dramaTTS Documentation

Thies Hecker

Jan 31, 2020

Contents:

CHAPTER 1

About

dramaTTS parses scripts (plain text files) for theatre/screen plays and converts them into a multi-voice audio plays (wave-files).

While the script parsing functionality is provided by the *dramaTTS* program itself, it relies on external tools for the audio processing:

- The Festival Speech Synthesis System[1] (herein referred to as *Festival*) is used for speech synthesis
- Sound eXchange (SoX)[2] for audio post-processing.

SoX, *Festival* as well as voices and lexicons for *Festival* have to be installed in order to create audio output with *dramaTTS* (see *Getting started*).

1.1 Introduction

dramaTTS is intended to provide a free solution for converting a drama / screen play text into a multi-voiced audio recording using text-to-speech synthesis.

While *dramaTTS* provides a text parser to identify the speakers of each line/paragraph of the play and a bunch of configuration options (e.g. for speaker voices) the actual text-to-speech synthesis and audio post-processing relies on the external tools Festival Speech Synthesis System[1] (short *festival*) and Sound eXchange (SoX)[2] (short *sox*).

dramaTTS reads a text file, stores the voice configuration and schedules separate *festival* processes to render each line/paragraph (i.e. a continuous text section read by on speaker) into wave-files, which are afterwards post-processed/merged using *sox*.

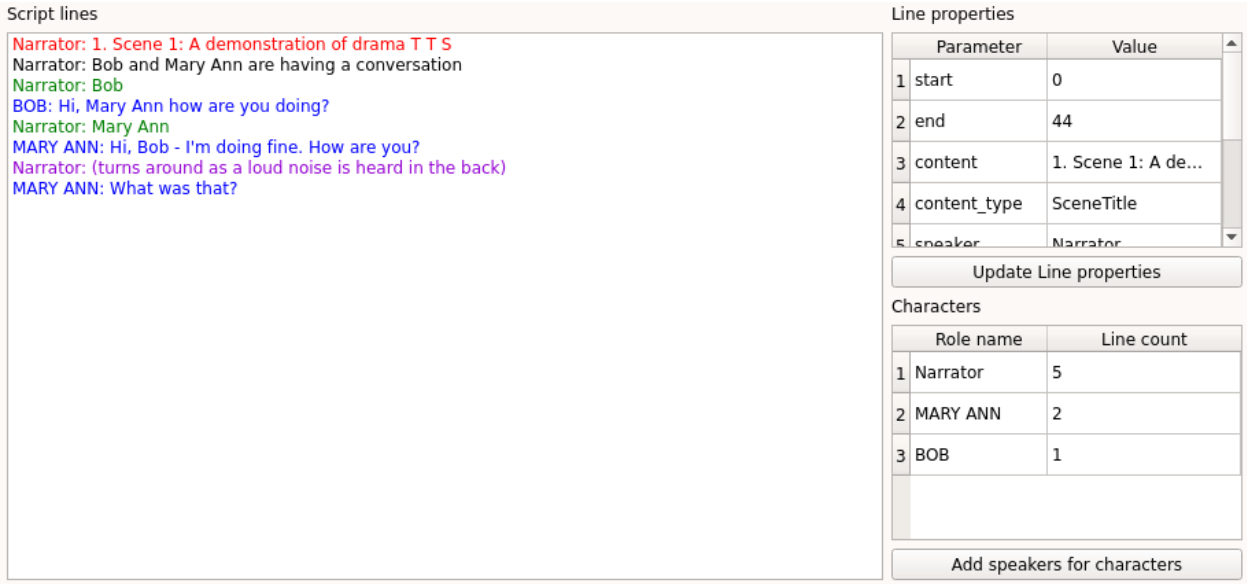
The basic working principles of *dramaTTS* shall be illustrated on a short text example - a detailed explanation of the different configuration options and features can be found in the *User guide*.

Let's consider following example text (stored in a plain text file - e.g. example.txt):

[1] http://www.cstr.ed.ac.uk/projects/Festival/
[2] http://sox.sourceforge.net/Main/HomePage
[1] http://www.cstr.ed.ac.uk/projects/Festival/
[2] http://sox.sourceforge.net/Main/HomePage

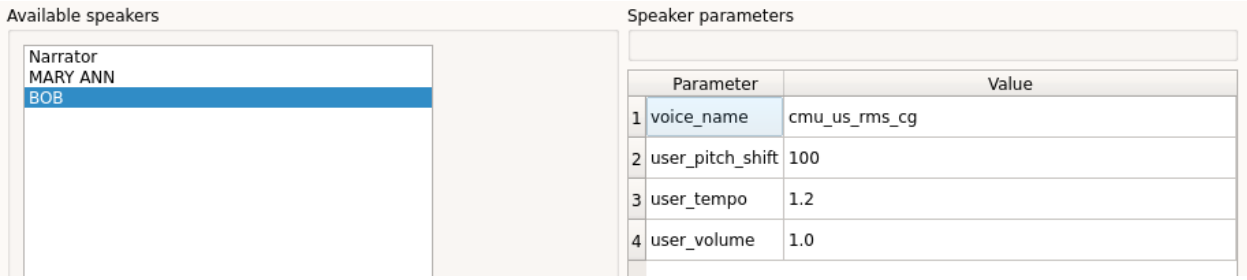
```
1. Scene 1: A demonstration of "drama T T S"

Bob and Mary Ann are having a conversation

BOB

Hi, Mary Ann how are you doing?

MARY ANN

Hi, Bob - I'm doing fine. How are you? (turns around as a loud noise is heard in the␣
↪back) What was that?
```

When this text is imported, *dramaTTS* will analyze it for different continuous text section of the same content type (which can for instance be a new scene title or content of a dialogue) and analyze the speaker of this section. I.e. for the example above *dramaTTS* will identify following structure:

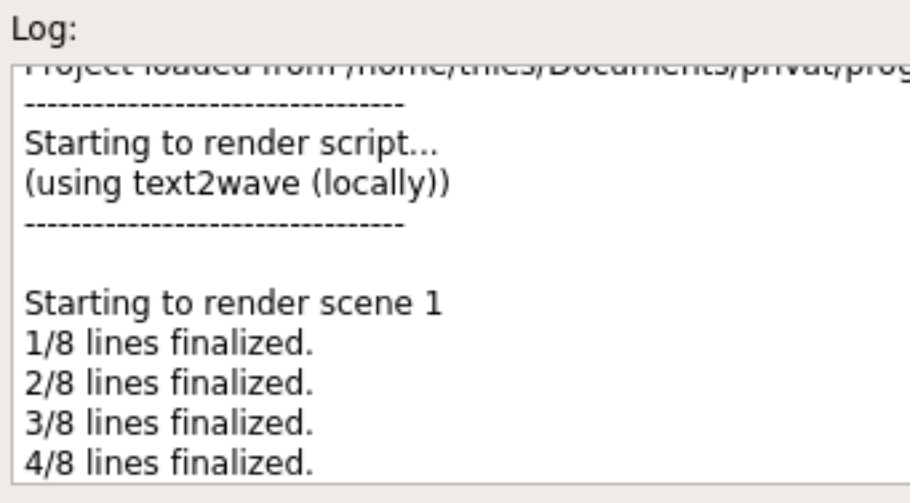


The different colors indicate different content types - as can be seen the characters/speakers in the play and their line count is identified and displayed in the table on the bottom right.

As a next step the voices for each character/speaker will be defined.



Afterwards the text can be rendered to audio files.



The audio output can be played below.

1.1.1 Features

As mentioned above *dramaTTS* consists of 2 main components: a script parser and a scheduler/configurator for the audio-rendering.

The script parser features:

- configurable input file formatting (see *Working with "Content identifiers"*)
- syntax highlighting (identifies different content like new scenes, dialogue lines, narrative descriptions,...)
- text string substitutions supporting regular expressions
- some utility functions like sorting speakers according to their number of text lines

The audio-renderering part basically provides a front-end to *Festival* and *SoX* with following features supported:

- Altering of *Festival* voices (pitch, tempo and volume)
- support for multiple CPU cores to accelerate audio rendering (dispatches parallel processes for individual lines)
- using a *Festival* server for rendering is supported
- some post-processing: normalize all voices, combine audio files (lines -> scenes -> single project file)
- (re-)rendering of individual scenes or speakers

1.1.2 Licenses

dramaTTS is realized using:

PyQt[4], Copyright (c) Riverbank Computing Limited

[6] https://gitlab.com/thecker/dramatts/blob/master/COPYING
[3] https://gitlab.com/thecker/dramatts/blob/master/LICENSE
[7] https://gitlab.com/thecker/dramatts
[4] https://wiki.python.org/moin/PyQt

and

setuptools_scm[5], Copyright (c) Ronny Pfannschmidt.

While *dramaTTS* is a standalone application, it is of limited use without *Festival* and *SoX* being installed, which provide the audio rendering (only script parsing including syntax highlighting, etc. is available).

While the *Festival* application itself and *SoX* are released under free software licenses as well, specific components, which are commonly bundled with *Festival* (i.e. certain lexicons and voices) may be released under non-free licenses.

For instance the *festlex-OALD* lexicon, which can be found among other files (incl. the source code of the latest *Festival* release) on the Festvox 2.5 release page[8] lexicon is restricted to non-commercial use only.

