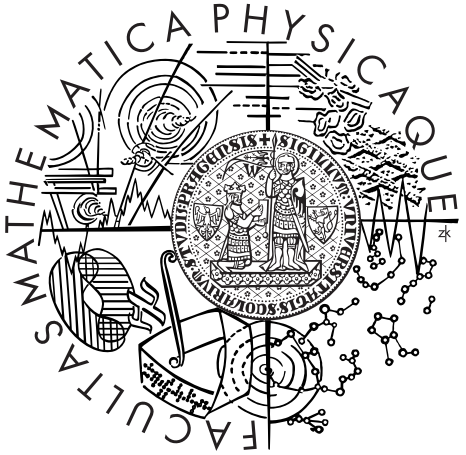
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Peter Šípoš

Scripting in Audacity Audio Editor

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: RNDr. Tomáš Pop
Study programme: Computer Science
Specialization: Programming

Prague 2013

Dedication. I wish to dedicate this thesis to my parents for their support and for my supervisor for his patience.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Scripting in Audacity Audio Editor

Autor: Peter Šípoš

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Tomáš Pop, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem této bakalářské práce byl vytvoření doplňku k programu Audacity, což by umožněl používání scriptu během editaci zvukovych nahrávek.

V první části práce je popsaná architektura Audacity, jaké jsou možnosti pro doplnění funkčnosti a jaké alternatíva jsou k dispozici. Poté popíše kritické rozhodnutí, které byly učiněné před implemetaci doplňku. Tyhle rozhodnutí byly ve spojení s výběrem scriptovacího jazyku, spravování chybových stavů a návrhem grafického rozhraní. Na konci práce pomoci příkladů je ukázan funkčnost doplňku.

Na závěr, aplikace byl naimplementován, jako ukázka technik, které byly popsané v práci.

Klíčová slova: Scriptování, Audacity, Editace zvuku

Title: Scripting in Audacity Audio Editor

Author: Peter Šípoš

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Tomáš Pop, Department of Distributed and Dependable Systems

Abstract: Audacity is popular and widely used audio editor available for all the main platforms including Windows, Mac as well as various Linux distributions. Audacity functionality can be controlled via sophisticated user interface, but the editor suffers from a lack of support for an automated execution of scripts and therefore, using audacity e.g., to perform the same action on multiple files can be tedious.

The thesis aims at extending Audacity editor to allow using scripts in the audio editing workflow.

The first part of the thesis overviews Audacity's architecture, and discuss, how Audacity can be extended and what alternative applications are available. Then, the thesis describes the most important decisions that were taken, including the choice of scripting language, managing errors and designing user interface. Finally, the extension functionality is shown on several examples reflecting a typical use-cases.

Keywords: Scripting, Audacity, Audio Editing

Contents

Introduction

In the beginning of computer history the computers were dedicated to scientifical calculations or office work. Later, they became capable of displaying pictures, playing audio and video files. However, these computers were not enough powerful to create the audio and video materials, thus for this purpose special equipment was used.

As the computating power of home computers was increasing, creating multimedia materials became easier in home environment. In their time the most advanced home computers, that were capable of audio editing were Amiga computers from Commodore. These computers could be extended with digitization boards, that made them capable of recording audio signals. Moreover, audio editing software was available for them, so they were fully equipped to become a home cut studio. Because of their reasonable pricing these computers became very popular.

The IBM-compatible personal computers were also becoming more powerful, so playing and recording audio using them became real, too. Several audio editor softwares were created. Amongst them, maybe the best-known applications were *Cool Edit* and *Soundforge*. These applications are also being developed nowadays. During the years the Cool Edit was taken over by Adobe, that releases it under name Audition and the Sound Forge is property of Sony Corporation now.

Beside the proprietary software several open-source alternatives appeared. I would like to mention two of them: the *Audacity* and the *SoX*. While the SoX is a command-line application, that focuses on playing and recording audio files, but it supports some basic editing task, such as mixing, the Audacity is a complete audio editing software.

The Audacity is a fairly well-known audio editing software, which is used by a lot of users. It's a multiplatform application, that can run on Windows, Linux and MacOS operating systems. It's functionality is pretty extensive. It supports e.g. multitrack editing, filters, plugins .... Through its graphical interface the commands can be easily accessed. However, its mightiness,focusing on graphical user experience,is its biggest deficiency: repeatedly performing the same operations on several audio files is inconvenient.

This thesis aims to tackle this deficiency, which means, that the execution of "batch commands" become more convenient. This can be reached by creating a scripting interface, that,through a selected scripting language,makes the Audacity's inner functionality available for scripting purposes.Thanks to this interface the user will be able to control Audacity with a script, that performs the task by calling the Audacity's functions.

