CM2005: OBJECT ORIENTED PROGRAMMING

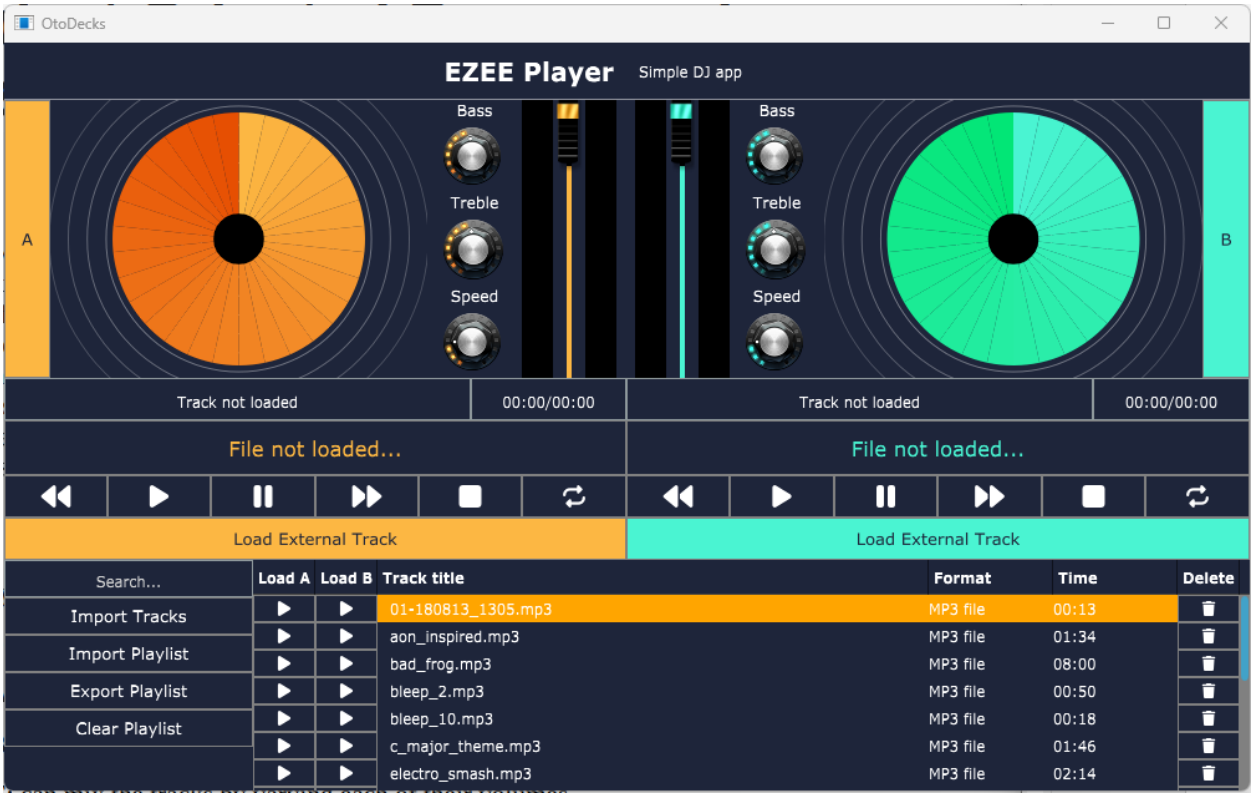
DJ Application Project

Student Name: Shudhanshu Gunjal

Student Number: 190321572

Introduction of DJ Application (EZEE Player)

EZEE Player is a basic DJ application that can play two music tracks simultaneously and allows user to control their “Bass”, “Treble”, “Speed”, and “Volume”. The player contains usual player buttons like play, stop, pause, backward, forward, and loop buttons. This player contains a playlist component where the user can load the track to Deck “A” or “B”, delete the track from the playlist, and on the left side, there are action section of the playlist where the user can search, import, export, clear the playlist.



Component structure:

[1]	MainComponent	Initializes the application.
[2]	DJAudioPlayer	This component plays the track as per the control adjustment
[3]	WaveformDisplay	Generates the waveform of the loaded track
[4]	PlaylistComponent	It is a playlist with Load (A/B), Delete and Actions feature
[5]	DeckGUI	This component handles the GUI of the Deck players

[6]	DiskArt	This component handles rotating disk GUI
[7]	Header	Header contains the DJ player title and tagline which is placed at the top
[8]	LevelMeter	Level meter handles the GUI of level meters according to volume dB
[9]	MyLookAndFeel	This component customizes the look and feel of the components
[10]	Utilities	This component contains helper functions.