The *Installing Festival without non-free components* section will provide an example for a *Festival* distribution based on *free* components only.

Please see the COPYING[6] file in the source code repository for details on licenses and copyright disclaimers of the individual components.

1.2 Getting started

1.2.1 Installing *dramaTTS*

You will need a python3 distribution installed and for most convenience you should have either the *pip* or *conda* package manager installed.

On linux you will most likely have python and pip already installed - if not you should be able to install them with distributions package-manager.

E.g. for *debian* based system like *ubuntu* just run:

```
sudo apt-get python3-pip
```

or on *arch* based systems:

```
sudo pacman -S python-pip
```

For Windows users I would recommend to install Anaconda or miniconda, which will provide the *conda* package manager (make sure to get the python3 - not the python2 - version!).

To install *dramaTTS* with pip:

```
pip install dramatts
```

Note, that on some distributions you may install python2 and python3 in parallel. In such cases you should make sure, that you not using a pip for your python2 environment to install *dramaTTS*. Eventually you need to use pip3 as a command. You can check if you are using the correct pip by calling:

```
pip --version
```

To install *dramaTTS* with conda:

```
conda install -c thecker dramatts
```

In both cases pip or conda should download all required dependencies and should be able to launch the program. To do that just type:

[5] https://github.com/pypa/setuptools_scm/
[8] http://festvox.org/packed/festival/2.5/

```
python -m dramatts.dramatts_gui
```

The GUI should pop up and you can import text files, define roles etc., but you will not be able render audio unless you have installed *Festival* (and its components) and *SoX*.

1.2.2 Installing Festival without non-free components

While many linux distributions include pre-built packages for *Festival* they often include non-free components like *festlex-OALD*. Therefore the safest way to create a *free Festival* distribution is to compile from source. To form a *free* distribution following components could be used:

- Festival 2.5 (main application)
- Edinburgh Speech Tools (EST) - required to compile *Festival*
- festlex_CMU (lexicon)
- festlex_POSLEX (lexicon)
- festvox_cmu_us_slt_cg (female voice)
- festvox_cmu_us_rms_cg (male voice)

All components can be downloaded at CMU's (Carnegie Mellon University) Festvox 2.5 release page. The source code of *Festival* and *EST* can also be cloned from the Festvox github page.

To compile the code follow the instructions in the INSTALL file included in *Festival*.

Note, that more voices can be found at the *Festvox* page (although some might require e.g. additional lexicons and thus won't be working with the selected components above). Additionally voices may also be altered in tempo and pitch in *dramaTTS* (by post-processing with *SoX*) to create more than one speaker per voice.

Building *Festival* from source is based on the autotools-toolchain - so it shouldn't be a problem on GNU/linux, but may be complicated on MS Windows.

Fortunately the eGuideDog team has created compile-instructions for Windows and even provides a Festival 2.5 version including precompiled binaries for Windows (which does not include the problematic *festlex-OALD* lexicon).

In order to use *Festival* under Windows with *dramaTTS* you will need to copy the *text2wave.bat* (see the /utils folder) to your *Festival* installation.

Make sure to adjust the paths in *text2wave.bat*, if you did not install *Festival* in C:\Festival.

1.2.3 Installing SoX

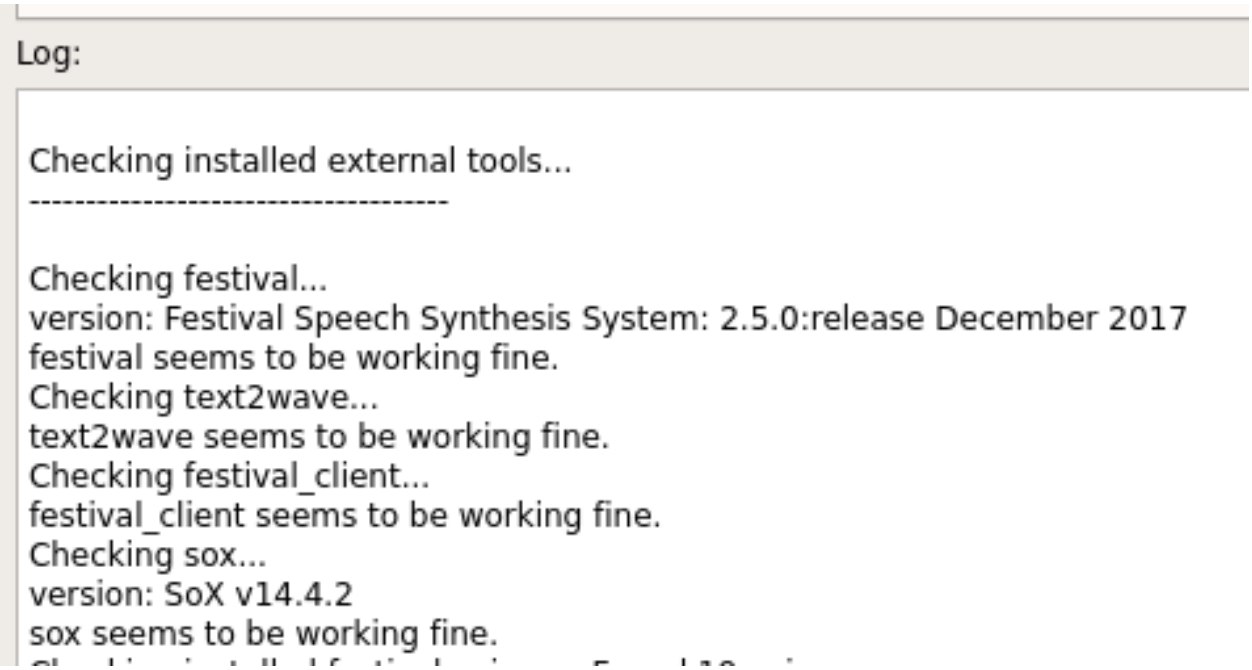
Under linux you will most likely have a pre-build package for *SoX*. Building from source is probably not required.

Binaries for Windows can be found on the SoX sourceforge page.

1.2.4 Configuring locations of external tools in *dramaTTS*

dramaTTS will try to determine the install locations of *Festival* and *SoX* automatically. This should most likely work under linux, if you installed the tools from the official packages (or put the location of the binaries in your PATH).

You can see, if the tools where found in the log windows of *dramaTTS*. A check will be performed each time *dramaTTS* is started - if all tools are configured correctly you will see messages like this in the log.



Under windows you will most likely have to define the tool locations manually.

To do that, just go to the *Adjust program preferences* in the *dramaTTS* GUI and specify the file locations.

If you used the *Festival* version provided by the link above the pre-compiled binaries are located in:

..Festival\src\main

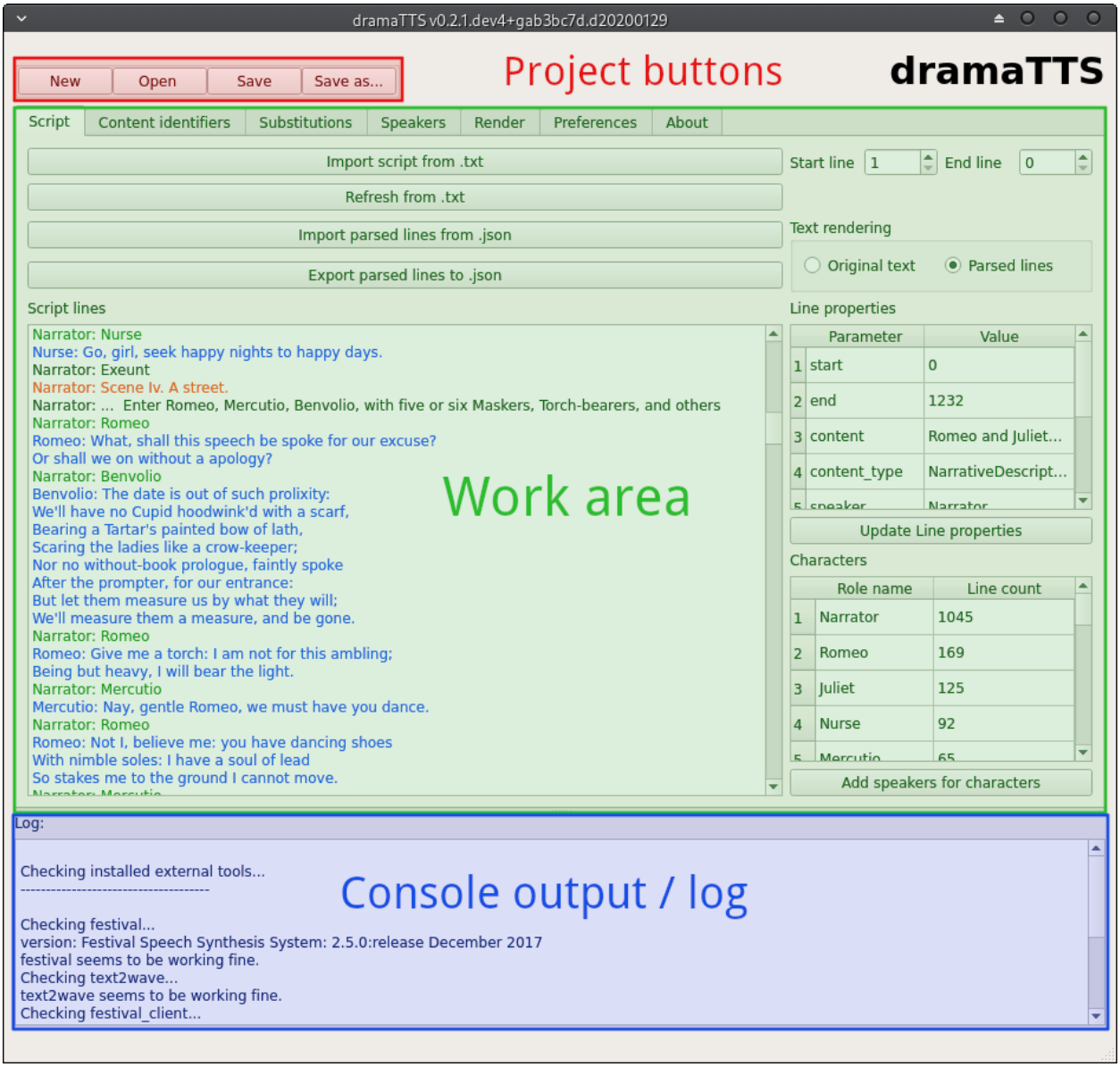
After you specified a new tool location, you should save the preferences and restart *dramaTTS* to make the changes become effective.

