WarpText` component to any `Text` or `TextMeshProUGUI` object.

2. In the `TextID` field of the `WarpText` component, input the `TextID` that you want the text to display.

When the game starts, the LanguageSystem Pro will automatically update the text in real time based on the selected language.





Make sure to follow this process for all texts that you want to be translated within your game.

Reminder: LanguageManagerEditor Tool

We offer a **free LanguageManagerEditor tool**, specifically designed for the LanguageSystem Pro format. It has a friendly interface and supports easy import, editing, and export of files in CSV, JSON, and XML formats. You can use this tool to export global files for the **InternalDatabase** and **GlobalExternal** modes or individual files per language for **ExternalFiles** mode.

For more details on using this editor, see the LanguageManagerEditor section.

ExternalFiles Mode

ExternalFiles Mode

In **ExternalFiles Mode**, language files must be stored separately for each language in the `StreamingAssets/Languages` folder. Each language should have its own file (e.g., `en.json`, `pt-br.json`, etc.), and the files should contain translations for that specific language.

This mode allows you to easily manage translations externally, especially useful if your project needs to scale with multiple languages. The system will automatically load the appropriate file based on the selected language.

We also offer a **free editing tool** specifically built for the LanguageSystem Pro format, featuring a user-friendly layout that supports CSV, JSON, and XML formats. The tool makes it easy to edit your language files, and you can export them as a global file for use in **InternalDatabase** and **GlobalExternal** modes, or as separate files per language for **ExternalFiles** mode. For more details, check the LanguageManagerEditor section.

Preparing Language Files:

1. Inside the `StreamingAssets/Languages` folder, create separate files for each language.
2. Each file must follow this format:

```
{
  "Translations": [
    {
      "TextID": "greeting",
      "TranslatedText": "Hello World"
    },
    {
      "TextID": "farewell",
      "TranslatedText": "Goodbye"
    }
  ]
}
```

- **File Naming**: The file should be named according to the language code, such as `en.json` for English, `pt-br.json` for Brazilian Portuguese, and so on.
- **TextID**: This is the unique identifier for the text entry that will be used in your game.
- **TranslatedText**: This is the actual translation for the `TextID` in the selected language.

> Note: The system will automatically look for the file that matches the current language, so make sure each language file contains the relevant translations.

Using WarpText with ExternalFiles Mode:

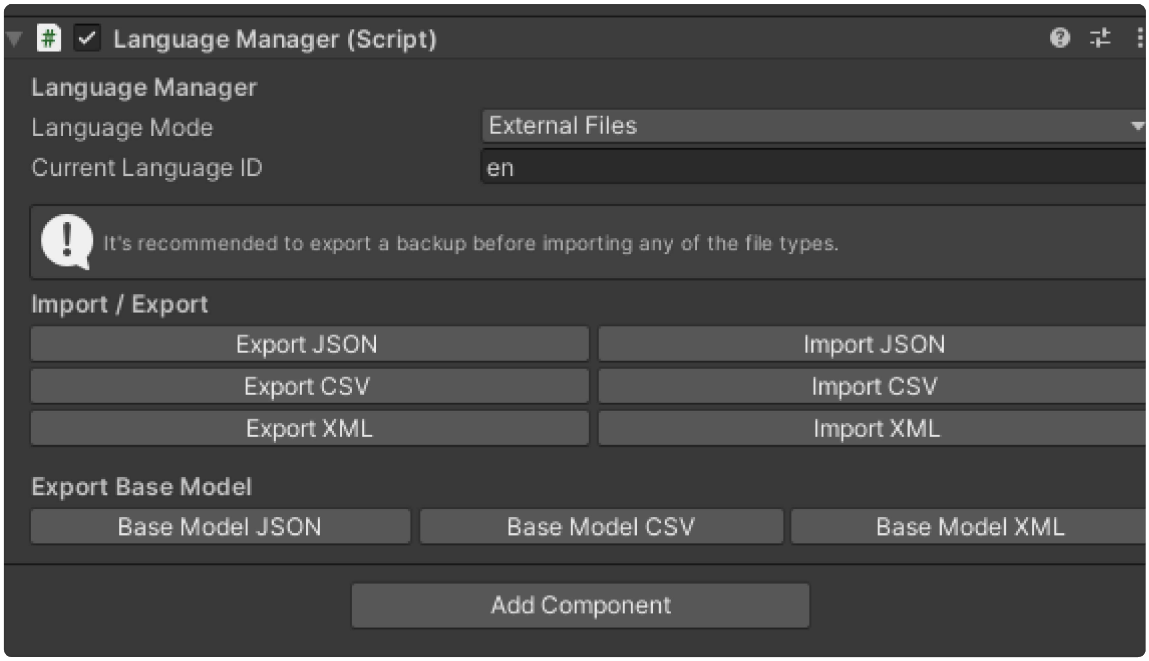
Once your language files are set up, you can use the `WarpText` component to display translated text in the game.

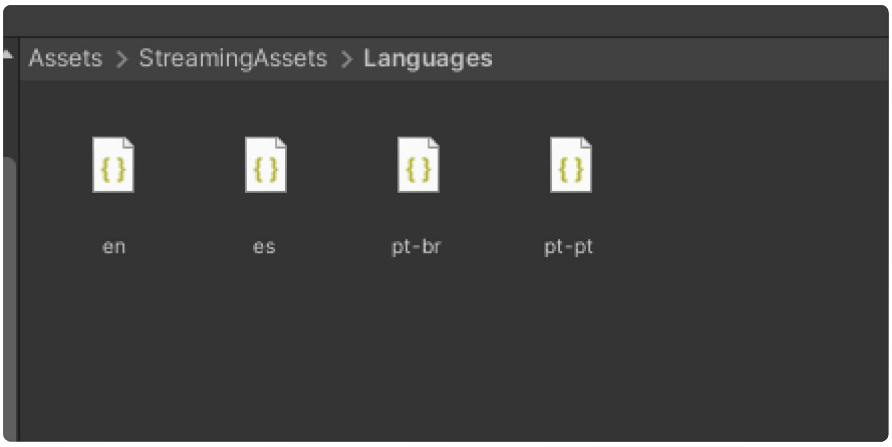
1. Add the `WarpText` component to any `Text` or `TextMeshProUGUI` object in your scene.
2. In the `TextID` field of the `WarpText` component, enter the `TextID` that corresponds to the translation in your language file.

