ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Liasko** Jméno: **Dmytro** Osobní číslo: **452745**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

VST plugin pro segmentaci hudby

Název bakalářské práce anglicky:

Music Segmentation VST Plugin

Pokyny pro vypracování:

Prostudujte standard VST pro realizaci pluginů v kontextu digital audio workstations (DAW). Prostudujte metody návrhu a realizace adaptivní hudby. Realizujte VST plugin pro segmentaci hudebního signálu pro účely adaptivní hudby v prostředí DAW. Tato segmentace nechť je řízena uživatelem pomocí grafického rozhraní tohoto pluginu. Výstupem pluginu nechť jsou data kompatibilní s některým z existujících systémů pro realizaci adaptivní hudby (např. FMOD Studio). Integrujte tento systém do pluginu tak, aby bylo možné ověřovat použitelnost výstupu pluginu během uživatelovy interakce s pluginem.

Seznam doporučené literatury:

[1] Snoman, R. (2012). The dance music manual: tools, toys and techniques. Focal Press.
[2] Rumsey, F. (1994). MIDI systems and control. Butterworth-Heinemann.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

doc. Ing. Adam Sporka, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **13.02.2019** Termín odevzdání bakalářské práce: **24.05.2019**

Platnost zadání bakalářské práce: **20.09.2020**

doc. Ing. Adam Sporka, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

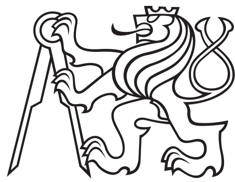
prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat bakalářskou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v bakalářské práci.

.
Datum převzetí zadání Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Music Segmentation VST Plugin

Dmytro Liasko

Supervisor: doc. Ing. Adam Sporka, Ph.D.
May 2019

Acknowledgements

I would like to express my gratitude to my thesis supervisor doc. Ing. Adam Sporka, Ph.D. for his support and advice throughout my work on the project.

Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 21, 2019

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 21. května 2019

Abstract

This bachelor thesis aims to design and implement a VST plugin for segmentation of audio signal for the purpose of adaptive music creation. The plugin receives the audio signal input from a Digital Audio Workstation. The segmentation will be based on the data the plugin receives by the user via the graphical user interface of the plugin. The plugin produces output compatible with Avalon music engine. The plugin was written in C++ programming language using JUCE framework the plugin is based on the plugin that was made by me as a part of my semestral project.

Keywords: VST, MIDI, Adaptive music, Audio signal processing, JUCE, C++

Supervisor: doc. Ing. Adam Sporka, Ph.D.

Abstrakt

Cílem této bakalářské práce je navrhnout a implementovat VST plugin pro segmentaci hudebního signálu za účelem tvorby adaptivní hudby. Vstupní audio signál plugin získává z programů typu Digital Audio Workstation. Proces segmentace je řízen na základě dat, který plugin získává od uživatele pomocí grafického rozharni. Výstupem pluginu jsou data kompatibilní s systémem pro realizaci adaptivní hudby Avalon. Plugin byl implementován v jazyce C++ (s použitím frameworku JUCE) a je založený na pluginu, který jsem implementoval v rámci semestrálního projektu.

Klíčová slova: VST, MIDI, Adaptivní hudba, Zpracování audio signálu, JUCE, C++

Překlad názvu: Music Segmentation VST Plugin

Contents

1 Introduction 1
1.1 Adaptive music common techniques 1
1.1.1 Horizontal re-sequencing 1
1.1.2 Vertical re-orchestration 2
1.2 Objectives 2
1.3 Technology 3
1.3.1 VST 3
1.3.2 MIDI 3
1.3.3 JUCE 4

Part I
Harvester VST

2 Plugin specification 9
2.1 Description of plugin midi events 10
2.2 Audio 12
2.3 Export 12
2.3.1 GUI 13
2.3.2 Extra 14

3 Design 15
3.1 Audio and MIDI input-output processing 15
3.2 Serialization 16
3.2.1 SerializedPattern 16
3.2.2 PluginSerialization 16
3.3 GUI 16
3.3.1 AudioProcessorEditor 17
3.3.2 PluginListBox and PluginListBoxModel 17
3.3.3 PatternInfoComponent 17

4 Realization 19
4.1 Plugin state 19
4.2 Timings 20
4.3 Illustration of GUI 20
4.3.1 Detailed description 20

Part II
Harvester 2 VST

5 Plugin specification 25
5.1 GUI 25
5.2 Audio 26
5.3 Export 26
5.3.1 .avalon_module 26
5.3.2 .avalon_bank 28
5.4 Collisions 28
5.5 Hotkeys 29

6 Design 31
6.1 Class diagram 31
6.2 Audio processing 32
6.3 GUI 32
6.3.1 PluginEditor 32
6.3.2 ModulesTableListBoxModel 32
6.3.3 TimelineButtonsComponent 32
6.3.4 TimelineLeftColumnComponent 32
6.3.5 TimelineComponent 32
6.3.6 WaveformComponent 33
6.3.7 ElementDetailComponent 33
6.4 Serialization 34
6.4.1 Models 34
6.4.2 PluginSerialization 34
6.5 Data flow 34
6.6 Extra 35
6.6.1 Audio player 35
6.6.2 PluginUtils 35

7 Realization 37
7.1 Used software 37
7.2 Implementation 37
7.2.1 Audio processing 37
7.2.2 GUI 38
7.2.3 Audio player 40
7.2.4 Serialization 41
7.2.5 Export 44
7.3 Testing 45
7.4 GUI illustration 45
7.4.1 Overall 45
7.4.2 Pop-up menu 46
7.4.3 Metadata element editing 46

8 Conclusion 47

Appendices

A Bibliography 51

B CD contents 53

Figures

1.1 Cross fading scheme. 2
1.2 Direct switch scheme. 2
1.3 Vertical layering scheme. 2
1.4 Main screen of the Projucer. 4
1.5 Projucer IDE screen. 5

2.1 Example of project in DAW Reaper [Cocb] 9
2.2 Example of MIDI mapping. 11
2.3 Example of possible changes. . . . 12
2.4 Lo-Fi GUI prototype. 13

3.1 The Harvester VST structure. . . 15
3.2 Structure of GUI. 16

4.1 GUI of synthesizer SAWER [Ima] 19
4.2 GUI. 21
4.3 Parts of GUI. 21
4.4 Different time display modes. . . . 22

6.1 The structure of the Harvester 2 VST plugin. 31
6.2 The data flow of Harvester 2 VST. 34

7.1 Example of waveform thumbnail. 39
7.2 Collision types. 43
7.3 The plugin hosted in DAW Reaper. 45
7.4 The context menu. 46

Tables

2.1 Description of module MIDI events. 10
2.2 Description of pattern MIDI events. 10
2.3 Description of point and region MIDI events. 11

5.1 The pattern types and representing letters 26

Chapter 1
Introduction

The music production process is especially complicated when it comes to a soundtrack creation. Unlike the common music, where the composer expresses his feelings, vision, thoughts and ideas, soundtracks are music tracks accompanying and synchronized to the images of a motion picture, television program, or video game.

The difficulty of the soundtrack composition is primarily based on the type of media content it plays along to. In soundtracks for movies, the composer knows all details about the context of the ongoing scene. This allows a composer to create music that stimulates the viewers' emotions at exactly the right moments. Videogames soundtrack creation process is more complicated because we cannot accurately predict what is going to happen on the screen next. It is impossible to know in advance how long will it take for the player to pass through a location or what amount of health points the player will have in 10 seconds after the battle started. That is the point where adaptive music comes in. Adaptive music is background music that is dynamically changed by reacting to some type of control input coming from the game.

1.1 Adaptive music common techniques

Among all of the adaptive music techniques, Horizontal re-sequencing and Vertical layering are considered to be the most common [Kä18]

1.1.1 Horizontal re-sequencing

Horizontal re-sequencing is a method of adaptive composition where the pre-composed music segments can be reshuffled according to the player actions.

Cross-fading

Cross-fading is one of the most common horizontal re-sequencing techniques. Its idea is to create a smooth transition from one music track to another where the first track volume is lowered to zero while the second track volume is getting raised up.

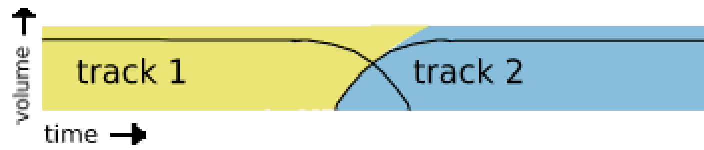


Figure 1.1: Cross fading scheme.

Direct switching

Direct switching is the simplest horizontal re-sequencing technique. The transition from one music track to another is performed by stopping the current track at the end of the current beat and starting the next track at the same moment.

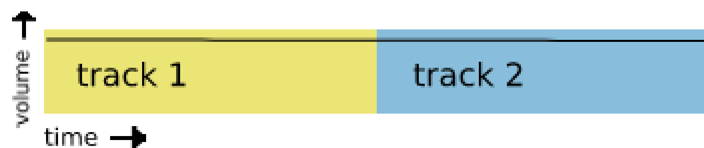


Figure 1.2: Direct switch scheme.

1.1.2 Vertical re-orchestration

Vertical layering is the adaptive technique where the music track is split into multiple layers that are intended for one or a group of musical instruments. The layers can be added or removed by certain events, that were predefined by the game designer and composer.