Requirements:

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

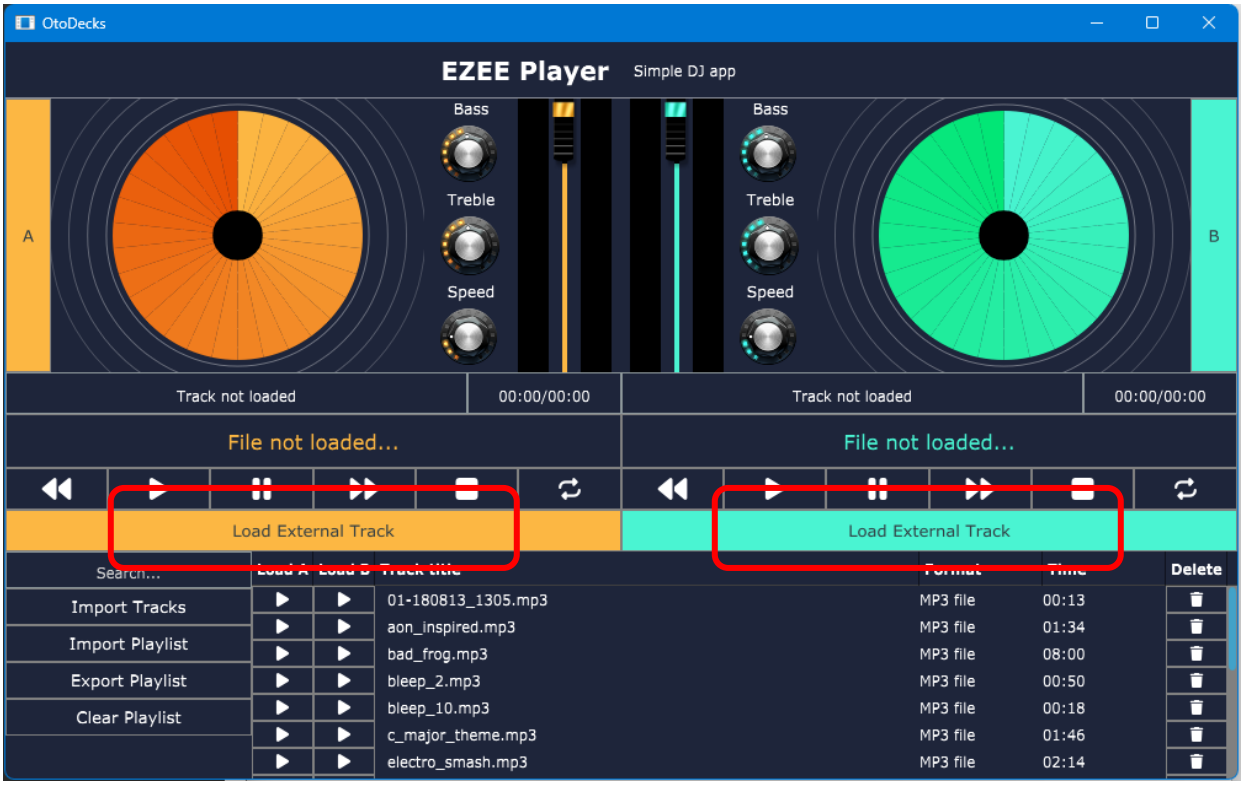
R1A: can load audio files into audio players

There are three methods to load track inside the deck

[1] Using the “Load External Track” button user can load external track directly to the decks
[2] Using “Drag and drop” user can load the track directly to the deck
[3] Using the playlist’s “Load A” or “Load B” Buttons

Let’s go through them one by one

[1] Using the “Load External Track” button



From the red highlighted rectangles, we can see there are two “Load External Track” buttons one is for deck A and one is for deck B. When the user clicks on the button it opens a FileChooser window and after selecting a valid music file it gets loaded in the specific deck using the loadTrackToDeck function.

