H3rtz.stdio

Report #2

Jean "Areas" Bou Raad
Kevin-Brian "KB" N'Diaye
Thanh Lam Nguyen "Velellah"
Yabsira Alemayehu MULAT "Yabs"

Contents

0.1 Introduction

In this second report, we will present the main achievements of our team during this second period. We learned from the first period, fixed major issues while finishing up both the WAV codec and the GUI and started the core of MP3 encoding.

Now that we have better knowledge about audio decoding, we are more aware of the difficulties that are to come.

The first part of our report will be about the extensive rework of the pulse-audio backend and the WAV encoding. While they did their jobs correctly, some issues made us rethink a few aspects. We can move on to the rest of the project with solid foundations.

The second one will bring details on the implemented first and second phase of the MP3 encoding. These parts are almost fully implemented. This part will detail the steps and concepts around the implementation of the MP3 encoding.

Finally, the progress on user interface will be shown. The part showcases the integration between the decoder and the UI, and the work done with the new feature: the playlist.

0.2 PulseAudio back-end rework

0.2.1 Introduction

For the first defense, we provided a rough implementation of a PulseAudio back-end. It can play raw audio on any computer. This part will explain the features added to the previous iteration of this back end. We recommend reading our first report to understand the big picture around this part of our project.

0.2.2 Motivations

Our previous implementation of PulseAudio was unstable for some tasks (e.g: changing the position in an audio file).
Indeed, the audio back-end played corrupted audio if the offset computed was not a multiple of the bit sampling size of the file. Also, if we changed the offset while the player was draining, it would have caused a hard crash. Those instances of bugs are only a few examples, and we will go through most of them in the later parts of this report.
Finally, the best motivation for this rework was better thread safety with the experience earned from the previous work.

0.2.3 Structures

After the first defense, we also realized that the structures linked to our back-end were sometimes not coherent. Hence, we decided to move, add, remove some attributes. Below, you can find some comparisons between the old structures and the new ones:

```
typedef struct pa_player
{
    wav_player *player; // modified: variable objects for backend
    pa_objects *pulseAudio; //modified: static objects
    pa_info *info; // modified: for volume
    state pa_state; // modified: global PulseAudio State
    fileType type; // wav original or mp3
    //pa_info *info; moved into another structure
    //pa_time *utility; deleted, attributes moved to
} pa_player;
```

Figure 1: New structure definition of a PulseAudio player

```
// Per track data, will change over the program's lifetime
typedef struct wav_player
{
    wav *info; // contain header and data pointers for the file played
    file *track; // I/O attributes
    unsigned char *data; // points to the beginning of the audio samples
    pa_stream *stream; // moved: Opaque PulseAudio object to launch operations
    drain *drainer; // new: holds information about the draining status
    pa_time *timing; // new: position in the current file, latency
    playerStatus status; // new: see enum
} wav_player;
```

Figure 2: New structure definition of a *wav_player*

```
// Static PulseAudio objects during the program's lifetime
typedef struct pa_objects
{
    // removed stream object
    pa_context *context;
    pa_threaded_mainloop *loop;
    pa_mainloop_api *api;
    char *sink;
    pa_server_info *server;
} pa_objects;
```

Figure 3: New structure definition of *pa_objects*

In terms of organization, the main change is a rule applied: the dynamic objects go in the *wav_player* structure, and the static ones go in the *pa_objects*.
It improves the quality of life while coding and allows more straightforward incremental updates. The objects are not spread all over the place anymore.

0.2.4 Drain: a particular operation

When a player reaches the end of a track, the program needs to verify that the audio server has played the samples. PulseAudio does not provide a function to get that information. Instead, it provides a method that drains the buffer until it is empty: pa_stream_drain[1]. This operation runs in the background, and the deferred callback runs to send a signal to the main thread to unlock it. The deferred callback runs after completion or error.
This scheme does not work if the drain starts and we try to rewind the audio in the meantime. The thread would freeze and cause a crash because the operation could not complete (no signal sent to unlock).
However, PulseAudio's API's developers knew that it could happen so the callback triggers with an error after a certain amount of time. It also causes a crash because it should not happen.

To fix that problem, we preferred to cancel the operation asynchronously to avoid triggering any assertion that would cause the program to crash.
To do so, we needed a structure to access the operation easily with other data. You can find it below:

```
1 // Object used to track draining process
2 typedef struct drain
3 {
4     DrainStatus state; // enumeration: indicates if there is an
      operation running
5     pa_operation *drain; // the PulseAudio operation pointer to
      cancel
6 } drain;
```

Figure 4: Structure definition of *drain*

The situation is as follows: the GUI in another thread calls the change of offset function in the track. In the meantime, another thread is draining the audio samples from the buffer. It is the process to cancel the operation:

1. Cancel the operation from the other thread using *pa_operation_cancel*, with as reference the pointer in the *drain* structure[2]

2. Once canceled, send a signal to the main thread. It unlocks a refresh concerning the state of the operation.

3. The main thread sees then that the operation is canceled. It will then restore the player to its normal state and unlock the *mainloop* object.

[1] https://freedesktop.org/software/pulseaudio/doxygen/streams.html
[2] See fig. 4

To illustrate these explanations, you can find below a simplified snippet of code:

```
pa_threaded_mainloop_lock(loop);
// starts a draining operation, it is asynchronous
// callbackDrain is the callback function
pa_operation *op = pa_stream_drain(stream, &callbackDrain, loop
);
// sets a pointer to the operation
player->player->drainer->drain = op;
// indicates that a drain is currently running
player->player->drainer->state = DRAIN_ACTIVE;
// we wait for it to be done
pa_operation_state_t state;
while ((state = pa_operation_get_state(op)) !=
PA_OPERATION_DONE)
{
    if (state == PA_OPERATION_CANCELLED)
    {
        // resets the playback to its previous state
        // ... (sets some callbacks)
        player->player->drainer->state = DRAIN_INACTIVE;
        // unlock the mainloop to allow other operations to run
        pa_threaded_mainloop_unlock(pa->loop);
        return;
    }
    // waits for signal from either deferred callback or normal
 operation
    pa_threaded_mainloop_wait(loop);
}
// normal operations continues, unlock the mainloop and return
...
```

Figure 5: Extract of code from *drainStream* function

0.2.5 States

Are you ready? Are you playing? Can you play a file? To avoid bugs, the player must have these answers. For instance, if we are not playing any audio, then there is no stream to pause. Trying to do such an operation will cause a segmentation fault[3].

We partially answered these questions with the first version. But, it showed its limitations when trying to implement parallel states.
This kind of situation happens when the player is playing audio but is also draining to close the stream. We want to have both information.
To hold that information, we have implemented multiple enumerations:

1. The first one indicates the status of PulseAudio objects over their lifespan:

```
typedef enum state
{
    BABY, // before mainloop initialisation
    READY, // before stream initialisation
    ACTIVE, // while playing/paused
    TERMINATED, // killed stream but track in memory
    equivalent to ready
    FINAL, // killed mainloop, cannot go back
} state;

```

Figure 6: Enumeration definition for *pa_objects*

   These states allow easy condition checking for some operation. For instance, if the player is *TERMINATED*, the playing function denies the operation.