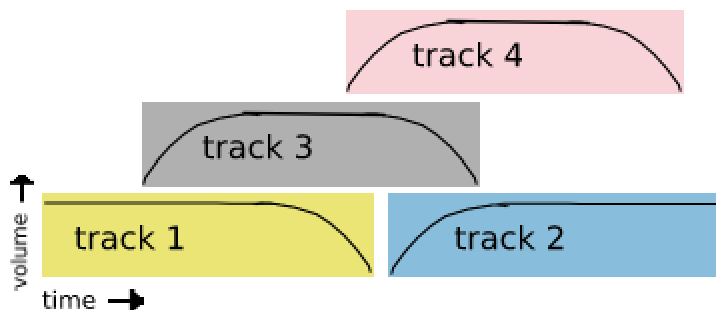


Figure 1.3: Vertical layering scheme.

1.2 Objectives

The main objective of this is project is to create a VST plugin for segmentation of audio signal inside a Digital Audio Workstation environment and integration of Avalon music engine[wel18c] as a system for adaptive music creation into

it. The plugin that is created in this work is based on the plugin that was made by me as a part of the semestral project.

1.3 Technology

1.3.1 VST

Virtual Studio Technology (VST) is a digital interface standard that is used to connect and integrate different software emulations of sound effects and synthesizers in Digital Audio Workstation (software that is used for recording, editing and producing of audio files) developed by Steinberg in 1996.

Listing 1.1: The description of the standart from the official documentation [Ste99]

```
 In the widest possible sense a VST-Plug-in is an audio
process. A VST Plug-in is not an application. It needs a
host application that handles the audio streams and makes
use of the process the VST plug-in supplies. Generally
speaking, it can take a stream of audio data, apply a
process to the audio and send the result back the host
application. A VST Plug-in performs its process normally
using the processor of the computer, it does not
necessarily need dedicated digital signal processors. The
audio stream is broken down into a series of blocks. The
host supplies the blocks in sequence. The host and its
current environment control the block-size. The VST
Plug-in maintains all parameters & statuses that refer to
the running process: The host does not maintain any
information about what the plug-in did with the last
block of data it processed.
```

In the year 1999, Steinberg updated interface specification to version 2.0. This update made possible for plug-ins to receive MIDI data and introduced to us new plug-in format - VSTi (Virtual Studio Technology Instrument).

1.3.2 MIDI

Musical Instrument Digital Interface (MIDI) is a technical standard, a digital interface that describes a protocol for communication between electronic musical instruments, computers, and related devices [MID96]. The main thing is that sound is never sent via MIDI, just digital signals known as event messages. MIDI messages can be split into channel messages (messages that are transmitted on individual channels) and system messages (messages that are not channel specific, such as start, stop, system reset). The type of a message can be identified by its first byte that has its hex value between 80 and FF and followed by 0 to more data bytes. For example, message 90 48 7F is a channel message that means “play note C5 on channel 1”.

1.3.3 JUCE

Most of VST plugins are written in C++. First of all, because the official SDK is available for C++ only. There a lot of frameworks that make VST plugins development easier, the most popular are Cockos WDL[Coca] and JUCE[ROLb]. For this project, The JUCE framework was chosen because it provides a large set of functions for audio processing, XML and JSON parsing, networking and user-interface creation. One of the main features of JUCE is its support of multiple plugin formats (VST/VST3/RTAS/AAX/AU) and different platforms (Windows, Mac, Linux, Android, iOS).

Setting up the environment

The creation of JUCE application starts in Projcuer. This is an application that allows a user to develop, manage and deploy cross-platform applications.

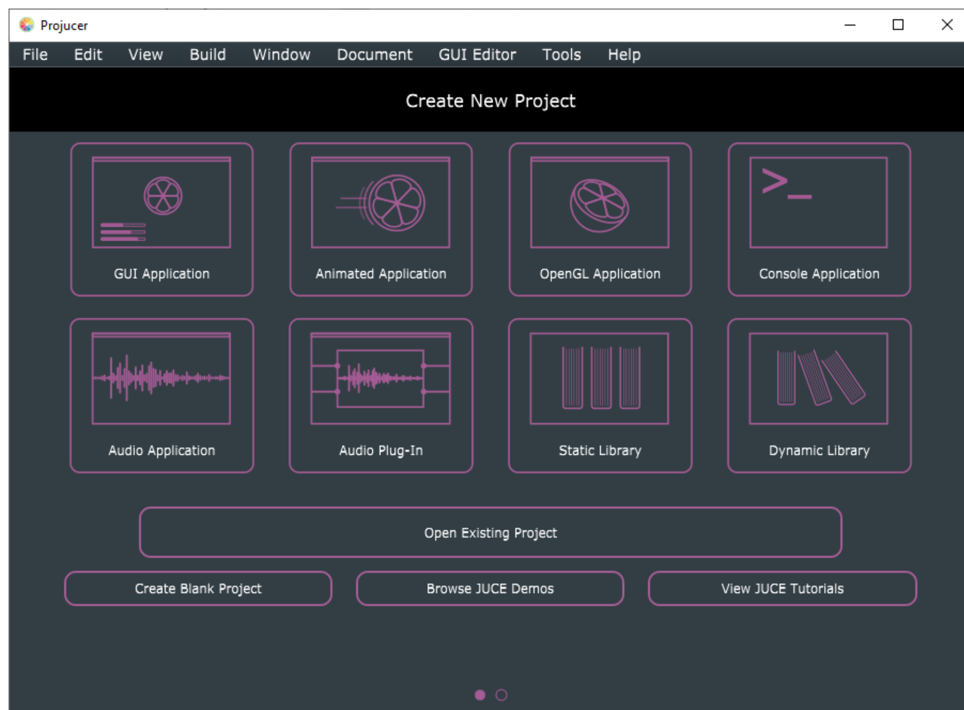


Figure 1.4: Main screen of the Projucer.

Projucer is also can be used as IDE or as an exporter to user-preferred IDE. It supports export to Visual Studio, CLion, CodeBlocks, XCode.

The basic structure of JUCE plugin

The heart of every audio plugin developed in JUCE are the following classes:

- AudioProcessor

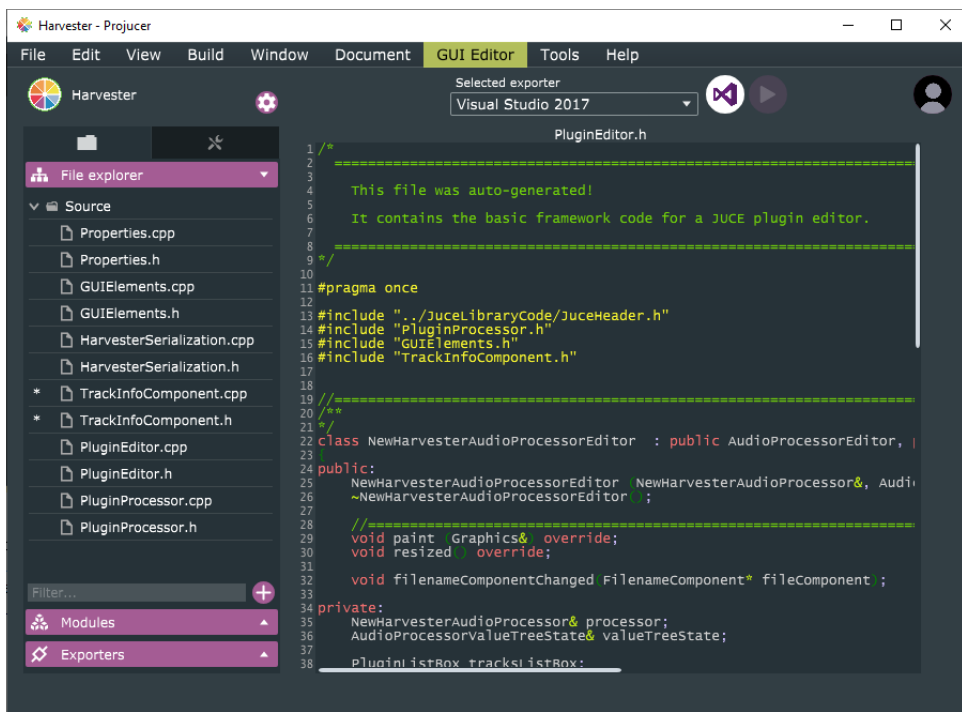


Figure 1.5: Projucer IDE screen.

This class handles the audio and MIDI IO and processing logic. The main methods that every plugin must have implemented are:

Listing 1.2: Declaration of the prepareToPlay method

```
void AudioProcessor::prepareToPlay(double sampleRate,
    int samplesPerBlock)
```

This method is being called every time before playback starts, to let the processor prepare itself. Parameter sampleRate is the sample rate that was set by the host and can't be changed during the playback. Parameter samplesPerBlock is a number of samples in every block passed to the processBlock method.

Listing 1.3: Declaration of the processBBlock method

```
void AudioProcessor::processBlock(AudioBuffer<float> &
    buffer, MidiBuffer & midiMessages)
```

This is the key method of every AudioProcessor that is called each time a block of samples and MIDI data need to be handled. Parameter buffer is a multi-channel buffer containing floating point audio samples. Parameter midiMessages is a multi-channel buffer containing time-stamped midi events.

The number of AudioProcessor class instances is strictly limited to 1.

- AudioProcessorEditor

  This class is responsible for GUI handling. Every JUCE audio-plugin can have multiple instances of this class. It is best to consider the processor as the parent of the editor because every editor has a reference to the processor.