Choosing Audacity as a target of this thesis is beneficial, because of its large user base. These users won't need to get used to a new audio editor and meanwhile they can use scripts to make their tasks easier.

Moreover, to make the workflow more convenient the scripting interface created as a part of this thesis will contain a script editor. This editor will provide the basic editing functionality including syntax-highlighting.

1. Background

In this chapter I would like to describe some techniques, that are used in my application and in Audacity itself.

1.1 The Architecture of Audacity

I use Audacity because it is a well-known, continuously developed and mature audio editor. However, it is designed as an application with graphical user interface, currently with restricted capabilities at scripting. Fortunately, Audacity supports plugins. The main purpose of these plugins is to add some sound-aware capabilities, such as generating DTMF tones. These plugins uses documented function calls and have access to all components building up the editor.

Fortunately, the developers of Audacity project are aware of this weakness. The experimental plugin mod-script-pipe can be used to access the Audacity's functions from an external application. Its operation is more detailly described in Section 2.1.

Audacity uses wxWidgets library to create its user interface by implementenging all necessary functions to draw the components and to handle events. This style of design comes to be useful for mod-script-pipe. This plugin uses a technique called window-hijacking to take control over the GUI components. This means that the plugin creates its own instances of these components, so it has access to their functions.

1.2 Interprocess Communication

This thesis does not aim at comprehensive overview of interprocess communication, but since IPC is a very crucial part of my application I would like to describe this technique.

There are nine options for IPC in Windows operating systems, which are the following [1] : clipboard, component opject model[1], data copy, dynamic data exchange[2], file mapping, mailslots, pipes, remote procedure call[3] and Windows sockets.

Each process in the system can communicate with other processes through one of the IPC methods. The simplest example from user perspective is copying an image from webpage to the clipboard and after pasting it to a text document. However, in the backgrund several requirements must be fulfilled. First af all, both processes must be prepared for the clipboard feature (this is not problematic nowadays) and the receiver must support the format of the data in the clipboard. For example, copying a video clip from video editing software to the Notepad will not work. Another problem with the clipboard is that every single process can access it. This is acceptable when the user perform the above mentioned tasks,

[1] Also known under acronym COM
[2] Also known under acronym DDE.
[3] Also known under acronym RPC.

but it can be really problem when some automated mechanism use it, because the user interaction can interfere with them.

As I will describe in Section 2.1 the implemented application in this thesis will use named pipes to communicate with Audacity. Named pipes,beside the anonymous pipes, are a specific type of pipes. While the anonymous pipes are best suitable for redirecting standard input and output between the child and parent process [1], the named pipes can be used between fully distinct processes, even if they are running on separate computers [2].

1.3 Scripting Languages in General

It tends to be hard to define, what scripting languages are. Usually scripting languages are such computer languages that are interpreted rather than compiled [3]. Another way to describe the scripting languages is definition of scripts itself. While the source code of an application written e.g. in C++ calls functions implemented in the same language, the source code of a program (or more likely the script) written in a scripting languages usually calls functions implemented in another language [3].

Another option to define the scripting languages is specify them as domain specific languages. Usually they are prepared for a specific environment, while the general purpose programming languages, as their name predicts, can be used for a wide spectrum tasks. For instance the bash is considered as a scripting programming language. It is designed to control the Unix operating system by executing commands, which are usually implemented in a different programming language. The Microsoft's Powershell is very similar to the bash and not surprisingly it is also considered as a scripting language.

The third option to distinguish the scripting languages from the rest of language's world is making difference between compiled and interpreted languages. Scripting languages are usually interpreted languages, which means that the source code is not translated to a machine code or an intermediate code, but the interpreter directly interprets it. For instance the bash is interpreted language. However in some cases this is not so straightforward. The above mentioned PowerShell can e.g. access .NET libraries, which are precompiled to an intermediate language. Moreover, the new JavaScript implementations in web browser before the execution compile the script, because of performance issues.

In the previous paragraph I mentioned the JavaScript. Despite the slightly unusual behaviour of its new interpreters it is considered as being a scripting language. The code written in JavaScript more likely calls functions that are implemented in a different language. For instance an HTML5 game uses JavaScript code to perform painting, but these functions are basically implemented in a different language. What is more, the JavaScript's initial purpose was enriching websites, which declares it to be a domain specific language.