2. The second enumeration indicates the states for the dynamic part of our playback objects:

```
typedef enum playerStatus
{
    NOT_READY, // if state is != ACTIVE
    PLAYING, // can be drained
    PAUSED, // corked stream
} playerStatus;

```

Figure 7: Enumeration definition for *wav_player*

   This enumeration enables us to enable or disable some operation if the state does not meet the requirements.

3. Finally, there is an enumeration to indicate if a draining operation is running:

[3] PTSD talking here...

```
typedef enum DrainStatus
{
    DRAIN_INACTIVE, // no operation running
    DRAIN_ACTIVE, // operation running -> playing state
    DRAIN_FINISHED, // can interrupt stream
} DrainStatus;

```

Figure 8: Enumeration definition for *drain*

All that work is all about synchronizing threads and avoiding unwanted operations. And it works now as expected, many bugs were fixed and the overall stability has greatly improved.

0.2.6 Error handling

Since our player interacts with the GUI, it needs a way to provide error messages instead of hard crashes using the error functions from the standard library. To do that, we have reworked many functions to have an integer return type to indicate if an error happened. Following the error code, the GUI will determine a message to display shortly after this defense.

0.2.7 Forwarding/Rewinding

Forwarding/Rewinding was ready for first defense with a few hiccups. To change the position in a file, we need to discard the audio data in the buffer and overwrite it with new data from the new offset in the file. It can be done using the function *pa_stream_flush* function. It flushes the buffer during an asynchronous operation. Then, we change the offset to write as soon as possible data on the buffer. There were two major bugs with our implementation:

1. The audio would get corrupted in some conditions.
2. The program would crash if the player is draining, and we changed the offset.

The first problem was linked to a bug in the offset computation. It was not every time a multiple of the sampling size of the track. We implemented a single line fix:

```
player->player->timing->offset = offset -
                offset % player->player->info->fmt->samplerate;
```

Figure 9: Fix for the first problem

The second problem is what has motivated the rework. We were unable to cancel a draining operation. The change of offset function will cancel any draining operation running if the program calls it at that particular moment.

```
    // If there is currently a drain, we stop it because it will
    fail
    // the draining operations times out after x seconds
    if (player->player->drainer->state == DRAIN_ACTIVE)
    {
        cancelDrain(player);
    }
```

Figure 10: Simple condition checking to solve problem #2

0.3 WAV encoding rework and improvements

0.3.1 Introduction

In order to complete the WAV codec, we needed to make sure the entire codec was put to use. The relevant information lies within $header \rightarrow list \rightarrow infos$, a linked list that stores metadata following the XMP standard so tags like:

1. Artist(s)
2. Copyrights
3. Genre
4. Name
5. Album
6. etc...

The information the user will be able to change is the one displaying on the previous list. Therefore, we wanted to add a feature where information could be added to the file.

0.3.2 First version

The first version only rewrote the same file somewhere else. However, some improvements could be made to the functions. The issues are the following:

1. Inefficient write (large buffer could create some errors)
2. Not-linked to the GUI (still runs in the terminal)
3. Doesn't do much for the user

The wav_encoding functions heavily relied on the write(2) function in order to write both in binary and strings using the same fd. But, as we have seen, write has some cases where it doesn't write as much as needed. In these cases, we would lose some data resulting in either a broken sound or lost metadata which will crash the parser later on.

That's why the main thing to fix on the encoding was to write a rewrite function. This version is a bit different from practicals because we need to know where the error is coming from as soon as possible.

```
1  void rewrite(int fd, const void *buf, size_t count, char *err_msg)
2  {
3      int r;
4      size_t offset = 0;
5      while (offset < count)
6      {
7          r = write(fd, buf + offset, count - offset);
8          if (r == -1)
9          {
10             errx(EXIT_FAILURE, "failed to write %s into fd",
       err_msg);
11         }
12         if (r == 0)
13         {
14             break;
15         }
16         offset += r;
17     }
18 }
```

The new parameter err_msg will allow us to know which particular part of the writing process failed. This method proves very useful when writing data from the linked list, putting the infoID signals where to debug right away.

0.3.3 New version

The linking and UX are part of the second version of the WAV encoding.

When the only thing the codec does is copy-paste, there's no point in using it which renders our work useless.

Additionally, if it's useless to the user it doesn't have its place in the GUI and thus, the final part.

Therefore, we needed a way to link the WAV codec to the GUI.

Adding information

The main reason someone might want to tinker with audio files to begin with is to add new information which is what we will do with the WAV codec.

The user will be able to change and/or add information from the file if they choose to do so.

The information is "limited" to $header \rightarrow list \rightarrow infos$ linked list as it contains most of the metadata.

```
for (; current != NULL; current = current->next)
{
    int art = strncmp(current->infoId, "IART", 4);

    if (art == 0 && strcmp(argv[0], "No changes\n") != 0)
    {
            unsigned int diff = strlen(argv[0]) - strlen(current->
    data);
            current->size += diff;
            header->list->chunk_size += diff;
            header->riff->fileSize += diff;
            current->data = argv[0];
            check[0] = 1;
            continue;
    }
}
```

This template mostly explains how we can update the information from the info list.

The parameters are the list of arguments called argv which the info given by the user. It follows a convention we created which goes as follow:

0 Artist

1 Copyrights

2 Genre

3 Name

4 Album

If no information is given by the user (a '$\backslash n$' using the terminal), the string is given "No changes $\backslash n$" to compare to something more consistent than a null string.

If the element already exists (artists, name, etc...), it's just being replaced after some updates to every relevant size parameter. The check list needs to know whether the string was used.
Otherwise, we have to the create the element ourselves like so:

```
strcpyn((unsigned char *)"IART", (char *)artist->infoId, 4);
artist->size = strlen(argv[0]);
artist->data = argv[0];
artist->next = NULL;
header->riff->fileSize += artist->size;
header->list->chunk_size += artist->size;
current->next = artist;
current = current->next;
```

Mostly the same thing but here we have to use the linked list structure carefully as to not lose any data by simply replacing a node instead of adding one to the list. The relevant size parameters are still being updated and we can write all that new information to a new file.

Linking

How do we link those functions to the GUI then? We create a button to show different fields where the user can change them, press enter and have those changes written into a new file.

Those fields will not be empty if the information already exists. This part was planned for this defense but we are a bit late on the schedule.

0.3.4 Conclusion

After this part is done and implemented into the GUI, it will finally conclude our final version of the WAV codec. What remains is the MP3 encoding which focuses on filtering, deleting and compressing data. Filtering and deleting data while conserving a clear, distinct sound is the next challenge for us.

0.4 MP3 Encoding

0.4.1 Time-Frequency Filterbank

Applying filter to signals has the main advantage of getting rid of the useless aspects of the signal and reduce the size of the signal.

The MP3 standard recommends a high-pass filter to improve the sound quality while removing lower frequencies.