Harvester VST

Chapter 2

Plugin specification

The main purpose of this part of the project is to create a plugin that will automate the process of export of sound and metadata from DAW based on defined midi-mapping. Every project in DAW consists of tracks and sequences of different patterns arranged to them. Under the term "pattern" we imply a sequence of midi messages or a recorded audio sample.

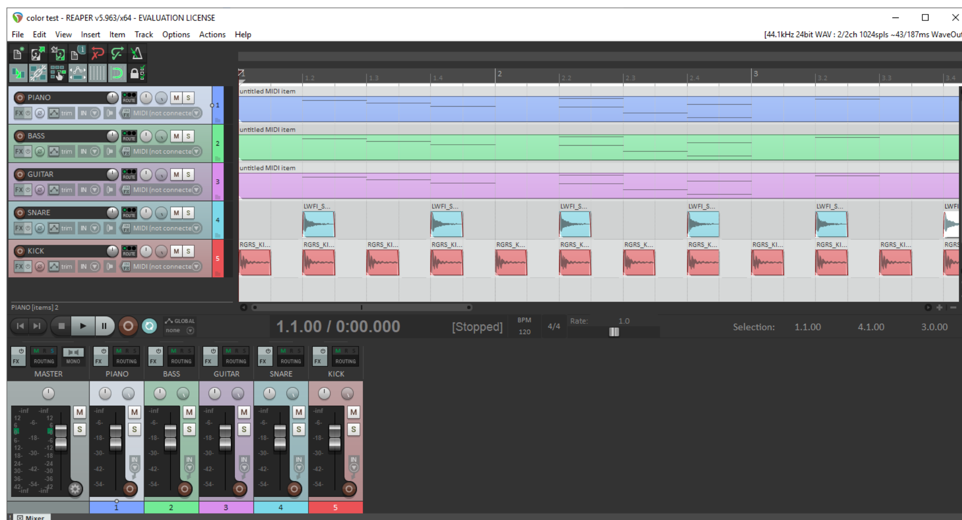


Figure 2.1: Example of project in DAW Reaper [Cocb]

The project, where the Harvester VST plugin is used, has the following structure:

- Project has at least one module
- Modules are consist of patterns
- Patterns have points and regions

2.1 Description of plugin midi events

According to the Avalon music engine documentation [wel18b], a module is a principal unit of music, other words, for easier understanding, we can call it a song.

Midi number	Note	Description
84+	C6 and up	Module delimitation

Table 2.1: Description of module MIDI events.

As every song consists of different structure elements, such as introduction, verse, bridge, an Avalon module consists of the patterns. Avalon music engine defines 6 types of patterns, some of them are mapped to the classic song structure.

- Pickup
  This type represents the introduction part of a song, the part where the song begins.
- Main
  This type represents the verse part of a song. It can be described as a repeatedly played main melody of a song.
- Ending
  This type represents the outro part of a song. This part ends, completes the played song.

Midi number	Note	Description
77	F5	Pickups (could be more than 1) Note-on means ä pickup starts here" Note-off means "that pickup ends here"
78	F#	Mains (could be more than 1)
79	G5	Endings (could be more than 1)
80	G#	Layer AUX (could be more than 1)
81	A5	Timed stinger AUX
82	A#	Random stinger AUX

Table 2.2: Description of pattern MIDI events.

Every patterns consists of special points and regions.

MIDI number	Note	Name	Description
65	F4	In-region	Note-on event means the start of the in-region, note-off means the end. The point of alignment to the origin beta is the end of the in-region
67	G4	Out-region	Note-on event means the start of the out-region, note-off means the end. The point of alignment to the destination alpha is the beginning of the out-region
68	F#	Stinger point	For timed stingers Note-on of the event matters
69	A4	Alpha point	Note-on of the event matters
71	B4	Beta point	Note-on of the event matters
74	D5	Downbeat	Note-on of the event matters. The point of the first beat of the bar.
75	D#	Beat	Note-on of the event matters. The point of the non-the-first beat of the bar.
No special MIDI markings needed		Leftmost Rightmost	The beginning of the pattern The end of the pattern

Table 2.3: Description of point and region MIDI events.

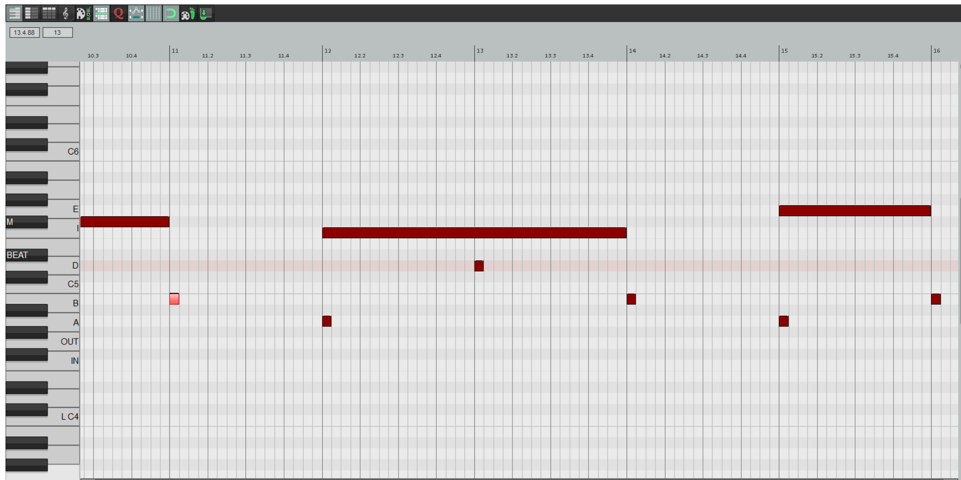


Figure 2.2: Example of MIDI mapping.

During the processing of the input MIDI messages, the plugin must extract the information about the "event type" and "timestamp in samples" properties, the rest of the message description is not important.

2.2 Audio

The plugin should work in a project with a sample rate of 48000 samples per second (48 kHz). The sample rate value is choice of the creators of Avalon music engine. If the sample rate was set to another value, the plugin will show to a user an error message.

The plugin must be able to export audio in Waveform Audio File Format (.wav) with 32 bits per sample depth.

2.3 Export

Users must be able to set the location for audio and metadata export. If the user didn't, the plugin will use its default folder that was created inside the user Application Data folder of OS. To prevent such situation when different plugin instances share the same default folder, the plugin will use the timestamp of its initialization as the default folder name.

Patterns audio and metadata files will be named according to the pattern appearance in the DAW sequencer timeline in format WWhXXmYYsZZZ where WW stays for hours, XX for minutes, YY for seconds and ZZZ for milliseconds.

The plugin must react to any change made to any pattern timeline and replace the result of previous export with the new one. Briefly, the plugins default folder must keep only the result of the last export.

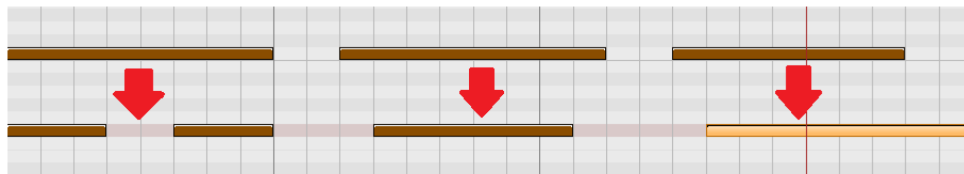


Figure 2.3: Example of possible changes.

During the export, the processed metadata must be converted to JSON file format.

Listing 2.1: Example of exported module

```
{
  "moduleName": "Module one",
  "productionCode": "M01",
  "leftmost": "0",
  "rightmost": "658286",
  "number": "",
  "alpha": "0",
  "beta": "658286",
  "inregions": [
    "324000",
    "329143"
```

```
  ],
  "outregions": [
    "82286",
    "87429",
    "246858",
    "252000",
    "411429",
    "416572",
    "576000",
    "581143"
  ],
  "trackType": "78",
  "beats": [
    "downbeat,5.142854",
    "downbeat,6.857146",
    "downbeat,8.571438",
    "downbeat,10.285708",
    "downbeat,12.000000",
    "downbeat,13.714292",
    "downbeat,15.428562"
  ],
  "locked": false,
  "timedStingers": []
}
```

2.3.1 GUI

The user interface of the plugin will be split into 2 main parts. On the left side of the screen, there will be an input field for setting up the plugins work folder and list of the patterns that have already been exported and basic information about it (pattern name, pattern type, module name and module code). The right side will show information about the pattern user clicked on.

```
+---------------------------------------------------+  Module name: [ TAVERN_          ]
| 00h15m00s102 P TAVERN         0004K               |
| 00h17m20s000 M                                    |  Pattern time info           Pattern role:
| 00h18m00s000 E                                    |  Leftmost:                   ( ) Pickup
| 00h21m50s000 P MEADOW         0004L               |  Alpha:                      ( ) Main
| 00h23m40s000 M                                    |  Beta                        ( ) Ending
| 00h24m10s000 E                                    |  Rightmost:                  ( ) Custom: [ _        ]
| 00h26m40s000 M GUARDS         0007A               |
|                                                   |  Out-regions
|                                                   |  +----------+
|                                                   |  |          |
|                                                   |  +----------+
|                                                   |
|                                                   |  In-regions
|                                                   |  +----------+
|                                                   |  |          |
|                                                   |  +----------+
+---------------------------------------------------+
```

Figure 2.4: Lo-Fi GUI prototype.

