VAMPORIUMLANGUAGE

Pseudo Language Translator

Simulate pseudo languages

Translating only parts of a text, depending
on the level of knowledge in that particular language

Different text formatting possibilities for the translated parts

Table of Contents

Features

- Quickly create pseudo languages that seem like real languages
- Test it right in the Inspector
- Gradual translation of a text's parts, depending on the knowledge level in that particular language
- Different text formatting (rich text) for the translated parts
- Automatically detects upper and lower cases

How to Translate

Quick Starting Guide

After downloading and including the asset into your project there are only 3 simple steps to use it:

1. To create a language, right click on the folder you want to create it in, select “Vamporium/Language/Language Data" and name your file as you wish
2. Set up and test your language
3. Translate your text by invoking one of the two overrides of the PseudoLanguage.Translate static methods

More Advanced Use

- Create a Component on your Player that holds the Player’s knowledge in different languages using the PseudoLanguage.Knowledge class or using the .
- For each document (scroll, book, etc) that uses a pseudo language, create and add a **Language** component with a LanguageData field that will hold the reference to a language in which this document was written. If you already have a component on it, you can include this field in your Component.
- Before the document appears, you can translate its text by invoking the PseudoLanguage.Translate method.

How to Set Up A Language and How It Works

After creating a LanguageData asset, you can begin setting up and testing your language.

This asset consists of some basic settings and three lists, one for *words*, one for letter *clusters* (groups) and one for each individual *letter*. It is really hard to rely only on words since there are quite a few thousands of them in each language, not counting their different forms, like plural, tenses, gluttenated, etc. This is why this asset is a pseudo language translator which simulates a translation by using the *Clusters* and the *Letters* sections.

The text given to be translated first is tested with the list of words, then the untranslated parts are tested with the list of *clusters* and finally whatever remains untranslated will be handled by the list of *letters*.

Words

If you want to have some specific words to be translated precisely, you can define them here, however – as mentioned before –, you will need to provide as many forms of the same word as possible… otherwise, for example, the plural can be translated differently from the specified form of the word.

Clusters

Here you can add letter clusters to be translated. For example, if “ight” should always be translated as “y” or “ugh”, this is the place to specify. Clusters must have at least two letters but the translated result can have any number of letters.

This is also the place where you should add common two-letter groups , such as “oo”, “ie” or “ee”. Or letter groups of even bigger numbers, such as “ight”, “ing”, etc.

Letters

Here are the single letters that are translated individually. However they can be translated into multiple-letter groups, for example “g” can become “kl”.

Tips

- If you want your pseudo language to remain speakable, avoid changing vowels (“a”, “e”, etc) into consonants (“m”, “”k”, “z”, etc… however you can go the other way: the more vowels, the less tongue twisting your language becomes
- You can entirely get rid of a word, cluster, letter by simply translating it into a “space” or even “nothing”
- You can create artificial word spaces in your text by adding spaces in your translated text; for example if you translate “m” into “al”, “ight” into “k yr” and “y” into ”m”, the word “mighty” will be translated into “alk yrm”
- In the inspector of each LanguageData asset you can test out the translation. Test several type of texts before you start the game, until you are close to what you’ve imagined the pseudo language should look like
- Using TextMeshPro texts you can make really nice additional formattings to the translated bits

Classes/Structs

PseudoLanguage

This class does most of the hard work. When a text needs to be translated, one of the static methods needs to be invoked with some arguments to tell which language and how much knowledge to use for the translation. Additionally the already provided Components can be used as demonstrated in the included example scene. You can also find descriptions of the LanguageKnowledge and LanguageDocument components.

public static string Translate(LanguageData data, string text, int knownLevels)
This method translates a text by providing the LanguageData, the text and the known levels in this particular language.

public static string Translate(LanguageData *data*, string *text*, float *learnt* = 0)
This method translates a text by providing the LanguageData, the text and the learn percentile (between 0 and 1) in this language.

public static string Translate(Knowledge *knowledge*, string *text*)
This method translates a text by providing a PseudoLanguage.Knowledge and the text.