This is the first step of the first phase, where low frequencies are simply being cut out from the signal by default.

0.4.2 Analysis Subband Filter

This filter is a polyphase filter. It turns a PCM signal (from a WAV file) with fs as default sampling frequency into 32 equally spaced subsection by sampling frequencies of fs/32.

This polyphase filter is later completed by the MDCT, those two create a hybrid filterbank.

Implementation

We have to follow the following steps:

1. Divide the audio into 32 samples
2. Create a list X of 512 elements where the first 32 elements are the audio samples then:

$$X_i = X_{i-32}, \text{ for } i = 511 \text{ down to } 32. \tag{1}$$

3. Multiply the each coefficient by an constant array C to create an array Z

4. Create an array Y following this equation:

$$Y_i = \sum_{j=0}^{7} Z_i + 64j, \text{ for } j = 0 \text{ down to } 63. \tag{2}$$

5. Create the 32 subband samples S by matrixing with:

$$S_i = \sum_{k=0}^{63} M_{i,k} * Y_k, \text{ for } i = 0 \text{ to } 31. \tag{3}$$

6. The final equation which will give us the coefficients of the final matrix by the following formula:

$$M_{i,k} = \cos\left[\frac{(2i+1)(k-16)\pi}{64}\right], \text{ for } i = 0 \text{ to } 31. \tag{4}$$

The constant array C draws this function:

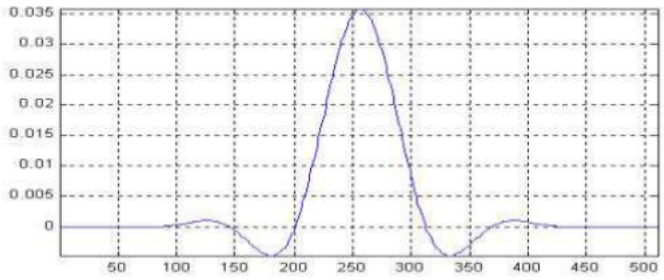

Figure 1: Coefficients from C_i

0.4.3 Psycho-acoustic model

Introduction

The MPEG standard encoding process is lossy. It means that the original audio, in addition to being compressed, is also being modified to reduce its size. The process discards also some data. To determine which data to keep and delete, the standard uses a psycho-acoustic model.

The term psycho-acoustic itself is a part of science that studies the relation between the perception of sound in the human ears and the sound being sent. Here, in our encoding process, a psycho-acoustic model tries to imitate the perception of the sound and select only the sound that the human can hear. For instance, generally speaking, we cannot perceive sounds above 20 kHz.Hence, the sounds above that range will be discarded by the model. But, there's more to it than that. The implementation of model 1 for the MPEG 3 - Layer 3 audio encoding process requires vast algorithms that this part describes.

Definitions

This part of the project is not only about computer science. Hence, we decided to provide some definitions for some keywords.

1. Sound Pressure Level (SPL): represents the relative loudness of a sound. It can be negative or positive. Its unit is the decibel (dB).

2. Masking: Our ears cannot distinguish two sounds that are too close. The sound with the higher sound pressure will mask the other. The model uses them to discard some data.

3. Critical Bandwidth (critical band): the critical band is the band of audio frequencies within which a second tone will interfere with the perception of the first tone by auditory masking[4]. It is the first block to detect the masking effect.

4. Maskers: represents the pressures required to hear a sound at a frequency *f*. In our implementation, it is an array that scans ranges of frequencies. Unit is again dB.

[4] https://www.wikiwand.com/en/Critical_band

Steps in implementation

This small subsection details the global process in which audio samples go through to get a mask. It will be a quick overview. The following parts will explain the notions in depth.

1. Using a Fast Fourier Transform (FFT) algorithm, we transform a time signal into frequencies with relative sound pressure levels. The result is called the SPL array, and the indices are called spectral lines.
2. Determination of the sound pressure level in each subband. This process is not yet complete because it interacts with future parts of the project.
3. Finding tonal and non-tonal components in our spectral lines.
4. Discarding data that cannot be heard by the human ear. They are either too weak or too close to another tonal component.
5. Calculation of the maskers using the relevant data. We compute two of them: one for tonal components and one for noises.
6. Computation of the global masker.
7. Determination of the minimum masking threshold in each subband.
8. Computation of the subband masking ration (SMR).

FFT: getting the sound pressures

As explained above, the first step is about transforming our audio samples from a unit of time into a range of frequencies. First, we must normalize our input:

$$x(n) = \frac{s(n)}{N * 2^{b-1}}$$

Where $s(n)$ is the input array, N the number of samples in our array, and b the number of bits per sample.

Then, with $s(n)$, we compute the FFT, which is as follows:

$$SPL(k) = PN + 10 \log_{(10)} | \sum_{n=0}^{N-1} x(n)h(n) \exp{(-i\frac{2\pi kn}{N})}|^2$$

Where PN is the power normalization term, for MP3, it is required to scale the values with a maximum of 96 dB, $h(n)$ is the Hann window[5].

[5]Please refer to https://www.wikiwand.com/en/Hann_function

In the C language, it translates to this piece of code:

```
// memory allocation here, Hann window is computed also above
long double max = -INFINITY; //macro from standard
for (size_t i = 0; i < HALF; i++)
{
    // FFT transform on NB_SAMPLES points => 1024 for MP3
    // sum allows to retrieve total value faster
    long double complex sum = 0;
    for (size_t j = 0; j < NB_SAMPLES; j++)
    {
        long double complex val = cexpl(-I* ((2*M_PI*i*j)/
NB_SAMPLES));
        long double complex q = window[j]*samples[j]*val;
        sum += q;
    }
    // conversion to real norm
    long double norm = cabsl(sum);
    long double sq = powl(norm, 2);
    long double res;
    // null logarithm is undefined!
    if (sq != 0)
        res = 10*log10l(sq);
    else
        res = sq;
    vector[i] = res;
    if (res > max)
        max = res;
}
// normalization to 96DB max
long double PN = 96.0-max;
addToArrayl(vector, HALF, PN);
```

Figure 1: FFT transform code for MP3 encoding

As you can see, we determine the PN using the result of the left-hand side term. The vector is 512 points long, which is half of the 1024 points used. In reality, there is a symmetry between the part before the middle and after the middle of the array.

These points represent the relative pressure with a maximum level of 96 dB. Moreover, the points scan a range of frequencies, which are $\frac{fs}{N}$ Hz. For instance, if $fs = 44100$ Hz, then each point scans a range of approximately 43.066 Hz. Meaning that at $SPL(112)$ gives the relative pressure of the sounds in the range of frequencies of 4823.44 Hz. These points are also called spectral lines.

