

Object Oriented Programming

Final Coursework

Ahmed Bahaj

October 2021 Semester

Student Number : 200484884

Screenshot of my application

.

**R1 : The application should contain all the basic functionality shown in class**

**R1A : Can load audio files into audio players**

There are 3 ways you can essentially load an audio file into the deck,

- 1- Using the load button : It's the most basic way and has been implemented in classes.

```
29 id DJAudioPlayer::loadURL(URL audioURL)
30
31 auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
32 if (reader != nullptr) // good file!
33 {
34     std::unique_ptr<AudioFormatReaderSource> newSource (new AudioFormatReaderSource (reader, true));
35
36     juce::String total_path = audioURL.getDomain() + "/" + audioURL.getSubPath();
37     transportSource.setSource (newSource.get(), 0, nullptr, reader->sampleRate);
38     readerSource.reset (newSource.release());
39 }
40
```

Function that loads files found in DJAudioPlayer.cpp

This function gets called when the listener of the load button listens to a click, it passes this function onto the player where the button is clicked.

- 2- Drag and drop : Also, has been implemented in classes

```
162 bool DeckGUI::isInterestedInFileDrag (const StringArray &files)
163 {
164     return true;
165 }
166
167 void DeckGUI::filesDropped (const StringArray &files, int x, int y)
168 {
169     if (files.size() == 1)
170     {
171         player->loadURL(URL{File{files[0]}});
172         waveformDisplay->loadURL(URL{ File{files[0]} });
173     }
174 }
175
```

Illustration of isInterestedInFileDrag and filesDropped found in DeckGUI.cpp

- 3- Load from the library : It's the third and probably the more preferable one for me as a user, I created two buttons in each row in the playlist where you can control in which deck you want the audio to be played.

```
124 void PlaylistComponent::buttonClicked(Button* button)
125 {
126     if (true)
127     {
128         std::string idStr = button->getComponentID().toString();
129         int id = std::stoi(idStr);
130         int trackIndex = std::stoi(idStr.substr(1));
131
132         if (idStr.at(0) == '3') {
133             player1->loadURL(URL{ trackTitles[trackIndex] });
134             waveformDisplay1->loadURL(URL{ trackTitles[trackIndex] });
135         }
136         if (idStr.at(0) == '4') {
137             player2->loadURL(URL{ trackTitles[trackIndex] });
138             waveformDisplay2->loadURL(URL{ trackTitles[trackIndex] });
139         }
140         if (idStr.at(0) == '5') {
141             if (trackIndex <= trackTitles.size() - 1 && trackIndex <= trackTitlesNames.size() - 1) {
142                 trackTitles.erase(trackTitles.begin() + trackIndex);
143                 trackTitlesNames.erase(trackTitlesNames.begin() + trackIndex);
144                 tableComponent.updateContent();
145             }
146         }
147     }
148 }
149 }
```

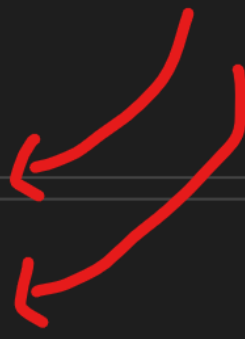


Illustration found in PlaylistComponent.cpp of R1A

## R1B : Can play two or more tracks

In my application, I have 2 players or 2 GUIs that each play their music, I've added them both to mixerSource using .addInputSource, this gives me the flexibility as I can add a third player and add it to the mixerSource which allows me to play 3 records at the same time.

```
40 //=====
41 void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
42 {
43     player1.prepareToPlay(samplesPerBlockExpected,
44         sampleRate);
45
46     player2.prepareToPlay(samplesPerBlockExpected,
47         sampleRate);
48
49     mixerSource.prepareToPlay(samplesPerBlockExpected,
50         sampleRate);
51
52     mixerSource.addInputSource(&player1, false);
53     mixerSource.addInputSource(&player2, false);
54
55 }
56 void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
57 {
58     mixerSource.getNextAudioBlock(bufferToFill);
59 }
60
61 void MainComponent::releaseResources()
62 {
63     // This will be called when the audio device stops, or when it is being
64     // restarted due to a setting change.
65
66     // For more details, see the help for AudioProcessor::releaseResources()
67     player1.releaseResources();
68     player2.releaseResources();
69     mixerSource.releaseResources();
70 }
71
```

Illustration code for R1B found in MainComponent.cpp

### R1C : Can mix the tracks by varying each of their volumes

As I mentioned earlier, my application have 2 players, each player have their own set of buttons and sliders, one of the sliders is specified for the volume, the application read the slider's value and passes it to setGain of the respective player.

