Making Learning Tracks - An Administration Tool

Simon Rylander (c04sr@student.lth.se)

December 15, 2012

Contents

1 Introduction **5**
1.1 Background . . . 5
1.2 Recording and Selling Learning Tracks . . . 6
1.2.1 Business Idea . . . 6
1.2.2 Setup and Planning . . . 7
1.2.3 Recording . . . 7
1.2.4 Rendering Mixes . . . 8
1.2.5 Delivery . . . 8
1.2.6 Backing Up Work . . . 8
1.3 Question At Issue . . . 8

2 Task **10**
2.1 Requirements . . . 10
2.2 Planning . . . 12
2.2.1 Investigation . . . 12
2.2.2 Implementation . . . 13
2.2.3 Testing . . . 13

3 Investigation **15**
3.1 File Formats . . . 15
3.1.1 Wave . . . 15
3.2 Audio Compresion . . . 16
3.2.1 MP3 . . . 17
3.2.2 AAC . . . 18
3.2.3 Most Viable Format . . . 18
3.3 The Physics of Sound . . . 18
3.4 Recording A Voice . . . 19

4 Implementation **21**
4.1 Modules . . . 21
4.1.1 MP3 Encoder . . . 21
4.1.2 Wave Editor . . . 21
4.1.3 Equalizer . . . 22
4.1.4 VST Effect Applyer . . . 23
4.1.5 Compressor . . . 24
4.1.6 FTP Client . . . 24
4.2 Processors . . . 24
4.2.1 Mp3 Decoder . . . 25
4.2.2 Equalizer . . . 25
4.2.3 Channel FX Applyer . . . 25
4.2.4 Wave Mixer . . . 25
4.2.5 Mix FX Applyer . . . 26
4.2.6 Mix Masterer . . . 26
4.2.7 Mp3 Encoder . . . 27
4.2.8 Compressor . . . 27
4.2.9 FTP Uploader . . . 27
4.3 Graphical User Interface . . . 27
4.4 The Tasks . . . 28

4.5 The Feedback System 29
4.6 Settings 29

5 Result/Conclusion **30**
5.1 Future Enhancements 30

Abstract

There are two ways to learn a capella music: Reading and listening. Reading music (ie sheet music) is very hard, and requires years of education and practise to be able to do it well. Listening and learning is easier, if there is a source of what is needed to be learned that can be listen to available. There might be recordings by other groups of the song, but unless it is the melody needed to learn, the task is very hard.

But what if there was a professional that can be hired to record the song, where one part is really loud, or panned away from the other parts? That way, it would be easy to listen and learn any part, while also being able to hear the full context! This is actually very possible! There are several companies out there that will record an a capella song upon order and create several mixes with each part played with a significantly higher volume, and more. This will allow choruses, quartets and choirs to drop the requirement on their members to be able to read music, hence they can expand in numbers and quality.

The companies that are in this line of business are mostly one-man companies where this one person is able to record all parts himself. This obviously requires a very wide voice range, and a lot of knowledge of music production. So the work for this man to produce learning tracks for one song is significant, and starts with singing, then mixing, and finally rendering of mixes. It is also very common for someone in this line of business to specialize in a certain sub-genre of a capella music. Therefore, a lot of the things done when recording these songs get repetative. Is there a way to optimize this process?

This project includes an in-depth look into all possible optimization possibilities in this business and a detailed description of implementation of software that supports this automation process. This software promises to take away up to 20

1 Introduction

1.1 Background

A capella music, which means music without instruments, is unique in the sense that music education is not needed to be a part of it. Being able to carry a tune with nothing but the voice is something that comes naturally to many people, and practising singing is something that can be, and is being done everywhere, i.e in the car, in the shower and in the kitchen. Hence, singing is something that many learn to do by just living a normal life. When learning an instrument, it usually requires a certain degree of music theory knowledge to be able to learn new music and/or create music alone by improvising. It is not sufficient to hear someone else play a tune to be able to copy it, one also needs to know the theory of how to produce the sounds coming from the instrument. Without knowing this theory, one will simply not be able to play an instrument very well.

Because of these facts, the voice is in one way the simplest instrument one can play, but in other ways also the hardest. To carry a tune with a low degree of quality is a simple task. Good singing however, is one of the hardest instruments there is. Creating music with the voice is, unlike many other instruments, all about very precise muscle control, and a high degree of concentration. And the only way to know if the right note is "playing" is by listening to oneself and adjusting accordingly. A common problem regarding the concentration issue is that many singers tend to become confused when hearing other voices trying to harmonize. Some have a problem differing their own voice from other voices, thus they do not know how to adjust their own voice. Others might be confused simply because they need to know what the harmonies are supposed to sound like before being able to be a part of it.

Choruses, choirs and other a capella singing groups all over the world that try to expand their numbers struggle with this problem. There are plenty of people being able to sing out there, but few that know the music theory required to learn music at a satisfactory pace. This means that the size of the chorus is related to the effectiveness of the learning process (among other things). The most common way of learning any music is either by listening and learning (which is not a good option in a capella music, due to the difficulty in keeping the different voices apart from eachother) or reading tabs or sheet music. Being able to read music, and also being able to sing that music, is something that requires a lot of musical training, both theory and practise.

In 1995, a man by the name of Chris Arnold[1] came up with an idea to enable "listening and learning" as an option for a capella music as well. Prior to the 90's, recording equipment was not something that every man simply had in their home, especially not multi-channel recording equipment. Therefore, this idea, as late as it was, could probably not have been put into practise much earlier. The idea was to use the standard Stereo-Mode Audio Playback method that is used in radios, TV's, car stereos and other music playback equipment, and have the voice part needed for learning in one channel, while having the other parts in the other channel. This allows hearing exactly what the part is, while

[1] http://www.qsvp.com/

also hearing the full context of the music. Also, no music theory knowledge whatsoever is needed to be able to learn, and sing a capella music.

1.2 Recording and Selling Learning Tracks

1.2.1 Business Idea

So to enable this idea, to make it possible to learn a capella music by ear, there has to be recordings of the songs to learn, to start with. Using any recordings that might already exist is not sufficient due to the fact that it is most likely not part-separated. The business idea is to create fresh recordings with multi-part recording equipment, creating the learning files by making different mixes of the recordings, and then selling this to singing groups. This would require a person or group that can read sheet music and is able to create music from it. Since most a capella music uses the full potential of the human voice range, both male and female, it would require many different voice types, or one person with a very good vocal range.