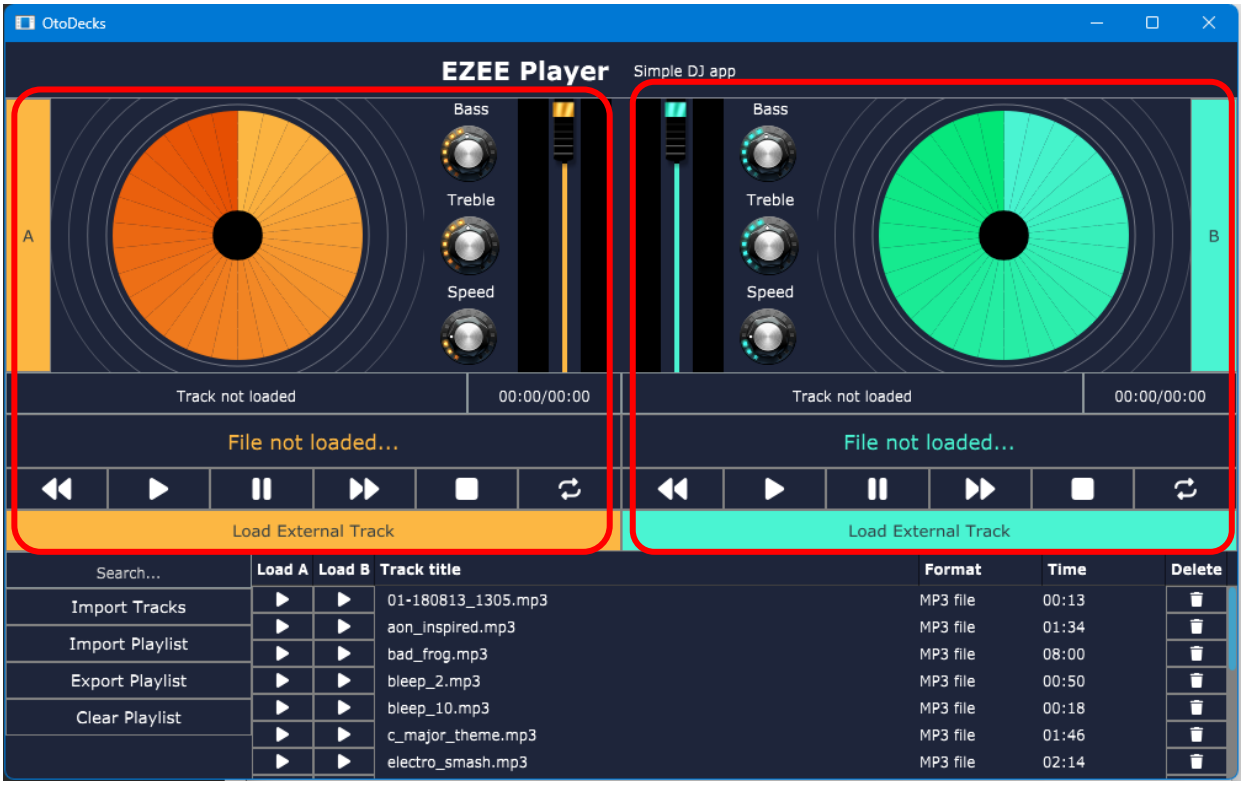
```
290     if (button == &loadButton)
291     {
292         auto fileChooserFlags =
293             FileBrowserComponent::canSelectFiles;
294         fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)->void
295         {
296             player->loadURL(URL{ localFile: chooser.getResult() });
297             waveformDisplay.loadURL(URL{ localFile: chooser.getResult() });
298         });
299         diskArt.setRotationSpeed(0);
300     }
```

loadButton click event - DeckGUI.cpp

```
354 void DeckGUI::loadTrackToDeck(File file)
355 {
356     player->loadURL(URL{ file });
357     waveformDisplay.loadURL(URL{ file });
358     trackTitleTxt.setText(newText: URL::removeEscapeChars(player->getTrackDetails()[0]));
359     trackDurationTxt.setText(player->getTrackDetails()[1]);
360     player->start();
361 
362     // Start disk rotation
363     if (player->isPlaying()) {
364         diskArt.setRotationSpeed(speedSlider.getValue());
365     }
366 }
```

Loading the tracks to the deck - DJAudioPlayer.cpp

[2] Using “Drag and drop”