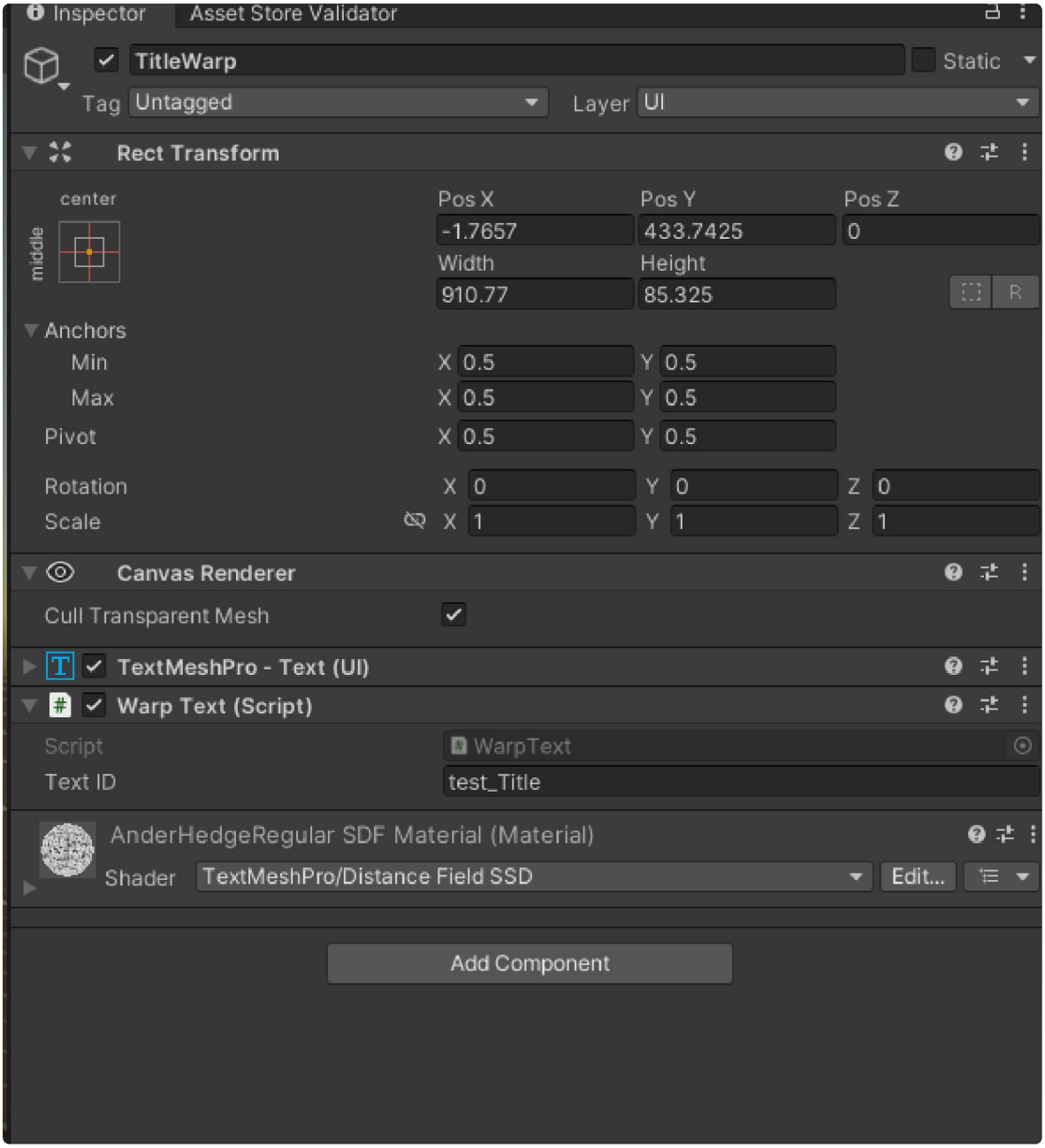
When the game starts, the LanguageSystem Pro will automatically read the appropriate file from `StreamingAssets/Languages` and update the text in real time based on the selected language.

Switching Languages:

To switch between languages, the LanguageSystem Pro will dynamically load the file corresponding to the selected language (e.g., `en.json`, `es.json`) from the `StreamingAssets/Languages` folder. You can change languages using a dropdown or any other input method, and the system will reload the text from the external files in real time.


Language Manager (Script)
Language Manager
Language Mode
External Files
Current Language ID
en
It's recommended to export a backup before importing any of the file types.
Import / Export
Export JSON
Import JSON
Export CSV
Import CSV
Export XML
Import XML
Export Base Model
Base Model JSON
Base Model CSV
Base Model XML
Add Component


Assets > StreamingAssets > Languages
en
es
pt-br
pt-pt



Make sure to follow this process for all texts that you want to be translated within your game.

Reminder: LanguageManagerEditor Tool

We offer a **free LanguageManagerEditor tool**, specifically designed for the LanguageSystem Pro format. It has a friendly interface and supports easy import, editing, and export of files in CSV, JSON, and XML formats. You can use this tool to export global files for the **InternalDatabase** and **GlobalExternal** modes or individual files per language for **ExternalFiles** mode.

For more details on using this editor, see the LanguageManagerEditor section.

GlobalExternal Mode

GlobalExternal Mode

In **GlobalExternal Mode**, a single global language file is used to store translations for all languages in one place. This file must be placed in the `StreamingAssets/Languages` folder and should contain all translations for all supported languages. This is a good option if you prefer to manage all translations in a single file rather than having separate files for each language.

Preparing the Global Language File:

1. Inside the `StreamingAssets/Languages` folder, create a global language file (e.g., `globalLanguages.json`, `globalLanguages.xml`, or `globalLanguages.csv`).
2. The global file should follow this format:

```
{
  "TextEntries": [
    {
      "TextID": "greeting",
      "Translations": [
        {
          "LanguageID": "en",
          "TranslatedText": "Hello World"
        },
        {
          "LanguageID": "pt-br",
          "TranslatedText": "Olá Mundo"
        }
      ]
    },
    {
      "TextID": "farewell",
      "Translations": [
        {
          "LanguageID": "en",
          "TranslatedText": "Goodbye"
        },
        {
          "LanguageID": "pt-br",
          "TranslatedText": "Adeus"
        }
      ]
    }
  ]
}
```

- **TextID**: The unique identifier for each text entry.
- **Translations**: This list holds translations for each `TextID`, where each entry has a `LanguageID` (e.g., `en`, `pt-br`) and the corresponding `TranslatedText`.

> Note: The system will load this file during game initialization and use it to switch between languages.

Using WarpText with GlobalExternal Mode:

As with other modes, the `WarpText` component is used to display text in your game based on the `TextID`.

1. Add the `WarpText` component to any `Text` or `TextMeshProUGUI` object in your scene.

2. Set the `TextID` field in the `WarpText` component to match the ID from the global language file (e.g., `greeting`, `farewell`).

When the game starts, the LanguageSystem Pro will load the global file from `StreamingAssets/Languages` and automatically update the text in real time based on the selected language.

File Formats Supported:

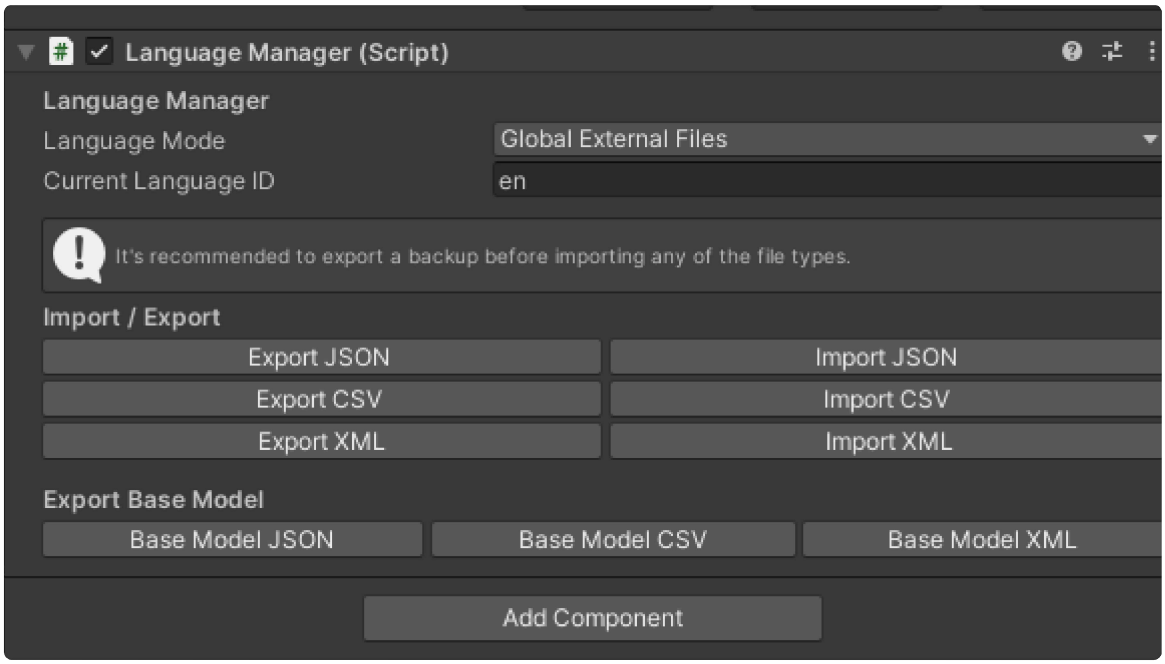
The global language file can be in any of the following formats:

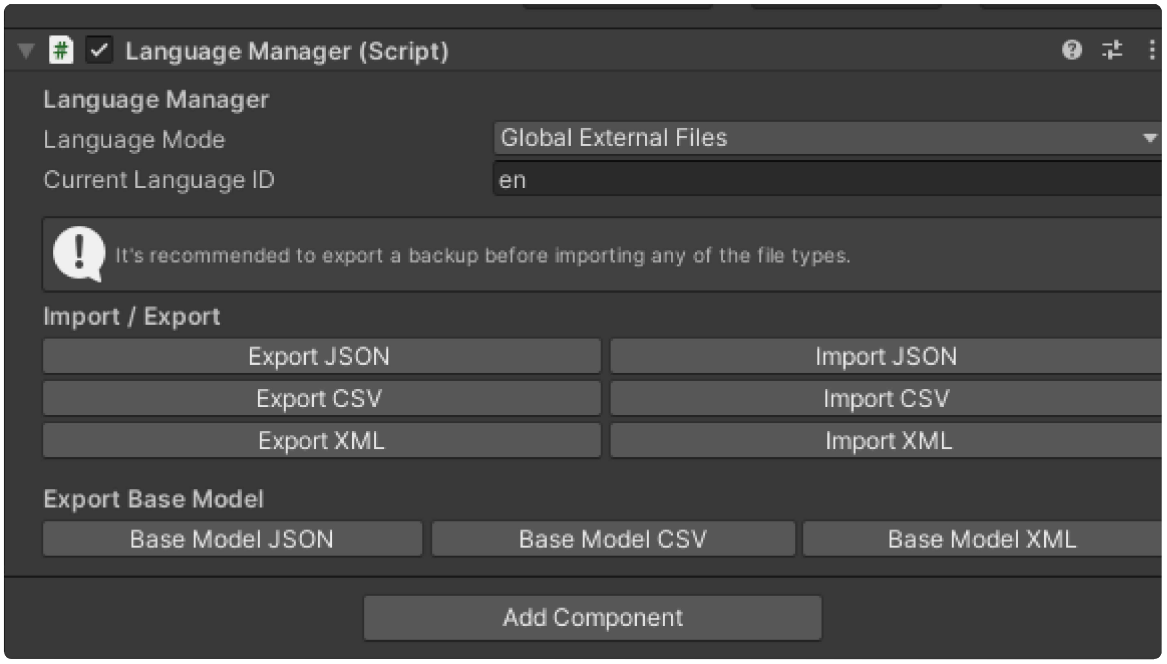
- **JSON**: `globalLanguages.json`
- **XML**: `globalLanguages.xml`
- **CSV**: `globalLanguages.csv`

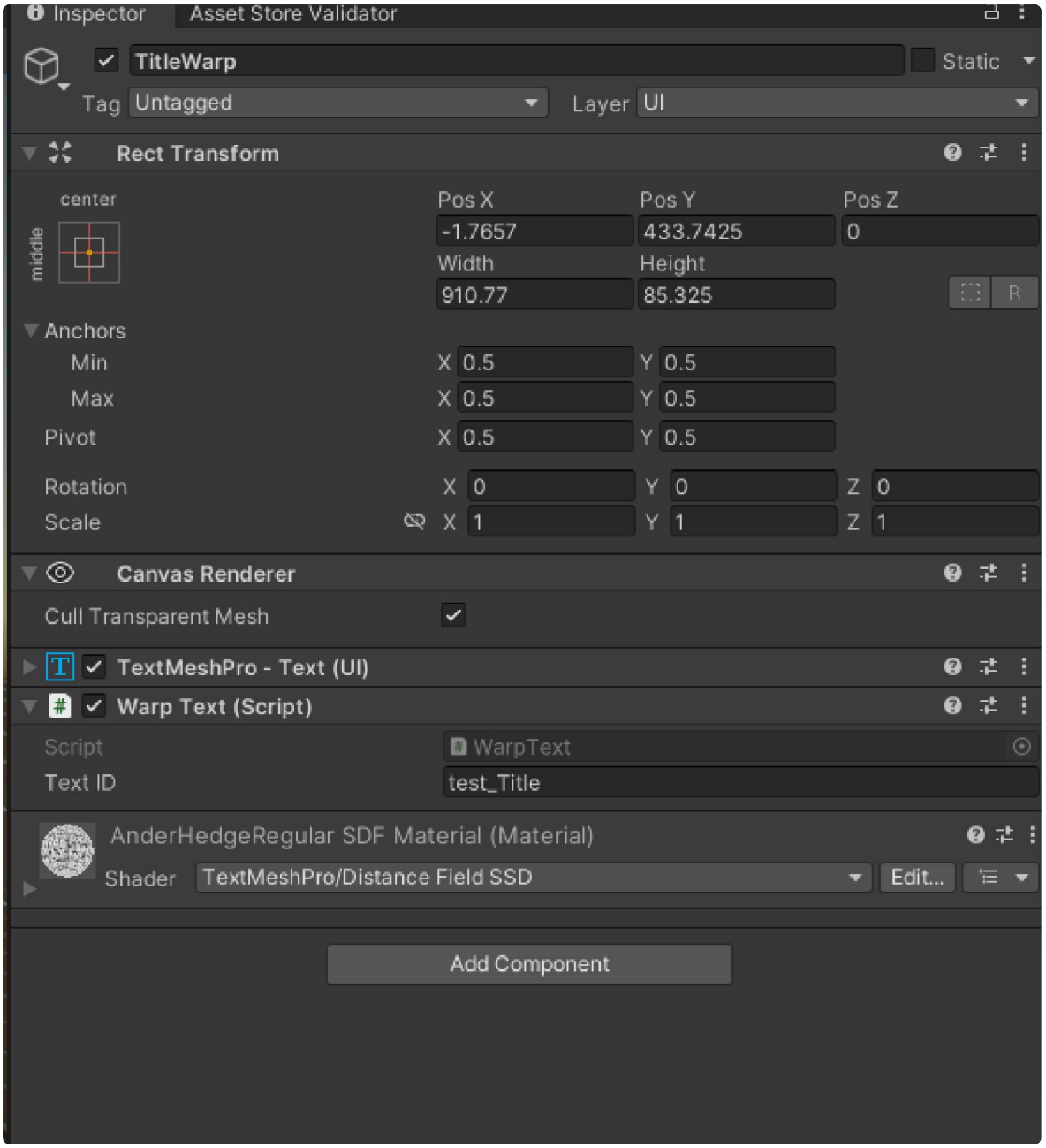
You can choose whichever format suits your workflow. The system will automatically detect and load the correct file during runtime.