2.3.2 Extra

There are 2 ways the user can assign production code and module name to a pattern:

- User will fill corresponding input fields in the pattern information part of GUI.
- User will be able to create a file called "manifest.txt" in plugins working folder. This file will be formatted as a CSV with the following structure:

```
[module MIDI note number],[production code],[module name]
```

Users will be able to "lock" patterns. Some items should not be re-exported, so if there is any new input in those regions, it's ignored.

Chapter 3

Design

According to the specification, the plugin structure can be split into 3 main parts: Audio and MIDI input-output processing, user interface, and serialization.

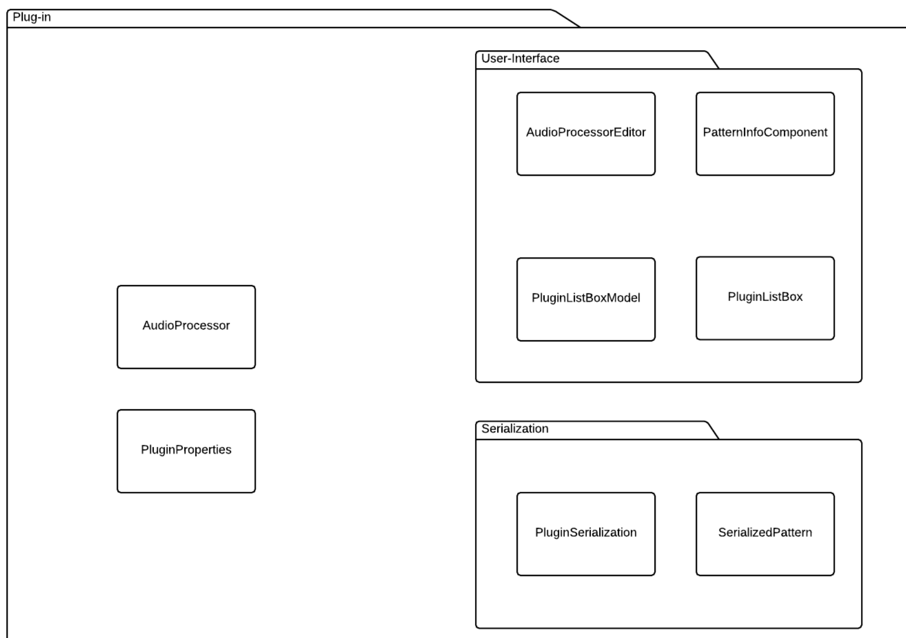


Figure 3.1: The Harvester VST structure.

3.1 Audio and MIDI input-output processing

As it was mentioned earlier, AudioProcessor class is responsible for all operations related to Audio and MIDI input-output processing. In our case processing logic is simple. It is based on a state machine that reacts to MIDI messages occurrences. When the plugin receives a MIDI message with a note number in the range of 77-82 (pattern types) and note-on event type a new instance of SerializedPattern class is created and the process of writing

the audio input to a file in the plugin working directory starts. If MIDI message number is a special point or a region and the process of export has already been started, it will be added to the suitable attribute of an instance of SerializedPattern class. Otherwise, the message is ignored. The process of audio export is handled by class AudioFormatWriter which is a part of the JUCE framework. AudioFormatWriter::writeFromAudioSampleBuffer() method is called every time when a block of audio samples must be written to a file.

3.2 Serialization

The process of serialization is split between the following classes:

3.2.1 SerializedPattern

The class can be described as a Data Transfer Object (DTO) and is widely used in all parts of the project. Process of (de)serialization of a pattern is handled by JSON and var classes which are part of the JUCE framework. JSON class has methods for converting JSON-formatted text to and from var objects. Var class is a wrapper class, that can be used to hold a range of primitive values.

3.2.2 PluginSerialization

The class is responsible for writing and reading a JSON formatted pattern to/from a file.

3.3 GUI

The graphical part of the plugin consists of the 4 classes.

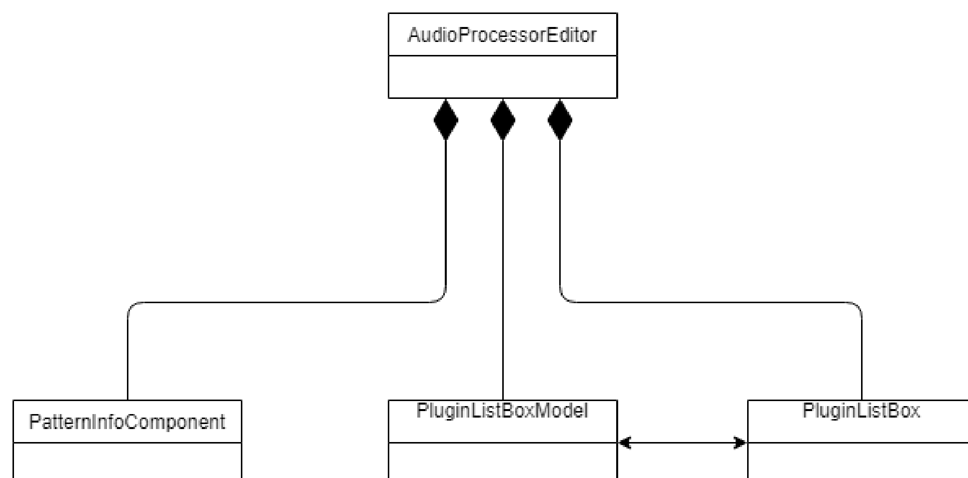


Figure 3.2: Structure of GUI.

3.3.1 AudioProcessorEditor

This class can be considered as a parent for all the graphical components in the plugin.

3.3.2 PluginListBox and PluginListBoxModel

JUCE's ListBox is the abstract class that allows displaying tables. The ListBox behavior is defined by ListBoxModel which describes the data model that needs to be displayed. PluginListBox and PluginListBoxModel are implementation of those classes.

3.3.3 PatternInfoComponent

This class is just a view that is updated every time user clicks on a PluginListBox item.

Chapter 4

Realization

4.1 Plugin state

Every VST plugin has its own memory where the entire plugins state can be stored. Mainly it is used to keep parameter changes made by plugin users. Just imagine that you have a synthesizer with more than 20 different knobs, switches, and buttons and you need to restore them every time you open the plugin again and again.

Figure 4.1: GUI of synthesizer SAWER [Ima]

In our case plugin memory is used not only as storage but as a method of communication between AudioProcessor and GUI. The only value plugin needs to be always stored is it's working directory. The idea of using this memory as a method of communication came to when I realized that there is no easy way to send data from AudioProcessor to AudioProcessorEditor,

but it was a necessary thing to make plugin update its GUI every time a new pattern is exported. The algorithm is consists of the following steps:

1. Register AudioProcessorEditor as plugin memory listener
2. Every time new pattern is exported AudioProcessor sends a notification that property memory's property "pattern" was changed

This algorithm was implemented with the help of AudioProcessorValueTreeState class which is a part of the JUCE framework.

4.2 Timings

One of the most important parts of the plugin implementation is timing precision. VST plugins work in real-time, getting as much data as possible from all available sources. Even small kind of processing delay, or just a situation when the plugin is out of sync, can be critical for the final output.

As a time tracking mechanism VST plugins use samples counting. When playback starts, plugin receives the sample rate of the project and the number of samples in every block which will be processed by AudioProcessor. The first reason is that the plugin can detect only the current position of the audio play head during the playback; the next reason is the way MIDI messages are processed.

Listing 4.1: The description of the MIDI messages processing order from the VST documentation [Ste99]

```
 Just as a side issue, the MIDI events resulting from
sequencer playback are actually scheduled into the
future, or pre-fetched, to accommodate for audio device
latencies. This actually means you get events some time
ahead of the actual playback time in pure MIDI terms.
```

That explains why all the metadata exported by plugin mainly in the "number of the samples from the start of the pattern" format.

4.3 Illustration of GUI

4.3.1 Detailed description

1. This is the component where the user can define the working directory for the plugin
2. This component is the list of all the patterns that were exported to the working directory of the plugin
3. This part consists of 2 blinking lights indicating that plugin receives MIDI and Audio input

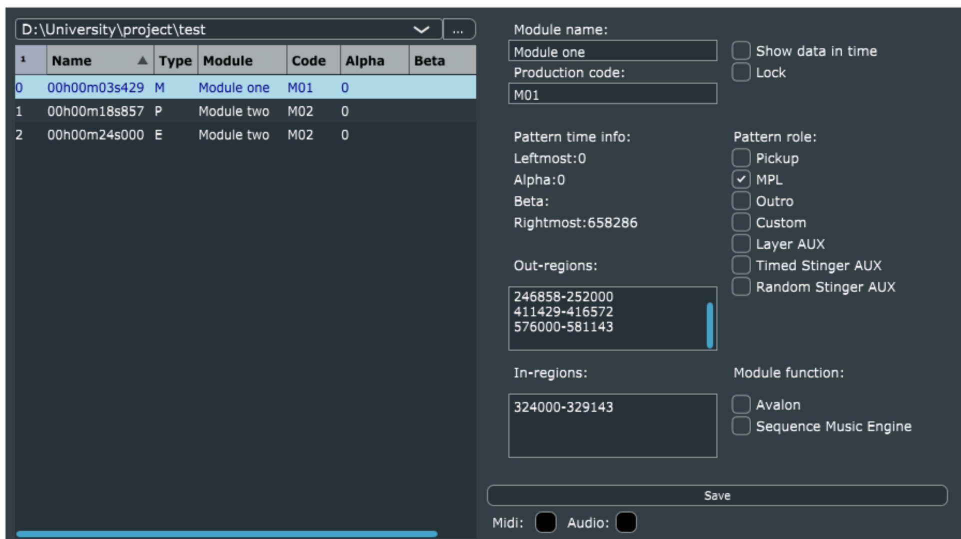


Figure 4.2: GUI.