PseudoLanguage.Knowledge

This is a subclass that can hold a character's/the player's knowledge in a particular language.

LanguageData *Data* : reference to the language this knowledge refers to

Float *Learnt* : the amount of knowledge in the language (between 0 and 1, where 0 being none and 1 being full knowledge)

LanguageData

A ScriptableObject that holds th01e dictionary and settings of a language. In the inspector of these ScriptableObjects there is a portion where direct testing of translations can be made.

Basic Info

String *Name* : the name of the language, mostly used to identify it for the player

String *Description* : you can attach a description to each language that can describe who speaks it, where the player found out about it, how hard it is, etc

Int *Difficulty* : this is the maximum level the player can learn in this language.

> NOTE: Once the player's knowledge in a language reaches this maximum, he/she knows it perfectly (all texts will appear fully translated to English, or any other language you want to communicate with your player). The progress of the player in languages can be held in the LanguageKnowledge Component or in one of your scripts. Then when translating you can invoke the PseudoLanguage.Translate method's override which accepts an argument of type int with the name of *knownLevels*.

Formatting

String *_formattingPrefix* : what to inject before a translated part (like a "" opening tag)

String *_formattingSuffix* : what to inject after a translated part (like a "" closing tag)

Bool *_simplifyFormattingI* : weather to simplify or not the formatting prefixes/suffixes.

> NOTE: This mode takes a bit more to translate a text but it's less messier. (Makes possible, for example, that instead of "translated" it will become "translated". In case of rich text tags it will have the same visual end result, but much cleaner, less characters.)

Testing

String *Test Text* : the text to be translated

Float *Learnt* : the amount of knowledge learnt in this language (between 0 and 1)

Button *Translate* : manually translate the test text (if the auto translation is disabled)

Auto Translation Toggle : toggles between auto and manual translation (disable it if your language has a really extensive database and/or the test text is really long, to not freeze the Inspector

String *Translation* : this is where the translated text will be shown

Bool *ShowFormatted* : toggles between the formatted (rich) text and the normal text (in this version all prefixes and suffixes will be visible in the text)

Database

Bool *_cacheDatabase* : whether to cache or not the list of *words*, *clusters* and *letters*. The cached lists will perform faster but always be case-sensitive, ignoring each entry's setting.

List<LanguageEntry> *words* : entire words to translate

List<LanguageEntry> *clusters*: clusters of two or more letters to translate (can translate two or more letter groups into one or more letters)

List<LanguageEntry> *letters* : translate letter by letter (can translate one letter into one or more letters)

LanguageEntry

A struct that holds an entry in the *words*, *clusters* and *letters* lists.

String *key* : what to translate

String *value* : into what

Bool *caseSensitive* : makes the search case-sensitive

Components

LanguageKnowledge

This one should be on the reader (usually the character) and holds the level of knowledge in each language.

LanguageDocument

This component goes on the GameObject of a document that will be translated. It supports the TextMeshPro TMP_Text component but really easily is editable to work with the default Text component. (Simply replace in the line 9 the text ***[SerializeField] private TMP_Text _textComponent = null;*** with ***[SerializeField] private Text _textComponent = null;*** and add the line ***using UnityEngine.UI;*** at line 4.)

String *Text* : the text that should be translated and shown in the Text or TTMP_Text component

LanguageData *LanguageData* : the language the text should be translated to

TMP_Text (or Text) *TextComponent* : the text component in which the translated text will be shown

Bool *TranslateOnEnable* : will translate the text each time this component or its GameObject will be enabled

Bool *TranslateOnLearn* : each time the player/reader will learn a level of knowledge in a particular language (via LanguageKnowledge) and the text on this Component is shown and is translated to this particular language, it will update reflecting the newly learnt knowledge