1.3 User guide

1.3.1 GUI layout

The GUI (Graphical User-Interface) consists of 3 main elements:

- project buttons
- work area
- log area (console output)



Project buttons

The project buttons will always be shown on top and allow to create a new project, open a saved project or save a project at any time.

Work area

The work area contains the tabs for the different work steps - e.g. for importing and viewing a script, defining options for the audio rendering, etc.

The individual tabs will be explained in more detail in the chapters below.

Log

The log shows the console output of the *dramaTTS* sub processes. It can provide valuable information like warning and error messages or the progress of the rendering process.

1.3.2 Input format basics

Although *dramaTTS* allows to configure the behavior of the script parser to support different formatting styles of the text source, it will always distinguish 5 different categories of content:

- An indicator for a new scene (content type 'SceneTitle')
- an indicator for a dialogue - which will be subdivided into:
- a speaker name which starts the dialogue ('DialogueIndicator')
- lines of dialgue ('DialogueContent')
- comments inside a dialgoue ('InlineComment')
- narrative description ('NarrativeDescription')

These types of content shall be illustrated based on the default *dramaTTS* format. Note, that this behavior can be adjusted (see *Working with "Content identifiers"*).

SceneTitle

A new scene is typically indicated by a single line with a scene number and a scene title text. In the default format each line starting with a number followed by a dot is assumed to indicate a new scene - e.g.:

```
23. A new scene
```

Scene titles will be read by the narrator.

Dialogue

A dialogue is typically indicated by a line giving only the speaker name, followed by dialogue content, which is terminated by e.g. the next double line break. In the default formatting a text section is assumed be a dialogue, if a line consisting only of UPPER CASE letters (for the speaker name) is encountered. The dialogue will end after the second blank line is encountered - e.g.:

```
BOB

Hi, I am Bob and this line is the my dialogue.
This line is part of my dialogue as well. (sighs) I guess it's time to end␣
↪my dialogue now.

Bob's dialogue was terminated by the second an empty line, which makes this␣
↪line belong to the narrator.
```

The parser will first identify the complete dialogue and then start to search for the 'DialogueIndicator' and 'InlineComments' within the dialogue. In the default formatting the 'DialogueIndicator' is defined by a single line with only UPPER CASE letters (i.e. the first part of the identifier for the complete dialogue) and 'InlineComment's are defined by any words encapsulated in parenthesis. Thus the parser will interpret the example above as follows:

DialogueIndicator (read by the Narrator) BOB

DialogueContent (read by the speaker - in this case BOB) Hi, I am Bob and this line is the my dialogue. This line is part of my dialogue as well.

InlineComment (read by the Narrator) (sighs)

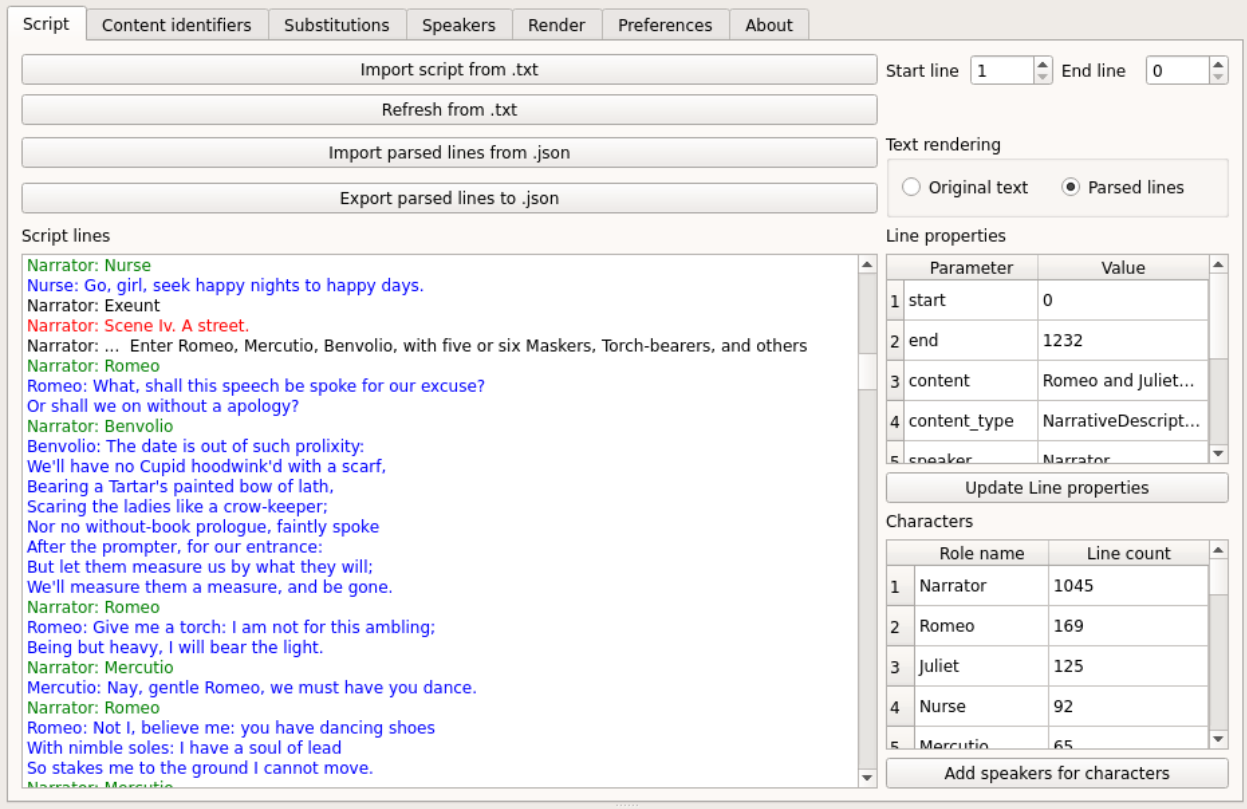
DialogueContent (read by BOB again) I guess it's time to end my dialogue now.

NarrativeDescription

Everything, that the parser did not identify as being a 'SceneTitle' or any part of 'Dialogue' will be assigned to the content type 'NarrativeDescription' and be read by the narrator - e.g. the last line in the example above.

1.3.3 Handling text files

If your script/text file is in accordance with the default *dramaTTS* format, you can go direcly to the "Script" tab and start importing your text file.



The script tab provides a viewer for the imported script, a table which displays the properties of each line (or more precisely each paragraph), an overview of the characters found in the script (including their line counts) and a couple of buttons for import/export.

View script parser results

The script lines viewer shows the result of the import. It uses color highlighting to distinguish the different types of content:

- black: Narrative descriptions (NarrativeDescription)
- red: New scene indicator (SceneTitle)
- green: A speaker name - i.e. a dialogue indicator (DialogueIndicator)
- blue: Text of a dialogue line (DialogueContent)
- purple: A narrator comment inside a dialogue - e.g. "he turns to . . . " (InlineComment)

Note that you can switch the text rendering mode between "Original text" and "Parsed lines". The "Parsed lines" option basically puts the name of the speaker at the beginning of each line.

If you click on a line in the viewer you can find more details on it in the "Line properties" table (see below).

Line properties table

The line properties table shows details on the properties of the selected line/paragraph including the content type (as described above), the start and end position in the text file, the scene the line belongs to, the speaker of the line,...

You can also modify the line properties by changing the property values in the table and clicking the "Update Line properties" button.

> **Warning:** If you re-import your script the changes made to the line properties will be lost. Thus this option should only be used as a final step in your editing process. A better solution to introduce changes (if you do not want to modify the source text directly) is to use the options explained in the *Working with "Content identifiers"* or *Substitutions* chapters.

Characters table

The characters table gives an overview of the characters/roles identified by the script parser. The characters are sorted according to their line count.