As this idea has evolved, more ideas of beneficial mixes for learning has arisen. Here is a list of other significantly relevant mixes that can be included in the product:

- **Full Mix** - All the parts are panned the way they should be panned according to the style's standard. This mix is for enjoyment purposes mainly, but also for providing a clear view of the intended interpretation.
- **Part Left Mixes** - One voice part is panned[2] all the way to the left, while the other three parts, at a softer volume level, are panned to the right. This mix is to enable the user using michrophones or separated speakers to hear his/her voice part separately while also hearning the other voice parts to be able to get a sense of the full harmony spectrum.
- **Part Predominant Mixes** - One voice part is panned to the center, with equal level in both speakers, while the original panning remains for the remaining parts. The audio level of the predominant part is significantly louder than the others. This mix is to enable learning even for people that for any reason at all do not have Stereo playback capabilities. Although this is not as effective as isolating the parts in separate channels, the specific voice part is still clearly audible, while the others are not completely missing.
- **Part Missing Mixes** - One voice part is completely muted while the other parts remain the same as in the full mix. This enables actually rehearsing singing with the recording, making sure that the voice part is completely learned, and to practise note accuracy.

With these four kinds of mixes (that, for 4 voice parts, result in 13 different mixes per song), most needs for learning the music are convered. There is also the possibility of other mixes, such as Double Part Missing tracks, for duette

[2]Panning, in terms of audio playback, refers to the balance between the left and the right channel. If a sound is panned 100% to the left, then the audio will solely appear in the left speaker

practising, or Part Right tracks which is convenient for car rides for customers living in a country with left-hand traffic (it is always more efficient to have the stand-alone voice part in the speaker closest). However, these mixes can be created upon special requests.

If a person is going to use these tracks to learn their music, they are going to have to listen to each song many times. It is also logical that the more enjoyable the music is, the more interested the person will be that listens to them, and the faster they will learn it. Also, a low-quality (or "bad") recording might be very exhausting to listen to several times in a row. Hence, a sense of quality is very important, both vocally and production-wise.

1.2.2 Setup and Planning

With today's technology, it is convenient to just use a computer and a multi-channel recording program to record the music. To ensure complete isolation of each voice part, one typically records one part at a time, starting with the leading voice part (in for instance Barbershop music, it is most often the Lead part, whereas in classical Male Choirs, it is more likely to be the First Tenor). Then, the bottom part is laid down and the rest recorded from bottom to top. But before commencing the recording process, it is important that one gets a general feeling for what the music is supposed to be. This can prove to be a very difficult task, just looking at the sheet music. However, for someone with significant experience in the music style, and good research capabilities, it is far from impossible.

Setting up the recording process requires that there is a clear vision of the outcome. For instance, one would typically want some special effects on the recording, such as a Reverb and Equalizer that improves the audio and makes it more enjoyable. If the recording artist is specialized in one or a few certain styles of a capella music, these effects and their settings vary little from song to song, and templates in the recording program can ensure that they do not have to do it for each new recording.

1.2.3 Recording

The recording process is in general very time-demanding, provided that the requirement of quality is significant. It might take 1-2 hours to record each voice part, considering that all notes, all dynamics, all the pronounciation and phrasing has to be perfect. However, once having sung all the parts once, creating the mixes for each voice part does not require any additional singing. The recording phase might include some mixing and pitch correction of the sung parts, depending on the method of proceeding, or it might be something that is done afterwards. In either case, to ensure that the harmonies are perfect, and not even-tempered[3], this is something that is usually done.

[3]In music, even-tempered tuning is an approximation of the just tuning, that allows instruments to change keys. However, since vocal harmonies become very dissonant when singing even-tempered harmonies, and since there is no need of approximation for the voice, all kinds of a capella singing uses just harmonies.

1.2.4 Rendering Mixes

Most multi-channel recording programs are designed to record and/or mix music and then render one single mix. Therefore, the rendering process is very slow, and in some cases even in real-time, which means that the rendering time is equal to the song length. Making 13 mixes for a song with 4 voice parts (or $3n + 1$ mixes for a song with n voice parts) will, no matter what recording program, take a lot of time. For each mix, there are settings that need to change, and the recording artist might have to listen a couple of times to make sure that the settings are satisfactory. Once all the mixes are produced, they need to be mastered. That means, the sound level needs to be optimized for each mix and each channel of the mix. This can, in extreme cases, even mean volume modulation of different frequencies. However, due to the nature of the tracks, this might be considered luxury.

1.2.5 Delivery

Once the music is created and all the mixes are rendered, they have to be edelivered to the customer. This process can for instance be handled by encoding the tracks to MP3 audio files, compressing all mixes in a ZIP file, uploading it to an FTP and providing the customer with a download link through e-mail. Very easy, but still time-requiring.

1.2.6 Backing Up Work

After everything is completed, the work needs to be saved and stored. The least space-consuming and most flexible way of doing this is to save each voice part, with no special effects added, separately on a harddrive. That way, the artist still has the option to change anything he would like fairly easy, by simply putting the parts back into a recording program and re-recording or changing anything he wants. While the very simplest way of keeping a recording project would be to just save it in the recording program, this can occupy a lot of space on the harddrive, which is not very effective. This also makes it a lot harder when considering adapting to another type of software for future recordings.

1.3 Question At Issue

When looking at the above stages of production and delivery, the question at issue is: **Is there any way to optimize this process by automation?** There are clear signs that some of these steps can be automated, hence resulting in less time consumption. Setting up a new project of recording can easily be automated with templates within a recording program. Planning the recording and analyzing the arrangement however, cannot. The actual recording of the voice parts is impossible to automate with today's technology, but who knows, in the future, there might be synthesizers that are able to read sheet music and produce musical instruments sounding like vocals. Backing up the work by rendering each channel as a separate audio file can in most multi-channel

recording programs be done in a few clicks, which means it is basically already as automated as it gets. But when looking at the remaining steps - rendering mixes, applying effects, mastering, encoding, compressing and uploading - which is a process that takes over an hour per song, out of which 95% of the time is waiting for the computer, it might be possible to create a program that performs all these steps automatically, based on nothing but the backup files. One would typically have to set some default values for the process before the rendering, but after that, the program should be able to handle everything in one single process.

2 Task

The task for this project would be to optimize and automate as much of the process as possible, without limiting the options for the creation of the tracks. The goal would be to create a method for selecting all the properties regarding the tracks beforehand, and then proccess everything at once. That way, the workload is minimized, and everything is handled in one single step.

The process of mixing and mastering the tracks seems to be something that requires a lot of listening and adjusting, which might be hard to automate. But after having recorded several hundreds of songs, this process starts repeating itself, and eventually one almost knows beforehand what needs to be done, and when. So after further analysis and help from experts, this task turned out to be far from impossible. All the steps mentioned above can most likely be automated successfully.

The automation of this process could be implemented in a computer program that takes in the individual voice tracks as input and generates the mixed files, uploads them and distributes them. The process can be run on any computer, and is typically only needed on a workstation, therefore, making it a standalone computer application with a *Graphical User Interface* (GUI) would be an ideal solution. The program could be written in most programming languages, and one that will fulfill all requirements, with an easily implementable GUI is Java, which will be chosen for this project.