1.4 Related Work

In the previous Section 1.3 I described the scripting languages by mentioning some of them. The bash and PowerShell are mainly used for interacting with

the operating system and its components (e.g. file copying, application installation, ...). Another widespread scripting languages are Perl and Python. One implementation of Python is related to this thesis. That is the *GIMP-python* [4], which enables creating plugins for GIMP using the Python language.

However, for GIMP there is another scripting interface. It is called the *Script-Fu* [5].

2. Analysis

In the following chapter I would like to discuss my decisions concerning the Audacity and the scripting interface in the early stages of development.

2.1 Overall Architecture

Audacity is designed to be controlled primarily using mouse through user interface or with a keyboard using shortcuts. However, we need to control it with commands sent from a script. In order to achieve this we need to utilize some kind of programming interface which can be called from our script.

Principaly there are two possible options to tackle this problem.First option is to implement new interface to Audacity, by modifyng its source code and adding necessary functions to existing parts of the application. The main benefit of this approach is having full control of capabilities accessible from script.

On the other hand, there is a major disadvantage, because it is hard to push patches directly to main Audacity source chain. This means that after new version release of Audacity the whole editor or some components may not work.

Second option is to create a standalone application, which communicates with Audacity using an existing interface. Implementing such an interface implies, that we can access to the Audacity using standardized functions. This enables us maintain our code independent from the Audacity's source, which can be handy in the future as the applications are being developed.

I decided to use the second approach and within this to use an interface which is an experimental module written by Audacity team. The module is called mod-script-pipe and is a plugin for the Audacity.

The module enables us to control the main application by commands sent via a system pipe and it is primarily designed to be called from scripts.

The advantages of the second approach are the following:

- We communicate with the application via a standard channel
- The plugin is going to be developed in future
- We need not modify existing Audacity source

The main disadvantages of mod-script-pipe are the following:

- The plugin is still only available in beta status
- It is not included in standard Audacity executables, it should be compiled from source code
- Already existing functions can change
- Its documentation is in early stages

Considering all advantages and disadvantages of these three approaches - full application implemented, using external plugin or implement a new one with a standalone application - I decided to implement an interface connecting to the plugin not touching the Audacity's source code.

2.2 Choosing the Implementation Programming Language

Audacity is written in C++ language using wxWidgets library to make it available for Windows and Unix-like systems, too. However, my decision to create a standalone application gave me freedom to choose a different programming language.

I decided to build the application based on .NET environment using C# languange. The main reasons for my decision were the following:

- My application won't execute calculation demanding tasks, so built-in optimalization for .NET code will be enough sufficient
- More convenient use, because the .NET environment controls some areas itself, e.g. disposing unused objects using garbage collector
- The chosen JavaScript interpreter is also written in managed code

2.3 Choosing Scripting Language

Choosing the language user uses to interact with my application is very important. In future, the application should be able to use its own plugins for different scripting languages, but in the context of this thesis I decided to use JavaScript. JavaScript is pretty well-known scripting languages among the users who use scripting in their everyday life, so they can get used to it very fast in this case, too. Moreover, the JavaScript supports features, that are necessary to interact with Audacity's scripting API, e.g. classes, functions.

Upon my decision to use C# language I had to focus on interpreters, that can be accessed from the .NET environment. I considered two JavaScript interpreters, which could be used for my application: the Node.js [6] and JavaScript.NET [7] . Both solutions have their advantages and disadvantages. Node.js is based on Google Chrome's JavaScript engine called V8, which is fast, supports a lot of functions (e.g. disk I/O, networking, encryption, ...). The JavaScript.NET is way more simple implementation of the language interpreter. It focuses purely on the language itself and we have to implement the classes we need to interact with. On the other hand, I found, that the main advantage of this interpreter is, that it can be attached to the main project as a DLL library, so no additional user interaction is needed, while the Node.js should be installed on the target computer.

According to the above mentioned causes I decided to implement my application using the JavaScript.NET interpreter.

2.4 Error Handling

User-created scripts can contain errors or errors can occur while the script is being executed and thus, error handling is crucial part of script interpreters.

Runtime errors can be handled in different ways. One option is implementing functions which always return a value, even when an error occured.In this case

the returned value contains information about the failure. Another method is let the function execution interrupt on when an error happened. In this case an exception is thrown to describe the errorstate.

The above mentioned approaches have their advantages and disadvantages. Managing exceptions consumes more resources than an usual execution flow. However, exceptions shouldn't be very frequent.