If you click the "Add speakers for characters" button, a new speaker (with the default speaker settings - see *Adjust program preferences*) will be added for each character which does not already have speaker settings defined.

Importing and exporting parsed lines

To start a new project you would usually "Import a script from .txt" - i.e. a plain text source. *dramaTTS* also allows to export and import the parsed lines (JSON-format).

Note: Importing/exporting the parsed lines is usually not required as the same information is stored inside the project file as well.

The "Refresh from .txt" button comes handy, if you have already imported a text file and either made modifications to the text file directly or to the content identifiers (see *Working with "Content identifiers"*) or substitutions (see *Substitutions*). Pressing the button will re-import the text file (applying the current parser configurations).

1.3.4 Working with "Content identifiers"

"Content identifiers" can adjust the behavior of the script parser.

The content identifiers are RegEx (Regular Expressions) patterns, which are used to identify a text section - 4 types of identifiers are defined:

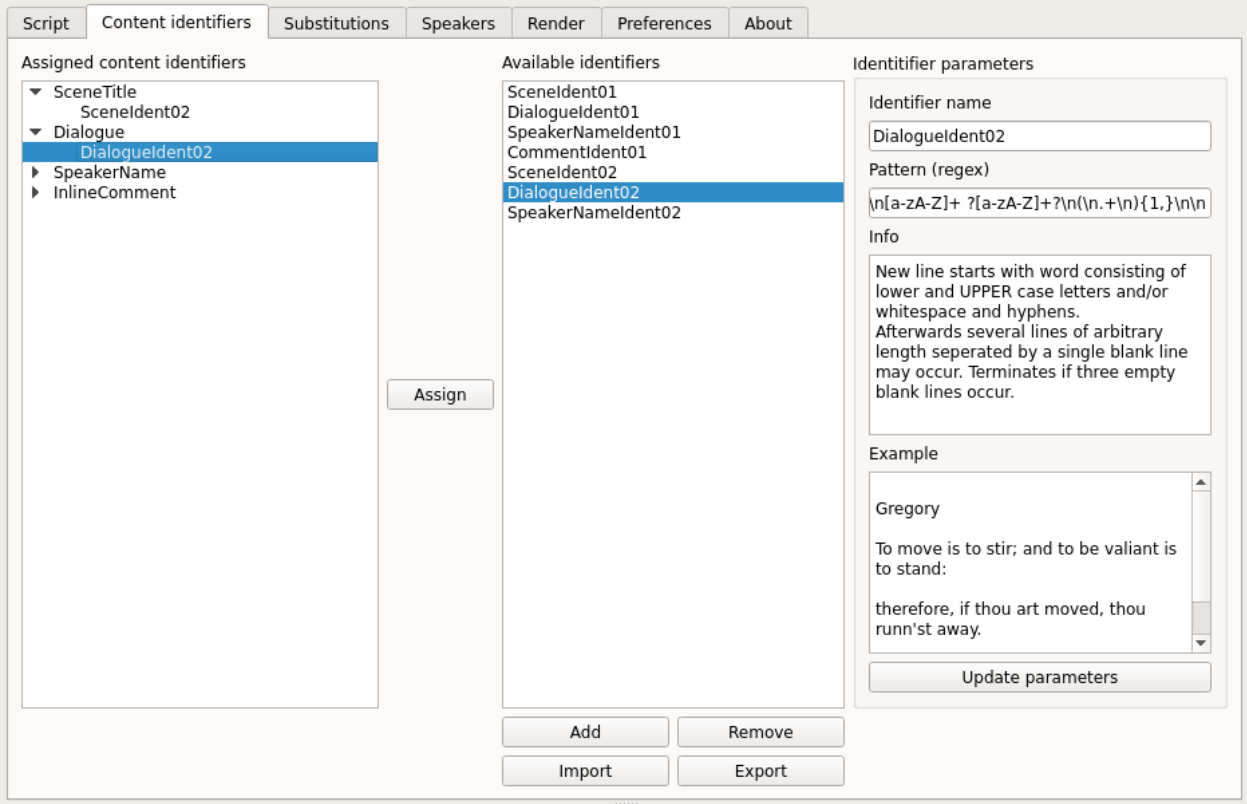
- SceneTitle
- Dialogue
- SpeakerName
- InlineComment.

Note: The identifier 'SpeakerName' is used to assign the content type 'DialogueIndicator'. These names will most likely be harmonized in a future version.

The behavior of the content identifiers was already explained in the *Input format basics* chapter. The list below shows the order of events within the text process.

1. Perform substitutions - see *Substitutions*
2. Search and extract SceneTitle contents
3. Search and extract Dialogue contents - this is the complete dialogue including the speaker name (SpeakerName), and dialog lines (including InlineComments)
4. Within each Dialogue it will extract the DialogueIndicator as defined by the SpeakerName pattern and split it from the rest of the Dialogue
5. In the remaining part of the Dialogue it will search for InlineComments and assign positive matches as InlineComment and text parts not matched as DialogueContent

Content identifiers can be configured in the "Content identifiers" tab.



The "content identifiers" tab basically consists of a tree view for the "assigned identifiers", a list of "available identifiers" and an explanation of the selected "identifier".

Assign content identifiers

The "Assigned content identifiers" view shows which content identifier has been selected for the different types of content in the current project. You can change the assigned identifiers by selecting the identifier type in the tree-view + an available identifier from the list and clicking the "Assign" button.

Browse available content identifiers

The "Available identifiers" list shows all content identifiers available. You can use the buttons below to add new identifiers, remove identifiers and import or export the complete list of identifiers.

Note: Changes made to the content identifiers will not persist through the sessions. In order to use a custom content identifier in a different project you should export the identifiers and import them to the new project. Content identifiers configurations are also stored in the project files.

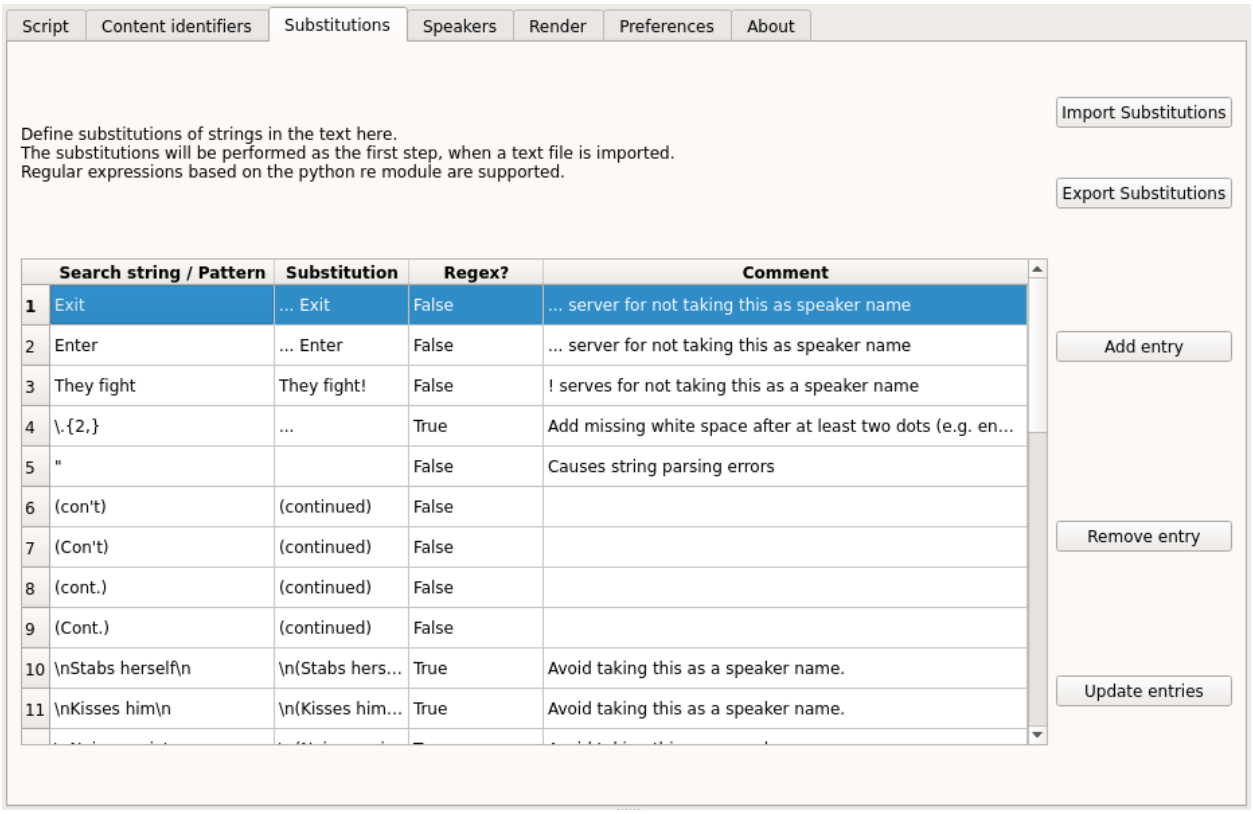
Identifier parameters

> Identifier parameters are shown to explain how the content identifiers work. To explain how the RegEx patterns works the "info" and an "example" field is available.
>
> If you create your own identifiers you should always add an example. When pressing the "Update parameters" button *dramaTTS* will perform a check, to see if your example is matched by the RegEx and display the result in the log.