2.1 Requirements

To be able to create a program that handles a sequence of tasks properly, and in the right order, a list of clear requirements needed to be established. Exactly what happens from the input is delivered to when the processing is done.

1. The program must have an easy-to-use graphical user interface that allows the user to understand what he is doing and lets him customize the generation precisely according to his preference.
2. The program should have a number of subsequent tasks to perform in a non-changeable, pre-determined sequence. Each task should take in input files of a certain file format, process them and generate output files of a certain format. This requirement is so that the processing can be cancelled if something goes wrong, and then re-established without having to start over.
3. The program should be able to accept any input file formats that any of the tasks accepts as input. When specifying the input files, the program must be able to make a qualified guess on what task(s) should be involved in the processing and creation of the tracks (from now on referred to as job). The user should also be able to re-specify what tasks, and how many tasks should be activated and inactivated for the job.
4. The program should be able to enqueue several jobs in the process queue, so that when started, all jobs will be processed without stopping between

them. All jobs in the queue should be editable and removable at any time during the customization phase.

5. During the processing phase, feedback should be given at all time, letting the user know exactly at what task and stage the processing is at.
6. During the processing phase, the user should be able to interrupt and cancel the processing at any time. If this happens, the last completed files should be stored on the harddrive so that the user can restart the process beginning at that same task.
7. Each task should have the option of whether or not to keep the output files on the computer after generation. However, the last task of the processing should always save its output files. The input files of the entire job should never be removed.
8. The tasks (in order of processing) should be as follows:
   (a) **MP3 Decoder task** - Should decode the backup files from MP3 to WAV format. This is in case the user stores all his backup files as MP3's. No settings necessary.
   (b) **Channel FX task** - Should apply channel-specific audio effects of the VST audio plugin standard to the wave files. This is to apply channel effects such as compressors that evens out the loud sections, and gates that reduce the noise. The user should be able to specify what plugins to use and settings for each one of them.
   (c) **Wave Mixer task** - Should turn the channel-specific wave files into learning track mixes. The user must be able to determine volume levels and pans of each of the channel files, and what mixes that should be created. The options should be Full Mix, Part Missing tracks, Part left tracks, Part Predominant tracks and Sample (the sample is for demonstration on the web page). There should be settings regarding volume level on the left and predominant parts, and starting time of the sample.
   (d) **Equalizer task** - Should equalize the mixes, enhancing or reducing any given frequencies by running them through an audio filter. The parameters for the equalizer should be editable by the user.
   (e) **Mix FX task** - Should apply mix-specific VST audio effects to the wave files. This is to apply mix effects such as reverb and echo, to enhance the listening experience. The user should be able to specify what plugins to use and settings for each one of them.
   (f) **Masterer task** - Should apply mastering VST audio effects to the wave files. This is to master the tracks with effects such as multiband compressors and maximizers, to enhance the volume level and even out any possible bumps that might stick out from the rest of the audio. This task should have options to normalize the waves before and/or after the applying of the effects.
   (g) **Pitch applyer task** - Should apply pre-determined *pitch files*[4] to

[4]Wave files containing a single audio signal indicating what key the song starts in. When

the beginning of some of the mixes. The user should be able to specify which mixes to apply the pitch note to.

(h) **MP3 Encoder task** - Should encode the mix files to MP3 files. This is to reduce the size of the files, making them deliverable to the customers. The user should be able to specify the bitrate of the MP3 files.

(i) **Compressor task** - Should compress all mixes to one single zip file. This is to be able to deliver the tracks in one single package. The user should be able to specify a file name, however a suggestion should always be generated based on the file names of the tracks.

(j) **FTP Uploader task** - Should upload the zip file to an FTP server. This is for the actual delivery of the tracks to the customer. The user should be able to specify what FTP server to connect to, username and password for the server, and what directory to put the file in.

9. There should be a function allowing saving of all the customization settings of all tasks that can have a default value (that is, a value that will most likely repeat itself from processing to processing), so that a new job is opened, these settings should be the default settings. This 'includes all parameter settings on each of the VST plugins.

10. The program must be able to process a job in a reasonable time frame, i.e less than an hour per song.

2.2 Planning

Before starting this project, a good project plan was necessary. The project from here on will consist of three major phases: Investigation, implementation and testing. Each of these phases requires planning of its own.

2.2.1 Investigation

This program is going to work with raw audio data, and the very first thing that needs to be done is research an appropriate audio format. This format will require individual research of the file structure and what different kinds of format settings that can be used internally. The goal is to have a final product that supports all kinds of format settings within the format, to ensure a high-quality end product.

This program will work with this audio format basically throughout the entire process. That means, it will be processed and mixed in many different ways. Before starting the implementation, everything needs to be clear on how processing of audio works.

To minimize the workload, research on what implementations might already exist is evident. Many of the steps within the processing of the files are very

singing a capella music, one often starts a song by using an instrument or device to play the key note, so that the singers are able to locate their starting note

basic, and might have been implemented and shared by others on the Internet. This is very common in Java programming, and might lead to an efficient implementation progression.

2.2.2 Implementation

The first thing that needs to be established is the system architecture. Regarding the architecture, starting from the deepest sections of the project and then working the way up is usually the best approach. With that approach, it is important never to have any *upward dependencies*[5]. That way, the need of implementing anything that depends on something that does not yet exist is eliminated.

It is clear, that to be able to perform these tasks, some bottom-level *Base Modules* within the project should be the first step. These modules should be standalone packages that depend on nothing, that have the basic functionality to be able to do the job, but implemented in a way that makes it universal, expandable and easily extractable. They must also have an easy-to-use *Application Programming Interface* (API). The modules themselves should not be intelligent at all, i.e they should accept any input settings and they will only do what they are told to do. These modules are going to be used by mid-level *Processors* that have a little intelligence of their own. These processors define the order of execution, i.e. what modules should do what with the files, and in what order. The processors will take in certain settings (such as audio levels for the mixes), but they know the basics of what to do with the modules (such as how to create each individual mix). For each of the processors, a high-level *Task*, that is directly connected to the GUI, controls the input-output and defines where in the sequence the processor is to be used. It should also display the settings available for the processor, and be responsible for setting these in the processor prior to processing.

On top of that, there should be a GUI. Aside from displaying the specific tasks, and the layout of the settings, the GUI will be very basic. It should contain the functionality of enqueuing jobs, saving settings, loading files and displaying status messages and feedback. This would be the very last step of the implementation phase, that connects everything together.

It is also likely that some classes and structure will be needed throughout the whole project, therefore there should be a common library that all packages can access at any time. This package could, among other things, contain a calculator that helps with byte-to-integer conversion and vice versa, and other mathematical aspects of the implementation.