Using a Jupyter notebook and some python magic[6], we get the following graph:

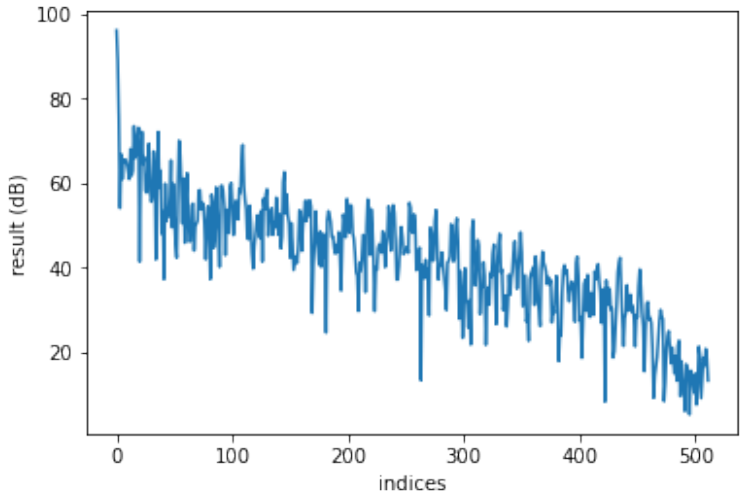

Figure 2: Example of result using an FFT

This result is similar to what one could get with specialized software (Audacity[7] for instance).

SPL determination for subbands

This part is described in the part 3.2.2 of the pdf available at:
http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf
It was only partially implemented because It requires data from external parts of the project not yet implemented.

[6] It's shiny...
[7] https://manual.audacityteam.org/man/plot_spectrum.html

Tonal and non-tonal components

To determine the masker, we need to find the tonal and non-tonal components. We determine them using the following method:

1. Determining local maxima, they satisfy this relation $SPL(k) > SPL(k+1)$ and $SPL(k) > SPL(k-1)$ with $k \in [1; N/2]$
2. Using the maxima, we check that they verify the following condition: $SPL(k) - SPL(k+j) \geq 7$ dB for all j in range:
   (a) if $2 < k < 63$ then for each value of j in $\{-2, 2\}$, the condition must be satisfied.
   (b) if $63 \leq k < 127$ then each of the previous j and the following one in $\{-3, 3\}$, the condition must be satisfied.
   (c) if $127 \leq k < 255$, then j must satisfy the previous conditions and for these values: $\{-6, 6\}$
   (d) if $255 \leq k \leq 500$, finally, we add: $\{-12, 12\}$
3. If the index k satisfies all these conditions, we add it to the list of tonal components.

For this part (and not only this one), we need an implementation of lists. We decided to go for a static list implementation, which follows the following declaration:

```
typedef struct static_list
{
    size_t *data; // stores indexes
    size_t size; // real size in memory
    size_t nb_el; // number of elements
} static_list;
```

Figure 3: Declaration of the *static_list* type

We implemented the following operations:

1. Append including extensions if the array is full.
2. Pop at index i.
3. Contains to search an item in the list.

To find noise components in the samples, we use another method. This method scans through critical bands. These follow an ISO norm and are provided in this pdf (table 3.8):

http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf

Each critical band in the table represents a range of indexes in the table of sound pressures. We sum the pressures of the frequencies not yet treated by the previous function (tonal process).
As a friendly reminder, please remember that the decibel is not a linear scale. To sum it, we must convert the values back to powers, sum them, and compute the resulting sound pressure level[8].
The next step for this algorithm is to compute the center of frequency matched with the index of the noises. It is the result of a sum weighted by the critical bands:

$$center = \frac{\sum_{n=cbi}^{cbi+1} 10^{SPL(n)/10}(z(cb(j)) - i)}{10^{power/10}}$$

Where power is the addition of each sound pressure, cbi and $cbi + 1$ are the ranges of the critical bands in the array of volumes. We round them to get the associated index. z is a function that provides the critical bandwidth associated with frequency f.

Finally, we get two lists with the tonal and non-tonal sounds. Those lists contain indices of the SPL array.

Decimation: remove useless sounds

As previously, our ears are not perfect. We cannot hear frequencies under a threshold in decibel, which varies for each frequency. Also, if two tonal sounds are too close, we only hear one.
We can compute the threshold of hearing thanks to this formula for each frequency f:

$$T(f) = 3.64\ (\frac{f}{1000})^{-0.8} - 6.5\ \exp{(-0.6(\frac{f}{1000} - 3.3)^2)} + 10^{-3}(\frac{f}{1000})^4\ (dB)$$

So for each tonal and non-tonal component, we check if the volume is above that threshold. If it is not the case, we discard it.

For tonal components, we need to compute their relative distance in barks. If $z(t[i+1]) - z(t[i]) < 0.5\ (bark)$ then we remove the one with the lowest volume according to the array of sound pressures. For the sake of simplicity in the formula, $t[index]$ represents the frequency associated with the index requested.

[8] See: https://www.wikiwand.com/en/Decibel

Masking thresholds for non-tonal and tonal components

We have now down-sampled the sound provided to a subset of tonal and non-tonal components. However, in reality, we can afford to have more data than this small subset. For a sampling rate at 44.1 kHz, we can have 130 ranges of frequencies with their masking threshold[9].
Rather than going through the numerous formulas, we will be giving a snippet of code to illustrate the process:

```
    // for the 130 critical bands do...
    for (size_t i = 0; i < crit->size; i++)
    {
        // initializes list of volumes
        masks[i] = initStaticListF();
        // critical band associated to frequency at index i
        long double zi = crit->barks[i];
        for (size_t j = 0; j < t->tonals->nb_el; j++)
        {
            // index of tonal component in the list
            size_t k = t->tonals->data[j];
            // map gives the nearest critical band from the
    frequency given by k
            long double zj = crit->barks[map[k]];
            long double dz = zi-zj;
            if (dz >= -3 && dz <= 8)
            {
                long double avtm = -1.525 - 0.275 * zj - 4.5;
                long double vf = 0;
                if (dz >= -3 && dz < -1)
                    vf = 17 * (dz+1) - (0.4 * SPL[k] + 6);
                else if (dz >= -1 && dz < 0)
                    vf = dz * (0.4 * SPL[k] + 6);
                else if (dz >= 0 && dz < 1)
                    vf = -17*dz;
                else if (dz >= 1 && dz <= 8)
                    vf = -(dz - 1) * (17 - 0.15 * SPL[k]) - 17;
                // append the sum to the list
                appendStaticListF(masks[i], SPL[k]+vf+avtm);
            }
        }
    }
```

Figure 4: Masking threshold computation for tonal components

The resulting array of lists contains the different masking thresholds for each component.