This way each player's volume can be changed without affecting the other.

```
123 void DeckGUI::sliderValueChanged (Slider *slider)
124 {
125     if (slider == &volSlider)
126     {
127         player->setGain(slider->getValue());
128     }
129
130     if (slider == &speedSlider)
131     {
132         player->setSpeed(slider->getValue());
133     }
134
135     if (slider == &posSlider)
136     {
137         player->setPositionRelative(slider->getValue());
138     }
139 }
140
```

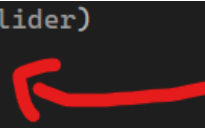


Illustration code for R1C found in DeckGUI.cpp

```
41 void DJAudioPlayer::setGain(double gain)
42 {
43     if (gain < 0 || gain > 1.0)
44     {
45         std::cout << "gain should be between 0 and 1" << std::endl;
46     }
47     else {
48         transportSource.setGain(gain);
49     }
50 }
51
```

Illustration code for R1C (setGain) found in DJAudioPlayer.cpp

## R1D : Can speed up and slow down the tracks

Similar to R1C, A second slider controlling the speed of the audio is Used and Is also found in DeckGUI.cpp, the slider Is controlling the double int 'ratio' and Is passed into resampleSource.setResamplingRatio to control the speed of the audio, It starts by calling the player's setSpeed with the slider's value which represents the 'ratio'

```
123 void DeckGUI::sliderValueChanged (Slider *slider)
124 {
125     if (slider == &volSlider)
126     {
127         player->setGain(slider->getValue());
128     }
129
130     if (slider == &speedSlider)
131     {
132         player->setSpeed(slider->getValue());
133     }
134
135     if (slider == &posSlider)
136     {
137         player->setPositionRelative(slider->getValue());
138     }
139 }
140
```




Illustration code for R1D found in DeckGUI.cpp

```
52 void DJAudioPlayer::setSpeed(double ratio)
53 {
54     if (ratio < 0 || ratio > 100.0)
55     {
56         std::cout << "ratio should be between 0 and 100" << std::endl;
57     }
58     else {
59         resampleSource.setResamplingRatio(ratio);
60     }
61 }
62
```

Illustration code for R1D (setSpeed) found in DJAudioPlayer.cpp

**R2 : Implementation of a custom deck control component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/start.**

**R2A : Component has custom graphics implemented in a paint function**

I used a simple stop/start app with a variety of pink and magenta colors in my app.

In addition to that, I made a forward and a rewind buttons, In my opinion they are the most used tools with audios and videos, making them was so simple, I adjusted transportSource using .setPosition.

Making these buttons was almost similar in making the 3 slider (speed, position, volume)

```
132     if (button == &rewindButton)
133     {
134         player->rewind();
135     }
136
137     if (button == &forwardButton)
138     {
139         player->forward();
140     }
141 }
142
```

Illustration code found in DeckGUI.cpp

```
100 void DJAudioPlayer::rewind()
101 {
102     // if after move is less than 0 or not for short audios
103     if (transportSource.getCurrentPosition() - 1 > 0)
104     {
105         transportSource.setPosition(transportSource.getCurrentPosition() - 1.5);
106     }
107 }
108
109 // Moving the playback forward
110 void DJAudioPlayer::forward()
111 {
112     //Calculating the last position on the transportSource - end of the track
113     double last_pos{ transportSource.getLengthInSeconds() };
114
115     // This prevents moving if the audio is almost finished
116     if (transportSource.getCurrentPosition() + 0.5 != last_pos || transportSource.getCurrentPosition() + 0.5 > last_pos)
117     {
118         transportSource.setPosition(transportSource.getCurrentPosition() + 1.5);
119     }
120 }
121
```

Code for making the rewind and forward actions found in DJAudioPlayer.cpp



## **R2B : Component enables the user to control the playback of a deck somehow**

As I mentioned in R2A, I created a rewind and a forward button, this is how they both work,

For the rewind action :

- 1- The function is called when the rewind button '<<' is on, it calls the player's `rewind()`; that exists in `DJAudioPlayer.cpp`
- 2- The rewind function in `DJAudioPlayer.cpp` starts with checking if the current position of the music minus 1 is more than 0, this prevents any weird work if the audio was extremely short.
- 3- If it's more than 0 then we decrement 1.5 from the audio's current position

For the forward action

- 1- 1- The function is called when the forward button '>>' is on, it calls the player's `forward()`; that exists in `DJAudioPlayer.cpp`
- 2- We begin by getting the audio's length, this will become useful in step 3
- 3- We start by checking if the audio's current position plus 0.5 does not equal the audio's length or if it is more than the audio's length

Tip : Audio's length is the same as the last second

- 4- If one of these conditions or both is true we increment the audio's current position by 1.5
- 5- The reason behind the if statements is to check if the audio is finished or almost finished.

**R3 : implementation of a music library component which allows the user to manage their music library**

**R3A : Component allows the user to add files to their library**

For this extension, dragging and dropping an audio file into the playlist is the only way to add to the user's library.

You can load an audio file into the deck but you need to drag it and drop it into the library.

```
151 bool PlaylistComponent::isInterestedInFileDrag(const StringArray& files)
152 {
153     return true;
154 }
155
156 void PlaylistComponent::filesDropped(const StringArray& files, int x, int y) {
157     if (files.size() >= 1)
158     {
159         for (int i = 0; i < files.size(); i++) {
160             trackTitlesNames.push_back(File{ files[i] }.getFileNameWithoutExtension());
161             trackTitles.push_back(URL{ File{files[i]} });
162             juce::String urljuce = URL{ File{files[i]} }.toString(false);
163             std::string urlstdt = urljuce.toStdString();
164         }
165         tableComponent.updateContent();
166     }
167 }
```

Illustration found in PlaylistComponent.cpp

When a file is dropped, it is pushed into trackTitlesNames which is a vector of juce::String, they are also pushed into other vectors for the usage of other tools.