2.2.3 Testing

The goal with any project is always to minimize the testing phase. The better the unit tests, the less acceptance testing and bug correcting will be necessary

[5]To maintain flexibility at all time in the system, there should never be both-way dependencies between any of the packages. Therefore, a "bottom" package is defined as a package with no dependencies to any other packages.

before the program is fully functional. The testing will naturally be performed with part specific audio files, trying to create a bundle of different mixes. It is important to try all kinds of different audio file formats, and make sure any format is supported. Trying all kinds of different task settings is the next step, and it is likely that each module is going to require some bug fixing. To make sure to be prepared, a lot of time should be reserved for this phase.

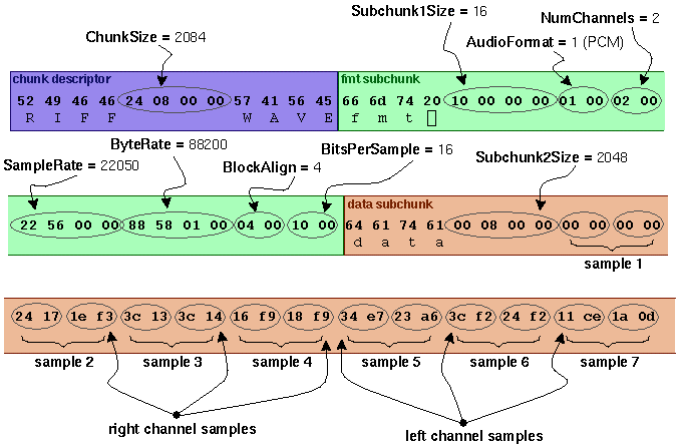

Figure 1: The WAV file format, and some standard values.

3 Investigation

3.1 File Formats

For editing of audio, uncompressed audio sources are the most optimal solution, as that is the fastest way to compute the changes. However, a large amount of files are to be shipped to the end customers, therefore an audio compression is required to the end result of the files for faster delivery and smaller storage. Hence, this program needs to support two audio file formats. The undisputedly most common format of uncompressed audio is the Wave file format.

3.1.1 Wave

The structure of the audio storage in a WAV file is basically the sampled sound wave, where the amplitude of each sample is stored in order. This format is not the optimal one, but is very easy to work with, hence most audio programs use it for editing. There is also a format header that specifies different settings, such as how many bits are used per sample (bitrate), how many samples that represent one second (sample rate) and how many audio channels are in the WAV file (usually one or two). Figure 1 explains the file format in detail.

The bytes of the file are, as in many other file formats, divided into so-called "chunks", that have different purposes. A chunk always starts with a four-letter name, and a 32-bit integer specifying the chunk size. All the number fields are encoded in Big Endian order, while the four-letter chunk names are specified in Little Endian order. For the standard WAV file format, the main header chunk (in purple) needs to be specified with those exact fields from the figure, and the chunk size needs to correspond to the number of bytes in the whole file. Since this field is only four bytes long, encoded the same way as a standard 32-bit integer (with one sign bit), this means that a WAV file may never be bigger

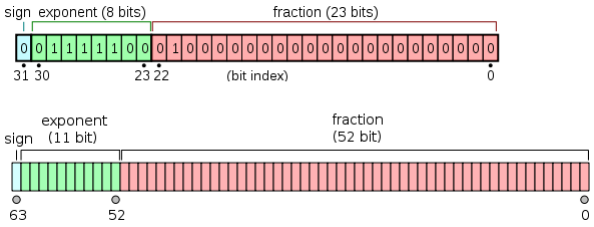

Figure 2: The bit structure of a 32-bit IEEE float (above) and a 64-bit IEEE float (below).

than $2^{31} = 2147483648B = 2GB$, which is a limitation to the format. However, these sizes are irrelevant to this project.

The second mandatory field is the format field (fmt). This field is typically 16 bytes big, but can be longer if specified in the chunk size field. The first 16 bytes however, are always encoded in the specific order of the figure. The number of channels field is self-explanatory (1 or 2 usually), the sample rate explains how many samples are played per second (standard values are any multiple of 11025), and the bits per sample field corresponds to how many bits are used to specify the amplitude of the sample (8, 16, 24, 32 or 64). The two fields "byte rate" and "block align" are somewhat redundant, since they specify multiples of the three previous fields. The byte rate specifies how many bytes are read per second (sample rate * num channels * bits per sample / 8), and the block align specifies how many bytes one sample is (num channels * bits per sample / 8).

There is also an audio format field, or a compression setting field that allows the user to specify if the data is stored in a different format, or with a certain compression. The different compressions are non-standard, and are usually program-specific, but there are two standard ways of encoding the sample amplitude. Either, they are encoded with binary integers, where the maximum amplitude corresponds to the highest value possible to encode with the given number of bytes, with one sign-bit. The second way of doing it is to encode the samples as IEEE Float values between -1 and 1. An IEEE Float is either 32 or 64 bits (4 or 8 bytes). Figure 2 explains the format of an IEEE Float in further detail. An IEEE float consists of one sign bit (just like the integer format), a small number of exponent bits and a bigger number of decimal bits. But since the WAV Float format only uses numbers between -1 and 1, the exponent bits are unused, hence there is a certain loss of precision compared to the integer correspondence of an equal number of bits. The format is standard however, and needs to be supported.

3.2 Audio Compresion

Although the WAVE file format is very optimal to work with, because of its non-decoded and raw nature, it is less than optimal when it comes to storage and file transfer, due to its size. Over the past 20 years, there has been a significant amount of research on the subject of audio compression, with and

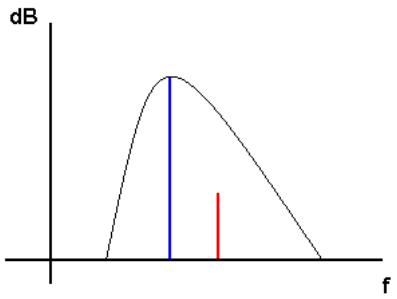

Figure 3: A simple display of frequency masking. The light blue audio tone masks all frequencies under the black line, such as the red audio tone.

without loss of quality, and the world-leading organization behind this research is called *Moving Pictures Experts Group* (MPEG), who have worked with both audio and video compression. Over the years, they have established a few new standards of different complexity, starting with MPEG-1. The research of MPEG-1 started in 1988, and the standard was finalized in 1992. The MPEG-1 standard consits of three different compression groups called *Layers*, each one of different complexity, the Layer 3 (also nicknamed MP3) being the most complex. This standard revolutionized the world when it was finalized, due to its ability to reduce the file sizes to a tenth of its uncompressed correspondence, without noticeable quality loss. Since then, it has become a standard in every music playing device in the world. There is in fact an even more efficient standard called *Advanced Audio Coding* (AAC), part of the MPEG-2 standard, which is about 30

3.2.1 MP3

The audio compression technique is mainly based on a specific psychoacoustic phenomenon called frequency masking. This phenomenon means that if a sound of a certain frequency is loud enough, sounds of frequencies near this sound will be masked, assuming that the frequency difference and loudness are not too big. Figure 3 illustrates an approximation of how frequencies are masked under each other. The black peak is a loud tone of a certain frequency. The light blue area indicates what frequency area this tone will mask for the human ear. Hence, weaker tones in nearby frequency areas such as the dark blue peak are inaudible, thus irrelevant to consider when compressing the audio.