[9]We cannot say it enough, please read http://www.mp3-tech.org/programmer/docs/jacaba_main.pdf at 3.2.6

Global masking threshold

The global masking threshold is the sum of all thresholds (from tonal and non-tonal components) for each critical band. Again, the sum of decibels is not a regular sum. We must merge the values and sum them all together using a function. Our implementation is as follows:

```
    // create a list of long floats
    static_list_f *g_masks = initStaticListF();
    // size = 130, the number of critical bands
    for (size_t i = 0; i < size; i++)
    {
        // merge all values into one array
        // its size is 1 (thr) + nb el in noise and tonal
        long double *values = calloc(1+tonal[i]->nb_el+noises[i]->
    nb_el, sizeof(long double));
        values[0] = table->thresholds[i];
        // copy at offset 1
        memcpy(values+1, tonal[i]->data, tonal[i]->nb_el*sizeof(
    long double));
        // copy at offset 1+nb values in tonal
        memcpy(values+1+tonal[i]->nb_el, noises[i]->data, noises[i
    ]->nb_el*sizeof(long double));

        // computes the dB sum
        long double final = add_db(values, 1+tonal[i]->nb_el+noises
    [i]->nb_el);
        // appends it to the list of global masks
        appendStaticListF(g_masks, final);
        free(values);
    }
    return g_masks;
```

Figure 5: Global masking threshold computation

Minimum masksing threshold

Now, we must go back to our subbands. They are only 32 for this encoding process. To reduce the previous 130 samples, we get the minimum between each subband. In the C language, we get:

```
long double *getMinimumMaskThr(static_list_f *global, size_t *map)
{
    long double *mask = calloc(NB_SUBBANDS, sizeof(long double));
    for (size_t i = 0; i < NB_SUBBANDS; i++)
    {
        // gets the minimum between index first and last of global
    masks thr
        size_t first = map[i*SUB_SIZE];
        size_t last = map[(i+1)*SUB_SIZE -1];
        long double min = INFINITY;
        for (size_t i = first; i <= last && i < global->nb_el; i++)
        {
            if (global->data[i] < min)
                min = global->data[i];
        }
        // default value is 0
        mask[i] = min == INFINITY ? 0 : min;
    }
    return mask;
}
```

Figure 6: Minimum mask value for each subband

Signal to Mask Ratio

We compute the value by removing the array from the previous part to the array given by the SPL determination. However, it is not yet fully implemented. It is the final part of our psycho-acoustic model. It allows us to compute the bit allocation required.

Conclusion

The psycho-acoustic model for the encoding is already quite vast. It is an old model but still illustrates the complexity of the science behind it.

According to Github, this part of the project represents around 1200 lines of code written by at most two person. It does not include the multiple tools and hours of research needed to understand the concepts behind the code. For instance, we coded the equivalent code in python to test our code and create a test suite.

Moreover, the MP3 standard was a proprietary codec[10] for a long time. It means that one cannot find easily on the Internet resources and standards. That led to lengthy research in the depth of Google to find decent documentation to do our implementation. It is a difficulty that might undermine our capacity to finish the encoding process.

[10] https://www.wikiwand.com/fr/MP3 ISO standard: 11172–3, 13818–3

0.5 GUI: Graphic User Interface

0.5.1 Introduction

In the previous defense, we were able to show some basic features of the user interface such as the play-pause buttons, file chooser, progress bar, volume button, and some features which were not fully functioning. For this defense, we were able to implement an information bar and playlist for the audio chosen by the user.

The first section which will be explained is about the information bar. The information bar is a section of the interface that shows different information about the audio chosen by the user. More details will be shown below. The second section which is about the playlist implementation will cover different mechanisms used to implement a basic playlist for a list of songs. It explains how we used pulseaudio and some GTK features to make our user interface dynamic and professional.

After explaining all the features which are implemented for this project, we will go through some ideas that can be done for the final defense and hand in a final product. This will be explained in the progression section to give an idea of where we are in the project and also what the final project project would look like.

Finally, there will be a conclusion part to give a summary about what is done for this defense and for also the remaining for the final defense which is approximately in a month.
[1]

[1] https://developer.gnome.org/gtk3/stable/

0.5.2 Information on songs

In this sub-part of the User Interface, one can find bellow the defined structures that will be used in later shown functions. They will give a better understanding.

```
typedef struct playlist_t
{
    pthread_t *threads;      // check if the task is finished
    wav **w;                 // headers
    file **f;                // files IO
    size_t nb_el;            // nb of elements currently
    size_t size;             // total size of the list in memory
    size_t index;
} playlist_t;

typedef struct UserInterface //To avoid global variables (widgets)
{
    GtkWindow* window;
    GtkButton* play_pause;
    guint ID;
    GtkVolumeButton *volume;
    GtkScale *slider;
    GtkAdjustment *adjustment;
    char *name_chooser;
    GtkListStore *dialog_list_store;
    GtkTreeView *dialog_tree;
    GtkTreeSelection *select;
    GtkLabel *name;
    GtkLabel *genre;
    GtkLabel *album;
    GtkImage *song_image;
    GtkFileChooser *audio_chooser;
} UserInterface;

typedef struct gtk_player   //Main structure to access data
{
    char *filename;
    pa_player *player;
    file *data;
    UserInterface ui;
    playlist_t *playlist;
} gtk_player;
```

In this defense, H3rtz.stdio worked on making a better looking User Interface. For that, the team chose to display the most important information which are :

1. Album
2. Genre
3. Artist4
4. Name

The next sub-subsections will give more details about these features. But first, here is a picture that quickly shows the features of the interface that will be explained soon in the next sub-subsections.

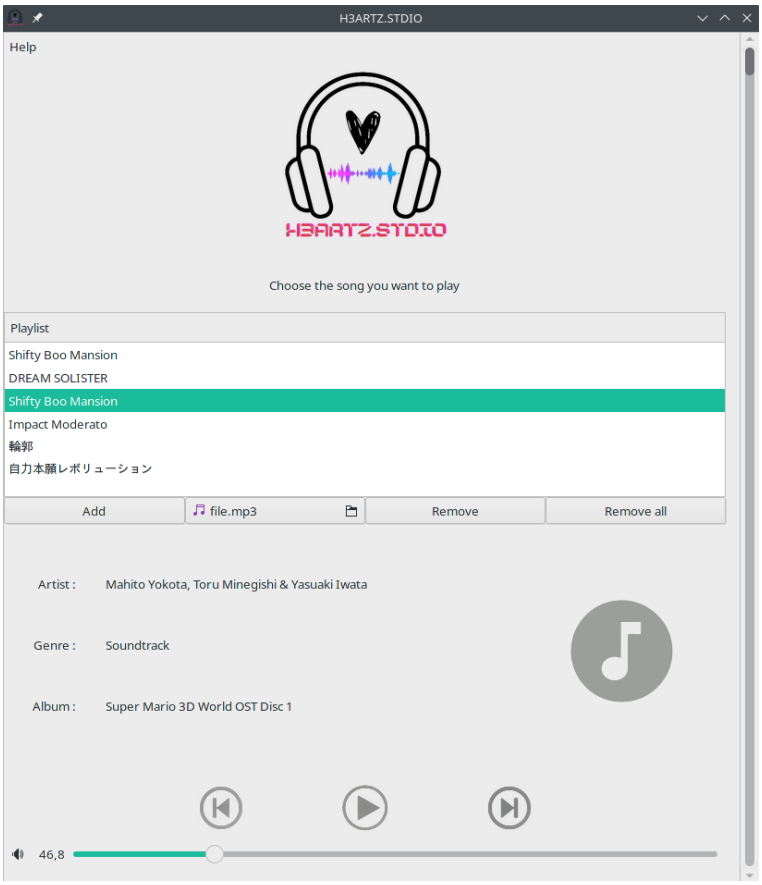

Figure 3: Main function of the album getter

Album of the song

The used data are obtained by parsing with a function that transforms the information of the header into an object of strings. When they are retrieved, to access the album, the code checks whether that specific info is given or not. If it is, the right album will be displayed in the label GtkLabel *album. Otherwise, it will display "Unknown".

```
1 void album(gpointer userdata)
2 {
3     gtk_player *player = userdata;
4     fileInfo *info;
5     info = getFileInfo(player->player->player->info->list);
6     gtk_label_set_text(player->ui.album, info->album ? info->album
      : "Unknown");
7     free(info);
8 }
```

