Translating Ren'Py games: A guide, by moskys

moskys#0576

¡Hola!

Call me moskys. As you surely know, if you have come to this guide, Ren'Py is a free and open source game engine (used for creating visual novels, generally) that has become one of the most popular software used by amateur creators to develop their gaming projects. Based on Python programming language, its simplicity and flexibility is a great advantage, since it allows to obtain more than acceptable results without prior programming knowledge. And one of the many functions that Ren'Py includes is to facilitate the translation of games created with this program.

So, after almost three years playing and manually translating into Spanish (and for PC) several games of diverse complexity, I have encouraged myself to write this guide to explain the process to anyone who want to start translating or are curious enough to know what I spend my spare time on.

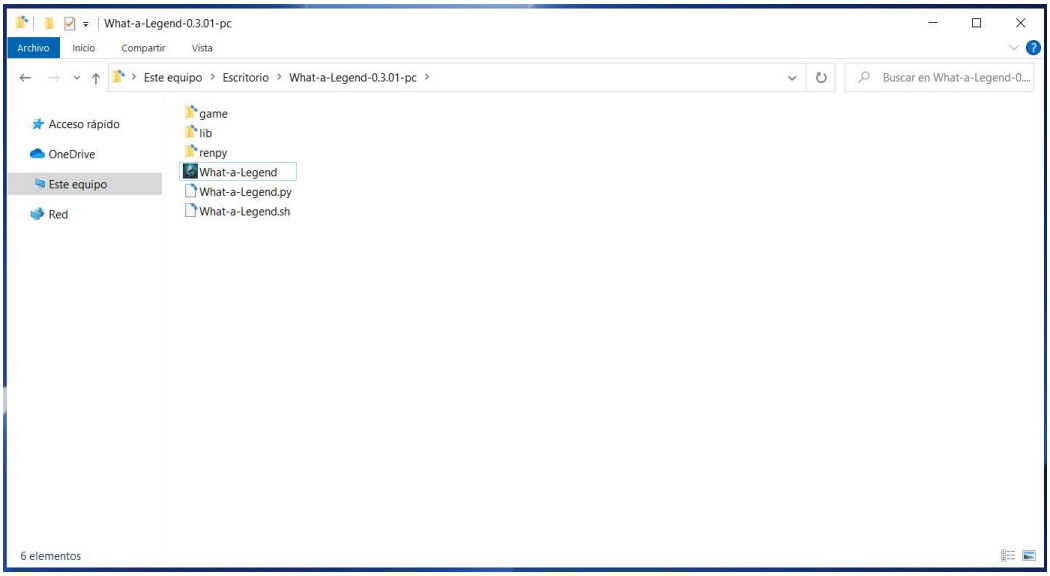
ESSENTIAL (AND FREE) TOOLS – versions as of 31-Dec-2020

- **Ren'Py SDK 7.3.5. ->** https://www.renpy.org/latest.html
- **UnRen v.09.dev ->** https://f95zone.to/threads/unren-bat-v0-8-rpa-extractor-rpyc-decompiler-console-developer-menu-enabler.3083/ *(links to a forum with adult content)*
- **Text editor:**
  - **Notepad++ 7.9.1 ->** https://notepad-plus-plus.org/downloads/
  - **Atom ->** https://atom.io/

1.- INTRODUCTION

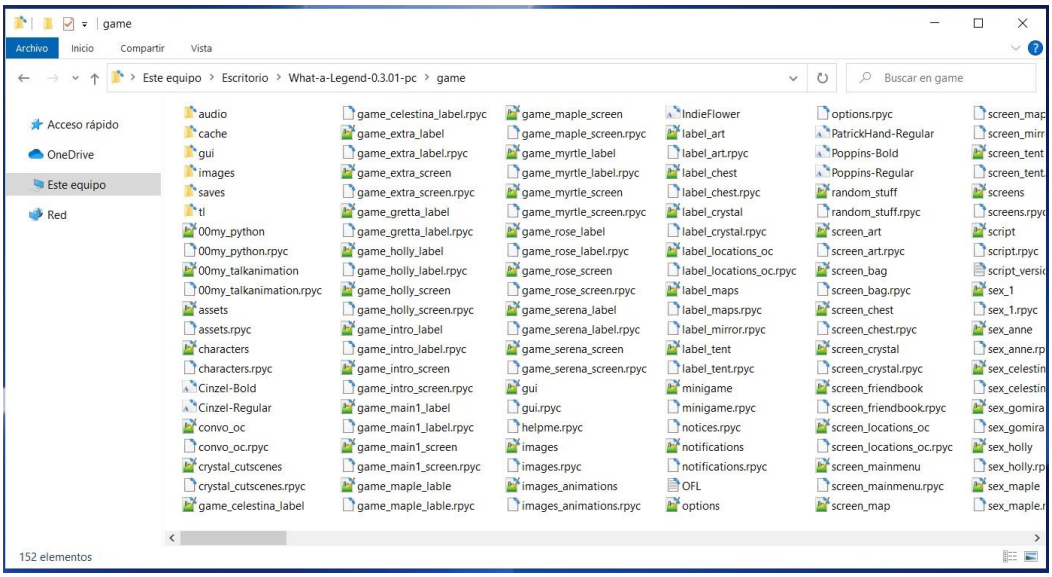
1.1.- Under the hood of a Ren'Py game

It is obvious, but in order to translate a game, you must first have a game. So look for any Ren'Py game that you have downloaded and take a look at its root folder. You'll see something like this:



Usually, we just run the executable file and play. But now let's look at the tree subfolders we see there, which is where the magic happens. Broadly speaking, in the "renpy" folder we can find the files containing the preprogrammed functions, and in the "lib" folder we have all the technical files that make the games run on the various operating systems. These two folders allow us to play the game without downloading any other specific program for it; we could say those folders turn Ren'Py games into self-executing programs.

The game itself is located in the "game" folder. Don't panic, this one is especially complex:



There are several subfolders that we can overlook for now to focus on individual files. When packaging their games, Ren'Py recommends creators to leave all the files in plain sight, as "What a Legend!" devs do. This allows us to see a whole series of apparently cloned files, some with a .rpy extension and others with a .rpyc extension.

|Top|

1.2.- Three basic concepts and two commandments

These files in the "game" folder are the **scripts**. It's in those scripts where developers include all the game's programming code and texts. To do this, they just write what they need to write in a text editor and save the file with a Ren'Py-specific extension named .rpy.

So, using a basic program like Windows Notepad (or, preferably, a better one, like Atom or Notepad++, to name two free and simple ones used in programming), we can open those .rpy files and see what creating a Ren'Py game is all about. Using "What a Legend!" as an example, if we open the file named "convo_oc.rpy", in its first lines we can see this:

```
1  # ==================== OLD Capital Base Conversations ========================================
2  # Being stopped at the gate of the old capital ==============================================
3  label convo_gate:
4      call hide_ui from _call_hide_ui_104
5      call silent from _call_silent_42
6
7      scene scene_oc_bridge_gate_talk
8      if current_hour == "Night" or current_hour == "Evening":
9          show kevin at g_cright:
10             xalign 0.6
11         show pov at m_left
12         with quickfade
13         show pov ewide bup mdislike hfshock hbshock
14         show kevin hbstop eangry
15         kevin "STOP!" with vpunch
16         show kevin hbpoint edoubt
17         show pov -mdislike hfneu1 hbneu1
18         kevin "Did you manage to get a passage permit?"
19         show pov mno bdoubt eneub hfhead
20         show kevin eneu hbneu
21         pov "Umm..."
22         show pov eneu bsad msad
23         show kevin esad hfspearmove
24         kevin "I'm sorry, buddy..."
25         show pov mcry eflat bneu hfneu1
26         kevin "...but no permit, no passage. That is the law."
```

Anything to the right of a # symbol are comments that will not appear in-game, but that creators use to guide themselves through their code. Later we will see how useful can this symbol be for translators.

In line 3 we see a word, **label**, which is going to be of importance. Labels are portions of the script. The size of these portions depends on the game developer, but they generally correspond to a specific scene. In this case, this label corresponds to a conversation that will appear on screen whenever players do some action that activates it. Due to Python language, everything happening in this scene is coded with an indentation.

> **Ren'Py's First Commandment:** you will always respect indentations and will NEVER mark them with Tab, but with the space bar (4 spaces, usually). If Ren'Py detects a Tab or an incorrect indentation somewhere in the scripts (including translation files) the game will not start.

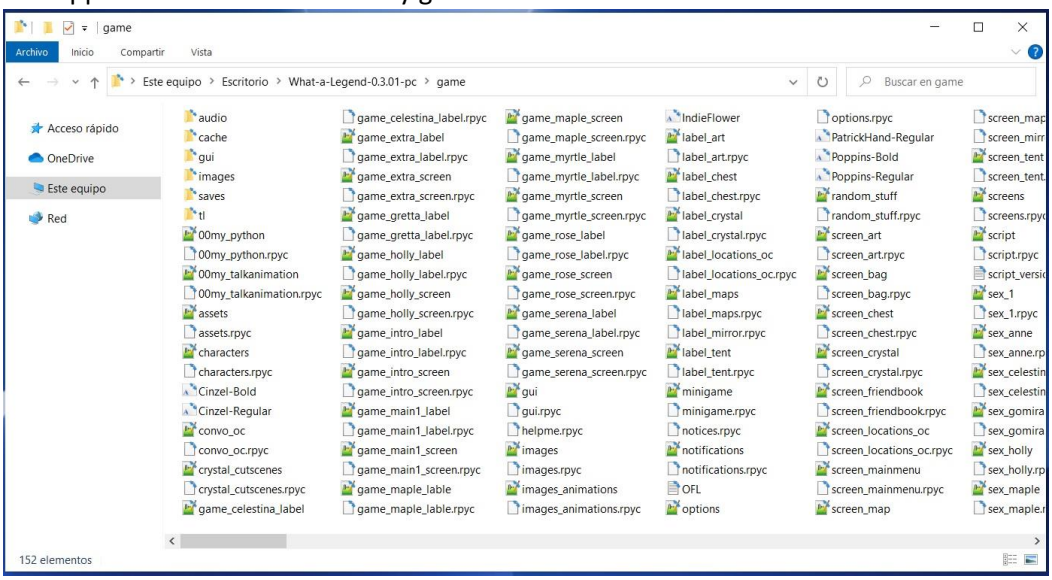
If we scroll down the script, within this very same "convo_gate" label we'll see several functions to show and hide images and to determine which part of the conversation will be displayed based on the game's variables triggered by players during their playthrough. Also, we can see some quoted sentences: these are the dialogue lines that will appear on screen. Those quoted sentences are the ones we will have to translate, and they are called **strings**, or text strings.

> **Ren'Py's Second Commandment:** you will always close quotes. Unclosed quotes (or unopened quotes, for what is worth) will prevent the game from starting.

|Top|

1.3.- How (I think) Ren'Py works? The third commandment

Assuming that I am not an IT expert and it is very likely that I will say something stupid, I will try explain very quickly what happens when we run a Ren'Py game.



Do you remember the list of cloned files within the "game" folder? Well, they are not exactly clones. Ren'Py runs the .rpyc files, which are a compilation of the editable files saved with a .rpy extension. This compilation occurs for the first time when developers open the game in their development environment. An AST (abstract syntax tree) is generated based on the content available at that moment and this AST will be the base for everything that will be added later in successive compilations. Following its algorithms, Ren'Py will assign an identification to each programming element of the .rpy files based on its position in the script and its own nature (text, variable, function, etc.), and with those identifications it will create .rpyc files with the same name that will be read during the game.

Every time the game is launched, Ren'Py looks for .rpy files and, if any, compares them with their respective .rpyc, looking for the identifications created during the initial compilation. If there have been any changes introduced to the .rpy file since the last time the .rpyc file was compiled, Ren'Py will detect both the unchanged elements and those that have been relocated, it will respect their previous identifications and will update the .rpyc file with the changes. Simplifying a lot, let's imagine that line number 3 of the .rpy file is initially compiled in a way that makes this line to be identified with a “3” in the .rpyc file. If in future versions of the .rpy file that line is not modified but becomes line number 7, it will stay as “3” in the .rpyc file. Thus, even though the final .rpy file could have hardly any resemblance to the first one created by the game developer, its corresponding .rpyc file will always be compiled following the relationships that were created in the first AST that was originated, as some of those initial elements are still there.

If we delete them at some point, Ren'Py will generate new .rpyc files but using the current version of the .rpy scripts. And this means that the relationships and identifications that will now be created in the AST will not be the same as in the original files, because they are being created from a .rpy script that looks anything like the first one. Taking that absurd example of before, line 7 of the .rpy file that was called “3” in the deleted .rpyc because it was being carried over from previous versions, would now be identified with a “7” as the AST is being generated from scratch. The game will start and new games shouldn’t be affected, but functions that use the AST references, such as loading games saved in previous versions, might stop working properly. And, in certain cases that we will see later, translations could also suffer some problems.