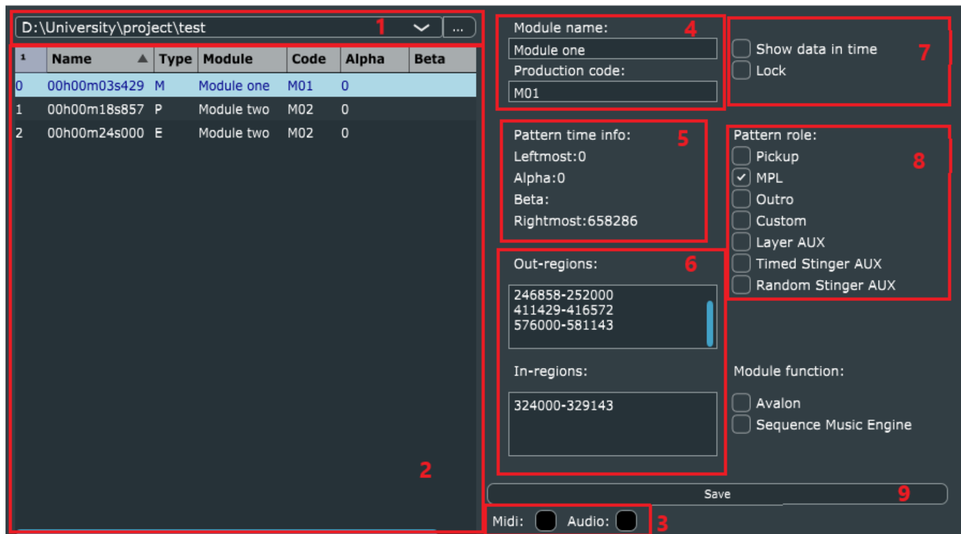


Figure 4.3: Parts of GUI.

4. This part has 2 input fields where user can arrange module name a production code to a pattern
5. This part displays information about pattern points
6. This part displays information about pattern regions. Basically, it's just a list of pairs in "start-end" samples.
7. This part has 2 checkboxes where you can lock the pattern to prevent reexporting and change the mode of how to show information about the pattern points, you can switch between samples and time format.
8. This part is a list of checkboxes that describe pattern type

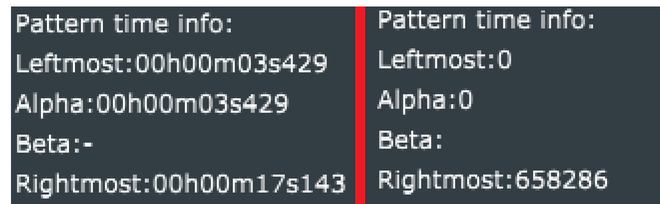


Figure 4.4: Different time display modes.

9. This button has to be clicked when the user made changes to pattern metadata and wants to keep them saved.

Part II

Harvester 2 VST

Chapter 5

Plugin specification

Despite the fact that Harvester VST works well in the production process, there are still some problems, most of them are from the UX point of view. For example, one of the main disadvantages of the Harvester VST plugin is its too much time-consuming workflow. Harvester VST was designed in such a way that music and metadata creation process is performed in the DAW, the plugin takes responsibility for automatic audio and metadata export and metadata validation via GUI. Another problem is that neither audio format nor metadata format is not compatible with Avalon music engine, therefore, there must be some application that is used as a converter between these formats.

The main idea of the Harvester 2 VST is to guarantee that the module creation and export process will be performed only inside the plugin and make that process faster and much user-friendlier.

5.1 GUI

If we compare GUI of both plugins, we can say that the Harvester VST interface allows the user to see and verify what was created and could be further converted into an Avalon module, then interface of Harvester 2 allows the user to create, to see, to verify, to export. Briefly, the graphical interface of the new plugin is going to be like DAW inside DAW.

The graphical interface will be split into 4 main parts: File chooser - a component that migrated from the first version of the plugin. It allows resetting the directory where plugin instance data will be stored. Timeline component - a component that represents a timeline of the project. That is the component where a user will be creating metadata for corresponding audio slices. Control component - a component that consists of a set of buttons that control all the actions performed in the timeline component, buttons that control the built-in audio player and an editable box that shows and also can be used for setting the current playback position. Exported modules list - a component that represents the list of all modules that were created by the user and exported by the plugin.

Also, the timeline component must have a built-in audio player that will allow checking what audio user works with. One of the main requirements

for the player is buffering support because it is possible that the user can serialize several hours of audio data as one file. In that case, its file size can reach a few gigabytes and to store such amount of data in memory is not the best solution.

5.2 Audio

There is also a big difference in a way the plugins work with audio. Harvester 2 stores all the audio data that were played or rendered in the DAW. These audio data will be stored as Waveform Audio File with a depth of 32 bits per sample and will be used during the module creation phase.

5.3 Export

As in the previous version of the plugin, the user must be able to define the working directory which will be used foremost for storing the results of the export process, otherwise, the plugin will create and use a folder in user AppData directory. Audio and metadata files of modules that are exported must be in a format that is compatible with Avalon music engine.

An Avalon module consists of the following files:

5.3.1 .avalon_module

This is a proprietary JSON-like file format designed by welove.audio GmbH[wel18a] as a part of Avalon Adaptive Audio technology. This is the file, where the module's metadata are stored. It has information about production code, module name, sampling rate, number of channels and detailed description of each pattern that belongs to the module.

Pattern description always begins as the key-value pair, where the key is the word, that describes the pattern type and the value is corresponding to its type letter and the serial number.

Pattern type	Letter
Pickup	I
Main	M
Ending	O
Layer AUX	L
Timed Stinger AUX	T
Random Stinger AUX	R

Table 5.1: The pattern types and representing letters

The next entry is a signal_length that describes the length of audio files in samples.

The last value is an array of points that describes sample positions of inregions, outregions and alpha, beta points.

Listing 5.1: Example of .avalon_module file

```
module: "0000M"
{
    display_name: "Miami";
    sampling_rate: 48000;
    channel: 2;
    patterns
    {
        main: "M1"
        {
            signal_length: 2448000;
            points {
                alpha(0);
                beta(2448000);
                in(336000, 480000);
                in(1056000, 1200000);
                in(1536000, 1728000);
                in(1968000, 2112000);
                outs(39374, 75937);
                outs(146249, 236250);
                outs(281250, 348750);
                outs(405000, 461250);
                outs(489375, 542812);
                outs(576000, 768000);
                outs(880312, 1043437);
                outs(1099687, 1209374);
                outs(1295156, 1382343);
                outs(1483593, 1622812);
                outs(1694531, 1791562);
                outs(1875937, 2002499);
                outs(2046093, 2131874);
                outs(2195156, 2332968);
                outs(2400000, 2448000);
            };
        };
        ending: "01"
        {
            signal_length: 384000;
            points {
                alpha(0);
                beta(384000);
                in(44999, 191249);
                outs(5624, 67499);
                outs(112499, 196874);
```

```
                outs(236249, 298124);
                outs(336000, 384000);
            };
        };
    };
};
```

5.3.2 .avalon_bank

This is a proprietary file format designed by welove.audio GmbH[wel18a] as a part of Avalon Adaptive Audio technology. This file has a straightforward structure because it consists of a header and a set of audio files in the .ogg format that represent patterns that were described in the corresponding .avalon_module file.

Listing 5.2: Example of .avalon_bank file header

```
aaabank
bank {
    chunk: "0000M-L1.ogg" { length: 100113; };
    chunk: "0000M-M1.ogg" { length: 647693; };
};
```

The header consists of a string that determines that this file is actually Avalon audio bank and list of audio files with their name and size in bytes.

5.4 Collisions

In the context of the plugin, a collision is an event in which the range of one metadata element or audio segments intersects with the range of another. For example, there is a pattern that has 48 000 and 326 000 as its start and end samples positions. When the user creates another pattern that has 144 000 and 480 000 as its start and end samples positions, we have an intersection in the [144 000,326 000] samples position interval. As a result, we have two patterns that are layered on each other in the [144 000,326 000] samples position interval and can be merged into one pattern.

The plugin must be able to detect and fix all the collisions that may occur during the user's interaction with the plugin. There are 2 types of collision users can accidentally create - Audio and Metadata elements collision. Basically, these collision types are similar, but the difference is the way they must be treated. Audio elements collision is a much more serious issue because it can lead to usage of extra space on the hard-drive and further complications during the export process or usage of the built-in audio player.