Switching Languages:

To change the language in **GlobalExternal Mode**, the system will reload the `TranslatedText` values from the global file based on the selected language. You can implement a dropdown or a similar method to allow users to switch languages dynamically.




Language Manager (Script)
Language Manager
Language Mode
Global External Files
Current Language ID
en
It's recommended to export a backup before importing any of the file types.
Import / Export
Export JSON
Import JSON
Export CSV
Import CSV
Export XML
Import XML
Export Base Model
Base Model JSON
Base Model CSV
Base Model XML
Add Component


Inspector
Asset Store Validator
TitleWarp
Static
Tag
Untagged
Layer
UI
Rect Transform
center
middle
Pos X
Pos Y
Pos Z
-1.7657
433.7425
0
Width
Height
910.77
85.325
Anchors
Min
X 0.5
Y 0.5
Max
X 0.5
Y 0.5
Pivot
X 0.5
Y 0.5
Rotation
X 0
Y 0
Z 0
Scale
X 1
Y 1
Z 1
Canvas Renderer
Cull Transparent Mesh
TextMeshPro - Text (UI)
Warp Text (Script)
Script
WarpText
Text ID
test_Title
AnderHedgeRegular SDF Material (Material)
Shader
TextMeshPro/Distance Field SSD
Edit...
Add Component

> Make sure to follow this process for all texts that you want to be translated within your game.

> **Reminder: LanguageManagerEditor Tool**
>
> We offer a **free LanguageManagerEditor tool**, specifically designed for the LanguageSystem Pro format. It has a friendly interface and supports easy import, editing, and export of files in CSV, JSON, and XML formats. You can use this tool to export global files for the **InternalDatabase** and **GlobalExternal** modes or individual files per language for **ExternalFiles** mode.
>
> For more details on using this editor, see the LanguageManagerEditor section.

Previous
ExternalFiles Mode

Next
ComponentBased Mode

Last updated 1 day ago

ComponentBased Mode

ComponentBased Mode

In **ComponentBased Mode**, you do not need to manage external files or predefined lists of translations. Instead, this mode allows you to handle translations directly through the `LocalizedText` component on each text object. This provides a dynamic and flexible solution for smaller projects or when you want to manage translations directly in the Unity Editor.

Using the LocalizedText Component:

1. Add the `LocalizedText` component to any `Text` or `TextMeshProUGUI` object in your scene.
2. In the `LocalizedText` component, you'll find a field for `TextID` where you can define the unique identifier for each piece of text. This `TextID` will be used to refer to this particular piece of text across all languages.
3. Below the `TextID` field, you will find fields for translations. For each language, you can define the **LanguageID** (e.g., `en` for English, `pt-br` for Brazilian Portuguese, etc.) and provide the corresponding **TranslatedText**.
4. When the language is changed in your game, the `LocalizedText` component will automatically update the displayed text based on the current language. There's no need to manage external files, as all translations are stored within the component itself.

Example of Adding a LocalizedText:

1. **Add the Component**: Select your `Text` or `TextMeshProUGUI` object and click **Add Component**. Search for the `LocalizedText` component.
2. **Define the TextID**: Set a unique `TextID` for this text, for example, `greeting`.
3. **Add Translations**: For each supported language, add the `LanguageID` and the corresponding translation:
   - For English (en): "Hello World"
   -

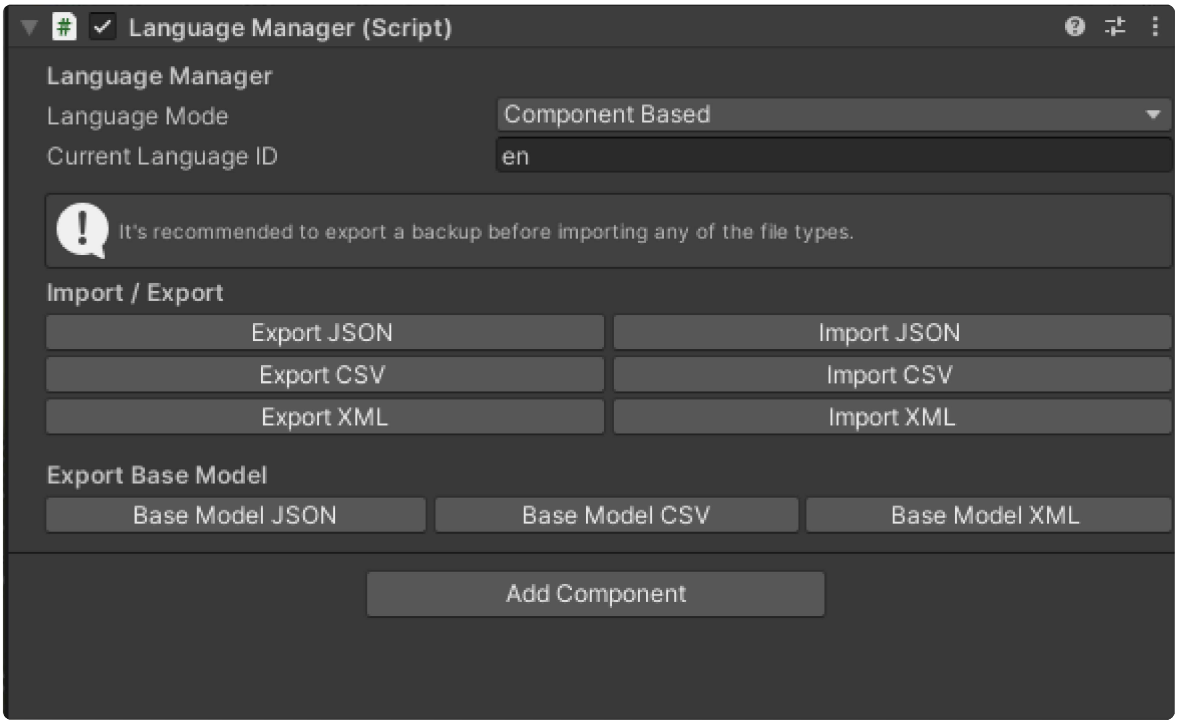
    - For Brazilian Portuguese (pt-br): "Olá Mundo"
    - For Spanish (es): "Hola Mundo"
4. **Automatic Update**: When the language changes in your game, the text will automatically update to the correct translation based on the `TextID`.