When looking at a spectral analysis of a wave file, there will be peaks in the most significant frequencies, and around them some noise that may or may not be audible due to this phenomenon. Research has shown that if analyzed, most of the data in the frequency spectrum is irrelevant due to man kind not being able to hear it. Hence, focusing only on the relevant frequency peaks, it can be encoded in such way that - without audible data loss - the size of the compressed data is significantly lower than the size of the raw data.

The encoding procedure starts with analyzing the wave with a psychoacoustic model and dividing the wave into 576 different frequency bands using a *Modified*

Discrete Cosine Transform (MDCT). After that, attempts of Huffman encoding are made to the frequency bands, which basically means that byte representations of the values are quantized, or "guessed" in a way so that they become extremely small. Then, the data is sent to a control loop that verifies that the sound is as close to the original sound as possible. When this process is completed, the bitstream is formatted into the selected compression rate, where the most common one, 128 kilobyte per second, is about a tenth of its original, unencoded size.

3.2.2 AAC

The AAC standard does not only offer even smaller file sizes to represent audio of the same quality, it also supports stereo, surround and other multi-channel standards. The most significant difference is the even bigger complexity in the coding process where the number of frequency bands is 1024, which is twice as many as the MP3 encoder, and the control loop consists of many more steps, such as adding of scale factors, rate/distortion controll process, improved Huffman encoding and noiseless encoding. This allows the standard to be able to represent audio to the same quality as MP3 at about 70% of the bitrate.

3.2.3 Most Viable Format

For this project, the importance of the spread of the format is valued the highest. All customers must be able to support the format, so a 30% dataloss does not justify usage of another format than MP3.

3.3 The Physics of Sound

All sounds are a composition of sine waves of different frequencies and amplitudes. For instance, a voice consists of one fundamental frequency, which is the lowest frequency and the frequency of the audible tone. Also, it consist of a number of harmonics in the frequency pattern 2x, 3x, 4x... where x is the fundamental frequency. The harmonic pattern of the voice is what enables different vowel sounds. Without them, a voice would just sound like a low whistle tone. So to compose the sound wave of the voice, all the sine waves of the fundamental frequency and all the harmonic frequencies are added together, with their volume level represented as the amplitude of the sine wave.

The same principle is used when adding together any sounds with eachother. Simply put, the sound waves (which are compositions of sine waves) are added together, and the resulting sound is a composition of the separate sounds. To do this mechanically, the sound wave from the WAV files are added together by adding the amplitude of each sample together with the same sample in the next WAV file, and the result is the two sounds combined.

The reason why the most common number of channels for a sound is 2, is that we have two ears. Since the human being only has two audio inputs, which is enough for our brain to determine exactly where a sound comes from, it is also

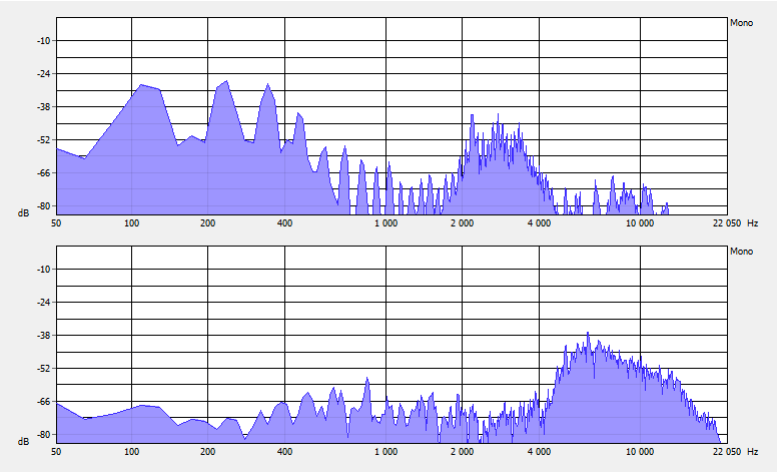

Figure 4: Two spectrum analyses, one of the vowel "E" (above) and one of the hard consonant "S" (below).

possible to create the illusion of a sound coming from anywhere with just two audio channels, assuming that each input (ear) only hears one of the outputs (speakers). Wearing headphones enables exactly that. The phenomenon that enables a sound to appear as if it comes from the side instead of the front is called Panning. A sound that comes from in front of a head, both of its ears pick up fully. However, if the sound moves slightly to the left, the right ear will become partially blocked from the sound by the head. To simulate this phenomenon, reduce the volume of one of the audio channels, and the sound will appear as if slightly moved to the other side. When listening to music, this phenomenon will increase the listening experience, because it feels like standing in the middle of the band, orchestra or chorus. In a capella music, the desired effect is to keep the different singing parts at different locations of the stage, to capture just that.

3.4 Recording A Voice

The audible frequency range of the human ear is approximately 20-20000 Hz. When listening to the vowel sounds of the human voice, the frequency is well-audible and generally pretty low in our audible range of frequencies. For men, the vowel sound of the human singing voice usually ranges between 60 and 500 Hz, and about 120 and 1000 for women. But the consonant sounds, especially the sharp ones such as S, T, K etc, is usually portrayed from 2000-10000 Hz. See figure 4. Have in mind, that any sounds with an amplifiation of -60 dB and below is very quiet, and mainly illustrates noise.

When recording a singing voice, it is important to use a michrophone of sufficient quality. The better (and more expensive) mic used, the better quality the recording will have, and the more realistically the michrophone will capture a sound. However, for a capella music, the "best" voice sound is not always the most realistic one. Enhancing the listening experience furthercan be done by

enhancing the high frequencies that represent the consonant sounds with an Equalizer. This gives a clearer picture of the voice, and gives the illusion of the articulation being better. An Equalizer would also allow the user to dampen the frequencies below 60 Hz, which are usually only the source of noise.

4 Implementation

The implementation was divided into three main phases. The first phase included making the different modules, which are standalone packages designed to do any basic tasks concerning its purpose. The second phase is creating the processors that are responsible over what needs to be done for this particular application with the modules, and in what order. The third phase means tieing everything together with a *Graphical User Interface* (GUI), and enabling the user to adjust settings within the processors.

4.1 Modules

Each module is designed to be able to perform not only the tasks that are required for this particular program, but all tasks that are generally considered standard for its purpose. That way, if they would ever be needed for another project, it is easy to reuse them. The project requires 6 different modules. Each module needs implementation that supports both processing of entire files, and partial processing of a specific number of bytes, that allows the user to process parts of a file at a time. This enables things like usage of a feedback system in the GUI, which is of advantage when processing might take a while.