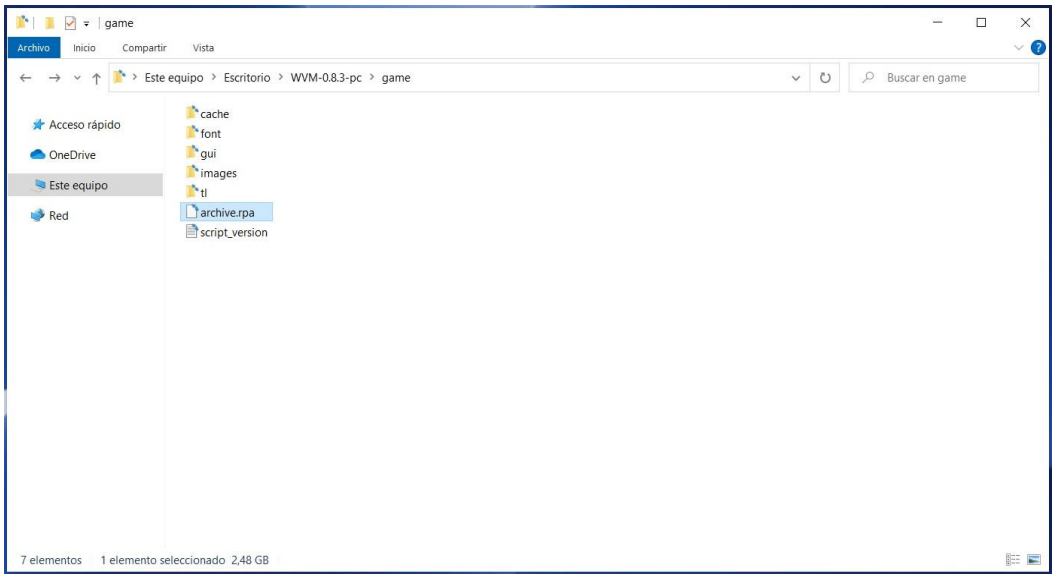
> **Ren'Py's Third Commandment:** Don’t delete .rpyc files from the original game, as this can originate unintended problems to players.

|Top|

1.4.- UnRen and .rpa archives

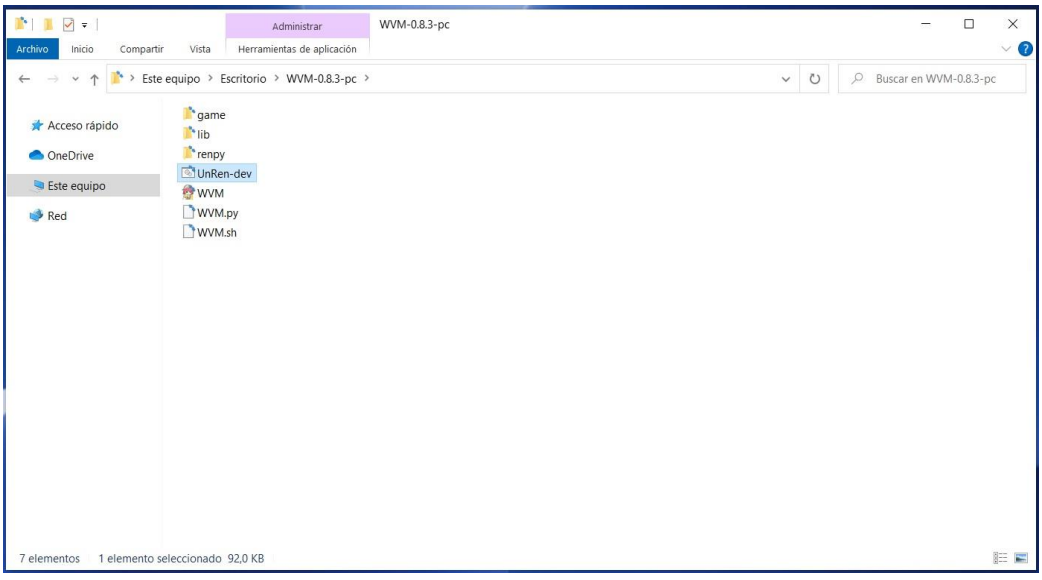
It's possible that, when you opened the "game" folder, you may have not seen anything of all that I just mentioned. That's because, when it comes to build the game for release, Ren'Py offers developers several options. The recommended one is to include all the individual scripts both in .rpy and .rpyc format, as we have seen in the "What a Legend!" example, but that's just a recommendation. As a Ren'Py game only needs its .rpyc scripts to work, some developers only include the .rpyc scripts. That way the game has a slightly smaller size and players have no direct access to the game's code (which might be a more important reason).

And there's another possibility (a quite common one, I'd say), as we may find out that in the "game" folder there are no scripts with either .rpy or .rpyc extensions, as the developer has chosen another option given by Ren'Py to build the game: compressing the scripts and other files (such as images) into a .rpa file. That way, instead of a list of .rpy and .rpyc files, we could have something like this in the "game" folder:

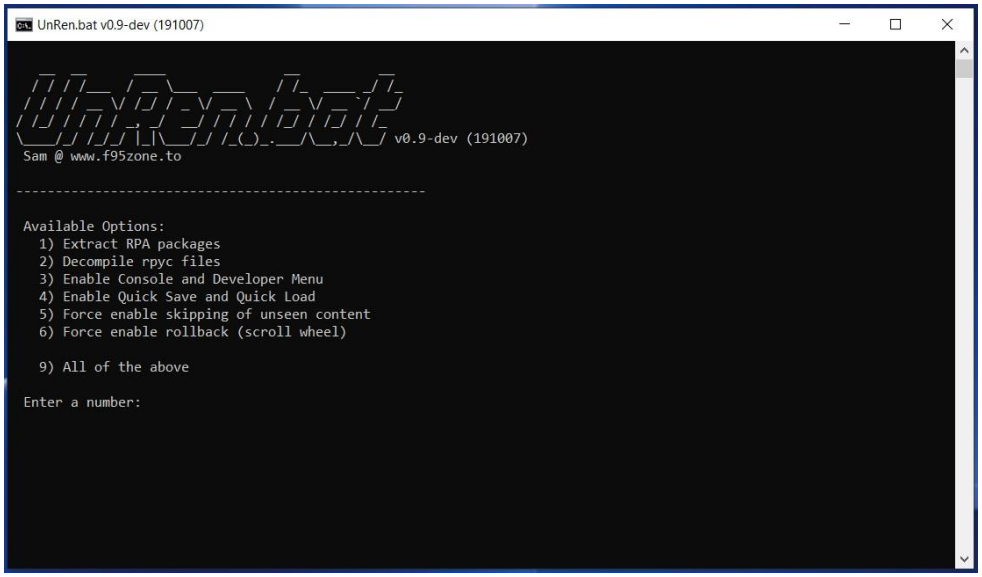


First of all, we can confirm that, as you probably already know, this is not a problem. The game will run perfectly as it is, but in order to translate it we will have to give some extra detour, since we need the scripts in .rpy format. Fortunately, there are several tools that perform the necessary tasks for it without us having to learn Python language. For its simplicity, I think the most manageable is **UnRen**. LINK *(beware: link to a forum with plenty of adult content)*

Once UnRen is downloaded, we unzip it and paste it into the root folder of the game of our interest, at the same level as the game executable. I guess it's easier to understand if you see it:



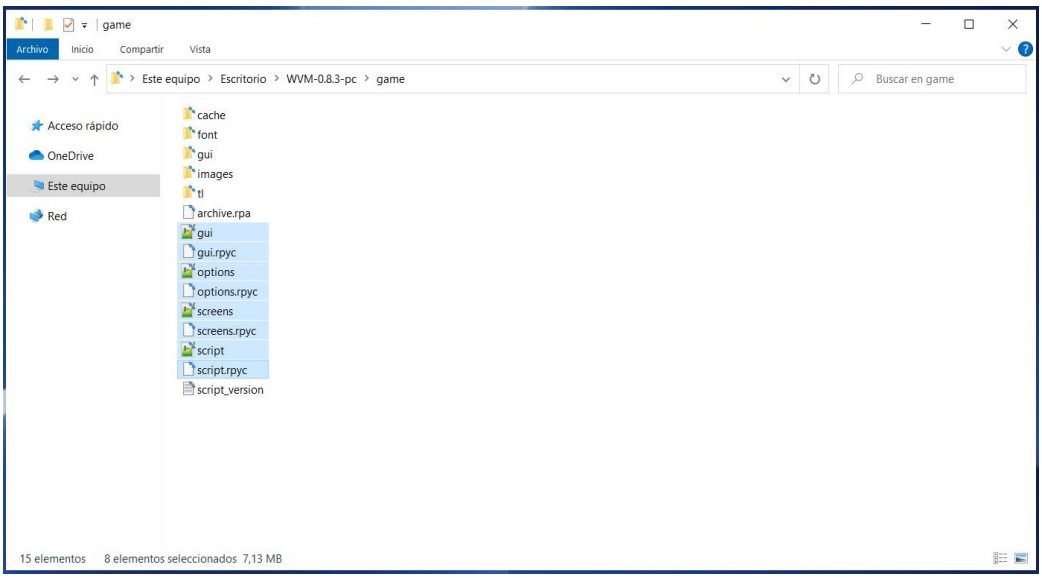
Then it's only a matter of running it and select what we want to do. This is UnRen's start screen:



As you can see, here we are only going to use the keyboard. We just need to press the key corresponding to what we want to do. Option 1) will "unzip" the .rpa files, extracting all its content in the "game" folder. It may be the case that within that .rpa file there were only scripts with a .rpyc extension, so, after finishing that option 1), we would have to ask UnRen to execute option 2), which consists in decompiling the .rpyc files to create some scripts in .rpy format that we could already use for our translation.

We can simply select option 9), which in a single step does the same as 1) and 2) and also enables the option to open the game's console and the developer menu (that's useful to examine and modify variables, to access specific parts of the script, and also to reload the game automatically when we introduce a change in the translation), as well as other options that the game's dev might have disabled.

Well, we let UnRen to do its job and at the end, if we go back to the "game" folder, we will see that, where previously there was only a file with a .rpa extension, there now appear the scripts that came compressed inside it.



Now we could start translating. But, firstly, let me use a paragraph to explain what is going to happen with the game. Now, inside the "game" folder there are scripts with .rpy extension, scripts with .rpyc extension and, in addition, inside the "archive.rpa" file there will be those very same .rpyc scripts (and perhaps even the .rpy ones). So now we do have cloned files.

Fortunately, Ren'Py is designed to solve these cloning's issues in an easy way: **if it finds two scripts with the same name, it runs the most recent one**. And, in this case, the most recent ones are those we've just extracted with UnRen. Therefore, we could delete the "archive.rpa" file (and it would be even recommended, as long as we have a way to recover it later, just in case).

And yet another clarification before going on. If the .rpa file contained both the .rpy and the .rpyc scripts, the ones we see now in the "game" folder are exactly the same that came from the game developer's computer, as they have just been extracted by UnRen. But if in the .rpa file there were only scripts with .rpyc extension, the .rpy scripts that we now have in the "game" folder are just an approximation created by UnRen based on the identified algorithm used to compile the .rpyc scripts. So they may not be exactly like the original .rpy scripts: they are just suitable to create the current .rpyc scripts. For example, all comments that the developer would have jotted down after a # symbol are lost, since they are never compiled in the .rpyc scripts and logically UnRen, reading only the .rpyc's, cannot tell if there were any comments in the original .rpy files.

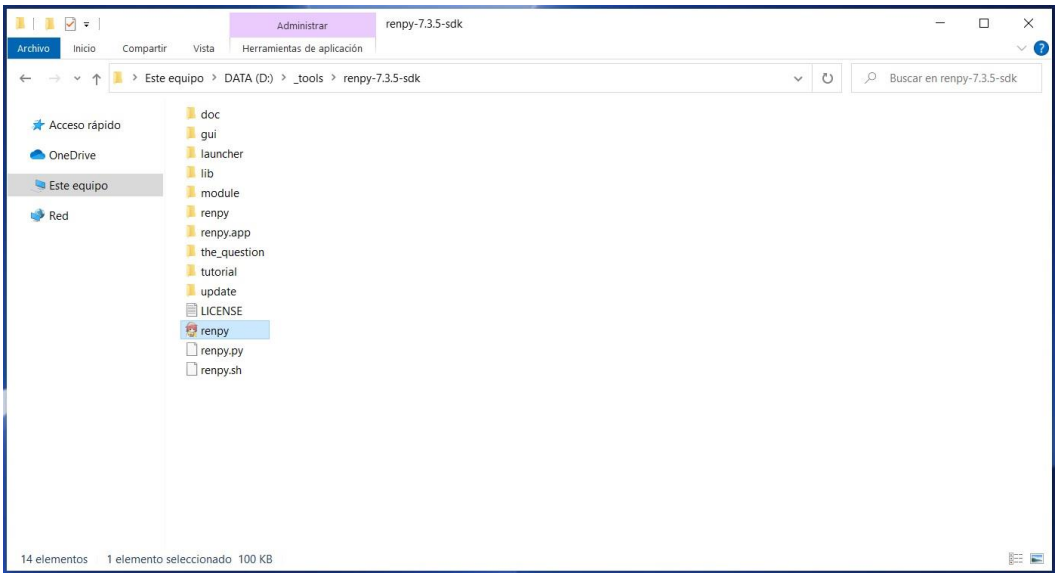
|Top|

1.4.- Ren'Py SDK

As we have seen, Ren'Py games run without installing any extra software to read them. They are self-executable and also allow us to modify their scripts with a simple text editor. Logically, however, to create them you do need a very specific software.