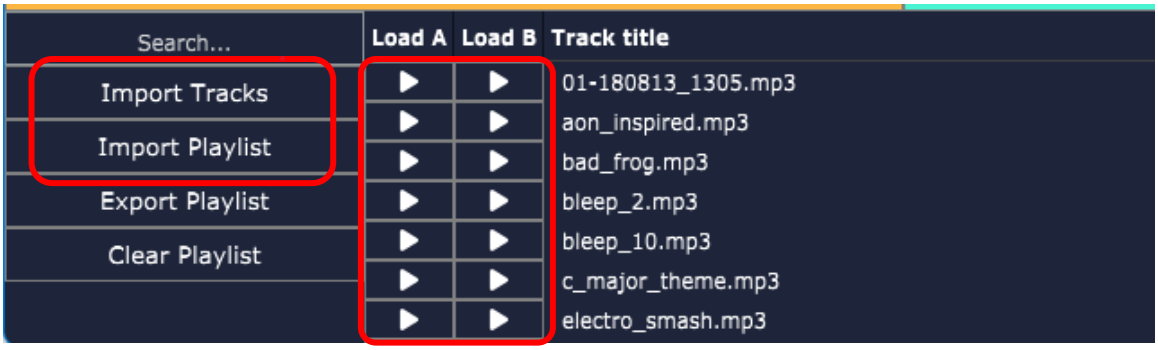


Users can drag and drop the audio track directly to the decks if it is valid then it will be loaded in the specific deck. This process is very similar to “Load External Track” only the difference is it uses the filesDropped function instead of FileChooser then uses the loadTrackToDeck function to load the track inside the deck.

```
338 bool DeckGUI::isInterestedInFileDrag(const StringArray& files)
339 {
340     std::cout << "DeckGUI::isInterestedInFileDrag" << std::endl;
341     return true;
342 }
343
344 // Load first track after drag and drop of the file in the deck player
345 void DeckGUI::filesDropped(const StringArray& files, int x, int y)
346 {
347     std::cout << "DeckGUI::filesDropped" << std::endl;
348     if (files.size() == 1)
349     {
350         loadTrackToDeck(File{ files[0] });
351     }
352 }
```

Dropping the track files to specific decks - DJAudioPlayer.cpp

[3] Using the playlist’s “Load A” or “Load B” Buttons



Loading music from a playlist is a bit more complicated than the first two options. First, we need to import tracks in the playlist using the “Import Tracks” or “Import Playlist” action from the left side of the component. Then after adding the tracks to the playlist we can load the tracks by using the “Load A” or “Load B” buttons which are highlighted in the right rectangle.

The “Import Tracks” button triggers the importTracksToPlaylist() function which opens a File chooser to select music tracks and then the tracks added to the playlistArr array.

```
256 void PlaylistComponent::importTracksToPlaylist()
257 {
258     auto flags = FileBrowserComponent::canSelectMultipleItems;
259     musicTracksChooser.launchAsync(flags, [this](const FileChooser& chooser)->void
260     {
261         auto file:Array<File> = chooser.getResults();
262         playlistArr.addArray(file);
263         tableComponent.updateContent();
264         tableComponent.repaint();
265         tableComponent.selectRow(rowNumber: 0);
266     });
267 }
```

Importing the tracks to the playlist table - PlaylistComponent.cpp

“Import Playlist” is very similar to “Import Tracks”, “Import Playlist” button triggers the importExportedPlaylist() function and opens a File chooser to select a previously exported .txt file that contains music track paths. After selecting the .txt file the data gets inserted inside the playlistArr.

```
321 void PlaylistComponent::importExportedPlaylist()
322 {
323     auto flags = FileBrowserComponent::canSelectMultipleItems;
324     importPlaylist.launchAsync(flags, [this](const FileChooser& chooser)->void
325     {
326         auto result:URL = chooser.getURLResult();
327         auto name:String = result.isEmpty() ? String()
328                          : (result.isLocalFile() ? result.getLocalFile().getFullPathName()
329                          : result.toString(includeGetParameters: true));
330 
331         if (!result.isEmpty())
332         {
333             autoImportDefaultPlaylist( result.getLocalFile().getFullPathName());
334         }
335     });
336 }
337 
338 void PlaylistComponent::autoImportDefaultPlaylist(String path)
339 {
340     File file(path);
341 
342     if (!file.existsAsFile())
343         return;  // file doesn't exist
344 
345     juce::FileInputStream inputStream(file); // [2]
346 
347     if (!inputStream.openedOk())
348         return;  // failed to open
349 
350     while (!inputStream.isExhausted()) // [3]
351     {
352         auto line:String = inputStream.readNextLine();
353         playlistArr.add(newElement: File(line));
354     }
355 
356     // Update the table component
357     tableComponent.updateContent();
358     tableComponent.repaint();
359 }
```