Figure 7: Main function of the album getter

Genre of the song

This part works just like the previous part, but here the genre will be displayed in the GtkLabel *genre.

```
void genre(gpointer userdata)
{
    gtk_player *player = userdata;
    fileInfo *genre;
    genre = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->ui.genre, genre->genre ? genre->
    genre : "Unknown");
    free(genre);
}
```

Figure 8: Main function of the genre getter

Artist of the song

The data is accessed in the same way as the previous part. To access the artist of the playing song, the code checks whether that specific info is given or not. If it is, the artist will be displayed in the label GtkLabel *name. Otherwise, it will display "Unknown".

```
void changeTitle(gpointer userdata)
{
    gtk_player *player = userdata;
    fileInfo *info;
    info = getFileInfo(player->player->player->info->list);
    gtk_label_set_text(player->ui.name, info->artists ? info->
    artists : "Unknown");
    free(info);
}
```

Figure 9: Main function of the artist getter

Name of the song

It is important to precise that the "name of the song" is not the display on the file chooser. It is actually given in the header, and retrieved by the function getFileInfo(), then accessed bellow with f→name. The displayed code bellow checks whether that specific info is given or not. If it is, the song's name will be displayed in the GtkTreeView (later explained in the section 4.3.1). Otherwise, it will display the path to the song.

```
void append(GtkWidget *widget __attribute__((unused)), gpointer
    userdata)
{
    gtk_player *player = userdata;  // Initialization of player
    structure
```

```
4       GtkTreeIter iter;                   // Value that hold the
        addresses of list items
5       gchar *str = player->ui.name_chooser; // Name of the file
        chosen
6       player->ui.dialog_list_store = GTK_LIST_STORE(
7           gtk_tree_view_get_model(player->ui.dialog_tree)); //
        Getting the model from glade
8       tuple data = ParseTrack(str);    // Gets the info about the
        track
9       if (!data.a || !data.b)
10          return;
11      player->playlist->f[player->playlist->nb_el] = data.a;
12      player->playlist->w[player->playlist->nb_el] = data.b;
13      player->playlist->nb_el++;
14      gtk_list_store_append(player->ui.dialog_list_store, &iter); //
        Appending elements to the list
15      wav *w = data.b;
16      fileInfo *f = getFileInfo(w->list);
17      char *entry = f->name ? f->name : str;  // Getting the filename
18      free(f);
19      gtk_list_store_set(player->ui.dialog_list_store, &iter,
        LIST_ITEM, entry, -1);
20      // Sets the value of one or more cells in the row referenced by
         iter
21  }
```

Figure 10: Main function of the song's name getter

0.5.3 Playlist

Before starting directly explaining about the implementation process we will introduce the tools that we used for this feature. Mainly, there are two tools that we used for this feature which are pulseaudio and GTK TreeView widget from GTK. In the previous report we gave brief introduction about pulseaudio and GTK. Hence, we will give the links to get more information about these two tools below[1] [2].

GTK TreeView

The GTK TreeView [3] is a widget used for displaying both trees and lists. It is part of the GTK Container tools hierarchically. To use this widget, we need to define a data structure. By data structure, it means either the lists or trees. For our project and specifically this defense, we used the list data structure. To define this structure in GTK, we used the GTK ListStore [4] which allows us to input a list inside a tree view build by the GTK TreeView. The ListStore structure contains different features like adding rows and columns, and with the integration of GTK TreeView, it makes items of a list clickable. It also helps the user to modify the items of a list.

We used two different structures of lists to take care of adding, removing elements and the stream of the loaded audio files. To be more precise, the GTK part is performing addition and removal of items from the list store while the other list structure concerned with PulseAudio is controlling the stream, moving to the next song or previous song and so on. These processes are running at the same time.

Basically, using the GTK TreeView widget we were able to add and remove different items in a list which is really important for the implementation of a playlist. The functions implemented are as follows: -

Adding item

```
1 void append(GtkWidget *widget __attribute__((unused)), gpointer
    userdata)
2 {
3     gtk_player *player = userdata;    // Initialization of player
    structure
4     GtkTreeIter iter;                 // Value that hold the
    addresses of list items
5     gchar *str = player->ui.name_chooser;    // Name of the file
    chosen
6     player->ui.dialog_list_store = GTK_LIST_STORE(
7         gtk_tree_view_get_model(player->ui.dialog_tree)); //
    Getting the model from glade
```

[1] https://www.freedesktop.org/wiki/Software/PulseAudio/Documentation/
[2] https://developer.gnome.org/gtk3/stable/
[3] https://developer.gnome.org/gtk3/stable/GtkTreeView.html
[4] https://developer.gnome.org/gtk3/stable/GtkListStore.html

```
8       tuple data = ParseTrack(str);  // Gets the info about the track
9       if (!data.a || !data.b)
10          return;
11      player->playlist->f[player->playlist->nb_el] = data.a;
12      player->playlist->w[player->playlist->nb_el] = data.b;
13      player->playlist->nb_el++;
14      gtk_list_store_append(player->ui.dialog_list_store, &iter);  //
         Appending elements to the list
15      wav *w = data.b;
16      fileInfo *f = getFileInfo(w->list);
17      char *entry = f->name ? f->name : str;   // Getting the
        filename
18      free(f);
19      gtk_list_store_set(player->ui.dialog_list_store, &iter,
        LIST_ITEM, entry, -1);
20      // Sets the value of one or more cells in the row referenced by
         iter
21  }
```