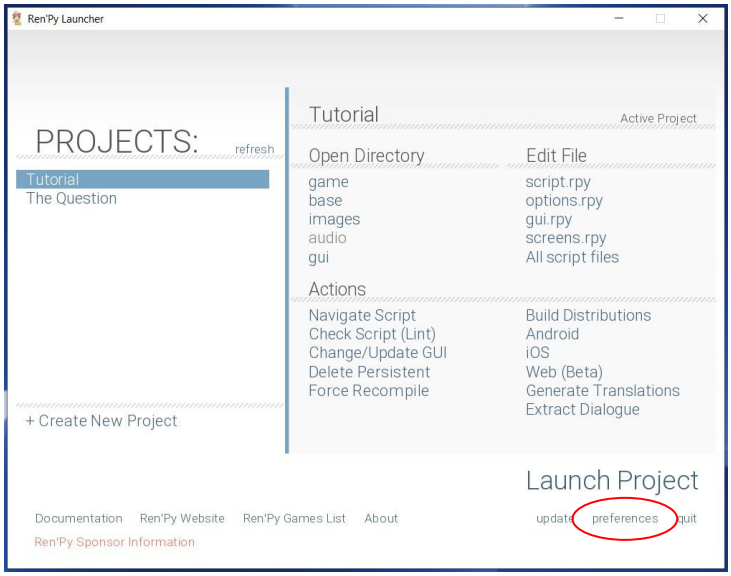
Ren'Py SDK is the application that allows us to create Ren'Py games - and their translations too. So the first step to translate any Ren'Py game is to download it. It's free, clean and fully trustable. LINK

Once you have downloaded and extracted it to your computer, open its folder and you'll see something like this:



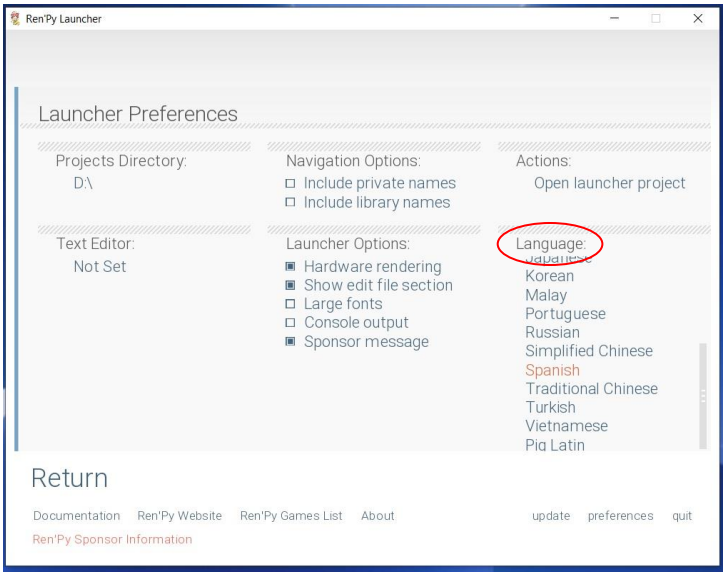
That is, a very similar structure to that of games, with several subfolders and an executable. Among the subfolders we have "the_question" and "tutorial", which are two tutorials to learn how to use Ren'Py. It doesn't hurt to take a look at them, but they do not add much to the translations.

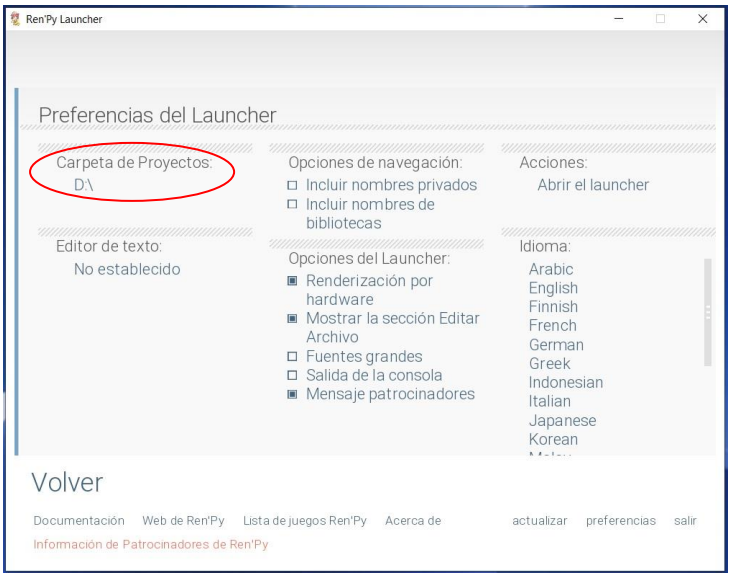
We just need to run the executable named "renpy" and start to work. After setting up some parameters, the main screen will appear, usually in English. So, first of all, let's change the app language to our own language.



First, we have to click in “preferences” at the bottom right of the window.

Under the label “Language”, on the right side of this “preferences” screen we will see a list of available languages. Ren’Py SDK can be run in all those languages, so we just have to look for our desired one. As soon as we click on it, the screen’s language will change.





Now we have the software running in our language (Spanish, in my case) and we won’t need to change it ever again. The circled option, called "Projects Folder”, is the directory where Ren'Py SDK will store and search for games. Here we need to select the directory where the root folder of the game we want to translate is located. That is, if you have "What a Legend!" in C/Documents, then select C/Documents. Click on "Back" and all games in that folder will appear in the “Projects” list, in addition to the two tutorials that are included in the Ren'Py SDK.

[Top]

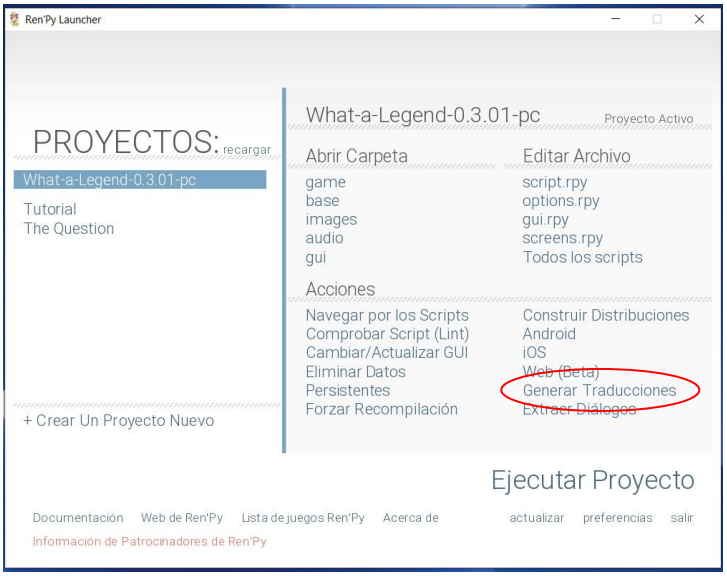
2.- LET'S TRANSLATE!

Either because the developer kept it easy for us, or because we have fight our way to achieve it, the important thing is that, at this point, we have our game with its .rpy scripts totally exposed and ready to be translated. But, unfortunately or not, Ren'Py doesn't translate anything, it just helps to extract the translatable texts and makes the translations appear where they belong. Ren'Py SDK generates the scripts that we'll use to create the translation, with all the necessary commands; however, getting it to fully extract all the translatable texts is not always as easy as clicking on a button, and sometimes we will have to do a deep editing job of the original scripts. But, since we have come this far, we are going to click on that button that surely you are already looking with greedy eyes: Generate Translations.

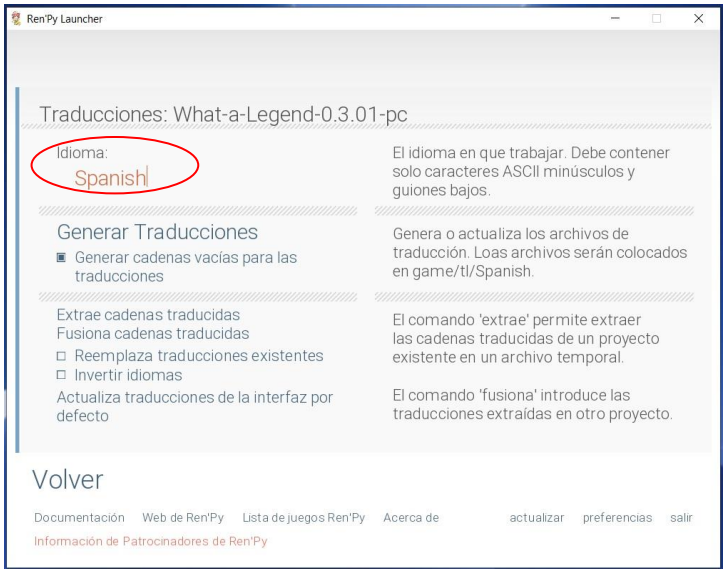
|Top|

2.1.- Generating translation files (and learning the fourth commandment)

After selecting the game that we want to translate (it will appear highlighted in the "Projects" section), click on the "Generate Translations" button on the right side of the screen.



In the "Language" box we must write the language we want to translate the game into. It's just an internal identification, so we can write whatever we want (note that no special characters are allowed, such as ñ or accented vowels). The most important thing is to remember what we have written here and to keep always in mind Ren'Py's fourth commandment:



> **Ren'Py's Fourth Commandment:** For Ren'Py, a capital letter is not equal to a lowercase letter, never, under any circumstance.

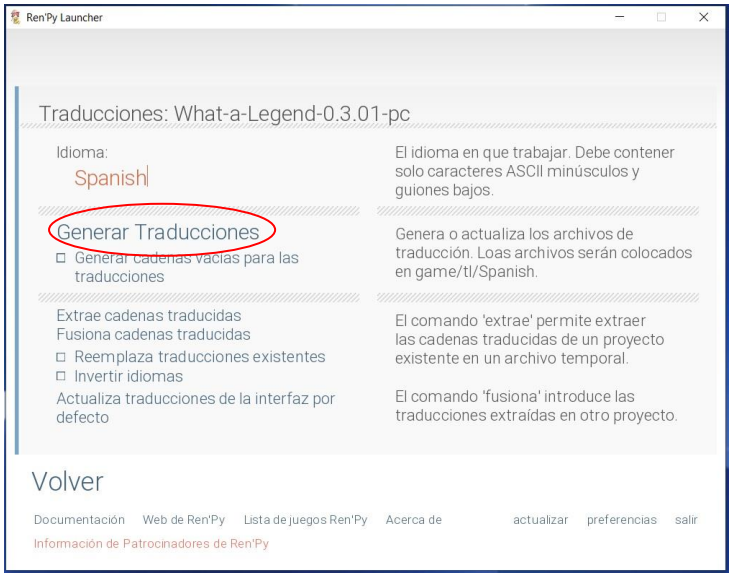
So you can write Spanish, spanish, espanol, French, french, francaise, ship, yellow or Monkey. Whatever you want. But, obviously, it is advised to use an understandable word. And, as I say, the most important thing is to remember what did you write there and how did you write it. For instance, I always use the word "Spanish", with a capital “S”, and that’s what you will see from now on in the examples provided.

Then there are several options, but the only one that matters to us is the one that reads "Generate empty strings for translations” (or whatever is said in your language). Here, as almost always, it’s all a matter of tastes. When translating, we will always have a visible line with the original text to guide us, but the idea is that, if we select that option, the translation scripts will be generated with a few blank lines for us to directly write our translation on them. If we do not select it, the scripts will be generated with those lines written in the game’s original language and we’ll have to delete those words and replace them with their translation. Here we can see a side by side comparison of how the translation files would look if we check that box (left) and if we don’t check it (right).

```
# TODO: Translation updated at 2020-12-11 13:16

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin ""

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov ""

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin ""

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin ""
```

```
# TODO: Translation updated at 2020-12-11 13:41

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "STOP!" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin "Did you manage to get a passage permit?"

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov "Umm..."

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin "I'm sorry, buddy..."

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin "...but no permit, no passage. That is the law."
```

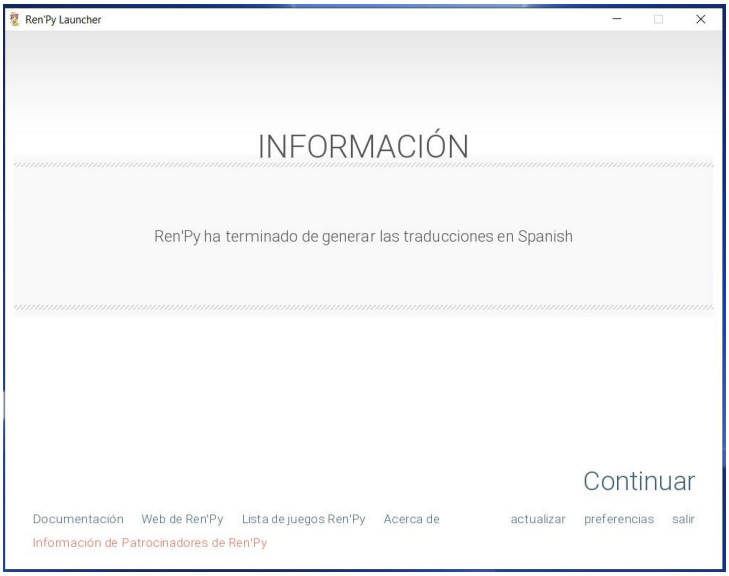
It may seem more convenient (it actually is) to generate empty strings, but, if somehow we forget to translate some line, when players reach that line in the game, absolutely no text will be displayed on screen. The other way, the text would be displayed in the original language, and that can be useful for us when testing an incomplete translation. This is especially important in menus, as it will allow us to have all the options active even when we haven't translated them yet.