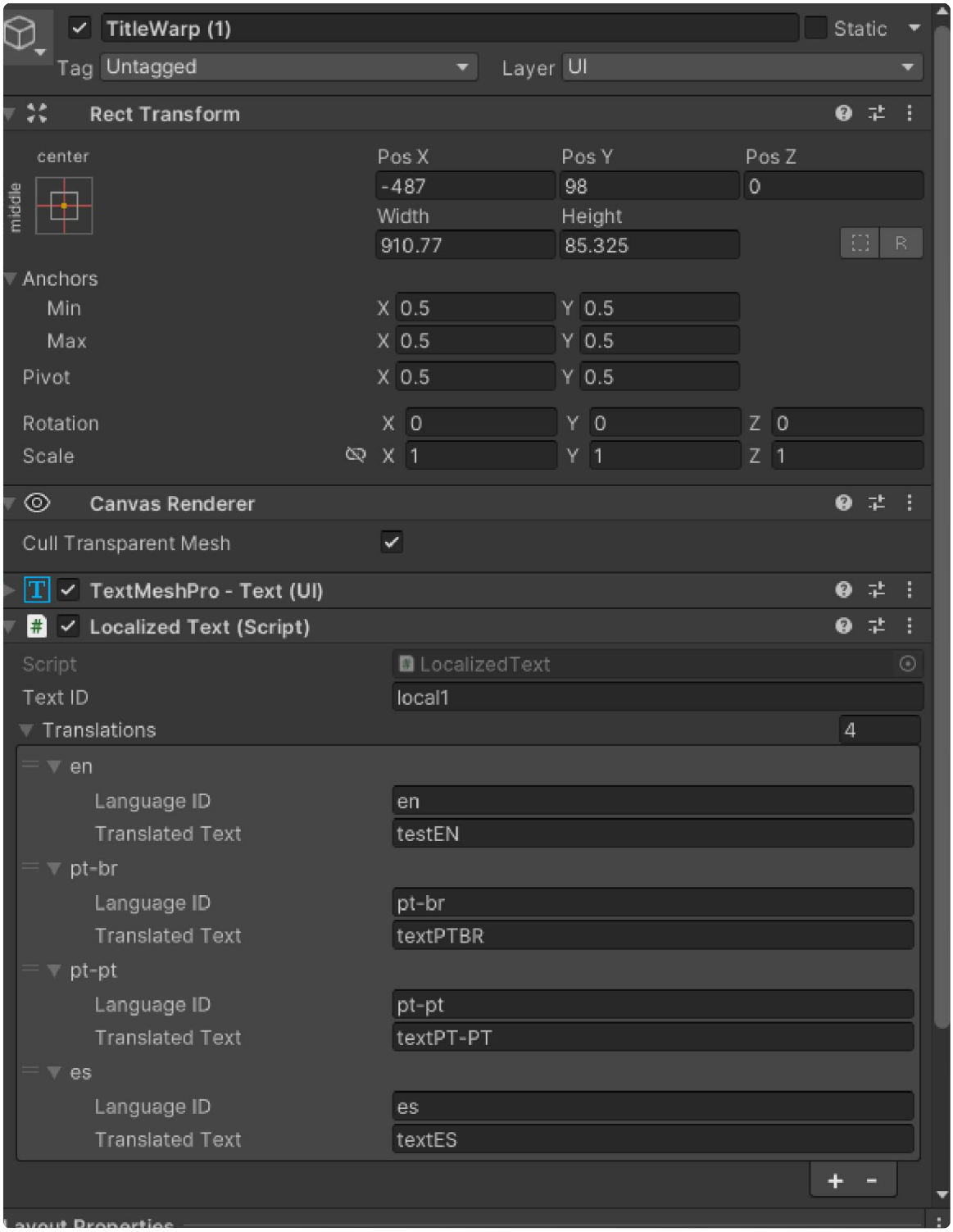
Benefits of ComponentBased Mode:

- **No need for external files**: All translations are managed directly in Unity using the `LocalizedText` component.
- **Dynamic updates**: When the language is switched, each `LocalizedText` component will update its displayed text without the need for reloading data from files.
- **Great for small-scale projects**: This mode is ideal if you have a limited number of texts and want a quick, efficient way to localize them.

Switching Languages:

As with other modes, you can change the language in **ComponentBased Mode** dynamically. Once the language is switched, all `LocalizedText` components in the scene will automatically refresh to display the appropriate translations.





Make sure to follow this process for all texts that you want to be translated within your game.

Reminder: LanguageManagerEditor Tool

We offer a **free LanguageManagerEditor tool**, specifically designed for the LanguageSystem Pro format. It has a friendly interface and supports easy import, editing, and export of files in CSV, JSON, and XML formats. You can use this tool to

> export global files for the **InternalDatabase** and **GlobalExternal** modes or individual files per language for **ExternalFiles** mode.
>
> For more details on using this editor, see the LanguageManagerEditor section.

Previous
GlobalExternal Mode

Next
Importing Translated Text into Custom Script

Last updated 1 day ago

Importing Translated Text into Custom Script

Importing Translated Text into Custom Scripts

Overview

The **LanguageManager** system not only handles in-game text translations for UI elements, but it also allows you to fetch translated text directly into your custom scripts. This can be useful when you need to handle translated content programmatically. In this section, we'll show you how to integrate translated text into your custom scripts, focusing on retrieving the translations using the **LanguageManager**'s `GetTextEntryByID()` method.

Example: Displaying Translated Error Messages

The following example demonstrates how to use the **LanguageManager** to retrieve a translated error message based on the current language and display it in a `Text` or `TextMeshProUGUI` component in Unity. This is useful for displaying dynamic content like error messages, notifications, or other in-game feedback that needs to adapt to the player's selected language.

Here's how you can integrate text translations into a custom script:

Step-by-Step Breakdown

1. **Assign Text Components**: Start by adding a reference to a `Text` or `TextMeshProUGUI` component in your script. These components will display the text retrieved from the **LanguageManager**.
2. **Get the Text ID**: The **LanguageManager** stores each text entry using a unique `textID`. In this example, we retrieve the translated text based on the `textID` provided.
3. **Retrieve the Translation**: Use the **LanguageManager** method `GetTextEntryByID(textID)` to fetch the translated text. This function returns the text corresponding to the provided `textID` in the player's currently selected language.

4. **Display the Translation**: After fetching the translated text, assign it to your UI element (such as `Text` or `TextMeshProUGUI`), and it will update automatically based on the current language.

Script Example

Here's a script that demonstrates how to import and use translated text in a custom Unity script:

```
using System.Collections;
using UnityEngine;
using TMPro;
using UnityEngine.UI;

namespace LanguageManager
{
    /// <summary>
    /// This script is used to display an error message from the LanguageManag
    /// on a Text or TextMeshProUGUI component.
    /// </summary>
    public class ErrorMessageTest : MonoBehaviour
    {
        public TextMeshProUGUI textMeshProUGUI; // The TextMeshPro component t
        public UnityEngine.UI.Text text; // The Unity Text component to displa
        public string textId; // The ID of the text in the LanguageManager

        /// <summary>
        /// Retrieves the error message from LanguageManager using the provide
        /// and displays it on the specified Text or TextMeshProUGUI componen
        /// </summary>
        public void ShowErrorMessage()
        {
            try
            {
                // Get the error message from the LanguageManager using the te
                string errorMessage = LanguageManager.Instance.GetTextEntryBy

                // Check if TextMeshProUGUI is assigned and set the error mess
                if (textMeshProUGUI != null)
                {
                    textMeshProUGUI.text = errorMessage;
                    textMeshProUGUI.color = Color.red; // Example: Set the tex
                }
                // If not, check if Unity Text is assigned and set the error m
                else if (text != null)
                {
                    text.text = errorMessage;
                    text.color = Color.red; // Example: Set the text color to
                }
                else
                {
                    Debug.LogError("No Text or TextMeshProUGUI component is as
                    return;
                }
            }
            catch (System.Exception ex)
            {
                Debug.LogError("Failed to retrieve or display the error messag
            }
        }
    }
```

```
}
```

Key Concepts

- **Using** `GetTextEntryByID()` : This is the core method for retrieving translations. Simply pass the `textID` associated with the text you want, and **LanguageManager** will return the correct translation for the current language.

  ```
  string translatedText = LanguageManager.Instance.GetTextEntryByID(textId);
  ```

- **Displaying in** `Text` **or** `TextMeshProUGUI` : You can directly assign the retrieved text to either `Text` (Unity UI) or `TextMeshProUGUI` components, depending on your project setup. This allows you to use either Unity's built-in text system or TextMeshPro, both of which are compatible with **LanguageManager**.