4.1.1 MP3 Encoder

The MP3 Encoder would typically be a module responsible for encoding MP3 files from WAV files, and decoding them. When encoding, there are a number of settings for the new MP3 file to choose, but all except one are extractable from the settings of the WAV file. That setting is the number of kilobits per seconds. This value determines the encoding bitrate, that is how large the final MP3 file will become, and the larger the size, the higher the quality of the audio. For the decoding, no settings whatsoever needs to be done.

4.1.2 Wave Editor

The Wave Editor module is probably the most extensive module, as it is responsible for all the editing of the wave files. A typical wave editor program can do volume change, cutting, pasting, panning, mixing and many other functions, and most of them are going to be used in this application. Each of the functions is very extensive, therefore this module is also the most extensive of all modules.

The volume change function is very basic. The user specifies a change factor between 0 and positive infinity, where 1 is equal to the original volume (ie no change). Then, the value of each sample is simply mutiplied by this factor, and the volume of the resulting sound will be adjusted according to that factor.

The cutting function means not only to exclude a number of samples from the wave file, but also that the wave file size needs to be adjusted in the WAV file header. The input for this function would be the starting time and the ending time, which then has to be converted to sample numbers through the sample

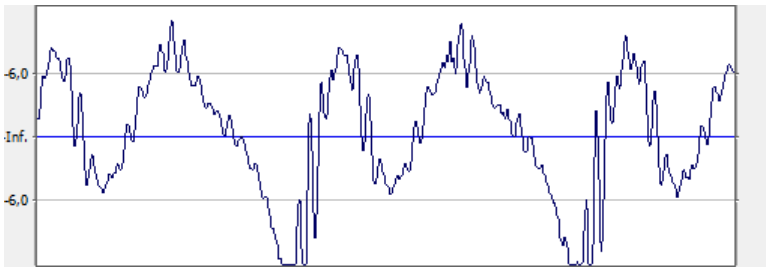

Figure 5: A distorted audio wave.

rate. After calculating the number of samples to cut, the header is changed accordingly, and the samples are removed.

Pasting samples from another audio is of the same principle as cutting. Instead of removing samples, they are added from another WAV file. A requirement for this to work is that the samples to paste must be from a WAV file with the same settings as the target WAV file. Otherwise, the samples will not be compatible with the new WAV file, which will lead to corrupt or twisted audio.

Panning an audio means two things. First of all, the WAV to pan needs to be a Stereo WAV file. If it is not, it needs to be converted to one. Secondly, the setting to this function should be a float value between -1 and 1, which defines in what direction to pan the audio and how much. In the case where this value is less than 0, the right channel volume should be reduced and vice versa. The reduction is in linear proportion to the absolute value of the setting.

The mixing function is slightly more advanced. After having verified that all the WAV files to mix together have the same settings, the length of the final file needs to be set to the longest of the files. Then, precautions are needed to avoid distortion. Distortion is when the amplitude of a sound wave is extended above the possible maximum data value of a sample. This will result in those samples being reduced to the maximum value, hence braking the "flow" of the wave. When there are sudden changes in the flow of a wave file, they present themselves as disturbances, which is called distortion. Figure 5 shows how distortion presents itself visually. When adding two WAV files together, two samples within the allowed sample range might exceed that range if they are added together. Therefore, to make sure that no parts of the audio gets distorted, the volume of each WAV file must be reduced by a factor of 1 divided by the number of WAV files to be mixed together. Doing this results in a certain loss of data precision, but with a high bitrate, this loss is unnoticable.

4.1.3 Equalizer

The equalizer module is a signal processing filter adapted for processing of audio signals. The settings of this module is the filter parameters, which can be any parameters, and as many as desired. The filter is an *Auto-Regressive Moving Average* (ARMA) filter, which allows enhancements and reductions of any desired frequency. The function of the filter is as follows:

$$A(z)Y(z) = C(z)X(z) \tag{1}$$

A represents the AR parameters as a polynomial, and C represents the MA parameters. X is the original sampled WAV input signal, and Y is the output for the filter.

Instead of specifying the parameters for the filter manually in the program, the typical way that a user wants to do it is to specify how the frequency spectrum should look. Therefore, the input for the module is the poles and the zeros as complex numbers, that act as roots for the filter. Those correlate much closer to the frequency spectrum than the actual polynomials.

This module requires two sorts of outputs. First of all, it should obviously be able to filter an input WAV file and return an output file. Secondly, it must also be able to display the frequency spectrum that the set poles and zeros will apply to the wave file, so the user is able to see what the equalizer is causing. The formula for the frequency spectrum is:

$$P(f) = \frac{|C(e^{j2\pi f})|}{|A(e^{j2\pi f})|} \tag{2}$$

In this formula, f denotes a frequency factor between 0 and 0.5, which is the fraction of the wave file's sample rate. According to the *Nyquist theorem*, it is not possible to represent a frequency higher than half the sampling rate, hence, the displayable frequency spectrum ranges from 0 to 0.5 times the sampling rate of the WAV file.

To implement this in an effective way, that allows for generation to be quick, the filter uses quantization of filter parameters. The number of quantization bits for each processing is calculated upon generation, and as many as possible are chosen to assure as little data loss as possible. So since the sample values from the input signal are already integers (and if they are not, they are converted to integers), pure integer multiplication followed by byte switching is the method of generation for this module.

4.1.4 VST Effect Applyer

An audio effect plugin of the VST standard is most often specified as a DLL file. The VST *Application Programming Interface*[6] (API) requires implementation in C++, which means that to be able to implement it in Java, a Cross-Platform implementation is required.

The VST Effect Applyer would typically require a number of VST Effects, along with corresponding settings for each VST Effect. To be able to provide settings for each effect in a simple way, they can be stored in a generated file. That makes it simple to provide only files as input settings to this module. For this

[6] An Application Programming Interface is an interface for programmers to implement for usage of the specific application in their own application. For instance, if the audio program is supposed to be able to use VST plugins, one must implement the VST standard API in the program.

idea to work, a VST Preset Managing function is required to generate these files with settings for the VST Effects. More about that in the GUI section.

The processing of WAV files to add VST effects can be done all at once. The bytes can simply be processed through one effect, and then the next one and so forth, until all effects are added to the bytes. Through this principle, multiple generations upon multiple effects is not required.

4.1.5 Compressor

The compressor module's sole purpose is to add all the track files to one single bundle. The module itself however, accepts any number of files of any file types, and the level of compression is selectable. The filetype used for compression is zip, since that is the standardized compression format in Microsoft Windows. The module can naturally both compress to zip and decompress from zip.

The Java language has standard classes for handing zip files and compression, therefore no research or specification was necessary for handling this task.

4.1.6 FTP Client

An FTP Client should typically be able to upload files, download files, browse folders on the FTP server and edit files within the folders. The FTP protocol is built around four-letter commands that are sent from the client to the server, upon which the server performs different tasks and returns what is requested. For extended security, the SFTP protocol could have been used instead of the regular, somewhat insecure FTP protocol. That would mean that the connection between the server and the client would be encrypted, and no one could snoop the network connection to achieve the transsferred data. However, it is not likely that this would happen in this line of business.