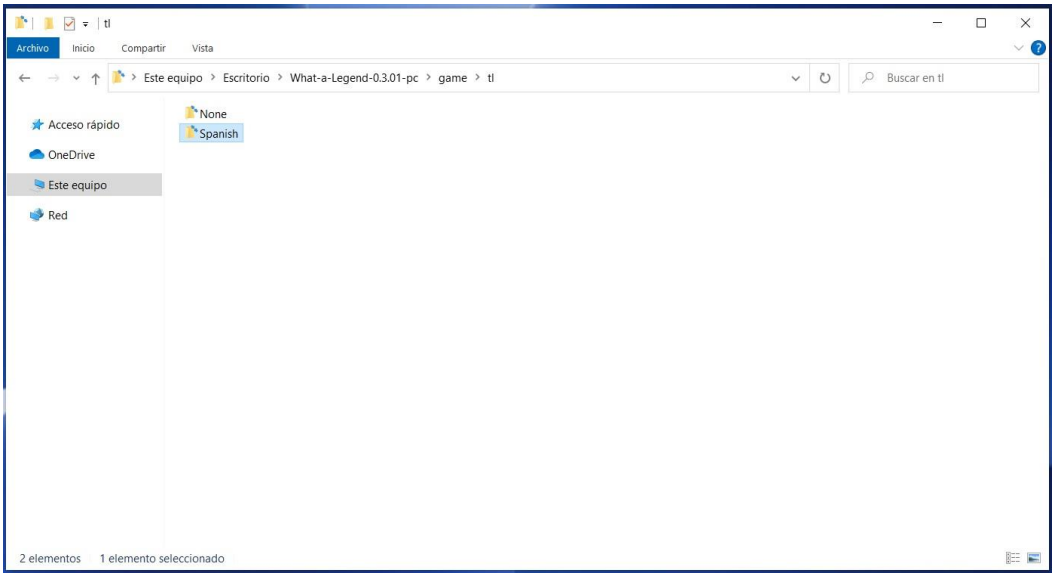
Personally, I prefer NOT to generate empty strings, so I leave that box unchecked, but if you are going to use machine translators you may be interested in generate empty strings.

Now we just have to click on the "Generate Translations" button (the circled one). As the informative text in the right column indicates, by doing so Ren’Py SDK will create the translation files inside the “game/tl/Spanish” folder (the name of said folder will be the one we have written in the "Language" box).

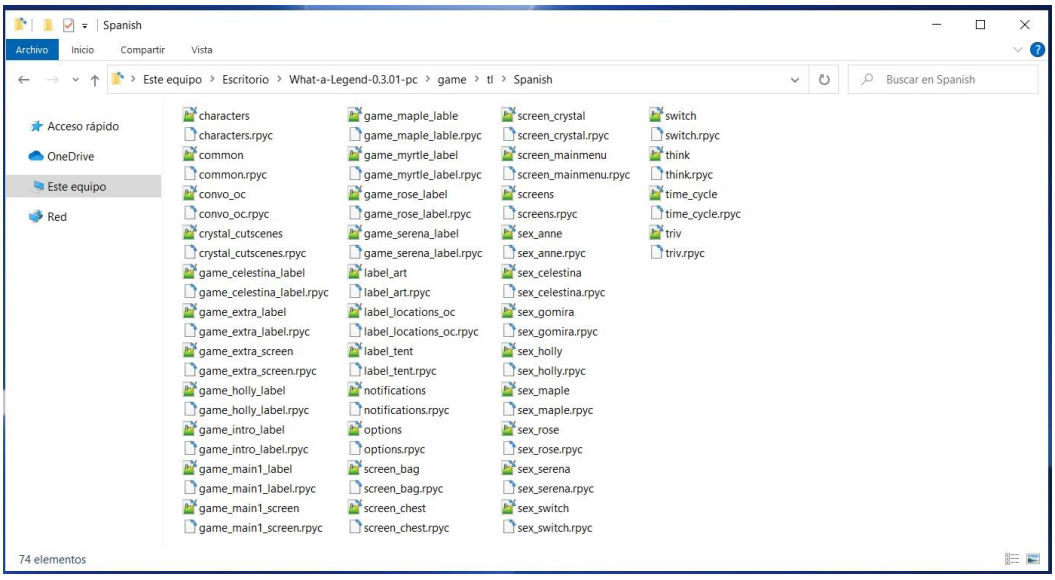
If there are no further problems, when Ren'Py SDK finish working its magic this message will appear. We can click on "Continue", to return to the main screen, or directly close the Ren'Py SDK app.



If we go to the "game" folder of our game, in the "tl" subfolder we will see two other folders: one named "None" and another one with the word we wrote in the Ren'Py SDK's language box. In this case, "Spanish":



And inside the "Spanish" folder we will find the translation scripts. What has Ren'Py SDK done to create them? Well, it has gone through all the original .rpy scripts, one by one in alphabetical order, and, whenever it found some translatable text in them, it has generated a .rpy file (and its corresponding .rpyc) with the same name as the original script, writing in it the strings that need to be translated.



Also, an additional script will have been generated: named "common.rpy", it will contain the translatable commands of Ren'Py menus that are not specific to the game. For instance, the alert message that appears when we close the game, asking us to confirm our choice, or the months and days of the week that will be used to name our saved games. It is a large file and a quite tedious one to translate, but there's a silver lining: usually, it's common to most games created with the same version of Ren'Py SDK, so it can be interchangeable from one game to another. So we can reutilize a translated "common.rpy" script that we already have in a previous game. Of course, before replacing it, we should always check that the strings match, as sometimes there are small changes and we could break something unintentionally.

At this point, translation it's only a matter of opening the .rpy scripts of the "tl/Spanish" folder with a text editor and start to work on them. We can change the name of these scripts (and even merge them in a single file), since Ren'Py will look for translations string by string and does not care in which file they are stored, but the standard practice is "don't touch anything": that way it's easier to identify each translated script with its original script.

|Top|

2.2.- The two translation functions (and yet another commandment)

It's not unusual that, the first time we start translating, we write a few lines in our language and then suddenly think: "Wait, where are the options for the player? Didn't the game ask me here if I wanted to do one thing or another?" Don't panic: in translation files, translatable texts do not appear in exactly the same order as in the original script. When it comes to extracting the translatable strings from each script, Ren'Py divides them into two categories: those that form the dialog block and those that refer to menu options, variables, character names, etc. Why? Well, because each one of these groups is going to use a different extraction and translation function.

The strings that form the dialog block will not generate any extracting issue because Ren'Py SDK will identify them without problems. What it does with them, however, will cause more of a nuisance and even some headaches when translating and creating our translation patch. Because Ren'Py encrypts them to save memory: using the MD5 hexadecimal encryption method to 8 bytes, each string is identified with an alphanumeric code that depends on the label where that text is located in the original script and the content of the string itself.

We can see it better with an example. First of all, let's remember the first lines of the "convo_oc.rpy" script of the game "What a Legend!":

```
# ==================== OLD Capital Base Conversations ==========================================
# Being stopped at the gate of the old capital ================================================
label convo_gate:
    call hide_ui from _call_hide_ui_104
    call silent from _call_silent_42

    scene scene_oc_bridge_gate_talk
    if current_hour == "Night" or current_hour == "Evening":
        show kevin at g_cright:
            xalign 0.6
        show pov at m_left
        with quickfade
        show pov ewide bup mdislike hfshock hbshock
        show kevin hbstop eangry
        kevin "STOP!" with vpunch
        show kevin hbpoint edoubt
        show pov -mdislike hfneu1 hbneu1
        kevin "Did you manage to get a passage permit?"
        show pov mno bdoubt eneub hfhead
        show kevin eneu hbneu
        pov "Umm..."
        show pov eneu bsad msad
        show kevin esad hfspearmove
        kevin "I'm sorry, buddy..."
        show pov mcry eflat bneu hfneu1
        kevin "...but no permit, no passage. That is the law."
```

The first line with translatable text (the first "string") is line 15 of the original script: under the "convo_gate" label, a character identified as Kevin says "STOP!" (with vpunch, an effect that will "shake" the player's

screen). The rest of the code up to that point does not contain any text to be displayed on screen.

When generating the translation script with empty strings, Ren'Py has transformed all that info on this:

```
# TODO: Translation updated at 2020-12-11 13:16

# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch

# game/convo_oc.rpy:18
translate Spanish convo_gate_e5ccb99b:

    # kevin "Did you manage to get a passage permit?"
    kevin ""

# game/convo_oc.rpy:21
translate Spanish convo_gate_bc693870:

    # pov "Umm..."
    pov ""

# game/convo_oc.rpy:24
translate Spanish convo_gate_6a0bcf57:

    # kevin "I'm sorry, buddy..."
    kevin ""

# game/convo_oc.rpy:26
translate Spanish convo_gate_26193626:

    # kevin "...but no permit, no passage. That is the law."
    kevin ""
```

We can see that, for each string, Ren'Py has generated a four-lines block. Let's analyze those blocks:

```
# game/convo_oc.rpy:15
translate Spanish convo_gate_fdd309da:

    # kevin "STOP!" with vpunch
    kevin "" with vpunch
```

Lines that start with the # symbol are merely informative and we could delete them without consequences. The first one tells us the location of that line in the original script: it is line 15 of the file "convo_oc.rpy" file, inside the "game" folder. The other line started with # is the original text to be translated, which is kindly offered by Ren'Py so that we know what to write in our language. And the last line is where we will be writing our translation (we'll only translate what is in quotation marks, we should leave the rest as it is). If we had generated the translation without empty strings, in this line the text would appear again in English.

The key is in the second line. This is what activates the translation function for this specific line of the original code. When we play a translated version, Ren'Py is running the game according to the original .rpyc files from the "game" folder, but it has been ordered to display the translated texts instead of the original strings. That command tells Ren'Py that, in each line of the dialog block, it has to search the scripts looking for a line with the command "`translate`" plus the active language ("`Spanish`", in this case, which is what we wrote when generating the translation files) plus the encryption code that corresponds to the original string. This encryption code starts with the name of the label where this specific line is located ("convo_gate") and continues with the result of applying the MD5 system to the line's content (apparently in that encryption system, "STOP!" equals to fdd309da).

If there were another identical string (another "STOP!") within that same "convo_gate" label, Ren'Py would have to assign it the same encryption code. But this could lead to conflicts, so it will add a _1 at the end of the second line's code, if there were yet another identical string it would be added a _2 and so on. Thus, the encryption codes are always unique for each string, even for identical strings within the dialog block.

But what if we go to the original script and now write "Stop!" instead of "STOP!"? Well, then the translation we have for "STOP!" will stop working, because the MD5 code of "Stop!" will be a different one and Ren'Py won't find it in the translation scripts, so the original text will be displayed instead. This causes some annoyances when translating new versions of games still in development, as it's common for creators to correct minor typos and rewrite a few sentences here and there. We will have to translate again certain strings that we had already translated before, because even the most minimal change will change their encryption code entirely and they will be completely different for Ren'Py engine. This also happens when its

label is renamed, even if the string's content is not modified.

> **Ren'Py's Fifth Commandment:** Any minimal changes made to the original string will invalidate that line's pre-existing translation.

Luckily enough, if what happens is a simple line relocation (the string goes from being on line 7 to be on line 8, but its content does not change and neither changes its label), the previous translation will keep working, since this relocation doesn't imply any change in the encryption code. The #commented line that tells us where that string is located in the original script will be outdated, but that has no major impact on our translation.

But, as I say, Ren'Py uses two translation systems. In addition to the code-based encryption, for the rest of strings that do not belong to the dialog block, a simpler system of direct translation is used. What strings are we referring to? Well, as I said at the beginning of this section, I'm talking about the options that are presented to the player during the game, but also about character's names, potential variables used in the game that have a text value, system menus (preferences, save and load game...), and so on.

When Ren'Py is running a translated game, all those strings will be replaced on screen by direct substitution, without any detours or encryptions. To achieve this, when generating the translation scripts, Ren'Py SDK groups them all at the end of the document, under the command "`translate Spanish strings`". This is the beginning of the second section of the "convo_oc" translation script from "What a Legend!":

```
8001 translate Spanish strings:
8002
8003     # game/convo_oc.rpy:107
8004     old "Permit"
8005     new ""
8006
8007     # game/convo_oc.rpy:107
8008     old "Old Capital"
8009     new ""
8010
8011     # game/convo_oc.rpy:107
8012     old "Helping pixies"
8013     new ""
8014
8015     # game/convo_oc.rpy:107
8016     old "Dungeon"
8017     new ""
8018
8019     # game/convo_oc.rpy:107
8020     old "Go back"
8021     new ""
8022
8023     # game/convo_oc.rpy:320
8024     old "Passage permit"
8025     new ""
8026
8027     # game/convo_oc.rpy:320
8028     old "Challenge"
8029     new ""
```

As you can see, here there are no strange codes. And if the "`Translate`" command was applied before to each line, now all of them are translated using just one command. We still have a first line with the # symbol to inform us about the location of the translatable string in the original script, but then it forgets about encryption codes and goes straight to show the original text with the "`old`" command; and then we have a "`new`" command before the text that should appear when the game is running a translation into the "`Spanish`" language (the one we wrote in the Ren'Py SDK when we were generating the translation scripts).