Importing playlist .txt file to the playlist table - PlaylistComponent.cpp

At the end of both functions there is tableComponent.updateContent() which refreshes the content in the playlist table using the paintCell() and refreshComponentForCell() functions. refreshComponentForCell() updates the Component IDs of the buttons to correctly point the tracks to load/delete events.

```
125 void PlaylistComponent::paintCell(Graphics& g, int rowNumber, int columnId, int width, int height, bool rowIsSelected)
126 {
127     // Create a temporary array to store the playlist array
128     Array <juce::File> tracksArr;
129     if (!searchInput.isEmpty())
130     {
131         tracksArr = filteredPlaylistArr;
132     }
133     else
134     {
135         tracksArr = playlistArr;
136     }
137 
138     g.setColour(Colours::white);
139 
140     if (columnId == 3)
141     {
142         g.drawText(tracksArr[rowNumber].getFileName(), x: 10, y: 0, width, height, Justification::centredLeft);
143     }
144     if (columnId == 4 || columnId == 5)
```

paintCell function for updating the textual content of the table - PlaylistComponent.cpp

```
174 Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int columnId, bool isRowSelected, Component* existingComponentToUpdate)
175 {
176     // If the existingComponentToUpdate component is null then update the component
177     if (existingComponentToUpdate == nullptr)
178     {
179         if (columnId == 1) {
180             DrawableButton* loadDeck1Btn = new DrawableButton{  buttonName: "LOAD_DECK_1", DrawableButton::ButtonStyle::ImageOnButtonBackground };
181             loadDeck1Btn->setImages( normalImage: playSvg.get());
182             loadDeck1Btn->addListener(this);
183             int index = playlistArrIndex[rowNumber];
184             String id = String(index);
185             loadDeck1Btn->setComponentID(id);
186             existingComponentToUpdate = loadDeck1Btn;
187         }
188 
189         else if (columnId == 2)
190         {
191             DrawableButton* loadDeck2Btn = new DrawableButton{  buttonName: "LOAD_DECK_2", DrawableButton::ButtonStyle::ImageOnButtonBackground };
192             loadDeck2Btn->setImages( normalImage: playSvg.get());
193             loadDeck2Btn->addListener(this);
```

refreshComponentForCell for updating custom components inside the table - PlaylistComponent.cpp

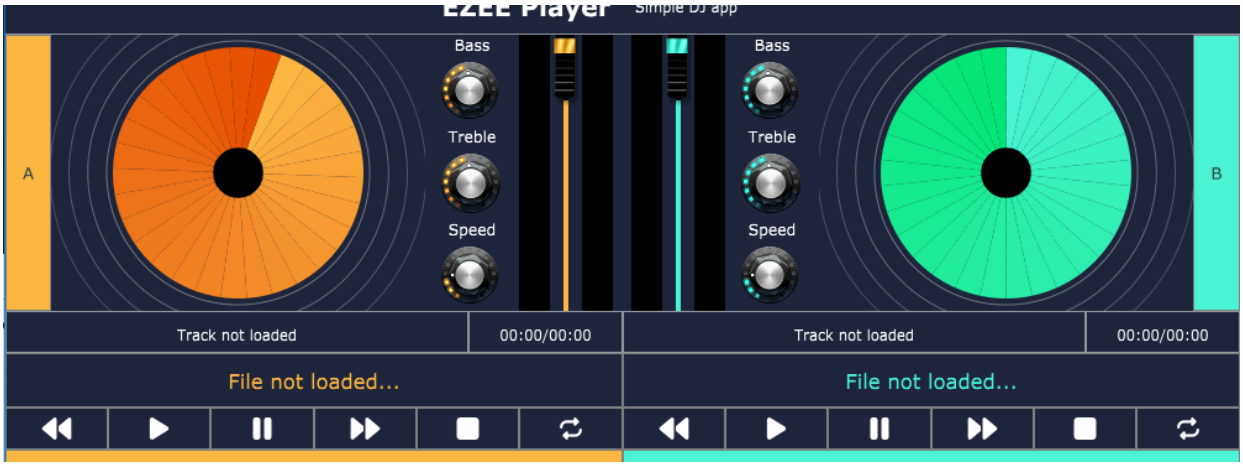
Now, Using the Load A and Load B buttons we can load the tracks to the deck, its logic happens in the buttonClicked() function. It checks the button text if it is “LOAD_DECK_1” or “LOAD_DECK_2” then the specific tracks get loaded in the deck by using their “id”.

```
414 void PlaylistComponent::buttonClicked(Button* button)
415 {
416     int id;
417     // Get the id of the button
418     if (button->getComponentID() != "")
419     {
420         id = std::stoi( _Str: button->getComponentID().toStdString());
421     }
422 
423     // Delete track from playlist
424     if (button->getButtonText() == "DELETE_TRACK")
425     {
426         deleteTrackFromPlaylist(id);
427     }
428 
429     else if (button->getButtonText() == "LOAD_DECK_1")
430     {
431         DBG("LOAD_DECK_1");
432         deckGUI1->loadTrackToDeck(playlistArr[id]);
433 
434         // ...
437 
438         // ...
445     }
```