Note: Although lots of different formats can be identified using (more or less complex) identifiers, the process still relies on having consistency throughout the text document and being "machine" distinguishable. E.g. if a speaker name (as a DialogueIndicator) and a narrative comment share the format ("\nBob Miller\n" vs. "\nEnter Bob\n" - where \n denotes a line break) the defined Regex might find "false" matches. Hence in some cases it makes sense make some changes to the document before importing or use substitutions (see *Substitutions*) to define some replacements (e.g. you could add some special characters to common director instructions like "Enter ..." to distinguish them from character names)

1.3.5 Substitutions

Substitutions provide the possibility to replace words in the text before the script parser is started. Substitutions can be defined in the "Substitutions" tab.

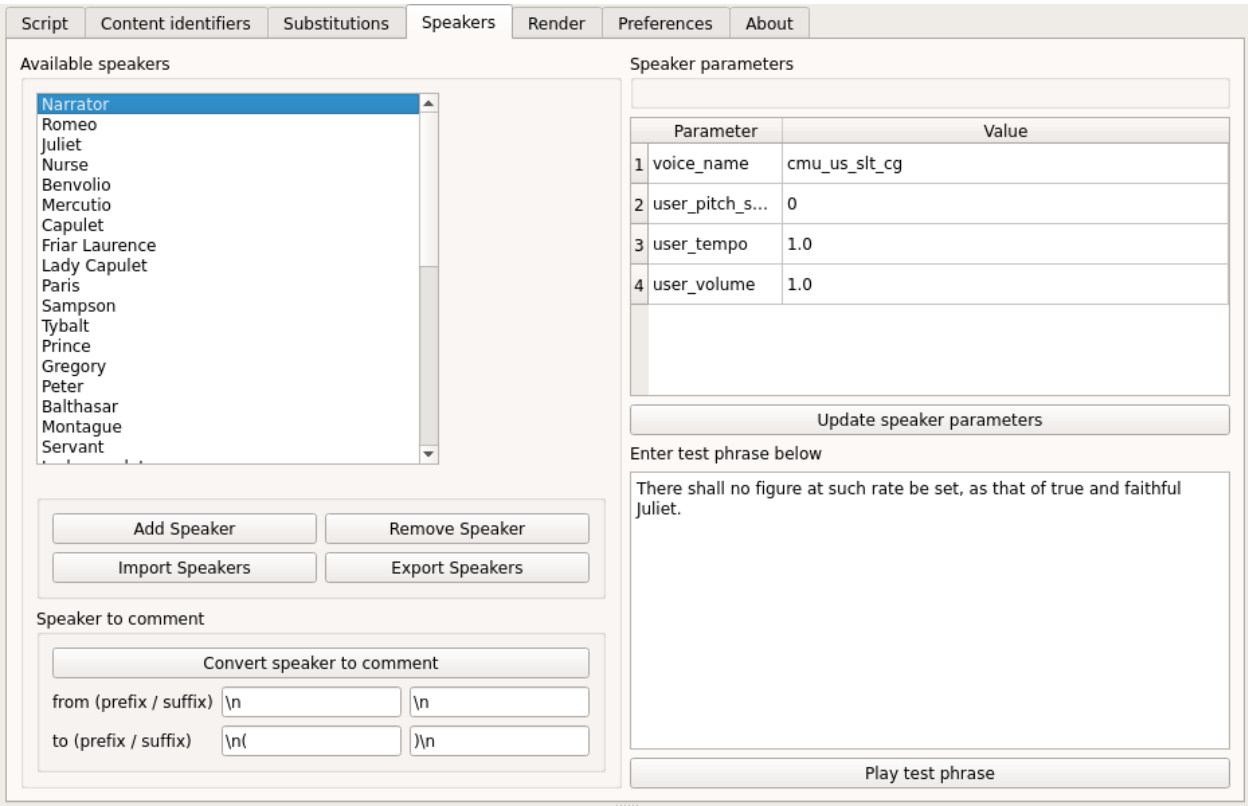


The substitutions consist either of a simple search text or a RegEx pattern. In both cases a substitution text has to be defined as a replacement.

Defined substitutions in the table can be modified and saved by clicking the “Update entries” button. The “Add entry” and “Remove entry” buttons can be used to extend the list or remove the selected substitution. Substitutions can also be exported and imported to a json-file to be shared between projects. You can also define a default substitutions file, which is loaded on startup - see *Adjust program preferences*.

1.3.6 Configure speakers

In order to give a character a distinctive voice, you will have to define its speaker settings. Speakers can be added, modified and ex- or imported in the “Speakers” tab.



Browsing, adding and removing speakers

The list box in the top left corner shows the defined speakers (if you have clicked the "Add speakers" button in the script tab - you should see all characters defined in the play here). With the buttons below the speaker list speakers can be removed or new speakers can be added.

Note: The speaker name must match the character name (case sensitive) - see also characters table in *Characters table*. Lines of a character, who does not have a speaker defined, will be read by the "Narrator".

You can also export and import the speaker definition's.

Speaker parameters

The speaker parameters table shows the parameters (voice name, pitch, tempo and volume) for the selected speaker. You can change the parameters and use "Update speaker parameters" button to save the changes.

Note: Although you can define the volume for each speaker to compensate differences in the loudness of individual voices, a more convenient way is to select the "normalize audio" option (see *Render audio*), which will automatically adjust the volume in the rendered audio files to a predefined dB-level.

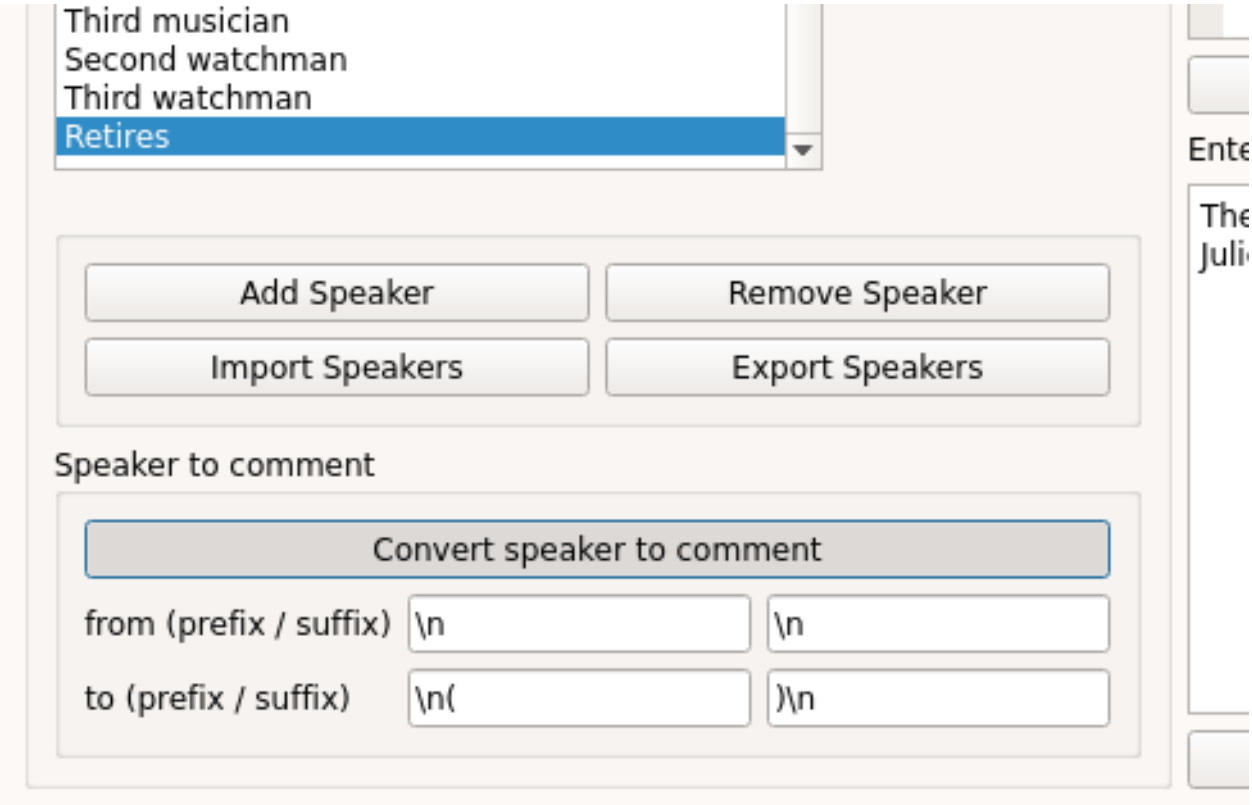
Play a test phrase

Below the speaker parameters you have a text field to define a test phrase. By clicking on the "Play test phrase" button the test phrase will be rendered with the currently defined speaker parameters.

Note: Rendering the test phrase may take a while for long test phrases and of course you need *festival* and *SoX* being installed and correctly configured (check the console output on start-up or see *Render audio*).

Convert a speaker to a comment

The "speaker to comment" button allows to automatically add a substitution for the selected speaker name. This can be useful, if a line has mistakenly been identified as a speaker name. E.g. in the example below a text line containing only the word "Retires" has been identified as a speaker and been added by the "Add speakers" method (since it matches the identifier for a speaker name).