Idea of functions always returning some values may sound good, but it has several drawbacks. The .NET Runtime signalizes errors by exceptions. This means, that in this particular case all exceptions should be handled inside our API, which brings several problems:

- There will be redundant code parts within the source code, caused by similar errors happening at different actions (errors at pipe communication can be such an example)
- Signaling error statuses needs our own system, which would basically be the same or very similar to the system implemented in .NET itself

Based on previously mentioned causes I decided to use the exceptions for error handling rather than creating some special return values for my API functions.

2.5 Audacity API

The mentioned Audacity extension uses its own commands to control the main program's interface. These commands can be supposed to be the Application Programming Interface (API) for the main program.

The commands form two groups:

General commands They control general functionality e.g. creating a new track or playing the audio file

Menu commands They control functions which are accessible from the menus e.g. using some filters

Menu commands control functions thar are usually invoked from the menu structure of Audacity's graphical interface. Such a function is applying a filter to our audio file. General commands can be described as everything else, including the menu commands. In fact, menu commands form subset of all commands, but they are special, because majority off executed commands is menu commands.

These commands are sent through one of the pipes created by the plugin. Their results are returned through the second pipe. However, these commands have to be specially formed their interpreter can withstand some malformation. On the other hand, these malformations are allowed mostly for whitespaces (see: example file for the plugin), so our custom interpreter really should respect the conventions.

2.6 Script Environment

It's important for users to have ability to customize the environment for their needs. This feature is needed because of their different needs based on different properties of their scripts. For example, some users have mainly simple, short scripts for that the default settings are sufficient. Other users creates more complex scripts, so they may need to adjust the environment. An example of this situation could be a script which is not necesseraly connects to Audacity depending on some function results. In this case there is no need to automatically connect to the Audacity, therefore automatic connection and cleanup is not necessary.

On the other hand, having only global settings may not be ideal, because users can obtain scripts from external sources, which may naturally be written for different environment settings. I considered the option of adding swiches to scripts' source code, which override the system defaults. The idea came from compiled programming languages, such as Pascal and C, where these switches adjust the compiler [9]. However I found, that implementing switches, that are available from the scripts, can be confusing, because of mixing several approaches of scripting languages and macros.

Finally, I decided to add a graphical option to control the automatic initiliazation and cleanup process.

2.7 Storage Settings

We want to keep our settings for script environment. To store these data the simplest way is to use a plain text file. In my opinion the amount of data is quite small, so I decided to use the plain text format. I also considered using XML format to save data, but serialized data may be too complicated for editing by hand, which can be useful in some cases.

The main problem was to decide, where to save the file. In this work I create the program and I provide testing on Windows operating system. This implies,that I have to count with the Windows multi user environment.

Since Windows Vista the Program Files directory is not accessible for writing without special permissions. Moreover, we would have to manage users within the AudaEditor. These factors mean, that using the application directory or simply one file in an user specified directory is inconvenient.

As a solution I found to use the user's appdata folder to store the settings file. To access this folder the Application [10] class is used from the .NET framework, in which the UserAppDataPath property stores the path.

As result, we use the Windows' user management and don't have to care about the problem of permissions, because the file is also accessible in user mode.

2.8 File Access

The scripting interface communicates with Audacity using pipes and Audacity loads input files on its own. However, in some cases we need to access files from the script. These file operations involve accessing to file system to retrieve the

directory content. After the list of files in the specific directory is ready we can go through it and pass each needed filename to Audacity.

However, passing file names to Audacity can be problematic. As I mentioned in Section 2.1, my scripting interface uses named pipes to communicate with Audacity. However, in the current version of mod-script-pipe there is a restriction, that filenames passed through the pipe must not contain spaces, because space is used as deliminiter.

As a solution I considered a workaround, that would involve copying the input file with invalid pathname to a temporary storage and after that the temporary pathname would be passed to Audacity. However, I found this solution problematic, because when script is being executed for multiple times for every run files should be copied and after execution deleted. This solution would be unusual and time consuming in most of the cases.

As a secondary solution I implemented an exception that is thrown, when an audio file with illegal file name is accessed. Thanks to this option user can know about the problem and act accordingly.

I would like to mention, that hhere is no support for file access in the current JavaScript.NET interpreter, but we can implement a wrapper for I/O functions. The wrapper can be internal part of the AudaEditor. Using this approach, wrapper is always accessible from script file. Another option is to create a standalone .NET library implementing the needed functions and a javascript wrapper to attach it to the actual project. This second approach is simply making a wrapper around a wrapper and because of file Input-Output operations tend to be frequent I decided to use the first option.