4.2 Processors

A processor is a mid-level component within the program that keeps track of what needs to be done with the files, using the available modules. The processors are, unlike the modules, application-specific, and their sole purpose is to represent one "step" of the generation. The processors keep track of not only what needs to be done, but what file types are required for the input files, and what file types the output files will be. There might also be several settings required for the processors, some of which are forwarded to the modules, and some that are used by the processors themselves for the generation. The processors are also responsible over continuously giving feedback about the generation within the module, which is handled with a listener solution.

Despite the relatively small number of modules, there will be a significantly bigger number of processors. Each module might be used in several processors, and one processor might use more than one module. The available processors and their purpose are listed below, in order of generation.

4.2.1 Mp3 Decoder

This processor uses the Encoder module to decode the backup MP3 files containing each part by itself. For decoding, no specific settings are required. The input file type for this processor is MP3, and the output is WAV. The WAV format will be kept throughout the entire processing phase of the generation, until the final mixes are encoded to MP3's again.

4.2.2 Equalizer

This processor uses the Equalizer module to equalize the audio for a more satisfying audio projection. The settings for the processor are the polar coordinates of the roots, which then will be converted to actual complex numbers and multiplied together to create the polynomial which will be forwarded to the module. This will be beneficial due to the nature of the user interface that will be associated with this processor, where adjustments adjustments can be made to the positions of the roots, rather than the polynomials themselves.

For the user interface, a number of roots of the AR and MA polynomials are placed as poles and zeros in the unit circle, and different sliders adjust the angle and distance from the center. The reason why this method is used is because of the clos relation to the frequency spectrum. Adjusting the distance of one pole will give one bump in the frequency spectrum, and changing the angle will reposition the bump. This makes it very easy to get the equalizer settings of choice.

4.2.3 Channel FX Applyer

This processor is meant to apply the channel-specific VST effects that might be desired, using the VST Effect Applyer module. Channel-specific effects are typically effects such as compressors, that make the part-specific voice recording more consistent in volume, or chorus effects, that give an illusion that there are more than one person singing the part. There might also be a desire to add certain effects to the bass part only, such as an equalizer that enhances the low frequencies. Therefore, support for adding bass effects is also included in this processor.

The processor takes in specifications of directories where the VST plugin files and the preset files are located, which are forwarded to the module. It also takes in directories for bass effects only, which are treated by the processor itself. The processor will commence generation of all WAV files except for the bass files (which are located using file names), using only the effects meant for all files. After that, the bass effects will be added to the WAV files identified as bass parts.

4.2.4 Wave Mixer

This is the processor that is responsible over using the part-specific files and generating all the different mixes. Since there are in total five different kinds

of mixes (full mix, part predominant, part left, part missing and sample), and each kind of mix has its own specific settings, this process requires a lot of settings. Firstly, the volume level of each part should be possible to adjust, so a percentage value of the original volume is selectable. Secondly, the pan of each part within the mixes that work with pan is also adjustable. On top of these two fundamental settings, there are also the mix specific changes. For the part left tracks and the part predominant tracks, there is a volume meter to adjust the volume level of the predominant part, i.e the boost percentage in volume. For the sample, the starting time of the song is something that will be different from song to song, therefore requires adjusting.

Each type of mix is an option. This means that there is a checkbox to choose whether or not the mix should be generated. And obviously, the fewer mixes the program has to generate, the faster the generation.

The process obviously works with the Wave Editor module, and the sequence of effects to apply to the files is always the same. The adjustable parts are the volume levels, and whether or not to perform certain generations at all. All the mixes are generated at the same time, due to the high reusage of waves. Therefore, to do it all at once saves a lot of time.

The first step of the generation of mixes is to create different versions of the part specific tracks with different pans. The pans necessary are 100

4.2.5 Mix FX Applyer

This module is using the Effect Applyer module the second time around, and is intended for mix specific tracks. The difference between this processor and the Channel FX Applyer is that with this effect applyer, effects are applied to the final mixes. For stereo effects such as stereo reverbs and chorus effects, it is a requirement that the mixes with the final pans are used for processing.

As before, the desired VST plugin is a DLL file, and the preset file created in the Preset Manager is pointed out in the file system. All effects will be applied to all mixes.

4.2.6 Mix Masterer

Yet again, this processor uses the Effect Applyer module, but not exclusively. This processor is intended for mastering effects that work with the volume level, such as compressors and maximizers. So aside from the plugins and the presets, this processor has the options to pre-normalize and post-normalize the tracks. These options takes up the volume level to its maximum for each mix without distorting a track, and it is done with the Wave Mixer module. This is a requirement for compressors to function properly, and it is a standard method of mastering any music.

The pre-normalization is done at the same time as applying the effects to save time. However, since it is impossible to know the highest peak of the pre-normalized and compressed audio, the post-normalization is done after the generation of each file. The reason why the normalizations are options is because

some might prefer having a maximizer or a volume booster to handle normalization for them, which might save time.

4.2.7 Mp3 Encoder

This processor uses the MP3 Encoder module to encode the WAV files back to MP3 files, which are more optimal for transmission over the Internet due to their lesser file size. There is one single option for this process, and that is the MP3 bitrate, which is typically any multiple of 32 less than or equal to 320 kilobits per second.

4.2.8 Compressor

The sole purpose of this processor is to use the Compressor module to put all the generated MP3 files into one single bundle. Since the MP3 compression is basically as compressed as can be, any compression rate besides "none" is redundant. The only option for this processor is the file name of the final file, which might be relevant to the method of delivery. The accepted input file type is mp3, and the output is zip.

4.2.9 FTP Uploader

The uploader processor uses the FTP Client module to connect to an FTP host and port with a user name and password, all specified as settings to this processor. The directory in which to put the generated zip is also specified as a setting, so that the processor can navigate to the correct folder and upload the file on its own. Since the file is uploaded to a server, there is no output to this specific processor whatsoever. The accepted input is zip.

4.3 Graphical User Interface

When designing the GUI, the philosophy was to keep it as simple as possible, and do not waste time on any fancy details. However, a certain logic is always needed for the cognitive part of the GUI, to make is as easy as possible to understand without having to read a manual. Figure 6 displays an overview of the entire GUI.

The GUI consists mainly of three different panels. The top panel is the panel to specify the input files to start working with, and load them into the program. It is also responsible over administration of the program settings. The middle panel is where all the generation tasks are handled, i.e where all the settings to each processor is specified. The bottom panel handles the queueing of jobs and when to start the generation, as well as displaying status messages to the user.