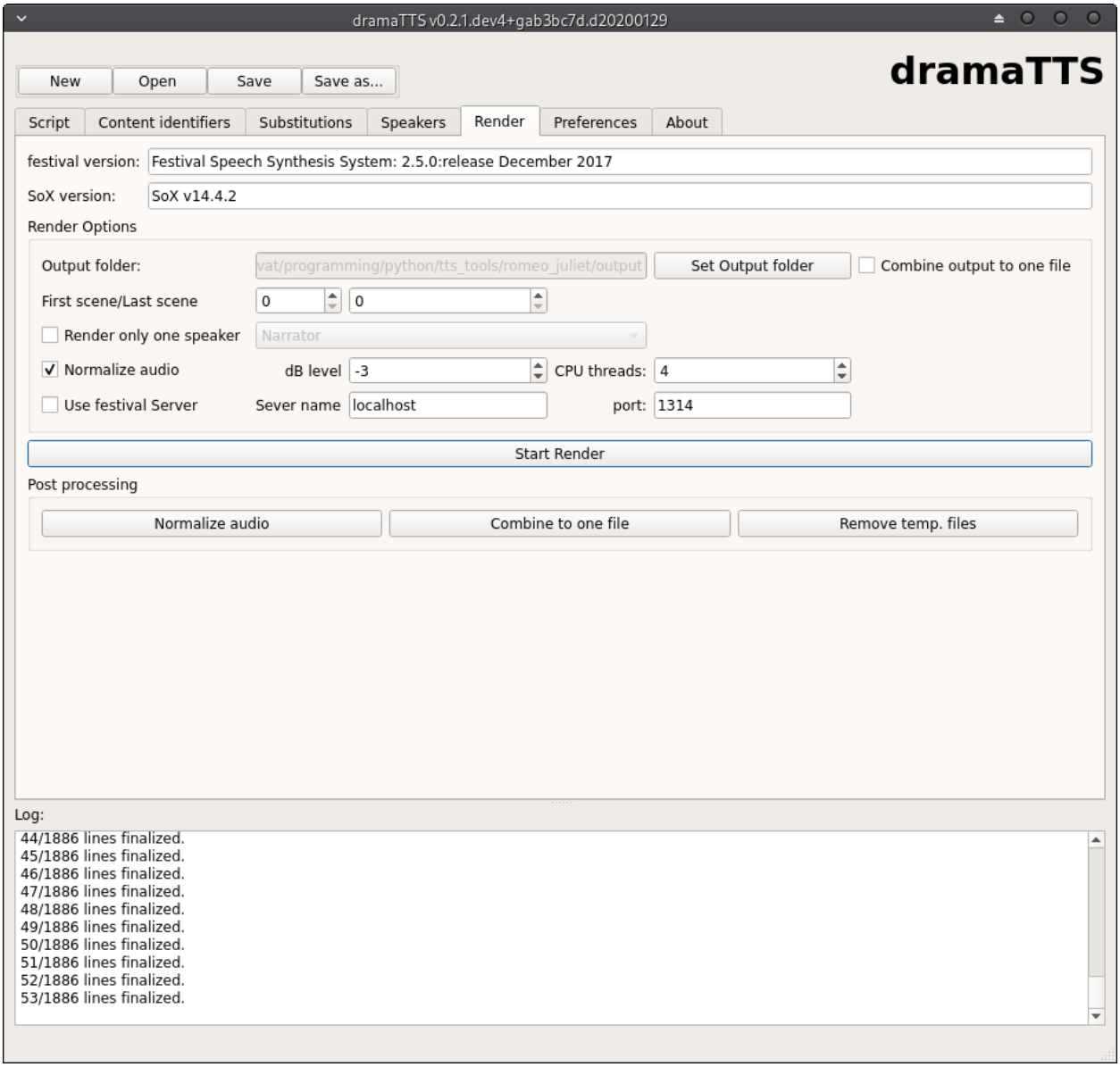
By selecting the wrong "speaker" and clicking on the "Convert speaker to comment" button, the "speaker" will be removed from the speaker list and simultaneously a substitution will be created by pre-/appending the prefixes/suffixes to the search and substitution string will be added to the substitutions list - in the example above this will result in:

32	Listen to this...		False	
33	\nRetires\n	\n(Retires)\n	True	Avoid taking this as a speaker name.

If the script is now re-imported "(Retires)" will not be treated as a speaker (assuming that Speaker names do not allow parenthesis in this case).

1.3.7 Render audio

To render the script to audio files go to the "Render" tab. Here you can configure the options for the audio renderer and start the rendering process.



The render tab also shows, if the *festival* and *SoX* applications are configured correctly. If true, you will see the version number of the applications displayed in the text boxes on the top.

Following options can be defined for the audio renderer:

Output files and folders

Before you start rendering it is mandatory to define an output folder to store the generated wave-files.

Inside the output folder *dramaTTS* will create a sub-folder for each scene - called "scene_XXX" (where XXX will be replaced with the scene numbers e.g. "scene_003").

In the rendering process a wave-file for each line/paragraph will be created. The wave-files will be named "scene_XXX_line_YYY.wav" - e.g. "scene_003_line_002.wav". After having rendered all lines of all scenes the individual wav-files will be merged to a single wave file for each scene - i.e. at the end of the rendering process you will see e.g. a "scene_003.wav" in the output-folder.

Note: After the render is completed you will actually have the recording in two versions - a) the scene files and b) the individual line files in scene sub-folders. You can of course delete the scene sub-folders - however this will prevent certain post-processing and corrective actions (see below) - so it is recommended to keep the files in the sub-folders as long as you plan to make modifications to the project.

Configure render options

Combine output to one file

> This option will - additionally to the scene and line files - create a single wave-file for the whole play, which will be stored on the same level as the output-folder and also have the name of the output-folder appended by a ".wav"

First scene/last scene spin boxes

> With these spin boxes you can limit the rendering process to a certain range of scenes - e.g. useful if you found a mistake in a specific scene and do not want to render the complete script again.
>
> ---
>
> **Note: Regarding the first scene**
>
> If the first line in the text source is not a 'SceneTitle', *dramaTTS* will create a new scene named "Preface" and assign all text up to the first SceneTitle to this scene (scene number 0). In this case the first real scene in the text would actually be scene number 1. This situation is the most common scenario.
>
> If however the text starts directly with a SceneTitle, then this first real scene will be scene number 0.
>
> **Regarding the last scene**
>
> Consider the numbering issues mentioned above as well. Setting the last scene to 0 means to render from the start scene to the end of the play.
>
> ---

Render only one speaker

> If the "Render only one speaker" option is checked you may select any of the characters found in the play and only the lines belonging to this character will rendered. This is useful, if you are unhappy with the rendered audio output of a specific speaker and want re-rendering the lines of the speaker with adjusted speaker settings. This option can also be used in combination with the scene range limiter described above.
>
> ---
>
> **Note:** If you want to use this option to re-render the lines of a specific speaker, you must not delete the individual line wave files in the scene sub-folders. After having re-rendered the lines of a specific speaker all scene files will be re-build as well.
>
> ---

Normalize audio & dB level

> If this option is checked the individual line files will automatically be normalized to the dB level defined by the dB level spin box.

Note: You can also normalize the files as a post-processing step (see below).

CPU threads

dramaTTS supports multi-processing for rendering the audio output, which can significantly reduce render times. In order to achieve fast render times you should select as many threads as (virtual) CPU cores you can spare.

Render times can vary significantly depending on your system, the length of the text and the voices used. To render a complete typical drama/screen play script with the default voices a core i5 CPU running on 4 threads might easily require 1-2 hours of render time.

Use festival server

The *festival* application can also work in a client-server mode - see the festival documentation[1] for details.

Note: If this option is selected *dramaTTS* will assume, that a *festival* server is running on the specified hostname (which can also be an IP-address) and port. To use server option the *festival_client* application has to be configured correctly - see *Adjust program preferences*.

Starting and stopping the render process

The "Start render" button starts the rendering process.

The progress of the rendering process will be displayed in the log.

When you press the "Cancel Render" button the program will cancel all lines, which have not yet been scheduled for rendering. However the currently running threads will not be killed immediately. The program will wait until they have finished their rendering tasks, which might take a while. A status message about the remaining running threads will be shown in the log.

Post processing

dramaTTS currently only supports limited post-processing options.