2.9 Graphical Interface

The Audacity is implemented in C++ using wxWidgets library for graphical user interface. Thanks to this library Audacity is a cross–platform application.There is an existing wrapper for wxWidgets, that makes possible to use the library in .NET environment, called WX.Net [12]. However, it seems, that development can be stoppedin the future, so I decided to abandon this solution and implement the scripting interface based purely on graphical elements available in .NET libraries. Fortunately, this does not necessarily means focusing on one platform, because thanks to Mono project, the application can also run on other operating systems.

2.9.1 Editor

The main area of the scripting interface is dedicated to the editor. Beside the script editing the editor also should provide at least a basic level of safety for our source code. To fulfil this need the editor should warn the user if the source code is unsaved and in this context dangerous situation happens, like creating a new file or simply closing the window of the application.

I decided to implement a basic syntax highlighting mechanism for the editor, which distinguish two types of keywords. One group of keywords consists of reserved words for JavaScript (e.g. *if*, *else*, *for*, *var*, *function*, *function*,...), and the other group consist of the keywords, that are used to describe the objects implemented as part of this thesis (*auda*, *filemanager*, *console*).

2.9.2 User Interface–Script Interaction

When we are running are script we want to get some feedback if there was a problem or everything was good. For signalizing problems during the editing I have chosen the messageboxes, but for script execution they can be disturbing if some errors appear more frequently and the flashng messageboxes became disturbing, thus I decided to use textboxes to show status messages or error information. Controlling the script execution is also connected with UI–Script interaction, which is in a more detailed way described in Section 3.4.

2.10 Projects And Files

Scripts are usually a short portion of code (source: wiki) , meaning that creating projects may not be necessary. On the other hand, grouping the requested resources for the source can easen the development by reusing once implementing code. In this case the projects mean a set of files, which contains the main script, javascript libraries (javascript files containing function definitions and requested function calls, settings), dynamic libraries with their javascript wrappers and the project settings.

The current javascript interpreter doesn't support execution of more than one script, but this problem can be solved by concatenating the codes and preprocessing the file.

3. Implementation

In this chapter I would like to describe several parts of my implementation of the scripting interface using the tools mentioned in previous chapters.

3.1 The Overall Architecture

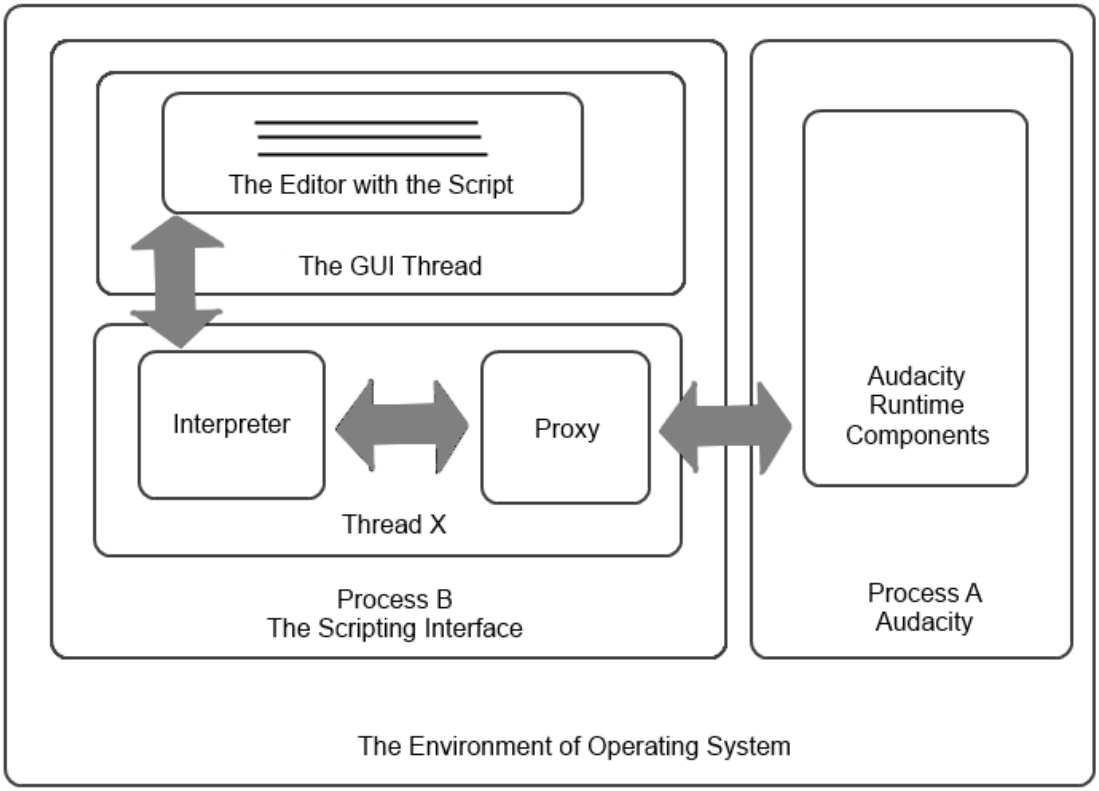


Figure 3.1: Architecture of the scripting interface

The mod-script-pipe plugin communicates with the outer world through pipes. My application connects to these pipes (one for incoming and one for outgoing messages) using a proxy component, which transforms the .NET function calls to matching mod-script-pipe commands and sends them through the pipe.