Figure 11: Function to add a song to the playlist

Removing item

```
1   void remove_item(GtkWidget *widget __attribute__((unused)),
        gpointer userdata)
2   {
3       gtk_player *player = userdata;   // Initialization of player
        structure
4       GtkTreeIter iter;                // Value that hold the
        addresses of list items
5       GtkTreeModel *model;
6
7       model = gtk_tree_view_get_model(player->ui.dialog_tree);  //
        Getting the model from glade
8       player->ui.select = gtk_tree_view_get_selection(player->ui.
        dialog_tree);   // Getting the GTK TreeSelection from glade
9
10      if (gtk_tree_model_get_iter_first(model, &iter) == FALSE) //
        Checking if the list is empty
11          return;
12
13      gboolean found = gtk_tree_selection_get_selected(player->ui.
        select,
14                                                       &model, &iter)
        ;
15      if (!found)
16          return;
17      GtkTreeIter iter_bis;
18      if (gtk_tree_model_get_iter_first(model, &iter_bis) == FALSE)
19          return;
20      gchar *key;
21      gtk_tree_model_get(model, &iter, LIST_ITEM, &key, -1);
22      ssize_t l = findIndex(player, key); // Getting the index of the
         songs in the list
23      if (l == -1)
24          return;
```

```
25      size_t i = (size_t)l;
26      if (i >= player->playlist->nb_el)
27          return;
28      if (i == player->playlist->index)
29      {
30          Pause(player->player); // Pause the audio
31          terminateStream(player->player);  // Terminate the stream
32      }
33      removeTrackAtIndex(player->playlist, i); // Remove from the
        PulseAudio list structure
34      gtk_list_store_remove(player->ui.dialog_list_store, &iter); //
        Remove from the Gtk ListStore
35  }
```

Figure 12: Function to remove a song from the playlist

```
1   void remove_all(GtkWidget *widget __attribute__((unused)), gpointer
        userdata)
2   {
3       gtk_player *player = userdata; // Initialization of player
        structure
4       GtkTreeModel *model;
5       GtkTreeIter iter; // Value that hold the addresses of list
        items
6
7       model = gtk_tree_view_get_model(player->ui.dialog_tree);  //
        Getting the model from glade
8
9       if (gtk_tree_model_get_iter_first(model, &iter) == FALSE)
10      {
11          return;
12      }
13
14      gtk_list_store_clear(player->ui.dialog_list_store); // Remove
        from GTK ListStore
15      if (player->player->pa_state == ACTIVE)
16      {
17          Pause(player->player);
18          terminateStream(player->player);
19      }
20      cleanPlaylist(player->playlist); // Remove from the PulseAudio
        list structure
21  }
```

Figure 13: Function remove all song from the playlist

Playlist Design samples

In this project we designed two layouts on how to show the playlist. The image samples are below :