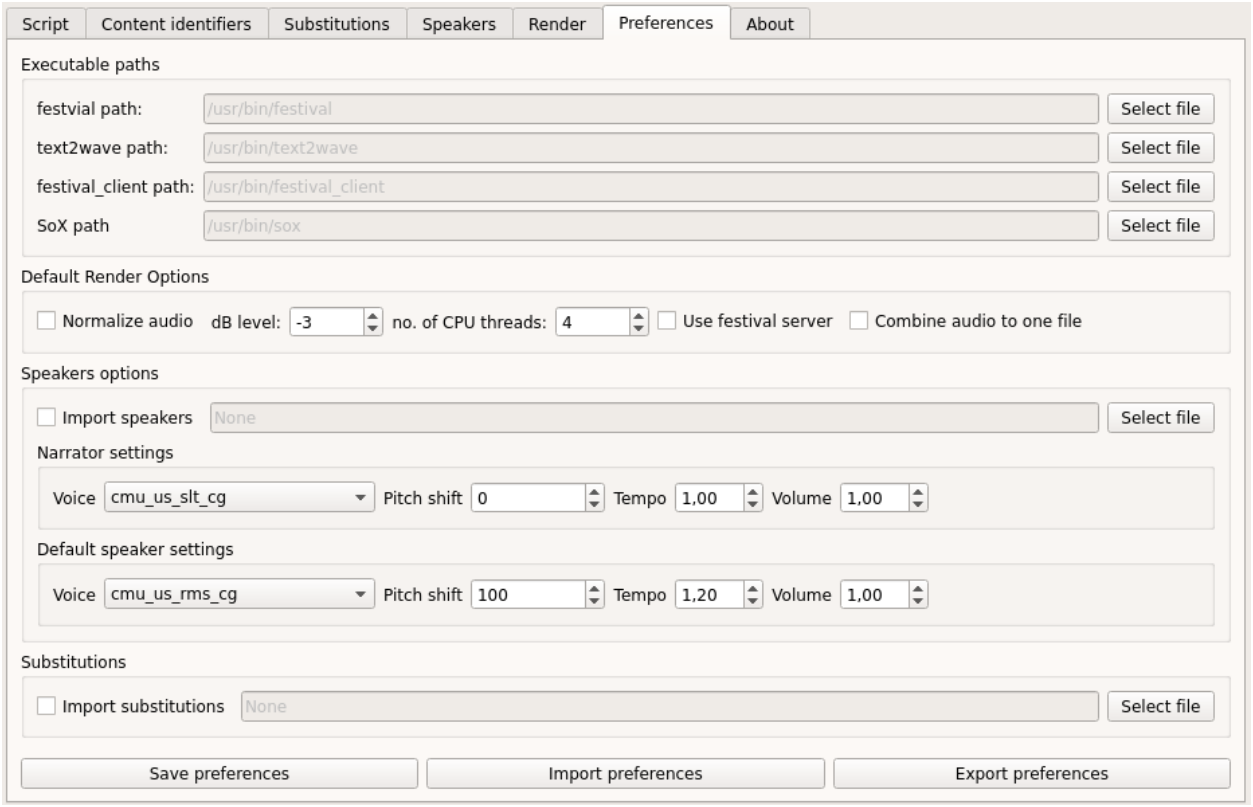
The "normalize audio" and "combine to one file" buttons will perform the same actions as described for the corresponding render options above.

The "remove temp. files" deletes the line wave files. Note that this prevents further post-processing and re-rendering strategies.

1.3.8 Adjust program preferences

The preferences tab provides some configuration options and default settings.

[1] http://www.cstr.ed.ac.uk/projects/festival/manual/festival_28.html#SEC129



Executable paths

Here you can defined the path to the required external applications for audio rendering - i.e. *festival* and its components (*text2wave* and the *festival_client* script) and *SoX*.

If the tools are added to your system PATH, *dramaTTS* should be able to determine the file locations automatically.

Default render options

For some render settings presets can be defined, which will be used when creating a new project (for details on the meaning of these options see the *Render audio* chapter).

Speaker options

If you check the "Import speakers" option, you can define a file with exported speaker settings, which will be imported on start-up.

You can also define the default configuration for the "Narrator" and the "default speaker". The "default speaker" defines the initial speaker settings, when a speaker is added (via the "Add speakers" or "Add speaker" buttons).

Substitutions

Analogous to the "Import speakers" option above, but for substitutions.

With the "Save preferences" button you can save the current preferences to *dramaTTS*'s config file (which is located in the current users home folder and named ".dramatts_config.json" - if you can't find it, you might need to change your file browsers view settings to show hidden files).

You can also export and import the preferences to another file.

1.4 API reference

This chapter contains the API reference for the dramatts.core and dramatts.voices modules.

1.4.1 ScriptParser

class dramatts.core.**ScriptParser**(*input_file=None*, *start_line=0*, *end_line=None*, *substitutions=None*)

A parser to identify content types inside a text file

Parameters

- **input_file** (*pathlib.Path*) – Path to text file
- **start_line** (*int*) – First line to import
- **end_line** (*int*) – Last line to import - None means last line of text file
- **substitutions** (*list*) – A list of text substitutions - details see Attributes

filename
Path to text file

Type pathlib.Path

start_line
First line to import

Type int

end_line
Last line to import - if None last line of text file

Type int

substitutions
A list of dictionaries with keys:

- 'search_text'(str): Text or RegEx pattern to search for
- 'subst'(str): Replacement text
- 'regex'(bool): If true - assume 'search_text' is a RegEx
- 'comment'(str): Additional information on this substitution

Type list

text
Raw text of the imported file

Type str

characters
List of characters in the play

Type list

lines_list
List of dictionaries for each line/paragraph identified in the text - dict keys:

- 'start'(int): index of first character inside the text
- 'end'(int): index of last character inside the text

- 'content'(str): Content of the line/paragraph
- **'content_type'(str):**
  - 'SceneTitle',
  - 'DialogueIndicator',
  - 'DialogueContent',
  - 'InlineComment'
  - 'NarrativeDescription'
- 'speaker'(str): Name of speaker for this line - e.g. a character's name

Type list

content_identifiers
List of dictionaries - for each content identifier with keys:

- 'name'(str): Name of the identifier
- 'pattern'(str): RegEx pattern
- 'info'(str): Explantation of the content identifier
- 'example'(str): An example matching the pattern

Type list

ident_assigments
Assignment of different identifier types to a content identifier - valid keys are: 'SceneTitle', 'Dialogue', 'SpeakerName' and 'InlineComment' - the value must be a content identifier dictionary (e.g. an element of the content_identifiers list)

Type dict

check_content_identifiers()
Checks the content identifiers patterns vs. the example

Returns True if example matches pattern

Return type bool

export_identifiers(*filename*)
Exports the content identifiers

export_parsed_lines(*filename*)
Exports the parsed lines

export_substitutions(*filename*)
Exports the substitutions

get_characters_from_parsed_lines()
Gets characters list from parsed lines and assign it to the characters attribute

Returns None

get_filtered_valid_lines(*first_scene=0*, *last_scene=None*, *speaker=None*)
Filters the valid_lines_list acc. to scene range and speaker name

Parameters

- **first_scene** (*int*) – Start scene

- **last_scene** (*int*) – End scene - if None the last scene of the play will be assumed
- **speaker** (*str*) – Name of speaker to limit filter to

Returns List of dicts - same format as valid_lines_list, but filtered to scene range and/or speaker

Return type list

get_lines_per_character()
Counts the lines/paragraphs for each speaker

Returns key = character name - value = no. of lines, sorted by the number by line count (descending)

Return type dict

identifier_names
Returns a list of available identifier names

Type list

import_identifiers(*filename*)
Imports the list of content identifiers from a file

import_parsed_lines(*filename*)
Imports parsed lines from json

import_substitutions(*filename*)
Imports the list of substitutions from a file

parse_lines(*filename=None*)
Parses the specified text file and assign result to lines_list attribute

Parameters **filename** (*pathlib.Path*) – Path to textfile, if None the filename attribute will be used (if defined)

Returns The value of the lines_list attribute

Return type list

scene_count
Number of scenes in the script

Type int

scenes_titles
List of the scene titles

Type list

substiute_text(*text*)
Substitutes elements in text (supports regex as defined by python re module)

Parameters **text** (*str*) – Text to check/modify

Returns Modified text

Return type str

valid_lines_list
Only non-empty lines/paragraphs of the lines_list attribute

Note: Has additional key-value pairs cmp. lines_list:

- 'scene_no'(int): Scene the line/paragraph belongs to

- 'scene_line_no'(int): Line number inside the side (where 0 is always the SceneTitle).

Type list

1.4.2 AudioRenderer

class dramatts.core.**AudioRenderer**(*voices=None*, *cpu_threads=4*)

This class provides configuration of the audio rendering and wrappers to launch the external tools (festival, sox) with predefined parameters for speech synthesis and post-processing

Parameters

- **voices** (*dict*) – Dictionary with key = speaker name, value is a VoiceConfig object
- **cpu_threads** (*int*) – Number of parallel CPU threads to use for rendering

voices
Dictionary with key = speaker name, value is a VoiceConfig object

Type dict

cpu_threads
Number of parallel CPU threads to use for rendering

Type int

norm_level
DB level to use for normalizing audio

Type int

sox_path
Path to the SoX application

Type pathlib.Path

festival_path
Path to the festival application

Type pathlib.Path

festival_client_path
Path to the festival_client application

Type pathlib.Path

text2wave_path
Path to the text2wave script

Type pathlib.Path

installed_festival_voices
List of voice names installed for festival

Type list

external_tools
A dictionary with the configuration setting for the external tools - keys: 'festival', 'festival_client', 'sox' and 'text2wave'. Each tool's key holds another dictionary with the keys:

- 'path'(pathlib.Path): Path to application
- 'version'(str): Version string

- 'installed'(bool): True if the tool was found to be working

Type dict

use_festival_server
True if server should be used instead of local render via text2wave

Type bool

festival_server_name
Hostname or IP-Address of the festival server

Type str

festival_server_port
Port number of the server

Type int

invalid_voices
List of speaker names with invalid configuration (e.g. defined festival voices not installed) - a dict for each invalid speaker with keys:

- 'name'(str): Name of the speaker
- 'index'(int): Position of the speaker in the voices attribute
- 'params'(list): A list of invalid parameter names - e.g. ['voice_name']

Type list

check_component(*path*, *tool_name*, *test_args*, *answer_str*, *answer_version=False*)
Checks an individual external tool by invoking the tool with defined test arguments and validating against an expected answer

Parameters

- **path** (*pathlib.Path*) – Path to executable (incl. executable name)
- **tool_name** (*str*) – Name of tool
- **test_args** (*tuple*) – Command line arguments to use for functions test (e.g. to get version string)
- **answer_str** (*str*) – A required str in the answer of the command line invocation with test_args
- **answer_version** (*bool*) – Extract version str form

Returns with keys: path, version and installed.