5.5 Hotkeys

As it was said earlier, UX is one of the most important parts of the Harvester 2 VST. The plugin must be able to react to the hotkeys that will trigger appropriate events in the timeline component. The hotkeys that will be used in the plugin are:

1. **Ctrl + MouseWheelUp** - zoom-in event
2. **Ctrl + MouseWheelDown** - zoom-out event
3. **Ctrl + Z** - causes undo event that will affect the last created metadata element
4. **MouseWheelUp** - scroll the timeline component scrollbar to the right
5. **MouseWheelDown** - scroll the timeline component scrollbar to the left

Chapter 6

Design

6.1 Class diagram

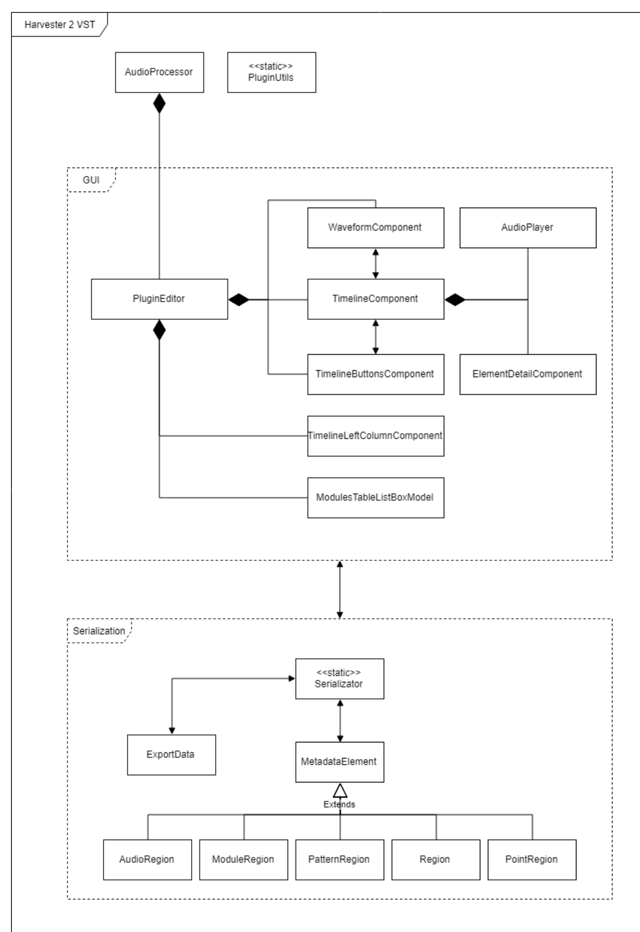


Figure 6.1: The structure of the Harvester 2 VST plugin.

6.2 Audio processing

This part of the plugin is represented only by AudioProcessor class. The class will be responsible for the processing of the audio input that will be passed to the plugin directly from the DAW during the playback. Due to the fact that this class is also used by the plugin hosting code as the wrapper around an instance of a loaded plugin, it will also be used for initialization of plugin Logger and retrieving working directory from the plugin state or creating the new one if the state is empty because the instance of plugin has not been created earlier.

6.3 GUI

This part of the plugin is represented by the following classes:

6.3.1 PluginEditor

The class is used as the main component where the other components will be placed.

6.3.2 ModulesTableListBoxModel

The class that implements a JUCE TableListBoxModel abstract class. It will be used as the data model for a TableListBox that will be placed in the PluginEditor and contain the list of the modules that were created by the user and successfully exported by clicking the corresponding button.

6.3.3 TimelineButtonsComponent

The class that will contain buttons and text editors that will provide an opportunity to control playhead of the builtin audio player or create metadata elements on the plugin timeline. The class will hold a reference to the plugin timeline component for easier audio player or metadata element creation events dispatching.

6.3.4 TimelineLeftColumnComponent

The class its only task is to represent labels for easier recognition of the type of metadata element that was placed on the timeline. The reason why such basic thing as labels is placed in a separate component is to eliminate the need for introducing the extra offset during the render of the timeline.

6.3.5 TimelineComponent

Except for the PluginEditor, this is the most important component of the plugin because this is the component where most of the interactions with the

plugin will be concentrated. The class will be responsible for the representation of a timeline, handling metadata element creation event. Also, the class is responsible for the creation of the built-in audio player and handling of the related to its events that can be emitted by TimelineButtonsComponent or just by clicking on the timeline. The class implements several mouse and keyboard listeners for hotkeys and different mouse events handling, for example, the user can perform undo action just by pressing Ctrl+Z combination or when the user right-clicks on a metadata element a popup menu appears where user can decide if he wants to delete or edit some properties of the chosen element. The entire component can be split into 3 parts:

1. Header - the part that helps the user to determine the position of the timeline

2. Metadata elements part - the part where the user can see, edit, delete or create metadata elements

3. Audio part - shows to the user available segments of audio data that were proceeded by the plugin and can be used during the audio bank creation.

The component will be provided with a scrollbar that will be used for setting the timeline position.

6.3.6 WaveformComponent

The main task of this class is to dynamically create waveform thumbnails of the available audio data segments based on the current timeline scrollbar position. The thumbnails are needed for increase of informativeness of the timeline and make the process of metadata elements creation more precise so plugin user can rely not just on ears, but eyes too.

6.3.7 ElementDetailComponent

The component that appears after the click on the corresponding menu item that can be seen after the right-click on a metadata element. That component consists of 2 types of input elements - common and situational. Common elements are 2 text editors that are responsible for showing and editing of metadata element start and end time. The second type of input element is called situative because their display is based on the type of metadata element that is being edited. They include 2 text editors for module name and production code input, one text editor for setting the pattern number and a dropdown select box for determining pattern type, and one slider for the in/outregion fade shape property. Alpha and points are the only metadata elements that use only common input type elements.

6.4 Serialization

As can be seen, the structure of this part is similar to the structure, the previous version of the plugin had. The difference is that the Harvester 2 has a header file where the audio segment model and models of metadata elements can be found. The reason for this separation is to make code cleaner, because, for example, this version of the plugin has definitely more models to describe, than the previous one had.

6.4.1 Models

According to the class diagram, it is clear that we have a class that is inherited by classes that describe the metadata plugin works with. Also, there are two DTO classes that will be used during the .avalon_module creation.

6.4.2 PluginSerialization

The header file that consists of only one class - Serializator. The class is responsible for all processes that are related to the serialization of the data plugin produces or consumes, e.g., save or get metadata element that was created by the user, fix metadata element or audio segment collision and of course convert and export the data in the Avalon music engine format.

6.5 Data flow

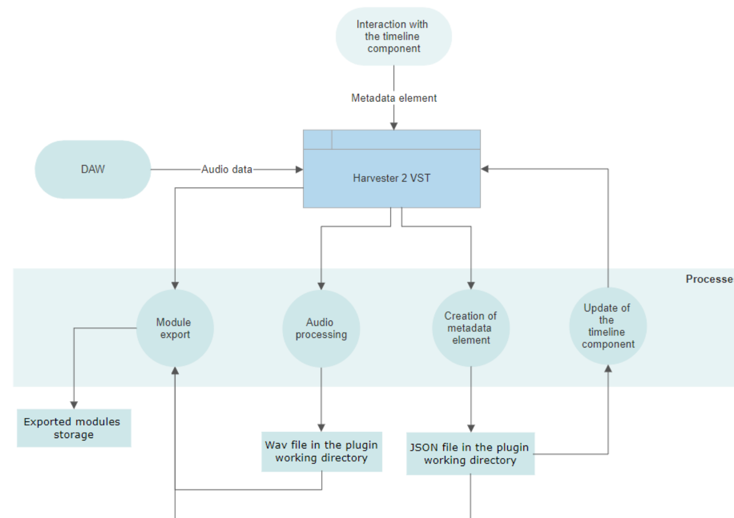


Figure 6.2: The data flow of Harvester 2 VST.

As can be seen in the diagram, input sources of the plugin are:

- DAW that provides audio data

- The user interaction with the TimelineComponent.

There are 3 data stores for the data plugin works with:

- JSON file in the plugin working directory that contains information about the metadata elements and audio segments that were created by the user
- WAV files in the plugin working directory that were created during the audio processing
- Exported modules storage is a subdirectory of the plugin working directory that stores all .avalon_module and .avalon_bank files that were created by the plugin

6.6 Extra

6.6.1 Audio player

The class that represents the built-in audio player that will be used in the TimelineComponent.

6.6.2 PluginUtils

This class is a substitution of HarvesterProperties class that was used in the previous version of the plugin. The class holds the path to the current working directory, as HarvesterProperties does, but also defines some static methods, that will be used in different components all among the plugin, for example, conversion from samples to human readable time format and vice versa.

Chapter 7
Realization

In this section, I want to describe software that was used during the plugin implementation and the way it was done.

7.1 Used software

AAAC_Editor. AAAC_Editor is a tool for testing of the Avalon modules.

Reaper. Reaper is a digital audio workstation and MIDI sequencer software created by Cockos. During the implementation process, Reaper was used as the main testing environment.

AudioPluginHost[ROLa]. AudioPluginHost is a digital audio workstation that comes with the JUCE framework code. During the implementation process, AudioPluginHost was used as an additional testing environment.

FFmpeg[FFm]. FFmpeg is the leading multimedia framework, able to decode, encode, transcode, mux, demux, stream, filter and play pretty much anything that humans and machines have created. It supports the most obscure ancient formats up to the cutting edge. In the plugin, FFmpeg is used as a tool for trimming audio files and conversion between WAV and .ogg formats.

Audacity[Aud]. Audacity is a free, easy-to-use, multi-track audio editor and recorder for Windows, Mac OS X, GNU/Linux, and other operating systems. During the implementation, process Audacity was used as a tool for checking the correctness of audio data that were serialized or merged during the collision fix process.