### R3B : Component parses and displays meta data such as filename and song length

I easily could Implement the file's name into the playlist. However, I found difficulties implementing the audio's length and could not do it unfortunately.

After dragging the file into the playlist we push it to trackTitles, moreover, we push it to another vector called trackTitlesNames that uses the function `.getFileNameWithoutExtension` to get the audio's name.

Then we use the `juce::TableListBox` `paintCell` function to write the audio's name into the playlist, see below.

```
151 bool PlaylistComponent::isInterestedInFileDrag(const StringArray& files)
152 {
153     return true;
154 }
155
156 void PlaylistComponent::filesDropped(const StringArray& files, int x, int y) {
157     if (files.size() >= 1)
158     {
159         for (int i = 0; i < files.size(); i++) {
160             trackTitlesNames.push_back(File{ files[i] }.getFileNameWithoutExtension());
161             trackTitles.push_back(URL{ File{files[i]} });
162             juce::String urljuce = URL{ File{files[i]} }.toString(false);
163             std::string urlstdt = urljuce.toStdString();
164         }
165         tableComponent.updateContent();
166     }
167 }
```

Illustration found in PlaylistComponent.cpp

```
73 void PlaylistComponent::paintCell(Graphics& g, int rowNumber,
74                                     int columnId, int width, int height,
75                                     bool rowIsSelected)
76 {
77     g.drawText(trackTitlesNames[rowNumber],
78               2,
79               0,
80               width - 4,
81               height,
82               Justification::centredLeft,
83               true);
84 }
85
```

paintCell code found in PlaylistComponent.cpp

### R3C : Component allows the user to search for files

The searching algorithm is easy and effective, first we check if the input is empty, if true then we deselect all rows meaning the searching engine is not being used, if it's not empty this where work starts :

- 1- A for loop is looping through all tracks using the trackTitlesNames vector that we used earlier.
- 2- The algorithm then checks if any of the trackTitlesNames contain the characters in the input.
- 3- If any similarities found then we select the rows and paint them with another color showing that they are different.

```
168
169 | PlaylistComponent::searching(String inputtext)
170 |
171 | if (inputtext == "") {
172 |     tableComponent.deselectAllRows();
173 | }
174 | else {
175 |     for (int i = 0; i < trackTitlesNames.size(); i++) {
176 |         if (trackTitlesNames[i].contains(inputtext)) {
177 |             tableComponent.selectRow(i, false, false);
178 |         }
179 |     }
180 | }
181 |
182 |
183 |
184 |
```

Illustration found in PlaylistComponent.cpp

```
23 | addAndMakeVisible(searchingTextBox);
24 |
25 | searchingTextBox.setTextToShowWhenEmpty("Search for music", Colour::fromRGB(127, 82, 90));
26 | searchingTextBox.onReturnKey = [this] {searching(searchingTextBox.getText()); };
27 | tableComponent.setMultipleSelectionEnabled(true);
28 |
```

Illustration found in PlaylistComponent.cpp

### **R3D : Component allows the user to load files from the library into a deck**

This was a fairly easy feature to make and also important since it's one of the 3 ways to load an audio into the deck, after adding an audio to the playlist you get 3 buttons,

- 1- Play to deck 1
- 2- Play to deck 2
- 3- Remove from playlist

We are currently discussing 1 and 2, We started by creating the buttons in 'PlaylistComponent::refreshComponentForCell' and each button has a columnId assigned to it and a componentId which we will use shortly.

Now we need to create the listeners and the actions for the button, Similar to deckGUI we use buttonClicked function to set the listeners.

We go on to create if statements to decide which button is being pressed, columnId helps us specify which button is being pressed, what comes after is similar to the loadButton functionality in deckGUI, we pass on the audio's URL to the player and the waveformDisplay.

Now it's DJAudioPlayer's job to make the sliders and the upper buttons function properly.

### **R3E : The music library persists so that it is restored when the user exits then restarts the application**

**NOT DONE**

#### **R4 : Implementation of a complete custom GUI**

##### **R4A : GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls**

To be honest I think I did fair job in this one, I worked really hard to choose the right colors and make them look as best as it can look, I used a variety of pink and purple with other darkish colors, I believe that sometimes music represents the darker colors.

Moreover I changed fonts' sizes and their colors.

In addition, I changed the sliders to make them look rotary instead of a plain simple slider, this makes the app look more advanced/professional.

##### **R4B : GUI layout includes the custom component from R2**

They are still showing and they look a bit similar to what has been shown in the classes, the 3 sliders and the 3 buttons (save, stop, load) still exist, I may have changed their positions and their sizes, moreover the waveform display still exist with a slight change of color and rectangle's size.

##### **R4C : GUI layout includes the music library component from R3**

The search box and the playlist are in the bottom side of the app.