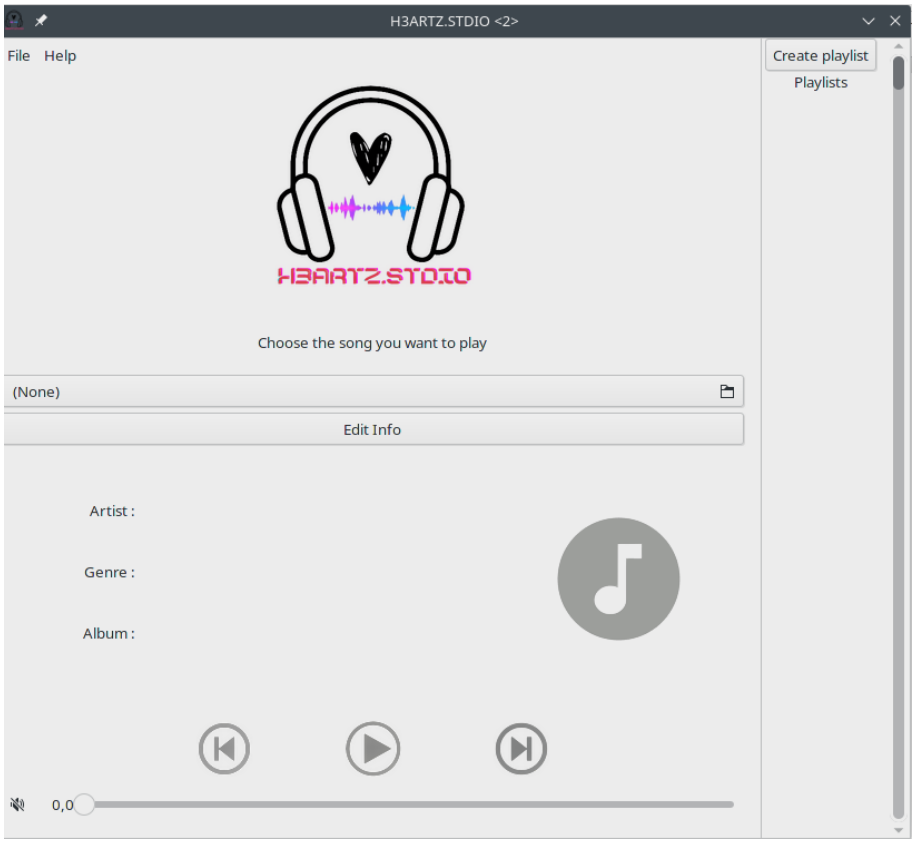

Figure 15: First version of the playlist's display

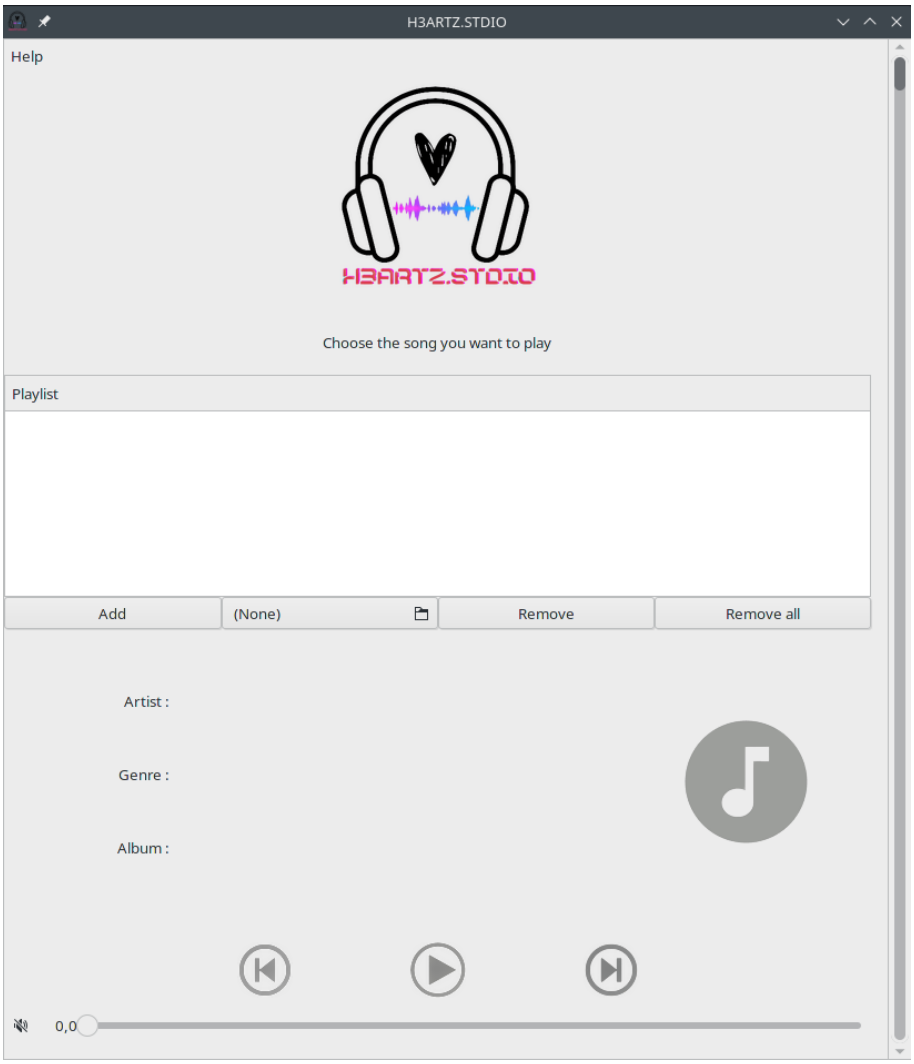

Figure 16: Second version of the playlist's display

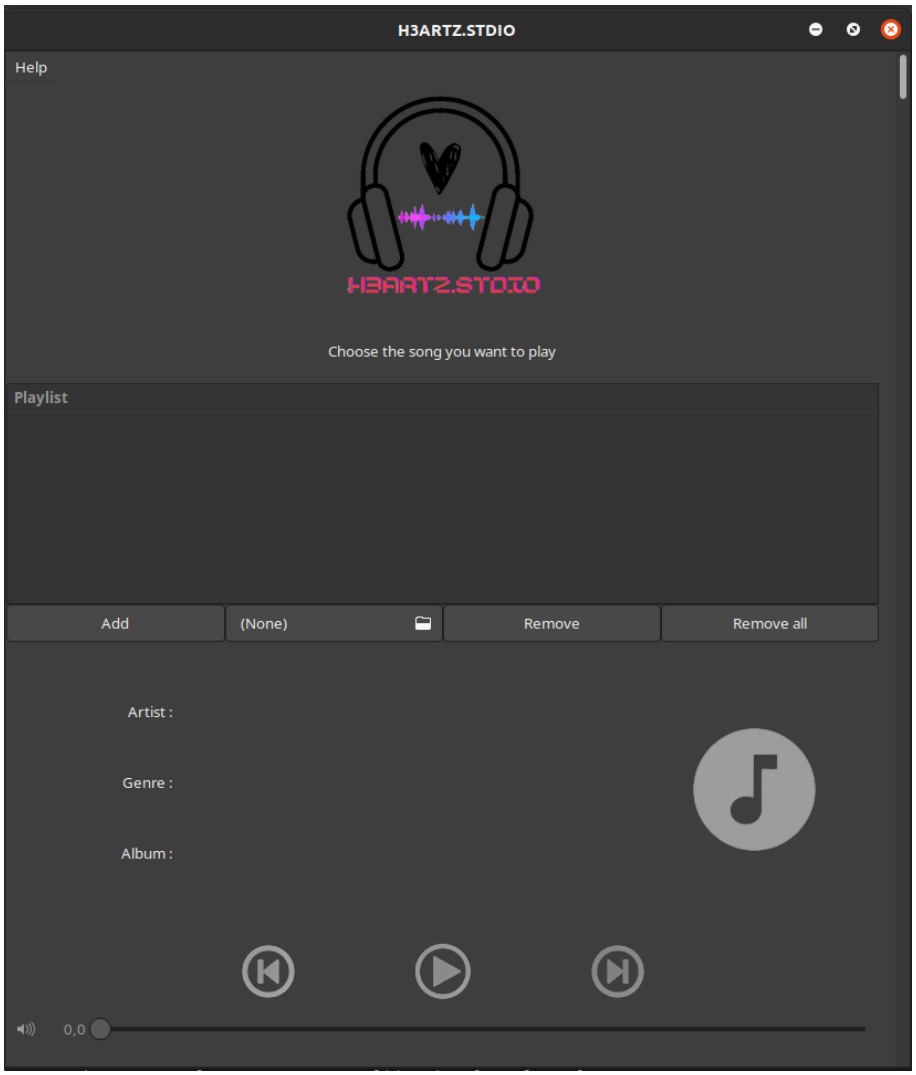

Figure 17: Second version on Ubuntu's dark mode

After a discussion upon the aesthetics and functionality of where the playlist should be put, we decided to use what you observe in the second sample. For this defense we're creating a playlist of songs even for a single song because it's more manageable which is also the reason why we picked the second sample. Furthermore, it is noticeable that there is a background color change in these three samples which comes from the theme that user uses. On Ubuntu, if one uses a dark theme, then the our application would look like the second sample, otherwise the first or second one is the default.

0.5.4 Conclusion

For this defense, H3rtz.stdio's main objective was to implement a playlist display for the user interface.

At first, it seemed complicated to implement it because there was many ways to implement a playlist, and also multiple elements that allows creating a playlist display. As usual, there are debates, votes, and reflections on how that should be displayed and implemented. Thanks to the team and a set objective, the user interface looks more attractive and complete.

For the next defense, the objectives for the interface are :

1. Feature to change song's information using a GtkDialogBox.
2. Try to change the file chooser such that it only displays wav and mp3 files while browsing files.
3. Display the playing song's image on the interface.
4. Feature "Help" button that opens the project's website.

And of course, once all these features are done, adding more features and improving the interface's aesthetic and backends will be realised.

0.6 Assignment tabular

We will assign a letter to every member to make our schedule more readable

1. KB : Kevin-Brian
2. J : Jean
3. L : Lam
4. Y : Yabs

Tasks	KB	J	L	Y
Multi-threading	×	×	×	×
Audio formats	×	×		
GUI			×	×
Website		×	+	

Cross : (×) Person in charge and Plus : (+) Assistant.
As you can see multi-threading will involve everybody.

0.7 Progression

Now, we need to consider how we are going to handle our time based on the three main deadlines :

1. The 1st presentation (March 29th - 2nd April, 2021)
2. The 2nd presentation (3rd - 7th May, 2021)
3. The 3rd presentation (14th - 18th June, 2021)

We made a tabular to make it easy to read. The 1st tabular below is the progression goals that we set from the beginning of the project :

Tasks	1st	2nd	3rd
Multi-threading	30	70	100
Audio encode - WAV	30	100	100
Audio decode - WAV	80	100	100
Audio encode - MP3	0	60	100
Audio decode - MP3	30	80	100
User Interface	40	70	100
Conversion Support	0	40	100
Website	100	100	100

And this is our actual progression on the project :

Tasks	1st	2nd	3rd
Multi-threading	75	90	100
Audio encode - WAV	70	100	100
Audio decode - WAV	80	100	100
Audio encode - MP3	0	35	100
Audio decode - MP3 (GST)	75	90	100
User Interface	50	70	100
Conversion Support	0	40	100
Website	100	100	100

0.8 Conclusion

For the second defense, our team kept the pace and hit major milestones with the User Interface. The playlist is functional and the entire interface's backend has been reworked.

The WAV codec is now over, the interface is entering its final stages so now we can dedicate the final phase to MP3 encoding. The pace will no doubt increase as the final deadline gets closer and closer.

Finally, we were not able to be on time with the MP3 encoding but if the entire group collaborates, we might be able to finish everything on time.

Thank you.