7.2 Implementation

7.2.1 Audio processing

In comparison to the previous version of the Harvester VST, the processing is pretty simple. The audio processing policy of Harvester 2 VST can be

described as “Everything that was played or rendered in the DAW has to be saved as a separate file in plugin working directory”. For the implementation of the policy JUCE classes AudioFormatWriter and AudioPlayHead were used. Inside the processBlock method of our AudioProcessor class, we update the instance of AudioPlayHead class to receive information about the position and status of sequencer moving play head. When the playback starts, the plugin easily detects that change and reinitializes the instance of AudioFormatWriter and writing of audio to a file begins. Also, during this process, the audio data sample length is being counted. When playback stops the instance of AudioFormatWriter is deleted to close outputStream and file the data are being written to.

7.2.2 GUI

TimelineComponent

Unfortunately, the JUCE framework doesn’t have a component that represents a timeline, so we need to implement our own. The timeline drawing process consists of the following steps:

1. The lines that are used as borders for the other parts of the timeline and the grey line that represent audio segments data are drawn
2. The vertical lines and the text elements that represent certain time positions are drawn.
3. Some parts of the gray line that represent audio are redrawn to green to identify what audio segments were processed by the plugin
4. Rectangles that represent metadata elements are drawn.
5. The red vertical line that depicts the current playback position of the audio player is drawn.
6. The content of the visual feedback buffer is drawn

The timeline is limited to show only 15 hours of data. First, there was an idea that the number of hours will be based on the length of the DAW project, but then it was discovered that there is no such functionality in the VST standard. The default settings define the hour-pixel ratio as 1 hour is 120 pixels. Thus the default visible area is limited to approximately 5 hours 27 minutes, but at the maximal zoom in level, the visible area is limited to 20 seconds. During the metadata element creation process, the user can see the visual feedback to the performed actions. After the click on a metadata element creation button, we start to listen to the user’s mouse actions. After the first click, we create the Rectangle, that is used as the visual feedback buffer, where x coordinate of the click position is initial x of the Rectangle, the width is calculated from the actual mouse cursor position, the height and y position properties are constant values defined

in the TimelineComponent class. The mouse listener of the component is also responsible for handling zoom in/out and visible position change events. The keyboard listener handles the Ctrl+Z combination press by calling the suitable method of the TimelineComponent class. The undo is performed by retrieving the last element from the list of types of the created metadata elements, based on the metadata element type, the last element of suitable metadata elements buffer is removed, the last element of the types list is removed as well.

WaveformComponent

The component that displays dynamically creates thumbnails of processed audio segments based on the current visible sample positions. The component is initialized in the PluginEditor class and then passed by reference to the TimelineComponent. The reason why initialization is not performed in the TimelineCompoent is to avoid extra work with mouse listeners, such as modification of mouseDown event coordinates to make the change of the playhead position by clicking on every point of TimelineComponent possible. Every time when the visible area of the TimelineComponent changes, the updateData method of the class is called. The method iterates over the list of visible audio segments and during the iteration calculates the position and creates corresponding waveform thumbnails objects. The paint method draws the thumbnails. The important thing is to use AudioThumbnailCache for the created thumbnails. The cache runs a single background thread that is shared by all the thumbnails that need it, and it maintains a set of low-res previews in memory, to avoid having to re-scan audio files too often. Thus, we can avoid additional audio files scanning for example during the zoom in/out events.

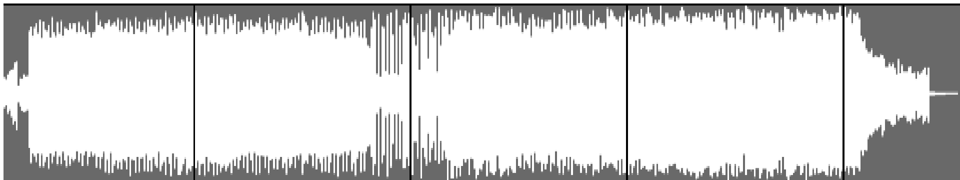


Figure 7.1: Example of waveform thumbnail.

ElementDetailComponent

The component is placed inside of the dialog window that appears every time the user clicks on the corresponding item of the metadata element menu. The component is adaptive to the type of the metadata element that is being edited, i.e. it changes the input elements visibility according to the element type.

7.2.3 Audio player

The audio player extends JUCE AudioAppComponent class. The class was designed first of all for the creation of desktop apps that will interact with the audio input and output devices, and it can be described as a simplified combination of AudioProcessor and PluginEditor classes. The main feature of this class is that it provides a basic AudioDeviceManager object so that we can shift the responsibility for initialization and setting of the audio output device onto the framework. The audio data reading and playback were implemented with the usage of the following classes:

AudioFormatManager

The class that holds a list of the audio formats JUCE framework can read or write. Also, it comes with a factory method that creates a suitable reader for the file that it takes as its only parameter.

AudioFormatReader

The class that is used for reading the samples of an audio file.

AudioSource

AudioSource is a class that can produce a continuous stream of audio data.

AudioFormatReaderSource

AudioFormatReaderSource is a type of AudioSource that will convert the output from AudioFormatReader into the audio data stream.

AudioTransportsource

This is a type of AudioSource that can be repositioned, played, stopped, started, etc.

Every time the user changes the playback position, the audio player receives the filename and a number of samples it must start file reading from, then AudioTransportSource instance is reinitialized according to the new parameters.

Listing 7.1: The method that is responsible for audio player initialization

```
void initialize(std::string filename, int64 playbackOffset)
{
 m_playbackOffset = playbackOffset;

 File audiofile(PluginUtils::getWorkingDirectoryPath()
                   + filename
                   + ".wav"
```

```
                );

  if (!audiofile.exists())
  {
    changeState(Stopping);
  }

  m_visualOffset = int64(std::stoi(filename));

  auto* reader = formatManager.createReaderFor(audiofile);

  if (reader != nullptr)
  {
    std::unique_ptr< AudioFormatReaderSource> newSource(
                                  new AudioFormatReaderSource
                                      (reader, true)
                                  );

    transportSource.setSource(newSource.get(),
                                  0,
                                  nullptr,
                                  reader->sampleRate);

    readerSource.reset(newSource.release());
  }
}
```

7.2.4 Serialization

Global metadata storage

The storage for processed audio segments and user-created metadata elements is represented as a JSON file. The reason for using JSON is the simplicity of the format and native support by the JUCE framework. The file must be named “globalMetadata.json” and located directly in the plugin working directory.

Listing 7.2: Example of globalMetadata.jsonenumerate

```
{
  "audio": [
    [
      0,
      1968128
    ]
  ],
  "modules": [
```

```
    {
      "start": 240000,
      "end": 1200000,
      "name": "Testovaci",
      "code": "0001T"
    }
  ],
  "patterns": [
    {
      "start": 242160,
      "end": 360000,
      "type": 77,
      "number": 1
    }
  ],
  "outregions": [
    {
      "start": 840000,
      "end": 870000,
      "fadeShape": 127
    }
  ],
  "inregions": [
    {
      "start": 600000,
      "end": 630000,
      "fadeShape": 34
    }
  ],
  "betas": [
    [
      360000,
      360000
    ]
  ],
  "alphas": [
    [
      1080000,
      1080000
    ]
  ]
}
```

Metadata element serialization and collision fix

The algorithm for serialization of metadata elements is trivial enough. For example, when the user creates a new module metadata element, the module is pushed to the suitable vector in the TimelineComponent and then the vector is passed by reference to the corresponding update method of Serializator class. The update method sorts the vector using the std::sort function and the process of collision fix begins. There are 2 types of collision that can occur.

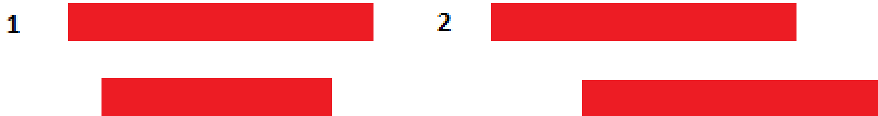


Figure 7.2: Collision types.

- In the first case, there are 2 metadata elements and according to their positions, we can see that one of them is inside of the other. That situation is fixed by removing the shorter element.
- In the second case, there are 2 metadata elements we can call them i, j, the collision occurs under the following conditions: i.start <= j.start <= i.end AND i.end <= j.end. The solution for that issue is merge of the elements.

The last part of the algorithm is the creation of JSON representation of the metadata elements vector and replacement of the suitable item of global storage with the new data.

Audio data segments collision

The process of audio data segments collision fix is more complicated than the process of metadata elements collision fix because it is not enough to update the global metadata storage, stored audio files must be updated as well. In the first case, when we have audio files layered, according to their positions, we can solve the issue just by deleting the shorter audio file. In the second case, the audio files must be merged. The merging process is implemented by using MixerAudioSource - a type of AudioSource that mixes together the output of a set of other AudioSources. The algorithm of the merging process consists of the following steps:

1. Calculate the offset of audio segments start positions
2. Create a zero-filled AudioBuffer (other words silence) with the length of calculated offset.
3. Create a temporary audio file that begins with the zero-filled audio buffer and ends with the audio data of the segment that has a bigger samples start position value.

4. Initialize AudioFormatReaderSources for the audio file that has a smaller start samples position value and the file from step 3.