The interpreter is a component which translates the user's script to API calls. In future, this component can be user defined, but for now JavaScript .NET is used.

The editor components simply provide some tools for script editing.

3.2 Application Properties And States

Application properties are user set preferences influencing the behaviour of scripts. Such an example is the property #autoinit, which causes automatical call of init()

function from editor's Audacity API. To store these properties I decided to implement a class that holds all data within it. My decision was based upon similar implementations in Java API for system global variables.

Application properties are firstly set to default values. After the first run if the user specific settings file is available,the values are read from that particular file.

While application properties are connected with script interpreting, application states are connected with the user interface. The application states contains data about the application and about the script. These states are similar to Windows Phone AppStates, where they contains information whether the app is running, tombstoned, etc. However, this editor doesn't need such feature for its own run, but I find it ideal to monitor the scripts' behaviour. In application states we can indicate whether the script is running,is stopped or an error occured.

Moreover, I also find it suitable to store data about file transactions, e.g. file loading error, project creation error. (if file open failed, by updating the gui we can disable controls, which otherwise would access the file's content)

3.3 Updating The User Interface

If at least one of the application states changes, the application must update its user interface to reflect the new values. Most frequently this happens, when a script is being executed.

3.3.1 Signalizing The Script Status

Since the script can be running for longer interval, it is useful to have an indicator of its status. The problem is, that Audacity doesn't send a respond until the command is finished. Moreover, we can't get information from the interpreter about the number of executed functions or the total number of functions.

Firstly, I considered using an infinite progressbar to indicate, that the script is being executed. Its advantage is simple implementation, but doesn't carry a lot information.

Secondly, I thought about creating an own control, which indicates the execution status with colour (red, green and grey). The advantage of this solution is, that it carries slightly more information and the red colour can warn the user to check the error log.

3.4 Termination of Script Execution

The JavaScript.NET interpreter is running in a separate thread, while the GUI uses its own thread. The interpreted continuously translates the script and calls the background functions. I decided to stop the interpret by adding a common object that contains a boolean variable. The proxy, which connect the interpreter to the Audacity check before every function call whether the script is nor interrupted by the user and then stops.

However, there is one drawback of this approach, that beacuse the Audacity is running in a different process it continues performing its last task. However,

this can be interrupted by user interaction.

4. User Guide

In this chapter I would like to provide an overview about how my application works and which tools are necessary to use it.

4.1 Installation

The Scripting Interface requires a functional Audacity application, with enabled mod-script-pipe support. Moreover, it was tested for .NET framework 3.5, but probably older versions are appropriate, too.

The program is started by clicking on AudacityScriptingInterface icon. If one of the settings is changed it creates a text file in the

A compiled version of Audacity is located on the attached DVD disc, which was used for testing. For a clean compilation tutorials are located on the following sites: Compiling Audacity for Beginners [13], Developing on Windows [13] and Audacity Scripting [8]

4.2 User Interface

As it can be seen on Figure 4.1 the majority of the user interface is dedicated to the editor. On the right side the control panel and the consoles accompany the editor. The consoles are dedicated to severeal printing functions, which are called from the script. Moreover, that tabs contain information about the number of unseen messages. These counters are reset, when the user checks the messages. The error console is usually used by the script interpreter to announce the errors, but it is also available for users. The script interaction states are also displayed on the script control panel, that is in the upper right corner.

The script control panel contains two buttons, which are dedicated to run and stop the script execution. Moreover, there is a square, which is by changing its colour signalizes the states of the application. The dark grey colour means inactive state, when no script execution is running, the green coulour means running script, with no errors, the yellow colour appears, when a warning message was sent to the console. Finally, the red colour means an error and in the error console appears the information about the error. Figure 4.2 shows the situation, when the script execution was interrupted. It already printed out a line to the simple output, but the error console shows the real problem. After clicking to the tab, we can see, that the scripting interface wasn't able to connect to Audacity. This is shown on Figure 4.3.