Unlike before, now no repeated string will appear not just in this script, but in the whole set of translation scripts. When Ren'Py SDK goes through the original scripts to generate the translation files, it will only extract each string the first time it is found and will overlook the duplicates that could potentially exist. So, when playing, whenever that string appears it will be replaced by the translation that we have introduced that one time. This saves us from repeating work, but becomes an inconvenient when we come across identical strings that may have different meanings depending on the context. For example, think of a string that is repeated several times during the game but it just says "Right": sometimes it could mean "Correct" and sometimes it can be referring to the right side of something, or indicating a direction, and maybe in our language we use a different word for each concept. But Ren'Py SDK would only extract the first "Right" string, so we could only translate it once and that one translation would appear always, so sometimes the translated text would not make any sense. We will see how to solve this problem later.

Using the example above, once we write in this script the translation for the string "Permit", that translation will appear throughout the game every time there is a string "Permit" outside the dialog block. But beware: according to Ren'Py's fourth commandment, if there were another string with the text "permit", we would need to translate it, because Ren'Py is perfectly case-sensitive.

Back to the topic, why does Ren'Py use two different translating functions? Well, for agility and efficient memory usage issues. Technically, this "direct replacement" translation system could be applied to absolutely all the strings in the game, but usually the dialog block is quite large and consists of much longer sentences that are hardly ever repeated. So when the program has to show a translation, it takes less time to find and replace thousands of encrypted codes than thousands of whole lines that are usually longer than those codes. However, these other menu strings are generally shorter and much less numerous, and it's possible that they could be more frequently repeated in the scripts, so Ren'Py can afford to do a specific search for the exact content of these strings without noticeable lags during gameplay.

Anyway, these are just some perfectly forgettable notions. The most important thing is to know that, for Ren'Py SDK, there are two types of "strings" and each one of these types uses a different translation system, and that's why they will appear separately in the translation scripts: first we'll find all the strings belonging to the dialog block, and then all those corresponding to menus, options and variables.

|Top|

2.3.- Helping and/or replacing the extractor

The reason for explaining in the previous point the two types of Ren'Py's translation functions is that, unfortunately, Ren'Py SDK is not always able to detect absolutely all strings that we should translate: yes, it will extract all those that integrate the dialog block (those that will be translated thanks to the encryption code), but it may not be able to identify all the strings that are translated by the direct replacement method (although it will extract the options that allow players to make decisions during the game).

For instance, to make Ren'Py SDK able to automatically extract characters' names, the game's developer should have defined those characters in a very specific way that, due to ignorance, many devs do not use. But if they don't, we can do it.

As the creators of "What a Legend!" did their homework, making life much easier for us translators, for these examples we are going to use the other game that we had already used to talk about .rpa files and UnRen extractor. So let's see how the "WVM" developer defines one of the characters involved in the story. In this case, in the file "script.rpy" he has created an "MC inner self" to indicate players that what we read in the dialog box is a thought of our own character:

```
224 define mcm = Character ("Your thoughts",color="#FFFFFF", who_outlines=[ (2, "#000000") ], what_outlines=[ (2, "#000000") ])
```

This is a typical line of Ren'Py code. It starts with a command (`define`) that indicates what this specific line does, in this case defining a variable. That variable is assigned a name (`mcm`) that will be used in the rest of the script to identify it, and it is ordered to behave as a character type variable (`Character`). Now Ren'Py knows that, whenever it finds the letters `mcm` before a string, it has to show a name above the textbox so the player knows which character is talking. And what will be shown there? What appears afterwards in parentheses: the character's name ("`Your thoughts`"), in a specific color, with a line surrounding the letters to highlight them.

Obviously, "Your thoughts" is a string that we should translate, but, if we generate the translation scripts with Ren'Py SDK, it would not appear anywhere. Ren'Py Mysteries. What can we do? Well, there are three options. The first option, which we plainly discard, is to just accept it and let the string to be always displayed in English. The second option is to make Ren'Py SDK understand that this quoted text is a string we want to translate and not just an internal code like the one that indicates the text color. Look:

```
224 define mcm = Character (_("Your thoughts"),color="#FFFFFF", who_outlines=[ (2, "#000000") ], what_outlines=[ (2, "#000000") ])
```

We have edited the original script to put the string "Your thoughts" in parentheses and we have typed an underscore _ before the parentheses. This combination of `_( )` symbols will allow Ren'Py SDK to identify the quoted text inside parentheses as a translatable string.

And if we generate now the translation scripts, we would find this in the direct translation's section:

```
translate Spanish strings:

    # script.rpy:224
    old "Your thoughts"
    new "Tus pensamientos"
```

That's the result after translating it into Spanish, of course. But the important thing is that, hadn't we edited the original script to replace "`Your thoughts`" for `_("Your thoughts")`, this string would not appear in the translation scripts.

That doesn't mean that we could have never translated it: as it is a direct translation, we could always write it in our translation script without Ren'Py SDK's help. This is the last of the three options I mentioned before: instead of editing the original scripts, we write in the translation scripts the translatable strings that Ren'Py SDK did not detect. To do this, we would go to the translation script's section where direct translations appear (the one that starts with the function `translate Spanish strings:`) and we would write a translation function with the same format we've seen above: always respecting the 4 spaces indentation, as well as quotes as the first two Ren'Py's commandments say, in a line we would write the command `old` followed by the string to translate (scrupulously respecting its writing, as Ren'Py's fourth and fifth commandments say), and then we would write below the `new` command followed by the translation. The line that starts with a # symbol wouldn't be necessary as it is merely informative.

Personally, I'm used to review and edit the original files first so that Ren'Py SDK can then do a full extraction job. Of course, sometimes I miss the occasional string and I have to regenerate the translation scripts several times, which is no more hassle than having to open again the Ren'Py SDK app and click on generate translations. Other translators prefers to edit the original scripts as little as possible, so they generate these translation scripts at the beginning and then modify them manually to include everything that Ren'Py SDK didn't extract. It's just a matter of personal tastes and working habits.

Whether you want to help the Ren'Py SDK beforehand or you prefer to write in the translation scripts the missing strings, you will have to go through the original scripts and look for these types of commands:

- `Character("...")`, used to define characters, as we have seen in the example
- `text "..."`, used to display text on a specific screen, outside of the textbox
- `textbutton "..."`, used for texts that perform an action when clicking on them
- `tooltip '...'`, used to display a pop-up message when hovering over a point
- `renpy.input("...")`, used to allow in-game typing (to let players name characters, usually)
- `$ renpy.notify('...', 'unlock')`, used to display messages after completing an action

In addition, we have text variables. Here we can find several possibilities, but none of them will be automatically extracted by Ren'Py SDK:

- `default` *variable_name* = "..."
- `define` *variable_name* = "..."
- `$` *variable_name* = "..."

So it's a matter of wrapping all those quotation marks with the `_()` symbol and generate the translation scripts, or copy its content (quotations included) directly into a translation script (no matter which one) with the `old` command and the `new` command below with its translation. Note that you have to copy everything that appears in quotation marks, not just the text to be translated, since sometimes there are tags inside `{}` symbols that are used to display the text in **bold**, *italics*, with another font or different size, etc. Those tags are also part of the string that Ren'Py will search and replace, so they have to appear behind the `old` command in order to help Ren'Py to correctly indentify that exact string.

Finally, just an extra remark: the `_( )` symbol **is only used to extract the strings** in parentheses. Once extracted to a translation script, we can delete that symbol and the translation will work fine, just as if we write those strings manually in the translation script. Therefore, when translating new versions of the same game, editing the original scripts again would not be necessary as we can reuse the old translation scripts in which those strings appear already translated. Also, there's no need to introduce that edited script in the translation patch, since players don't need the `_( )` symbol: once the translatable string appears in the translation script, the text will be displayed translated on screen. I'll admit that I have learned this relatively recently and, had I known earlier, I would have saved myself from a few hours of unnecessary work.

|Top|

2.4.- The Language-change option

Last thing we should do before starting to translate, or perhaps the first one, is to think about how we are going to activate the translation in-game, once created. Do we want the game to always start in our language? Do we add the option to change languages in the main menu, or in the options menu? Do we prefer to let players decide in which language they want to play each time they start the game? And how do we do all of this, with a plain text question or creating a screen with flags and other visual elements? As always, it depends on translator's personal tastes, available time and coding skills, and obviously also on the original game's setup.

The simplest way is to make the game always start in our desired language. To do this, we just need to open any game's script (preferably the "gui.rpy" script, as much of the game's config is defined there, but it actually doesn't matter and we can even create a new one) and write this line of code, without indentation:

```
define config.language = "Spanish"
```

Where `"Spanish"` is, let's remember, the word I entered in Ren'Py SDK when generating the translation scripts; and thus the reference used by the "`translate`" commands, so you use the term you chose. If we don't do anything else, the game will always start in our language and players won't have any in-game option to switch back to the original language: even after disabling this line of code (deleting it or writing a # symbol at the start of the line) the game would no longer boot in the original language, as by default Ren'Py games always start in the latest language in which they were played. A new `config.language` should be established with the original language, which for Ren'Py is always named `None`.

Best practice, in any case, is to always add a language change option within the Preferences menu. Assuming the game uses the preferences screen that Ren'Py incorporates by default, you'd have to go to the "screens.rpy" original script, look for the section of the "Preferences" menu and make it look like this:

```
718 init -501 screen preferences():
719     tag menu
720
721
722     use game_menu(_("Preferences"), scroll="viewport"):
723
724         vbox:
725
726             hbox:
727                 box_wrap True
728
729                 if renpy.variant("pc"):
730
731                     vbox:
732                         style_prefix "radio"
733                         label _("Display")
734                         textbutton _("Window") action Preference("display", "window")
735                         textbutton _("Fullscreen") action Preference("display", "fullscreen")
736
737                 vbox:
738                     style_prefix "radio"
739                     label _("Rollback Side")
740                     textbutton _("Disable") action Preference("rollback side", "disable")
741                     textbutton _("Left") action Preference("rollback side", "left")
742                     textbutton _("Right") action Preference("rollback side", "right")
743
744                 vbox:
745                     style_prefix "check"
746                     label _("Skip")
747                     textbutton _("Unseen Text") action Preference("skip", "toggle")
748                     textbutton _("After Choices") action Preference("after choices", "toggle")
749                     textbutton _("Transitions") action InvertSelected(Preference("transitions", "toggle"))
750
751                 vbox:
752                     style_prefix "pref"
753                     label _("Language")
754                     textbutton _("English") action Language(None)
755                     textbutton _("Español") action Language("Spanish")
756
```

The exact code to write would be this (always remember to mark the indentations with the space bar):

```
vbox:
    style prefix "pref"
    label _("Language")
    textbutton _("English") action Language(None)
    textbutton _("Español") action Language("Spanish")
```

The `_("English")` string assumes the game's original language is English; we should write the right one but leaving the rest of the line as it is, since Ren'Py will always consider that original language as the `None` language. And, obviously, `_("Español")` should be replaced by your language's name in your own language, while the `action Language("")` command should contain the word you used to generate the translation scripts.

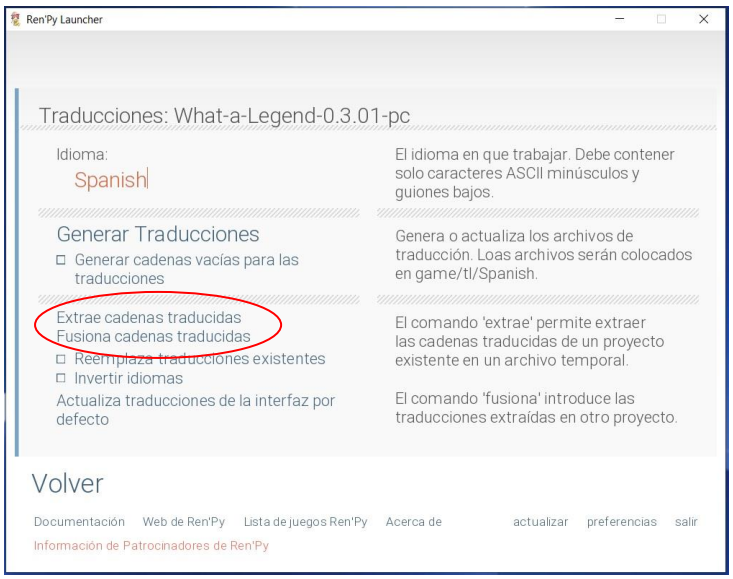
You may have noticed that there are several strings inside the `_( )` symbol, so Ren'Py SDK will extract them. Personally, I only translate the title, "Language", and leave the others untranslated. Why? Well, because that way the names of the languages will always appear written in their own language, regardless of which language we are playing in. For instance, if by any chance some player who doesn't speak our language find our translation and runs the game in our language, by clicking on the preferences menu they will quickly see the word "English" written in English and will easily understand that they can switch languages there.