5. Initialize MixerAudioSource and set AudioFormatReaderSources from step 4 as its input.

6. Write the output of MixerAudioSource to a new file.

7. Delete the temporary file that was created in step 3 and file that has smaller start samples position value.

8. Rename the output audio file from step 6 to a file that had bigger start samples position value.

7.2.5 Export

The process of export of the created metadata elements and the processed audio segments as an Avalon music engine module begins by collecting the list of processed audio segments positions because we need to exclude possibility of export of metadata elements that do not have corresponding audio data. Then we iterate over the module type metadata elements that can be found in the range of the existing audio segments positions. During the iteration, the instance of ExportData class is being filled with data of patterns, that are located in the range of module position, and their special points and regions. Then the ExportData instance is passed to corresponding functions for .avalon_moulde and .avalon_bank files creation.

.avalon_module creation

Due to the fact that the .avalon_module file structure is strictly defined, the task of its generation can be characterized as straightforward. We just need to convert the data that the ExportData instance holds into a formatted string.

.avalon_bank creation

The process of .avalon_bank file creation is more complicated. First of all, we need to trim processed audio files according to the sample positions of the pattern type metadata elements; then the trimmed audio files must be converted into the .ogg format. The trimming and conversion processes were implemented by using the following FFmpeg commands:

Listing 7.3: Example of the FFmpeg commands used during the .avalon_bank creation process

```
 //Audio file trimming by the sample positions
 ffmpeg -i in -af atrim=start_sample=0:end_sample=48000
out

 //Wav to .ogg conversion
```

```
ffmpeg -i audio.wav  -acodec libvorbis audio.ogg
```

The commands are being invoked by using the system() function.

When the trimming and conversion processes are over, we can start creating the header of the file. As it was described in the specification, the header consists of the string that determines that the file is an Avalon music engine module and the list of .ogg sound banks. After the header creation, we start to append the content of generated .ogg files to the .avalon_bank file end. In the end, all .ogg and wav files inside the export directory are deleted, so the directory will contain only .avalon_module and .avalon_bank files.

7.3 Testing

Testing is one of the most difficult and time-consuming parts of the VST plugin development process. Unfortunately, there are no testing frameworks for such a specific task, and the only option is to perform the tests manually. JUCE AudioPluginHost and Reaper were used as the host applications for debugging.

7.4 GUI illustration

In this section there are screenshots of different parts of the plugin.

7.4.1 Overall

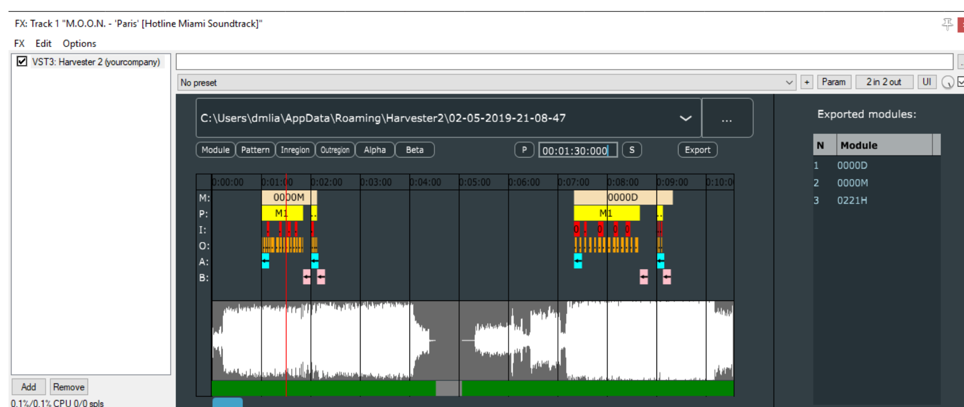


Figure 7.3: The plugin hosted in DAW Reaper.

The screenshot shows a plugin instance hosted in DAW Reaper. There are waveforms for 2 processed audio data segments and a set of data elements that were created by the user. On the right half of the screen the list of the exported modules located.

7.4.2 Pop-up menu

The screenshot shows the context menu that pops up when the user right-clicks on a metadata element.

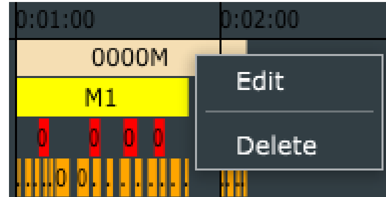
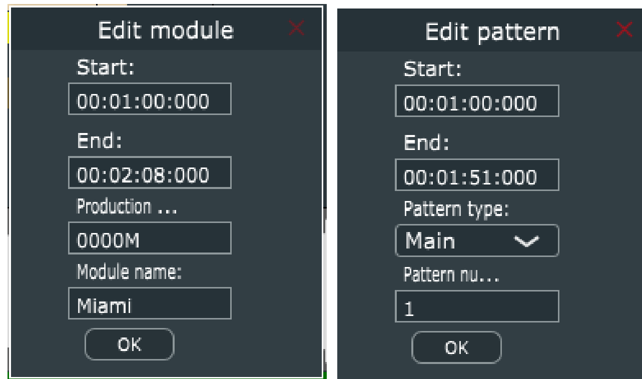


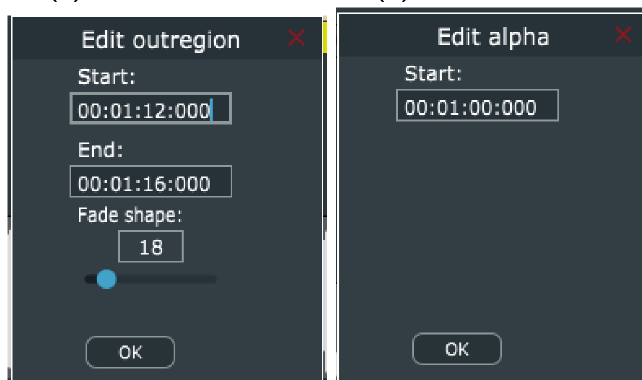
Figure 7.4: The context menu.

7.4.3 Metadata element editing

The screenshots show how the content of ElementDetailComponent changes depending on the metadata element type.



(a) : Module details **(b) :** Pattern details



(c) : Inregion details **(d) :** Alpha point details

Chapter 8 Conclusion

In this work, I have studied the principles and common techniques of the adaptive music creation process. I became more familiar with the concept of Digital Audio Workstation software, the principles of VST plugins functioning and MIDI standard.

As the result of this project I created 2 VST plugins: Harvester VST is a plugin for music segmentation based on the MIDI events mappings. Harvester 2 VST is a plugin for music segmentation based on the user's interaction with plugin GUI. Unlike the first version, Harvester 2 VST is directly integrated with Avalon Music Engine by producing data in the engine defined proprietary format. The functionality of both plugins was tested in different host applications. The correctness of the output was approved by testing in the special application for testing of Avalon Music Engine modules.

Both plugins are written in C++ programming language with the help of the JUCE framework.

This work may be continued in the future to bring more enchantments into the structure, GUI or workflow of both of the plugins. For example, the problem of beat detection. The beat detection of the Harvester VST plugin is based on the midi mapping that can be considered as a good and reliable solution from a functionality perspective, but unfavorable from the perspective of UX.

Appendices

Appendix A

Bibliography

[Aud] Audacity Team, *Audacity - Free, open source, cross-platform audio software*, [online] `https://www.audacityteam.org/`, Accessed: 15-05-2019.

[Coca] Cockos Inc., *Cockos WDL framework*, [online] `https://www.cockos.com/wdl/`, Accessed: 15-05-2019.

[Cocb] ———, *Digital Audio Workstation Reaper*, [online] `https://www.reaper.fm/`, Accessed: 15-05-2019.

[FFm] FFmpeg Developers, *FFmpeg - A complete, cross-platform solution to record, convert and stream audio and video.*, [online] `https://ffmpeg.org/`, Accessed: 15-05-2019.

[Ima] Image Line Software nv, *Synthesizer SAWER*, [online] `https://www.image-line.com/plugins/Synths/Sawer/`, Accessed: 15-05-2019.

[Kä18] Lassi Kähärä, *Producing adaptive music for non-linear media*, Bachelor's thesis, Tampere University of Applied Sciences, 2018.

[MID96] MIDI Manufacturers Association (MMA), *The complete midi 1.0 detailed specification*, 1996, [online] `https://www.midi.org/specifications-old/item/the-midi-1-0-specification`, Accessed: 15-05-2019.

[ROLa] ROLI Ltd., *JUCE AudioPluginHost*, [online] `https://github.com/WeAreROLI/JUCE/tree/master/extras/AudioPluginHost`, Accessed: 15-05-2019.

[ROLb] ———, *JUCE framework*, [online] `https://juce.com/`, Accessed: 15-05-2019.

[Ste99] Steinberg, *Virtual Studio Technology Plug-In Specification 2.0 Software Development Kit*, 1999, [online] `http://jvstwrapper.sourceforge.net/vst20spec.pdf`, Accessed: 15-05-2019.

[wel18a] welove.audio GmbH, *Avalon Adaptive Audio*, 2018, [online] `https://welove.audio`, Accessed: 19-05-2019.

[wel18b] ———, *Avalon Adaptive Audio. Technical documentation.*, 2018.

[wel18c] ———, *Avalon Music Engine*, 2018, [online] https://www.adaptive.audio, Accessed: 15-05-2019.

Appendix B

CD contents

```
./
├── BP_Liasko _2019.pdf ........................ Thesis
└── Plugin ................ Source code, projects, libraries
    ├── Harvester 2.jucer ............. Projucer project file
    ├── Builds ......................... IDE projects folder
    │   └── VisualStudio2017 ....... Visual Studio project folder
    ├── JuceLibraryCode ................ JUCE library code
    └── Source .................. The source code of the plugin
```