Usage Scenario

Imagine you are creating an in-game error popup or notification. Instead of hard-coding the error message in a single language, you can use **LanguageManager** to ensure that the message appears in the player's selected language, enhancing the user experience.

For example, if a player tries to perform an invalid action, you can fetch a translated error message using `GetTextEntryByID("error_invalid_action")` and display it dynamically on the UI.

Previous
ComponentBased Mode

Next
LanguageAudioManager

Last updated 1 day ago

LanguageAudioManager

LanguageAudioManager

Overview

The **LanguageAudioManager** is a singleton class in the **LanguageSystem PRO** that manages language-specific audio clips. It allows you to easily manage and retrieve audio files based on the current language setting in your game. You can use it to play localized audio, such as voiceovers or sound effects, which change depending on the selected language.

Key Features

- **Audio by Language**: Automatically plays the correct audio file based on the game's current language.
- **Audio Identification**: Each audio clip is assigned an `audioID` for easy reference.
- **Language-Specific Audio Management**: The system allows you to manage multiple audio clips for each language and audioID.
- **Audio Format Flexibility**: Works with standard Unity `AudioClip` objects, meaning you can use any supported audio format (e.g., MP3, WAV, etc.).

How It Works

The **LanguageAudioManager** stores audio clips using an `audioID` that maps to multiple language-specific `AudioClip` objects. When the audio is requested, it automatically selects the clip associated with the current language and returns or plays it.

Example Setup

- Add the **LanguageAudioManager** component to a GameObject in your scene (e.g., an empty GameObject or an audio manager GameObject).
- In the **Inspector**, configure the `audioEntries` list to include `audioID`s and associated language-specific audio clips.

AudioEntry and LanguageAudio Structure

Each **AudioEntry** consists of:

- **audioID**: A string identifier for the audio (e.g., "greeting", "gameOver").
- **languageAudios**: A list of **LanguageAudio** objects, where each entry links a language ID (e.g., "en", "pt-BR") to its corresponding `AudioClip`.

API Reference

`GetAudioByID(string audioID)`

Returns the `AudioClip` for the given `audioID` in the current language.

```
AudioClip clip = LanguageAudioManager.Instance.GetAudioByID("greeting");
```

- **Parameters**:
  - `audioID`: The ID of the audio file (e.g., "greeting").
- **Returns**:
  - The `AudioClip` for the specified audioID and the current language, or `null` if not found.

`AudioExists(string audioID)`

Checks if a specific audio clip exists for the given `audioID` in the current language.

```
bool exists = LanguageAudioManager.Instance.AudioExists("greeting");
```

- **Parameters**:
  - `audioID`: The ID of the audio to check.
- **Returns**:
  - `true` if the audio exists, otherwise `false`.

`PlayAudioByID(string audioID, AudioSource audioSource)`

Plays the audio clip associated with the given `audioID` on the provided `AudioSource`.

```
LanguageAudioManager.Instance.PlayAudioByID("gameOver", audioSource);
```

- **Parameters**:
  - `audioID`: The ID of the audio to play.
  -

- `audioSource`: The `AudioSource` where the audio will be played.

`GetAvailableLanguagesForAudio(string audioID)`

Returns a list of all available languages for the given `audioID`.

```
List<string> languages = LanguageAudioManager.Instance.GetAvailableLanguagesFo
```

- **Parameters**:
  - `audioID`: The ID of the audio for which to retrieve available languages.
- **Returns**:
  - A `List<string>` of language IDs that have associated audio clips.

Example: Play Localized Audio

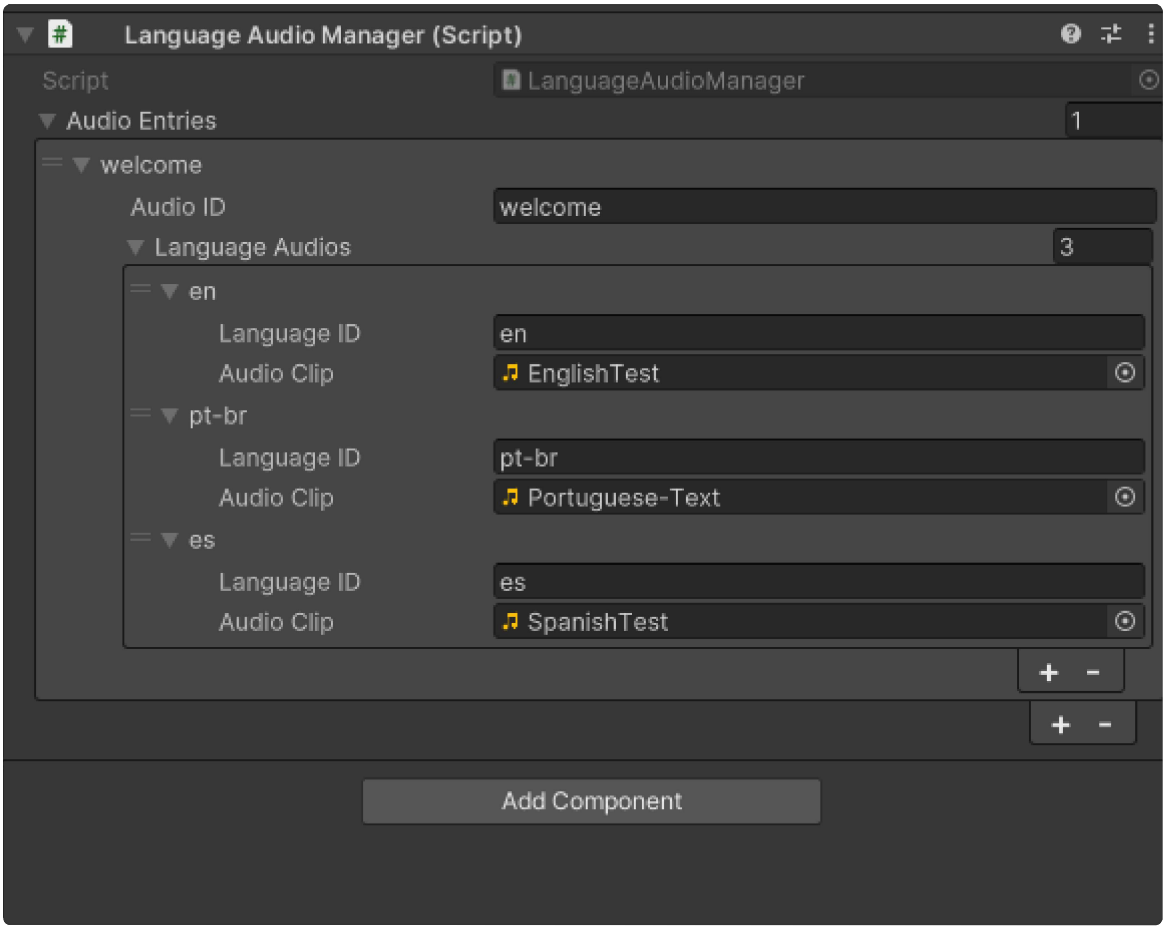
Here's an example of how you can play localized audio using the **LanguageAudioManager**.

```
public class PlayLocalizedAudio : MonoBehaviour
{
    public string audioID; // Unique audio ID for this sound
    public AudioSource audioSource; // AudioSource component

    private void Start()
    {
        LanguageAudioManager.Instance.PlayAudioByID(audioID, audioSource);
    }
}
```

Integrating with Custom Scripts

To integrate **LanguageAudioManager** into your custom scripts, use the `audioID` to retrieve and play audio clips based on the current language, as shown in the example above.