As for other options, the truth is that there are as many as games and translators, so I'm not going to detail all of them. Just take a look at how the game's preferences menu is coded (if it's customized), do some reverse engineering, search online forums and use Ren'Py tutorials to create screens, splashscreens, imagemaps, buttons... There is a whole world of possibilities, as complex and visually attractive as you want.

|Top|

2.6.- Translating updates

In these times of Patreon it's very common for games to not be released as completed projects, but in episodic format, and we are forced to update our translations as new versions are released. The procedure doesn't have major complications... unless we want to do it the way Ren'Py SDK suggests. The official Ren'Py app offers us an option to extract the existing translations of a game's old version and insert them into the new one, updating the translation scripts with the new content. Nice, uh? Well, not so much. Let's see.



To do this the "official" way, we would launch Ren'Py SDK, we would select in the Projects folder **the previous version** of the game we want to translate (so, the old version we have already translated) and, in the Generate Translations screen, we would click on **"Extract Strings Translations"**.

Once that process ends, we would return to the Ren'Py SDK's start screen, we would select the game's **new version** (the one we want to translate now) and, back in this Generate Translations screen, we'd click on **"Merge Strings Translations"**.

Theoretically, at the end of the process we would have, inside the "game/tl/Spanish" folder of the game's new version, our previous translation mixed with this new version's new translatable strings. The problem is that this official procedure does not work as it should, and translations simply don't merge: if an original

script has not changed at all from one version to another, its translation script will be carried over from the old version to the new one, but if the original script of the new version does not match the old, Ren'Py SDK will generate an entirely new translation script without any trace of the pre-existing translation (even for those strings that remains unchanged), forcing us to repeat the translation work.

So the most effective way is just to download the game's new version, copypaste inside its "game" folder the "game/tl/Spanish" folder that we have in the previous version (that is, our old translation scripts), and generate the translation files for the new version as if it was a completely new game. If we choose the SAME language of the previous version (that is, if we write in Ren'Py SDK's language box the very same word we used before, so "Spanish", in my case), and we DON'T check the "Replace existing translations" option, Ren'Py SDK will simply update the existing translation scripts with the strings for which it cannot find a valid translation. At the end of each translation script, both the game's new translatable strings (the new content) and those strings that have been modified since the last translated version will be added, divided again in two blocks (first one for the dialog block strings and second one for options and menus). Sorting translation scripts by its last modification date will quickly allow us to see which ones have new content to translate.

This same procedure is used for updating our translation scripts after having edited the original ones to include the _( ) symbol where necessary, in case Ren'Py SDK didn't extract at first all the translatable strings. That is, we would edit the original scripts and we would generate a new translation **without replacing the existing ones**, and then Ren'Py SDK will include, at the end of the translation script, a new list with all the strings that had not been previously detected.

|Top|

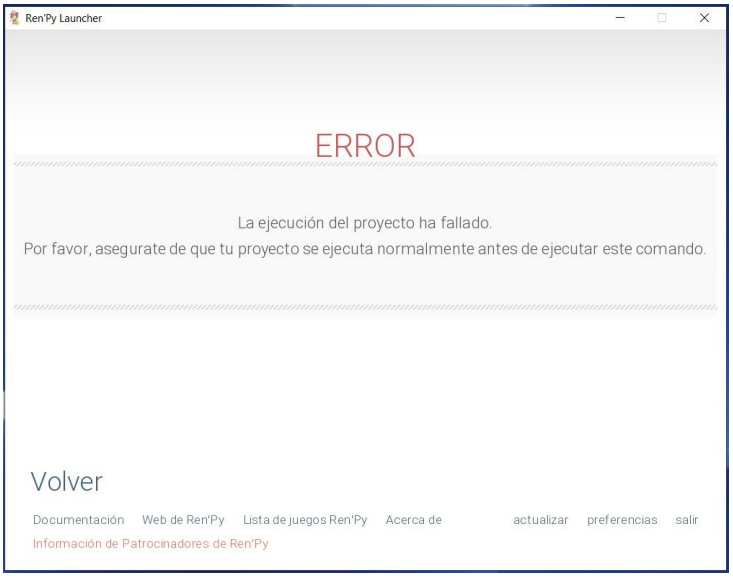
3.- WHY I CAN'T MAKE IT WORK

After telling you all of this, if you have been able to create your own translation and everything is working properly, then congratulations for being such an attentive pupil, and also for having chosen for your practices a quite simple game. Because the most normal thing to happen is that, when playing your translated version, every now and then some texts will still appear untranslated, even though you may have translated them in the translation scripts. Next, I will try to explain some of the most common incidents.

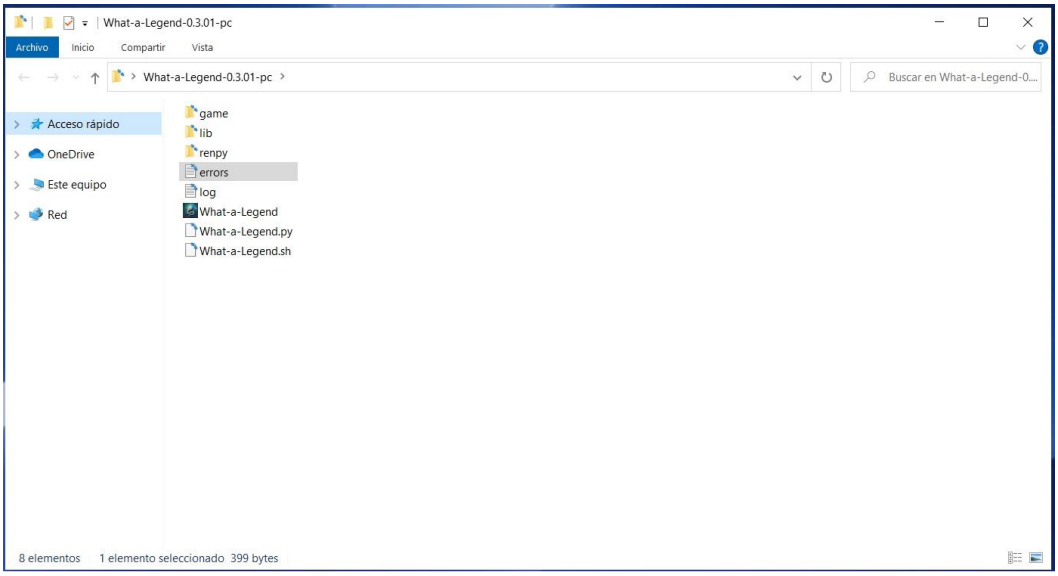
|Top|

3.1.- Bugs and errors detection

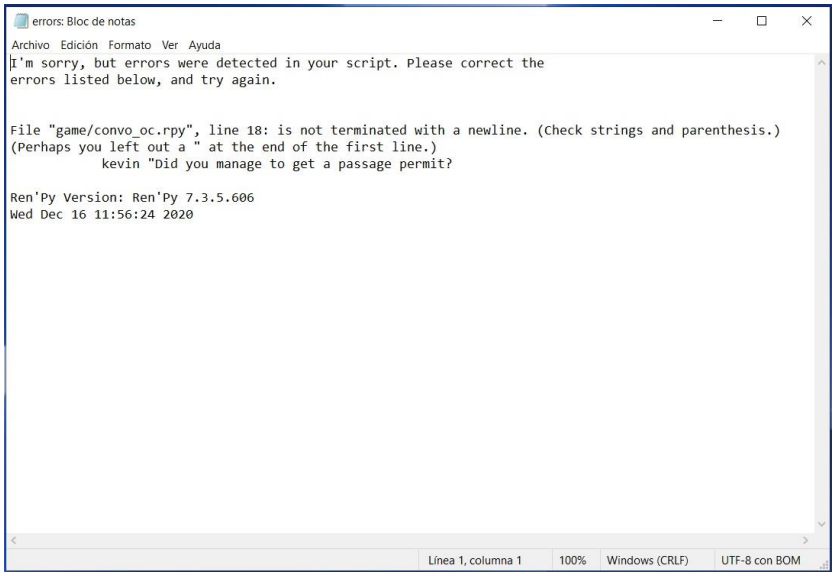
First things first, we should familiarize ourselves with the screen Ren'Py uses to warn us when something is wrong. In an ideal world, games would be released bug-free and without any critical errors, so the first fatal screenshot a translator might face should be the one that warns us that Ren'Py SDK could not generate the translation scripts. Which can only mean that we have made some mistake(s) while editing the original .rpy scripts. Here we have the Spanish version of that screen:



In order to see what exactly went wrong, we should check the root folder of the game. There, next to the game's .exe file, multiple .txt files will appear.



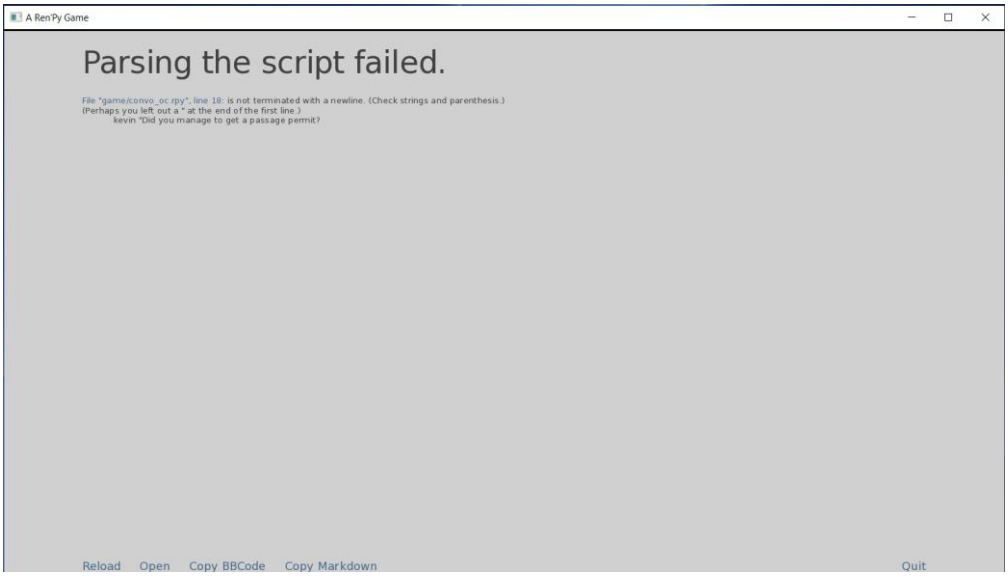
We should focus on the one named "errors.txt", which can be opened with any text editor. There we will see the error message showing the line or lines of code that caused the problem.



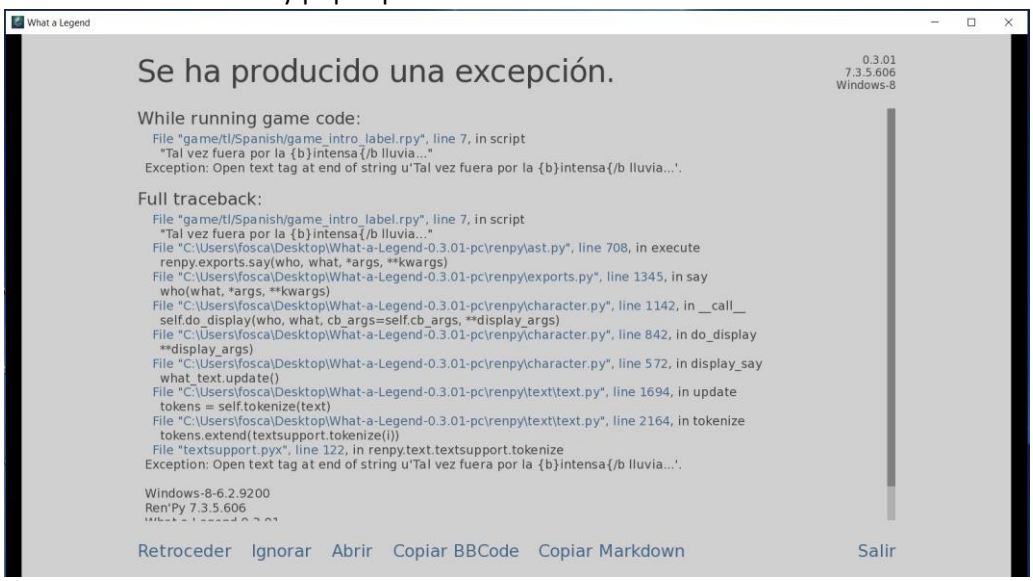
In this case, it's something as usual as forgetting to close quotation marks (on line 18 of the "convo_oc" script), but it could have been an indentation marked with Tab, or an unclosed tag, or whatever. We only need to open that file, fix it, save changes and return to Ren'Py SDK to try again what we were doing.

If you see a more complex message, or if it shows up even before editing anything, the problem usually arises because you have used an outdated version of UnRen or Ren'Py SDK.