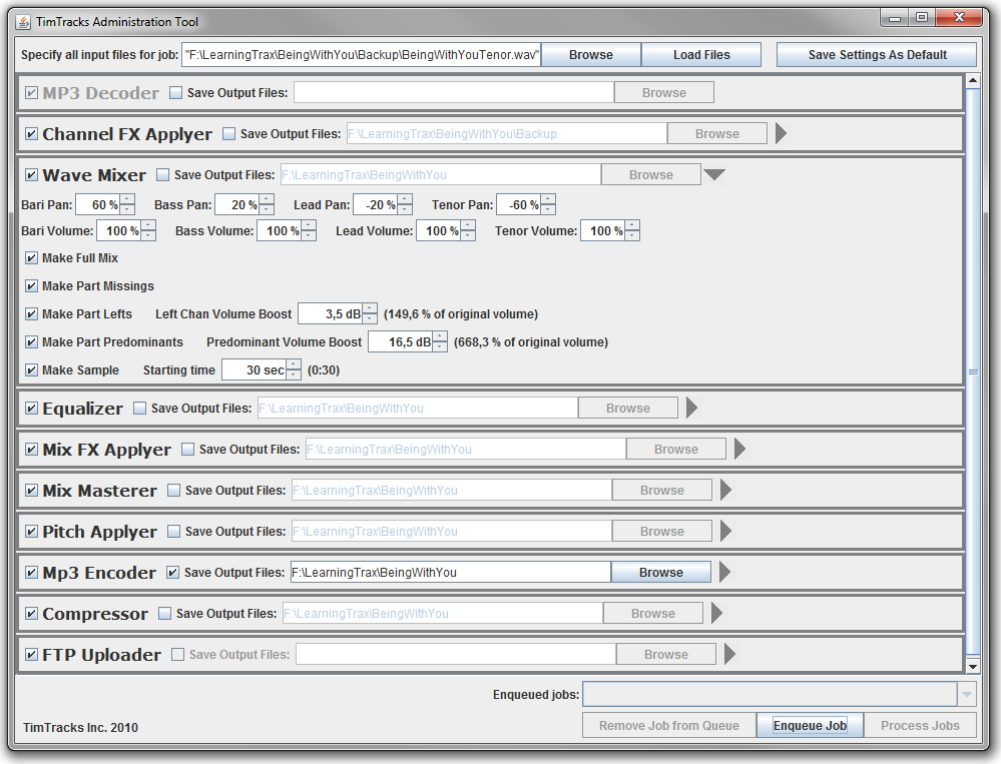

Figure 6: The graphical user interface in its entirety.

4.4 The Tasks

Each processor needs to be connected to the GUI, to allow the user to specify what settings they want for them. These connections are called *tasks*. Each task has its specific settings that can be changed in any way. They can also be enabled or disabled, and they are directly connected to their processor. Since each processor has its specific input and output file type, the tasks will automatically keep track of whether or not the order of generation is possible, and if not, disable themselves. For instance, if one task is disabled, and that task's output is different from its input, then the next task in the order will not have the correct input for the generation, and will therefore be disabled as well. However, if the disabled task has the same input file type as output, then the following tasks will not be affected.

The tasks are named the same as their corresponding processor, and the tasks and the processors have a one-to-one relationship. Since each task (or processor) produces temporary files, the generation can be interrupted at any time, and the last produced files are saved. That is a way to prevent having to do the whole generation again.

4.5 The Feedback System

Since the generation of tracks takes a long time (20-40 minutes for a 4-part song, depending on how fast the computer is), a good feedback system is required. The feedback system has its base in the GUI and establishes communication between itself and each of the processors. That way, the processors themselves report information about where in the generation they are at, and how much they have left.

The feedback will be displayed as a progress bar, a counter and a status message in the middle panel, replacing the task panel as soon as the generation has started. It will also contain a cancellation button that will stop the generation by calling the current processor's cancelling function. When a generation is cancelled, the last created temporary files will be saved, so that the generation can continue from its last point if necessary.

4.6 Settings

Each task has its specific settings, most of which have standard values that get reused from job to job. These standard values might change occasionally, for instance if the user decides to do something different with the audio effects, but when they do, the changes usually apply to all future projects, until further notice. Therefore, the system responsible over saving and loading the settings works with a default setup that loads upon each new project, that is changeable at any time. The goal with this is to keep it as simple as possible, while saving the user as much time as possible.

The only way to administrate the settings is a button in the top right corner that saves the current settings as default settings. When this button is hit, all the current settings will be saved, and loaded as default the next time the program starts.

5 Result/Conclusion

This project was initially divided into three parts: Planning, Investigation and Implementation. The planning stage was required to know what needed to be investigated, and the investigation was necessary to know how to do the implementation. However, in the planning stage, there was already a clear vision of the desired results of the research and a foundation of how the implementation would work. As it turns out, this vision turned out to match the reality and all three stages therefore went according to plan. The testing stage, which is a part of the implementation stage, was very short, and aside from a few minor implementation mistakes, everything worked fine.

Recording a standard 4-part song takes in average 3 hours. Mixing the song and applying necessary pitch correction required to achieve the desired quality takes in average another 2 hours. This is 5 hours in total. The manual process of generating all the mixes with effects, mastering, encoding, compressing and uploading takes in average 1 hour and 15 minutes. With this program, it takes less than 5 minutes to get the generation started, which then takes care of itself. So the working time for one project has been reduced from 6 hours and 15 minutes to 5 hours and 5 minutes, which is a 19% time reduction. This time reduction can be converted to an increase of hourly rate by 23%. This matches the initial expectations of the program well.

5.1 Future Enhancements

If this program were to be expanded with more features, the most appropriate enhancement would be investigating the possibility of dividing the generation between the processor cores, instead of doing everything in one core. Some of the processors could easily divide the work between a couple of threads, although it is uncertain if that is all it takes to use another processor core. It would also take an amount of research to find out how time-consuming each part of the generation is. It is likely that writing to disk is what takes most time, in which case it might not be relevant to speed up the calculations at all. It may also be so, that the calculations and writing to disk takes about the same amount of time, in which case it most definitely will save a lot of time.

To solve the writing to disk problem, one way of adapting the program would be to implement an option to process all the wave files at once. There would be some slight complications, such as certain effects that are not applyable without scanning the whole wave file first, and the fact that some processing has different output formats than others. But these are obstacles that can be solved, and might speed up the processing significantly.

References

[1] Monson H. Hayes, *Statistical Digital Signal Processing and Modeling.* 1st Edition, 1996.

[2] John G. Proakis, Dimitris G. Manolakis, *Digital Signal Processing: Principles, Algorithms, and Applications*. 4th Edition, 2006.

[3] Simon Haykin, *Adaptive Filter Theory*. 4th Edition, 2001.

[4] Karlheinz Brandenburg, *Low Bit Rate Audio Encoding*. Ilmenau Technical University, Germany, 2000.

[5] K. Brandenburg, Marina Bosi, *Overview of MPEG Audio*. Vol. 45, 1997.