Previous
Importing Translated Text into Custom Script

Next
AudioWarper Component

Last updated 1 day ago

AudioWarper Component

AudioWarper

Overview

The **AudioWarper** component allows you to associate multiple audio clips with different languages directly in the inspector. Unlike the **LanguageAudioManager**, this component manages its own audio clips and swaps them automatically based on the selected language, without depending on any external audio manager.

This is useful for situations where audio needs to be localized on a per-object basis, and you want each object to handle its audio internally.

Key Features

- **Localized Audio**: Each object can have its own localized audio clips based on language.
- **Automatic Switching**: The audio clips automatically change based on the currently selected language.
- **Flexible Setup**: Can be configured to play the audio either on an `AudioSource` component or using `PlayClipAtPoint` for 3D sound positioning.
- **Easy Integration**: Integrates seamlessly with the existing **LanguageManager** to track language changes in real-time.

How It Works

The **AudioWarper** component stores a list of `LocalizedAudioClip` objects, each of which contains a `languageID` and an associated `AudioClip`. When the language is changed in the game, the component automatically updates the audio clip being used.

Setup Instructions

1. **Add the Component**: Add the **AudioWarper** component to any GameObject that requires language-specific audio.

2. **Configure Localized Audio Clips**: In the **Inspector**, configure the `audioClips` list to include the `languageID` (e.g., "en", "pt-BR") and the associated audio clip.
3. **Optional - Auto Update AudioSource**: If you want the component to automatically update the `AudioSource` on the object, enable the `Change AudioSource Clip based on Language` option and assign an `AudioSource`.
4. **Play Audio**: Use the `PlayLocalizedAudio()` method to play the audio in the correct language.

AudioWarper Properties

audioClips

- A list of localized audio clips. Each entry includes a `languageID` and an `AudioClip`.

changeAudioSourceClip

- If enabled, the `AudioSource` on the GameObject will be automatically updated with the correct audio clip based on the current language.

audioSource

- The `AudioSource` component to update with the localized audio clip. If left empty, the component will try to get the `AudioSource` automatically from the GameObject.

API Reference

`UpdateAudioClip()`

Updates the audio clip of the associated `AudioSource` based on the current language.

```
audioWarper.UpdateAudioClip();
```

`PlayLocalizedAudio()`

Plays the localized audio clip for the current language. If an `AudioSource` is assigned, it plays through the `AudioSource`. Otherwise, it uses `AudioSource.PlayClipAtPoint()` to play the audio at the GameObject's position.

```
audioWarper.PlayLocalizedAudio();
```

`GetAudioClipForLanguage(string languageID)`

Retrieves the `AudioClip` associated with a specific language ID.

```
AudioClip clip = audioWarper.GetAudioClipForLanguage("en");
```

`StopAudio()`

Stops any currently playing audio on the assigned `AudioSource`.

```
audioWarper.StopAudio();
```

Example: Playing Localized Audio

Here's an example of how you can set up and use the **AudioWarper** component to play localized audio.

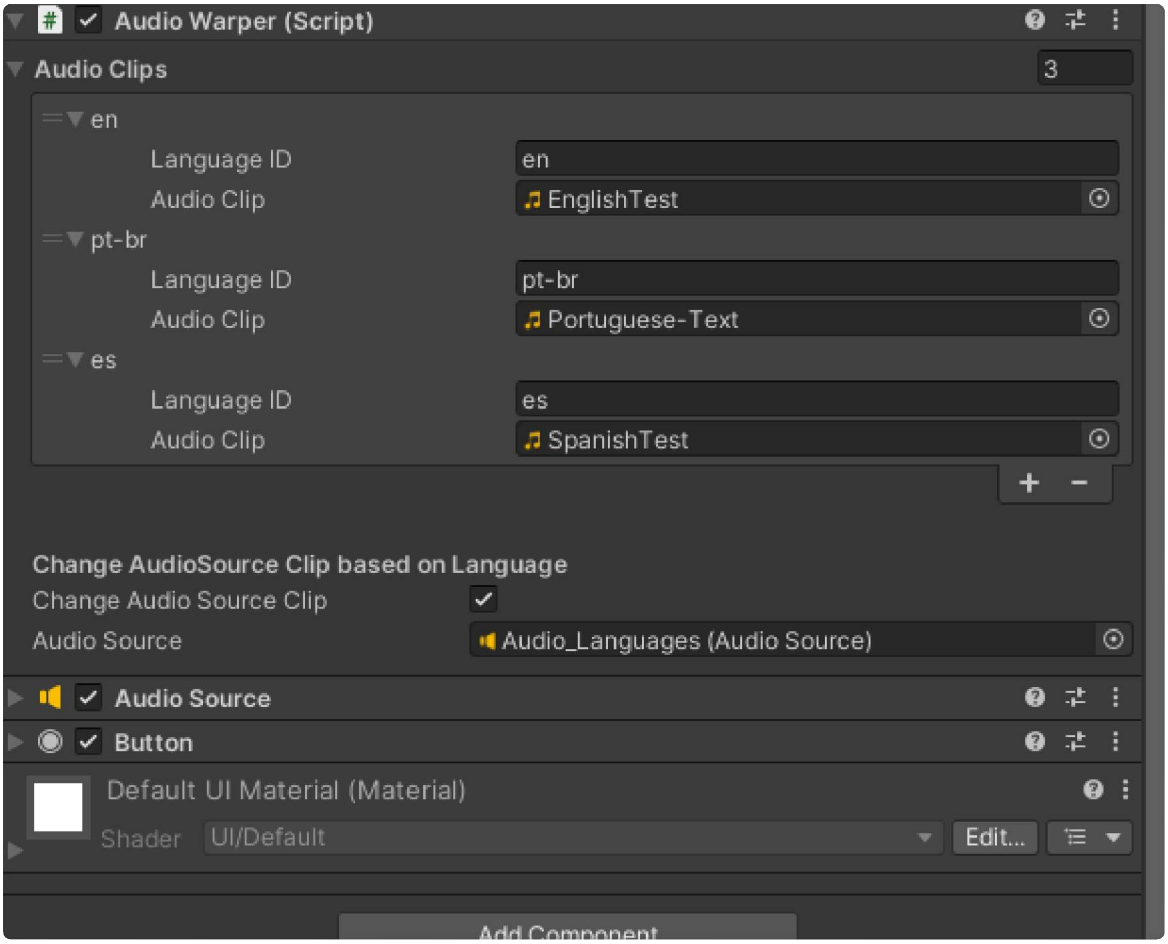
1. Add the **AudioWarper** component to a GameObject.
2. Add different audio clips for each language to the `audioClips` list.
3. Ensure the `AudioSource` is assigned or use the default settings.
4. Call `PlayLocalizedAudio()` when you want to play the audio in the correct language.

```
csharpCopiar códigopublic class AudioPlayer : MonoBehaviour
{
    public AudioWarper audioWarper;

    private void Start()
    {
        audioWarper.PlayLocalizedAudio();
    }
}
```

Use Case

The **AudioWarper** component is ideal for use cases where you want to manage localized audio clips directly on a GameObject without relying on a global audio manager. It gives you more granular control over individual audio sources, allowing each GameObject to manage its own language-specific audio assets.