Return type dict

check_installed_components()
Checks if external components are installed and updates the external_tools attribute

Returns: None

check_voices()

Checks the voice parameters for correctness (e.g. voice installed) and assigns invalid voices to the invalid_voices attribute

Returns List of dicts with speaker name, index and invalid parameter in self.voices

Return type list

create_voices_dict()
Create a dictionary with all voices

export_voices(*filename*)
Exports all voices to a json file

static get_ext_tool_path(*exec_name*)
Get path to external executable

Parameters **exec_name** (*str*) – Name of the application/executable

Returns Path to executable if found - else None

Return type str

get_festvial_voices()
list: List of the installed festival voices (str)

static get_tool_feedback(*command_name*, *path=None*, *cmd_args=('–version',)*)
Calls an external tool with certain arguments and returns feedback (for testing functionality)

Parameters

- **command_name** (*str*) – Name of command to check
- **path** (*pathlib.Path*) – Path to the executable, if None see below
- **cmd_args** (*tuple*) – Command line arguments to pass to tool

---

Note: If path is none the command will be invoked only with the commnand_name - assuming the executable's location is added to the PATH environment variable.

---

import_voices(*filename*)
Imports voices from a json file

play_audio_test(*voice*, *text*)
Plays a test text for the current voice

Parameters

- **voice** (VoiceConfig) – voice configuration
- **text** (*str*) – Text for testing

read_voices_dict(*voices_dict*)
Fills voices attribute dict with VoiceConfig objects defined by values from a dictionary

Parameters **voices_dict** (*dict*) – Dictionary with key for each speaker name. Each speaker key holds another dictionary with voice parameters with following keys: - 'voice_name'(str): Festival voice name - 'user_pitch_shift'(int): Pitch shift offset - 'user_tempo'(float): Tempo factor - 'user_volume'(float): Volume factor

render_audio(*voice*, *text*, *filename*, *normalize=False*)
Renders a specific script line (or part of a line) into an audio file

Parameters

- **voice** (*str*) – Name of speaker
- **text** (*str*) – Text to render
- **filename** (*pathlib.Path*) – Path object

- **normalize** (*bool*) – If true normalize the sound level

1.4.3 ProjectManager

class dramatts.core.**ProjectManager**(*script_parser*, *audio_renderer*, *output_folder=None*)

Manager for a dramaTTS session

Parameters

- **script_parser** (ScriptParser) – ScriptParser object
- **audio_renderer** (AudioRenderer) – AudioRenderer object
- **output_folder** (*pathlib.Path*) – Folder for the generated audio files

project_filepath

Path to the project file

Type pathlib.Path

script_parser

ScriptParser object

Type *ScriptParser*

audio_renderer

AudioRenderer object

Type *AudioRenderer*

output_folder

Folder for the generated audio files

Type pathlib.Path

speaker_to_render

Speaker name, if output for a specific speaker only shall be rendered (otherwise None)

Type str

normalize

If true the audio output will be normalized

Type bool

combine

If true the files for the individual scenes will be combined to single file too

Type bool

start_scene

First scene to render

Type int

end_scene

Last scene to render - if None the last scene in the play will be assumed

Type int

digits_scene_no

Number of digits for zero padding in the scene folders/file names

Type int

digits_line_no
Number of digist for zero padding of the line numbers in the wave files

Type int

preferences
Dictonary to store the program preferences - see create_prefs_dict method for details

Type dict

add_speakers_for_characters()
Adds default speaker for each character found by the script parser, who does not alerady have a speaker assigned

combine_audio(*input_path=None*, *output_path=None*)
Merges all files in one directory (excluding subdirs) into a single file

Parameters

- **input_path** (*pathlib.Path*) – Folder name of files to combine (defaults to output_folder)
- **output_path** (*pathlib.Path*) – Filename of combined file (defaults to output_folder + .wav)

convert_speaker_to_comment(*speaker_name*, *from_prefix='\n'*, *from_suffix='\n'*, *to_prefix='\n('*, *to_suffix='\n)'*)
Deletes and entry in the speakers dictionary and creates a substitution adding prefixes and suffixes

Parameters

- **from_prefix** (*str*) – Prefix before speaker_name to include in search text
- **from_suffix** (*str*) – Suffix after speaker_name to include in search text
- **to_prefix** (*str*) – Prefix before speaker_name to include in substitution
- **to_suffix** (*str*) – Suffix after speaker_name to include in subsitution

Notes

- This is useful, if the parser misunderstood director instructions for speaker names (DialogueIndicator)

create_prefs_dict()
dict: Creates a dictionary with the preferences settings

create_project_dict()
dict: Creates a dictionary representation of the current project settings

default_config_path
Default config path

Type pathlib.Path

get_line_no_str(*line_no*)
str: Returns a string representation of the line number

get_scene_folder_path(*scene_no*)
pathlib.Path: Returns the path to a scene subfolder

get_scene_no_str(*scene_no*)
str: Returns a string representation of the scene number

load_preferences(*filepath=None*)
Loads the preferences from a json file

Parameters **filepath** (*pathlib.Path*) – If none the default config file (see default_config_path property method)

load_project(*filepath=None*)
Loads a project from a file

Parameters **filepath** (*pathlib.Path*) – path to file name (if None self.project_filepath will be used)

normalize_audio()
Normalizes all audio segments and rebuilds files for each scene

process_prefs_dict(*prefs*)
Processes the preferences dict (see create_prefs_dict) and assings values to internal variables

process_project_dict(*project*)
Reads a project dict and assigns to internal variables

render_line_from_dict(*line_dict*)
Renders a line based on the dictionary provided by the script parser

Parameters **line_dict** (*dict*) – An element of the list returned by ScriptParser.get_filtered_valid_lines method

Returns Returns the argument

Return type dict

render_script()
Renders the script into audio files

save_preferences(*filepath=None*)
Saves the preferences to a json file

Parameters **filepath** (*pathlib.Path*) – If none the default config file will be used (see default_config_path property method)

save_project(*filepath=None*)
Save the complete project

Parameters **filepath** (*pathlib.Path*) – path to file name (if None self.project_filepath will be used)

1.4.4 Helper functions

dramatts.core.**handle_getstatusoutput**(*commandpath*, *cmd_args*)
Returns the output from subprocess.getstatusoutput taking into account the os.name

Parameters

- **commandpath** (*pathlib.Path*) – Path to executable including executable name
- **cmd_args** (*tuple*) – Tuple with command line arguments (str) to pass to the programm

dramatts.core.**convert_version_string**(*version_str*)
Converts a version string into integer values for major, minor, patch

Parameters **version_str** (*str*) – Version string like '1.0.0' or '1.0'

Returns Tuple of int values (major, minor, patch). Patch will be None, if two digit version_str provided.

Return type tuple

1.4.5 VoiceConfig

class dramatts.voices.**VoiceConfig**(*voice_name='cmu_us_slt_cg'*, *pitch_shift=0*, *tempo=1.0*, *volume=1.0*, *speaker_name=None*)

Creates a voice config that allows changing pitch and tempo of the festival voices

Parameters

- **voice_name** (*str*) – A festival voice name
- **pitch_shift** (*int*) – SoX pitch shift value (+-100ths of a semitone)
- **tempo** (*float*) – SoX tempo factor (1.0 = no change)
- **volume** (*float*) – Volume factor (1.0 = no change, >1.0 = louder)

create_voice_dict()

Creates a dictionary representation of the voice config

Returns

Dictionary representation of voice config with keys:

- 'voice_name'(str): Name of the voice in festival
- 'user_pitch_shift'(int): Pitch shift in 1/100th of semi-tone
- 'user_tempo'(float): Tempo factor
- 'user_volume'(float): Volume factor

Return type dict

pitch_shift

Final pitch shift

Type int

tempo

Final tempo

Type float

update_parameters(*voice=None*, *pitch_shift=0*, *tempo=1.0*, *volume=1.0*)

Updates the voice parameters

Parameters

- **voice** (*str*) – Name of the voice in festival
- **pitch_shift** (*int*) – Pitch shift in 1/100th of semi-tone
- **tempo** (*float*) – Tempo factor
- **volume** (*float*) – Volume factor

volume

Final volume

Type float

CHAPTER 2

Links

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Index

H

I

L

N

O

P

R

S

T

U

V