4.3 Examples

4.3.1 Example 1 - Filtering

In this example we merge two noisy audio files. Denoise filter is applied after the merger and the result is exported to a third file. We assume that the input files are stereo tracks.

Figure 4.1: User interface of my application with a simple script opened in it.

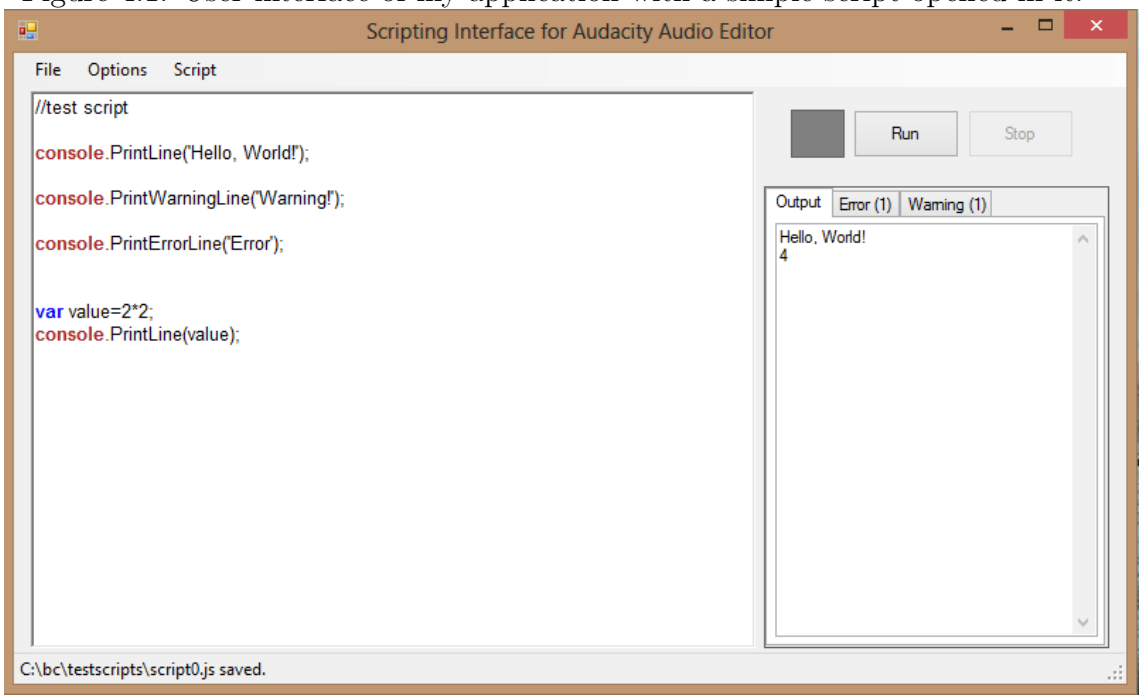


Figure 4.2: An error occured while the script was running.

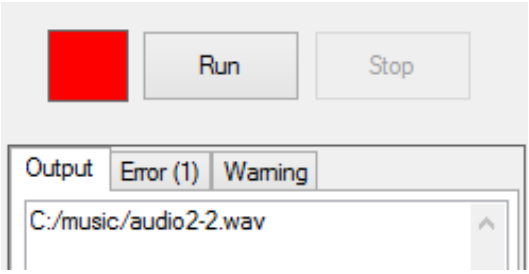
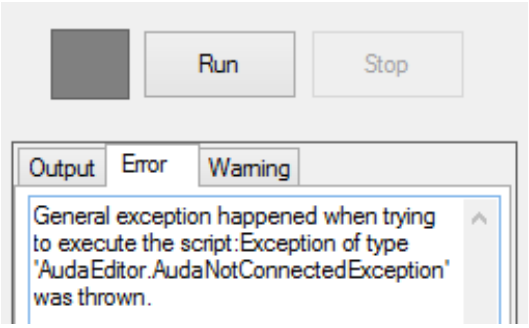


Figure 4.3: The cause of the error



Track A and B are opened. (Audio file C:/music/m.mp3 and C:/music/m-2.mp3 are used in this example) Track A correspondents to audio.mp3 file and track B to audio2.mp3

Listing 4.1: Step 1 and 2

```
1 auda.import('C:/bc/sample/audio.mp3');
  auda.import('C:/bc/sample/audio2.mp3');
```

Unnecessary parts are cut from the tracks: Position 0:00:00-0:01:00 (the first 1 minute) is cut from track A and position 0:01:00-0:03:00 from track B.