Previous
LanguageAudioManager

Next
Importing Audio into Custom Script

Last updated 1 day ago

Importing Audio into Custom Script

Importing Audio into Custom Script

Overview

The **LanguageSystem PRO** supports language-specific audio clips, allowing you to play different audio files based on the current language setting. This system is managed by the **LanguageAudioManager**, which uses audio IDs to retrieve the appropriate audio clip for the selected language.

In this section, you will learn how to:

- Retrieve and play an audio clip based on the selected language using its audio ID.
- Implement the **CustomAudioImport** component to integrate this feature into your custom scripts.

Retrieving Language-Specific Audio Clips

To access an audio clip based on the current language, use the **LanguageAudioManager**. It manages all audio clips by associating them with an `audioID` and a list of language-specific `AudioClip` objects.

Example: How to Retrieve and Play an Audio Clip

Here is a basic example of how to retrieve and play an audio clip using the **LanguageAudioManager**.

```
public class PlayLocalizedAudio : MonoBehaviour
{
    public string audioID; // Unique audio ID for this sound
    public AudioSource audioSource; // The AudioSource that will play the audi

    private void Start()
    {
        // Get the audio clip based on the current language and audioID
        AudioClip clip = LanguageAudioManager.Instance.GetAudioByID(audioID);

        // Check if the clip exists, then play it
        if (clip != null)
        {
            audioSource.clip = clip;
            audioSource.Play();
        }
        else
        {
            Debug.LogError($"Audio clip for '{audioID}' not found.");
        }
    }
}
```

In this example:

- The `audioID` is used to find the appropriate audio clip for the current language.
- The audio is played via the provided `AudioSource` on the GameObject.

CustomAudioImport Example

To make this process easier, you can use the **CustomAudioImport** component, which serves as a practical implementation of the **LanguageAudioManager** in a reusable format. This component automatically retrieves and plays the correct audio clip for the current language based on a given `audioID`.

How to Use CustomAudioImport

Follow these steps to integrate **CustomAudioImport**:

1. **Attach CustomAudioImport**: Add the **CustomAudioImport** component to any GameObject with an `AudioSource` component. If it doesn't have an `AudioSource`, you'll need to add one.
2. **Set the Audio ID**: In the **CustomAudioImport** component, set the `audioID` field to the desired audio ID (e.g., "greeting", "farewell").

3. **Automatic Playback**: Once set up, **CustomAudioImport** will automatically retrieve the appropriate audio clip for the current language when the GameObject is activated.

CustomAudioImport Script Example

```
csharpCopiar códigousing UnityEngine;

public class CustomAudioImport : MonoBehaviour
{
    public string audioID; // The ID of the audio to play
    private AudioSource audioSource;

    private void Awake()
    {
        audioSource = GetComponent<AudioSource>();

        // Automatically play the audio when the component starts
        PlayAudioManually();
    }

    /// <summary>
    /// Manually plays the audio associated with the current language and audi
    /// </summary>
    public void PlayAudioManually()
    {
        if (audioSource == null)
        {
            Debug.LogError("No AudioSource found on this GameObject.");
            return;
        }

        AudioClip clip = LanguageAudioManager.Instance.GetAudioByID(audioID);
        if (clip != null)
        {
            audioSource.clip = clip;
            audioSource.Play();
        }
        else
        {
            Debug.LogError($"Audio clip for '{audioID}' not found.");
        }
    }
}
```

Customizing Audio for Each Language

Using **CustomAudioImport** or manual scripting, you can customize audio playback for each language. This flexibility allows for localized voiceovers, sound effects, or

notifications tailored to your audience's language preference.

Summary of Key Functions in LanguageAudioManager

- **GetAudioByID(audioID)**: Retrieves the `AudioClip` for the current language and the given `audioID`.
- **PlayAudioByID(audioID, AudioSource)**: Plays the audio clip for the `audioID` using the provided `AudioSource`.
- **AudioExists(audioID)**: Checks if an audio clip exists for the given `audioID` in the current language.

Previous
AudioWarper Component

Next
Switching Languages

Last updated 1 day ago

Switching Languages

Switching Languages Using the LanguageButton and LanguageDropdown Components

1. LanguageButton Component

The **LanguageButton** component is a simple, user-friendly solution for switching languages when a button is clicked. Each button is tied to a specific language via the `languageID` variable. When the button is clicked, the language is updated in the game.

How to Use the LanguageButton Component

- **Step 1:** Add the **LanguageButton** component to a Unity Button in your scene.
- **Step 2:** In the Inspector, you will see a field to input the `languageID`. Enter the language code (e.g., `"en"`, `"pt-br"`, `"es"`) for the language that the button should switch to.
- **Step 3:** Now, when the button is clicked, the **LanguageManager** will switch the language of the game to the corresponding `languageID`.

Here's an example:

```
[RequireComponent(typeof(Button))]
public class LanguageButton : MonoBehaviour
{
    public string languageID;

    private Button button;

    private void Awake()
    {
        button = GetComponent<Button>();
        button.onClick.AddListener(OnButtonClicked);
    }

    private void OnButtonClicked()
    {
        if (LanguageManager.Instance != null)
        {
            LanguageManager.Instance.SetLanguage(languageID);
        }
        else
        {
            Debug.LogError("LanguageManager instance not found.");
        }
    }
}
```

Example Scenario

- You could create multiple buttons, one for each language, and assign each a different `languageID`. Clicking the button will instantly change the game's language.

---

2. LanguageDropdown Component

The **LanguageDropdown** component allows players to select a language from a dropdown list. It works with both Unity's built-in **Dropdown** and **TextMeshPro TMP_Dropdown**. When a language is selected, the **LanguageManager** updates the game's language accordingly.

How to Use the LanguageDropdown Component

- **Step 1:** Add the **LanguageDropdown** component to a Unity **Dropdown** or **TextMeshPro TMP_Dropdown** in your scene.
- **Step 2:** The component automatically populates the dropdown with the languages available in the **LanguageManager**.
-

- **Step 3:** When the player selects a language from the dropdown, the language is switched in the game.

Here’s an example:

```
public class LanguageDropdown : MonoBehaviour
{
    private Dropdown dropdown;
    private TMP_Dropdown tmpDropdown;
    private bool isTMPDropdown = false;

    private void Awake()
    {
        dropdown = GetComponent<Dropdown>();
        if (dropdown != null)
        {
            isTMPDropdown = false;
            PopulateDropdown();
            dropdown.onValueChanged.AddListener(OnDropdownValueChanged);
        }
        else
        {
            tmpDropdown = GetComponent<TMP_Dropdown>();
            if (tmpDropdown != null)
            {
                isTMPDropdown = true;
                PopulateDropdown();
                tmpDropdown.onValueChanged.AddListener(OnTMPDropdownValueChang
            }
            else
            {
                Debug.LogError("No Dropdown or TMP_Dropdown component found or
            }
        }
    }

    private void PopulateDropdown()
    {
        if (LanguageManager.Instance != null)
        {
            List<string> options = new List<string>();
            foreach (Language language in LanguageManager.Instance.Languages)
            {
                options.Add(language.LanguageTitle);
            }

            if (isTMPDropdown && tmpDropdown != null)
            {
                tmpDropdown.ClearOptions();
                tmpDropdown.AddOptions(options);

                int currentIndex = LanguageManager.Instance.Languages.FindInde
                if (currentIndex >= 0)
                {
                    tmpDropdown.value = currentIndex;
                }
            }
```