This message would also have been displayed if we had tried to run the game by clicking on its executable. Then the screenshot would be something like this, with the same information of the "errors.txt" file:



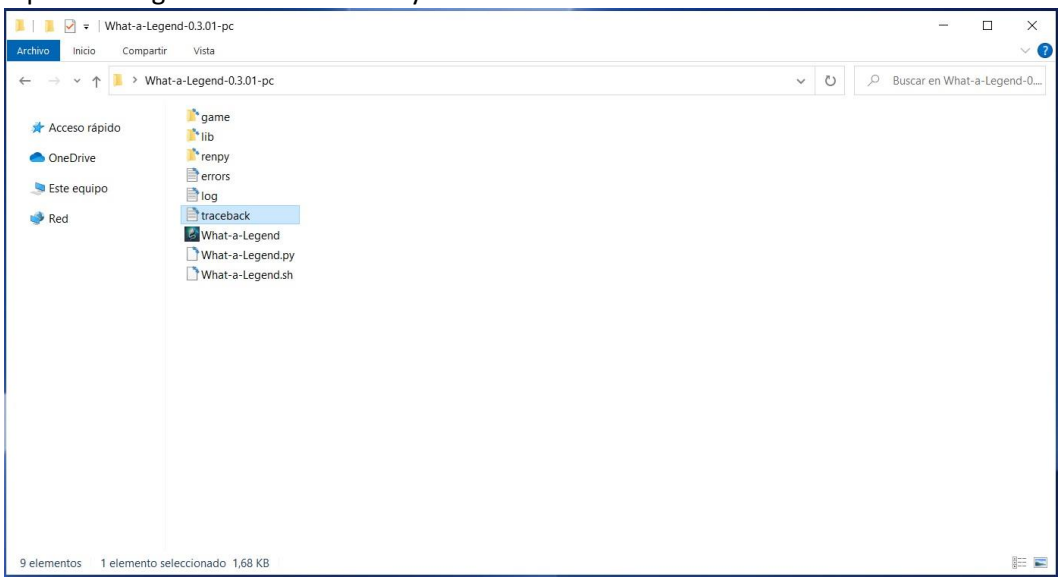
Sometimes, the error is not a critical one in the sense that it allow us to launch the game and generate the translation files, but it does generate an exception while playing the game. Generally, these failures are due to a misspelled label or a problem with the game's variables. That's when, during our gameplay, this gray screen with all those lines suddenly pops up:



On this screen we are given the option to try to ignore the exception and resume playing (with a high chance of getting further warning screens and game's crashes). We can also rollback to an earlier point to save our game before trying to fix what's wrong. In addition, we can copy this message with a BBCode format (that will allow us to insert it in online forums) or to copy it as Markdown (which then we could attach to a Discord message) just in case we want to ask for help.

However, these are almost always minor errors that, even if they are not originated by our work, are relatively easy to fix. To fix them, we always must pay attention to the message's first line, since it indicates the location of the code line that caused the problem. In this case, in the "game_intro_label.rpy" translation script (if you look closer, it's said that it's inside the "game/tl/Spanish" folder, and that's why I know this is the translation script and not the original one), line 7, there is an unclosed tag (a tag was opened with the command `{b}` to **bold** a word, but it wasn't properly closed). And in the last line just before the info section (where we can find the player's Operating System, the software's version and the date and time of the exception), the exception's cause appears again, although without any reference to the script where it occurred. Just by looking at these two lines, we will be able to locate and fix the mistake.

Now, taking another look at the game's root folder, we would see that a "traceback.txt" file has been generated. This file contains the same information we saw in the in-game exception screen. Both this "traceback.txt" file and the "errors.txt" file we saw before are regenerated every time an exception occurs, deleting its pre-existing content to show only the most recent issue.



In all cases, it's just a matter of locating and fixing the error, saving the scripts and trying again what we were doing when the exception appeared, hoping that was the only bug. Good luck with that.

|Top|

3.2.- Lines with too much text

Generally speaking, words and sentences written in English are usually shorter than in most languages. This can cause some translations to exceed the limits of the textbox that we see on screen, or to visually obstruct other elements in the user's interface. Game becomes hard or unpleasant to read and sometimes even the functionality of some buttons is altered. There's a triple and quite simple solution to this.

For starters, we can always try to rewrite our line to say the same with fewer characters. If the result does not convince us and we are translating a dialog block's string, we can take advantage of the huge freedom that Ren'Py gives us to write the translated text. Because, although Ren'Py SDK only offers us one line to translate the original string, we can split it into two or more lines, as long as we respect the Ren'Py's commandments regarding quotes and tabs. Look at the example:

```
# game/script.rpy:10
translate Spanish label_start_XXXXXXX
    # mc "Hello. My name is John."
    mc "Hola."
    mc "Me llamo John."
```

Thus, although in the original script the text is displayed in just one message, in the Spanish translation it will be split into two messages, with no further consequence. Actually, we can introduce any variation we can think of, like adding functions, modifying variables, etc., which will only be applied when playing in our language. That's because, although the translation function is intended to replace one text string with another text string in a different language, what it actually does is to replace a whole line of code with what we write in the translation script, so nothing can really prevent us from replacing that line of code (which happens to be just some text said by a character) with a whole different block of code.

The third option, a bit more complex, is to edit the original "gui.rpy" script, where among many other things game's devs can define the font size and the textbox width. For the font size, we should look for the "`define gui.text_size`" command and reduce the number that appears next to it. And to increase the width of the textbox, we would increase the number that appears next to the "`define gui.dialogue_width`" command. These workarounds will allow us to write more characters in each translated string and they usually work well in all games, although it can break the game's aesthetics and you may find some technical difficulties depending on the level of customization of the game's interface. But getting a satisfying result is just a matter of doing some research and testing your changes.

|Top|

3.3.- Fonts that don't allow special characters

Slightly related to the previous one, in the sense that one of the solutions is to edit the "gui.rpy" file, we have the issue of those games that uses a text font that does not contain some special characters such as accented vowels or any other characters that are quite specific to some languages. Therefore, when running the translated version, the words will appear on screen without those letters or with an odd symbol instead.

The quickest solution is to open the original "gui.rpy" file and look for the "`define gui.text_font`" command to replace the font used with DejaVuSans, which does not require any extra action since it's integrated into Ren'Py. This is how your code should look like:

```
define gui.text_font = "DejaVuSans.ttf"
```

If the font that does not support our language's special characters is the one used for the names of the game's characters, or some other specific font used in menus, you can locate it in the same script and modify it in the same way.

Another option is to set a more suitable font that we like more and has those special characters. In that case, in addition to editing the "gui.rpy" file as we have just seen (but obviously with the chosen font's name instead of DejaVuSans) we will have to include in our patch the .ttf file of said font, so that it remains stored inside the "game" folder.

Sadly, by choosing any of those two options, the game can not be played again with the original font in any language. So we can choose an intermediate solution, slightly more complex, which is to set a specific font just for our language, so that the game's original version would be displayed exactly as its developer had in mind even after applying our translation patch. In the original "gui.rpy" file we should write this:

```
init python:
    translate_font("Spanish", "myfont.ttf")
```

Where "`myfont.ttf`" is the font we've chosen for our language (and that language being "`Spanish`" in my case). If it is not one of those standard fonts supported by Ren'Py, we will have to include the .ttf file in our patch so that this time is stored in the "game/tl/Spanish" folder and not in the "game" folder.

In all the cases abovementioned, we must remember to include the edited "gui.rpy" file in our patch, so it replaces the one that the players who have downloaded the original game have within their "game" folder.

And finally, we also could go all-in and edit the original font with a font-editing software, to design the missing characters. Obviously, in this case we would also have to include the resulting .ttf in our patch to replace the original one in the "game" folder, but we wouldn't need to edit the "gui.rpy" file at all.

|Top|

3.4.- Translating images

One of the most time-consuming tasks to do is translating images that contain text. Sometimes the game's developers decide that an important part of the action need to be based on a SMS message, for example, or any other visual element (a computer screenshot, a newspaper headline, a poster on the wall or whatever you can imagine). Although Ren'Py offers several coding solutions to write these texts in the scripts and, therefore, make its translation easier, devs usually find more convenient to simply create an image with the text included on it.

If we are lucky enough that, while the image is displayed, there is also some dialogue, in our translation we can just add the text to the character's words, as if he/she was reading it aloud.

Or we could also "subtitle" the image thanks to a solution we used to fit the translated strings within the textbox: we can split that string's translation into several lines and write in them the translation of the image's message, maybe in *italics* or in some other way that helps the players know what's going on there.

```
# game/script.rpy:12
translate Spanish label_start_XXXXXXX
    # mc "Look. He sent me a message."
    mc "Mira. Me envió un mensaje."
    "{i}(And here we translate the message that is displayed on the image.){/i}"
```

If, sadly, we don't have any dialog lines associated with the image, we can create a blank one in the original script. To do this, the first step is to open the **original script**, find the line of code in which we think Ren'Py is ordered to show the image (we can use nearby text strings to find it) and then, respecting the indentation, we would insert a new line below, with a blank string (writing only quotation marks), like this:

```
scene black
show sms00 #This is the command used to show the image that contains text
"" # This is the new blank string we insert in the script, to translate the pic
pause
hide sms00.jpeg
mc "Wow."
```

Then we (re)generate the translation files and now we can include the image's text translation as the translation for that new blank string. Of course, for the translation to be correctly displayed, we would have to include in our patch the original script that we have just edited.

However, sometimes this option is simply not viable or doesn't result in a good aesthetic result. In that case, the only solution is to edit the original image (or to create a new one) with some picture editing software like Photoshop or GIMP (or even MS Paint, if it's a very simple editing). First, we would go to the original script to find image's name (generally whatever appears after the `show` or `scene` commands). Then we would search inside the "game/images" folder and edit it to write its text in our language. In an ideal world, we could kindly ask game's developer to provide us the base image, without any text on it, to make this editing work easier. Good luck with that.

Once the edition is finished, we shouldn't replace the original image in that "game/images" folder; instead, we need to save a copy with our changes, with the same name and image format, in an identical path but inside our translation folder (in my case, inside "tl/Spanish"). So, if the original image is stored as "`game/images/ch1/sms00.jpeg`", we **MUST** save the translated image as "`game/tl/Spanish/images/ch1/sms00.jpeg`". That way, when we are playing the game's translated version and the image named "sms00.jpeg" has to be displayed, Ren'Py will first look for it in the "images" subfolder of the translation folder, and only if the pic cannot be found there, it will show the one in the original "game/images" folder. And, logically, if we are playing in the game's original language, the original image will be shown.

|Top|

3.5.- Translating text variables

When we talked about Ren'Py SDK's limitations and how it has some troubles extracting all the translatable strings, we said that the variables that contain text as its value are not automatically detected by the extraction function. Regardless of how we finally manage to include them in the translation scripts, the problem is that, usually, that translation will not be displayed later on screen, when it should.

When a text variable appears on screen, it has been coded as an embedded element in a string. We will notice it in the scripts because, within the string, an object with the variable's name on it will appear between [brackets]. Sometimes the string consists exclusively of that object, and sometimes it's just part of a sentence. For example, in one of the original scripts of the game "City of Broken Dreamers" we find this:

```
56      show bg ch1sonja3 with dissolve
57      $ insult = renpy.random.choice(insults)
58      "Unknown Woman" "Come on now, [insult]."
```

In this case, the game's developer has decided that a certain character will use an insult that will vary randomly in each game. To do this, in another script (sometimes we have to do a lot of research), the dev has defined a variable which is actually a list of different insults.

```
139 default insult = ""
140 default insults = ["jackoff","fuck-face","cock muncher","retard","cumb dunt","dick mitten","fuck-knuckle"]
```

The function in line 57 will randomly select an insult from the list we see on line 140, and that chosen insult will be the value of the variable named "insult" (which has been defined as blank on line 139). Then, in line 58, the character "Unknown Woman" will use that insult while talking to our MC.

Therefore, our first task would be to translate the list of insults of line 140, which actually contains several strings, one for each quoted insult. Using our imagination, we translate each of them with the `old` and `new` commands and, in the dialog block's translation string, we add the appendix `!t` to the object with the variable's name on it:

```
571  # game/chapter1/ch1sonja.rpy:58
572  translate Spanish ch1sonja_cfc9ab60:
573
574      # "Unknown Woman" "Come on now, [insult]."
575      "Desconocida" "Venga ya, [insult!t]."
```

Thus, when playing in the game's original language the variable will be displayed in that language, since the original script has not been modified, while the translated version will display the translated content. If we just translate the insults but don't include the appendix `!t` in that other string, when playing the translated game the insult would still appear in English, as our translated string would keep telling Ren'Py to display the original variable `[insult]`, instead of its translated value `[insult!t]`.

Note that this only refers to real text variables, not to those that are used to name game's characters. Sometimes, for convenience when writing the strings, games developers replace some characters' full names by the definition of their respective "character" variable. We will notice it because the variable we see embedded in the string is the same that we see at the beginning of the dialogue lines of said characters, before the quotation marks. In that case, the translation of that variable (if necessary) is done automatically by Ren'Py and we don't need to do anything else. For instance, there are several characters in the game "Deliverance" who are known by a common, translatable noun, and not by their real first name.

```
8  define li = Character("Lieutenant", ctc="ctc_default",ctc_pause="ctc_default") #fowler
```

In this case, the variable named `[li]` refers to one of them (Lieutenant, as he's a Police Chief), but since it is a "character" variable Ren'Py will automatically display its original value or its available translation, depending on which game's version we are playing. **So we just have to translate "Lieutenant" using the `old` and `new` commands** and, whenever that variable appears in a translation string it is not necessary to add the appendix `!t`.

```
177  # game/script.rpy:350
178  translate Spanish interrogation_3864c01a:
179
180      # co "[li], you have a visitor. Mayor wants to see you."
181      co "[li], tiene una visita. El alcalde quiere verle."
```

With these notions we should have no problem getting all the translatable text of the game to be correctly displayed in our language.