In Audacity track A occupies the 0th and 1th mono track, which are set as the second to last and last parameters of select command. The selection range values are set in seconds.

Listing 4.2: Step 3 and 4

```
auda.select('Range',0.0,60.0, 0, 1)
auda.cmdDelete();
```

Track B is located on 2nd and 3rd mono tracks.

Listing 4.3: Step 4

```
auda.select('Range',60.0,180.0, 2, 3)
auda.cmdDelete();
```

Denoise filter (in Audacity called NoiseRemoval) is applied to all tracks (all 4 mono tracks).

Listing 4.4: Step 5

```
auda.select('All');
auda.effNoiseRemoval();
```

There is no explicit track join function in mod-script-pipe, because tracks are automatically merged into needed format on saving. The result is saved to C:/music/audio3.wav.

Listing 4.5: Step 6 and 7

```
auda.export('C:/bc/sample/audio3.wav');
```

4.3.2 Example 2 - Speed Up

This example script makes changes to all mp3 files in the specified directory. It cuts out the first 4 seconds of every track and speeds up each of them by 100 per cent.

Listing 4.6: Speed up recordings in a specified directory

```
var dir='C:/music';
var files=filemanager.GetFiles(dir,'*.mp3');
auda.init();

for(var i=0;i<files.length;i++)
{
  var fullpath=files[i];
  var filename=filemanager.GetFileName(fullpath);
  var exportpath=dir+'/'+filename+'-2.wav';

  auda.import(fullpath);
  auda.select('Range',0.0,4.0, 0, 1);
  auda.cmdDelete();
```

```
15    auda.select("All");
      auda.effChangeSpeed(100.0);
17
      auda.export(exportpath);
19
      auda.select("All");
21    auda.removeTracks();
  }
23 auda.cleanup();
```

5. Conclusion

The main goal of the thesis was to extend Audacity editor with the capability of automated scripting.

We have shown that scripting can be supported via experimental extension of Audacity called mod-script-pipe.

In particular, we have implemented scripting console in a separate process, executing scripts written in a well-known JavaScript language. We also created a graphical user interface, that makes creating and editing of scripts more user-friendly.

We have shown on several examples that even if scripting is not the most important part of Audacity functionality, it is in several cases very useful.

On the other hand, the current implementation can be extended in several aspects, for example future work includes support for other scripting languages and improvements in the interaction between the scripting interface and the Audacity.

A. API functions

In this appendix I would like to catalogize commands needed to use the scripting API. These commands can be accessed through pre-defined objects as shown in 4.3

A.1 Functions of *auda* object

A.1.1 *init* function

A.1.2 *cleanup* function

A.2 Functions of *filemanager* object

A.2.1 *GetFiles* function

A.3 Functions of *console* object

A.3.1 *Clean* function

A.3.2 *PrintLine* function

A.3.3 *PrintErrorLine* function

A.3.4 *PrintWarningLine* function

Bibliography

[1] MSDN Article, *Interprocess Communications*. Date: 10/27/2012 http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574%28v=vs.85%29.aspx

[2] MSDN Article, *Named Pipes*. Date: 10/27/2012 http://msdn.microsoft.com/en-us/library/windows/desktop/aa365590%28v=vs.85%29.aspx

[3] Rich Morin,Vicki Brown, *Scripting Languages*, MacTech. http://www.mactech.com/articles/mactech/Vol.15/15.09/ScriptingLanguages/index.html

[4] GIMP Python homepage, *Documentation*, http://www.gimp.org/docs/python/index.html

[5] Script-Fu Documentation, http://docs.gimp.org/en/gimp-concepts-script-fu.html

[6] Node.js API documentation, http://nodejs.org/api/

[7] JavaScript.NET Homepage http://javascriptdotnet.codeplex.com/

[8] Audacity Scripting, *Scripting*, http://manual.audacityteam.org/man/Scripting

[9] Freepascal compiler settings, http://www.freepascal.org/docs-html/user/usersu68.html

[10] MSDN Article, *Application Properties*, http://msdn.microsoft.com/en-us/library/system.windows.forms.application_properties.aspx

[11] Script-Fu Documentation, *Standalone Script-Fus*, http://docs.gimp.org/en/kinds-of-script-fu.html

[12] WX.Net Documentation, *WX.Net.Build Documentation*, http://wxnet.sourceforge.net/wxbuild/

[13] Compiling Audacity for Beginners, http://wiki.audacityteam.org/wiki/CompilingAudacityForBeginners

[14] Developing on Windows http://wiki.audacityteam.org/wiki/Developing_On_Windows