<u>What follows next is an emergency solution that should only be used as a last resort.</u>

There are quite complex games, and sometimes we will not be able to see the text properly translated because we don't know how to locate the string that contains the object with the variable (it can happen especially in custom menu screens, inventories, etc.). The emergency solution is to edit the original scripts and wrap that variable's definition with parentheses, adding a double underscore before the opening sign:

`define` *variable_name* `= __("...").`

This, on the one hand, will allow Ren'Py SDK to extract the translatable string, and on the other hand, by using the double underscore, our translation will always be shown on the screen... even when playing the game in its original language. It's a shame, but sometimes we just don't see another option. Of course, it is possible that, by doing this, we could be generating some internal logic issues in the game, as the value of that variable may be used in some logical expressions that determine the game's paths or the content to block or show to the player. In those cases, to avoid bugs and other inconsistencies, we must include the variable between the `__( )` symbols also in those expressions.

For example, let's imagine we have a variable named "fruit" whose original value is "Apple". Somewhere in the code we should find its definition:

```
default fruit = "Apple"
```

Later on, during the game, the player can change that fruit for another, so in the script we could find something like this:

```
$ fruit = "Orange"
```

Let's imagine that we have found those two strings and have translated them with the `old` and `new` commands in our translation scripts:

```
translate Spanish strings:

    old "Apple"
    new "Manzana"

    old "Orange"
    new "Naranja"
```

But in the game there's an inventory screen to show a list of various items in player's possession, one of them being the variable `[fruit]`, and for whatever reason we are not able to find in the original scripts that screen's coding lines, so we can't extract that specific string `[fruit]` and translate it as `[fruit!t]` in the translation script. This means that, even when playing the game's translated version, the text displayed on that inventory screen will be "Apple" or "Orange", still in English. The brute force solution would be to extract with the double underscore those strings we did locate:

```
default fruit = __("Apple")
$ fruit = __("Orange")
```

Thus, every time the `[fruit]` variable is mentioned in the original scripts, the value that will be displayed on screen will be our translation for those words, even if when playing in the original language, which is not optimal. But it may happen that this "`fruit`" variable is used at some point to display a certain dialog line or a different one, based on what fruit the player owns at that moment. Something like this:

```
if fruit == "Apple"
    mc "I like red apples."
if fruit == "Orange"
    mc "I like orange juice."
```

If we leave the original scripts like this, the game would crash when reaching this part of the script because the value of the "`fruit`" variable would be our translation for the words "Apple" or "Orange", not "Apple" or "Orange" in English, which is how the expression is originally coded. It might not cause a critical exception, and depending on the rest of the code the player may not even notice anything (or maybe just a small jump or incoherence in the dialogues), but of course this is not what the game's developer wanted. To solve this, we should also use the `__( )` symbols in the logical expression:

```
if fruit == __("Apple")
    mc "I like red apples."
if fruit == __("Orange")
    mc "I like orange juice."
```

Had we managed to properly translate the infamous `[fruit]` string, all this editing would not be necessary, since the logical expression would work internally with its original value (in the game's original language) even though the translation was displayed on screen.

|Top|

3.5.- Polisemy: different translations for equal words

This issue came up when talking about the two translation functions used by Ren'Py. Strings only support one translation: in fact, if we try to translate them twice (writing two `old` commands with the same string, for instance), the game simply won't start and the error message will say that there is a duplicate translation. In order to start the game, we will have to delete one of them.

But, as I said in that example, sometimes we find identical strings that need different translations, like the English word "Right" which, among other things, could mean "Correct" or just a side or a direction opposite to "left". The solution is to edit the original script to get those identical strings to stop being identical, without affecting the game in the original language. And for that we only need to use the # symbol.

As we have already stated, everything to the right of a # symbol will not appear on screen and is discarded by Ren'Py in all its processes... unless we put it inside a tag. Tags are commands embedded in the strings with `{ }` symbols, and generally are used to change the font's size and color, to highlight the phrase in **bold** or *italics*, etc. Ren'Py reads these tags as part of the string that contains them and executes what they order, but their literal content is not displayed on screen. So, if in one of those tags we include a text after the # symbol, the tag won't order Ren'Py to do anything and Ren'Py won't show its content, but now we would have a string which is different from the one that has no tag, and both of them will look identical on screen.

Therefore, we'll find the string that we want to translate differently and we'll incorporate a tag in it in which we will write something that will allow us to identify it later, such as the translation we want to apply. Thus, when reading and extracting strings, the text "`Right`" and the text "`Right{#Direction}`" are no longer the same for Ren'Py, although both will be displayed as "Right" on screen. Then we just have to regenerate the translation scripts (or manually include in them this new "tagged" string, with the `old` command) and assign it a different translation with the `new` command. Eventually, we should have something like this:

```
translate Spanish strings:

    old "Right"
    new "Correcto"

    old "Right{#Direction}"
    new "Derecha"
```

Remember, in this case writing this in the translation script is not enough: we also have to edit the original .rpy script to embed the tag into the string we want to translate differently so that, when playing the translated version, Ren'Py will look for the tagged string's translation. And, obviously, we need to remember to include those edited scripts in the patch.

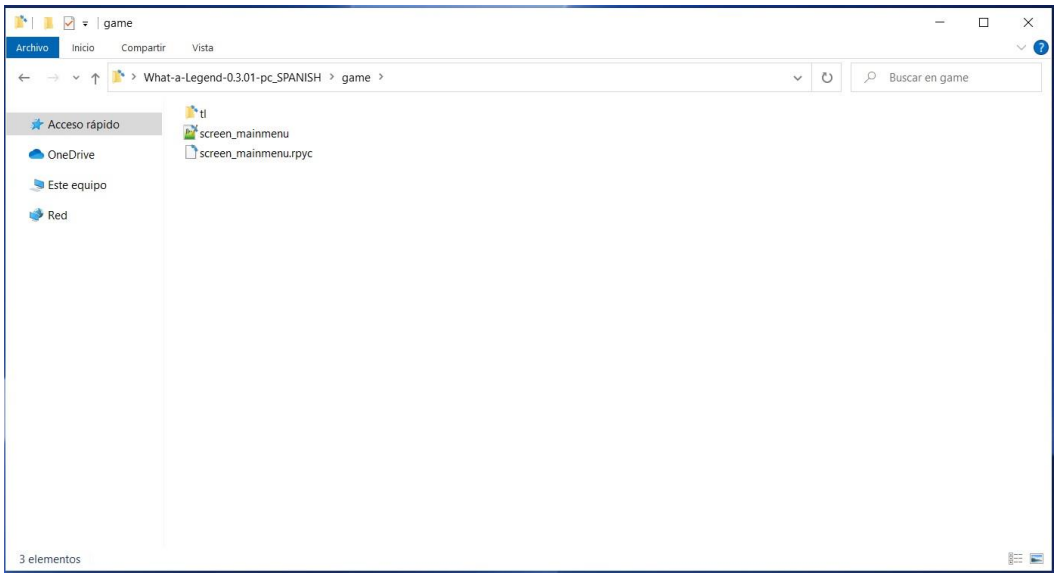
|Top|

4.- THE TRANSLATION PATCH

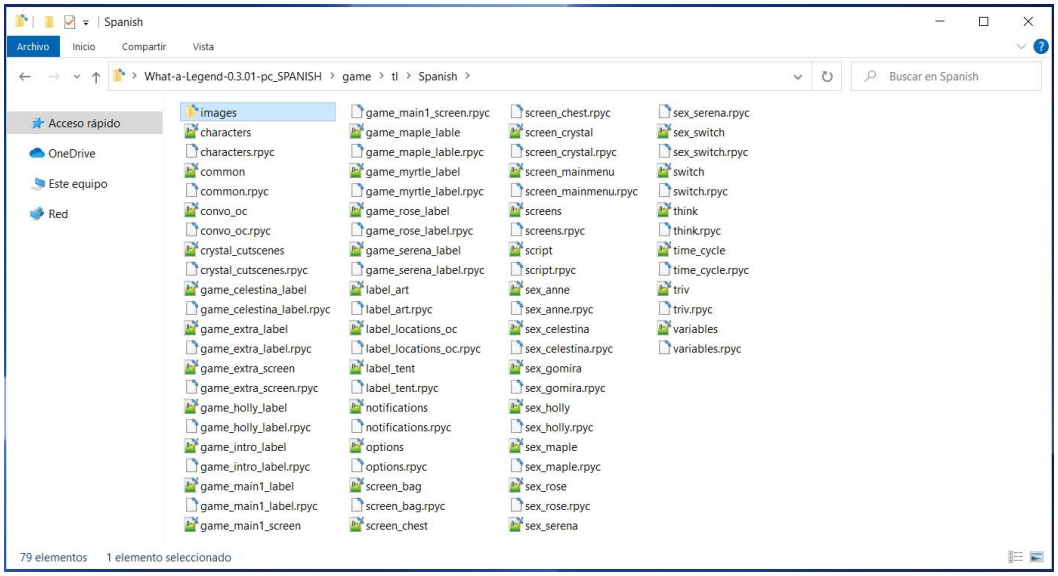
Once our translation is done (and tested), it is time to share it with the rest of the world. But, in order to work, our patch must respect the original game's folder structure. That is, the original .rpy scripts that we have edited (along with their respective .rpyc scripts) and the .ttf files of the fonts used to solve those problems mentioned on section 3.3 must end up in the "game" folder of the player, who must replace the original files with those in our patch. And, obviously, we also must include a "tl" folder, and within it another subfolder (named "Spanish" in my case, because that's the language I wrote in Ren'Py SDK) that will contain our translation scripts and, where appropriate, the necessary images and fonts, as it has been explained earlier in the guide.

The easiest way (for me, at least) is to create a "game" folder with all those files, so that players will only have to paste it into the root directory of their original game and accept to replace and/or overwrite all the matching files.

This is, for example, how the "game" folder of my "What a Legend!" translation patch looks like:



In this patch I only had to include the "screen_mainmenu.rpy" script, where I added the language change option. This file will replace the original "screen_mainmenu.rpy" script that exists inside the player's "game" folder. Logically, there is also the whole "tl" folder that I had in my computer, with the "Spanish" subfolder and my full translation inside, which looks like this:



There you can see all the translation scripts and also a subfolder named "images" with the game's images I had to translate, which are also stored in folders with the same name as those containing the original images in the "game/images" folder, as explained in section 3.4.

So, as a quick reminder, to let players enjoy all our work we just need to include in our translation patch the translated files plus all the original scripts that have been edited for something else than extracting the strings with the `_( )` symbols (remember that Ren'Py doesn't need that symbol to display the translated string, once that string is present in the translation scripts with the `old` and `new` commands). That is: the .rpy scripts we have edited to allow the correct translation of polysemic words, the "gui.rpy" and/or "screens.rpy" files edited to change language or the game's font (as well as those fonts' .tff files), and their respective .rpyc files, all those will be in the translation patch.

However, my patches usually include absolutely all the original scripts I've edited, so that other translators can use them to get all the translatable strings extracted by their Ren'Py SDK. But "What a Legend!"

developers already write their code with those `_( )` symbols where needed, so in this case there was nothing else to add.

Finally, one last comment. I **always** recommend including the .rpyc scripts in the patch (that is, those that correspond to the original .rpy scripts that we have edited, and that will stay in the player's "game" folder). If we have never deleted them since we downloaded the original game (or since we extracted them with UnRen in order to be able to generate the translation scripts), these .rpyc scripts we have in our computer will respect the AST created when they were originally compiled by the game's developer, even though they have been modified by us when editing their respective .rpy scripts. So, when players download our patch and replace the .rpyc files they have in their computer (which, if they have never deleted them, are also the same ones we had at the beginning of our translation process), the basic structure of the new .rpyc scripts will still be the same and they should have no problems to load games they had saved when playing the game's original version before applying our patch (anyway, it's always advisable to first load those old saves in the game's original language, and then change languages to keep playing from that saved point).

What would happen if we don't include the .rpyc scripts in the translation patch? It depends. Generally, if the original game were packaged as recommended by Ren'Py (with both the .rpy and .rpyc scripts exposed inside the "game" folder, as it was the case of "What a Legend!") there should be no problem: our edited .rpy scripts will replace the original ones that players have on their computer and, when launching the game, their .rpyc files will be updated with the changes included in our .rpy scripts. This process shouldn't break anything... assuming the player had never deleted the game's original .rpyc scripts (if they have been deleted by the player before installing our patch, we can not be responsible of whichever errors could arise due to an AST change when loading old saves).

The really important issue can arise when the original game's scripts are compressed into a .rpa archive: if we only include in our patch the edited .rpy scripts, Ren'Py will create new .rpyc files on player's computer, and there's a risk of them not being generated with the same AST structure that those we extracted from the original game, which were the ones we used to create our translation (and are also the ones that players still have compressed in their .rpa file). In such cases, not only old saves might be compromised: it may happen that the function that executes the "encryption code" translation (dialog block's strings) won't detect the existing translation. When that happens, the game's menus and options would be displayed in our language (because those strings are translated by direct replacement thanks to the `old` and `new` commands) but the dialogs of those edited scripts would be displayed untranslated, in the game's original language. It is a quite strange error but that can occur, especially with relatively old games that, perhaps, were originally compiled with an ancient version of Ren'Py SDK. Anyway, we'll avoid this situation including in our patch both the .rpy and .rpyc scripts: although edited and updated, they still follow the AST structure of the original scripts we extracted with UnRen, and that structure will match the player's old saves structure too. Better safe than sorry.

|Top|