buttonClicked event handler - PlaylistComponent.cpp

R1B: can play two or more tracks

Absolutely, there are two decks “Deck A” and “Deck B” on the DJ player which can play two separate audio tracks.



You can see there are several buttons to handle track controls which are present at the bottom of the waveform display their button events happening inside the buttonClicked() function. After the button click it triggers the player's function like player->play(), player->stop() functions.

```
233  // On button click event
234 void DeckGUI::buttonClicked(Button* button)
235 {
236     if (button == &playButton)
237     {
238         std::cout << "Play button was clicked " << std::endl;
239         player->start();
240         diskArt.setRotationSpeed(1);
241     }
242
243     if (button == &stopButton)
244     {
245         std::cout << "Stop button was clicked " << std::endl;
246         player->stop();
247         diskArt.setRotationSpeed(0);
248     }
249
250     if (button == &pauseButton)
251     {
252         std::cout << "Pause button was clicked " << std::endl;
253         player->pause();
254         diskArt.setRotationSpeed(0);
255     }   if (button == &backwardButton)
256     {
257         std::cout << "Backward button was clicked " << std::endl;
258         player->backward();
259     }   if (button == &forwardButton)
260     {
261         std::cout << "Forward button was clicked " << std::endl;
262         player->forward();
263     }
264
265     // Loop button toggle
266     if (button == &loopButton && player->isPlaying())
267     {
268         if (loopButton.getToggleState())
269         {
270             loopButton.setColour(TextButton::buttonColourId,
```

buttonClicked event handler - DeckGUI.cpp

R1C: can mix the tracks by varying each of their volumes

Yes, we can mix the tracks by varying each of their volumes by using the vertical slider. By default, the volume is set to 100%, the user can adjust the volume by changing the slider position.

These sliders trigger the sliderValueChanged() function from there an if statement check if it is a volume slider, if it is a volume slider then calls player->setGain() functions and updates the volume of the deck.

```
303  // Slider value change events
304 void DeckGUI::sliderValueChanged(Slider* slider)
305 {
306     if (slider == &volSlider)
307     {
308         player->setGain(slider->getValue());
309     }
```

sliderValueChange event handler for volume slider - DeckGUI.cpp

```
108  // Volume Control
109 void DJAudioPlayer::setGain(double gain)
110 {
111     if (gain < 0 || gain > 1.0)
112     {
113         std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl
114     }
115     else {
116         transportSource.setGain(gain);
117     }
118 }
```

setGain function to adjust the volume of the audio - DJAudioPlayer.cpp

R1D: can speed up and slow down the tracks

Of course, the user can speed up or slow down the track by using the speed rotary slider which is highlighted in the below screenshot. It has min 0.0 and max 5.0 values. By default, it is set to 1.0.

When the rotary slider value changes the sliderValueChanged() function gets triggered and checks if it is the speed slider if it is correct then updates the speed using player->setSpeed() function.

```
319     if (slider == &speedSlider)
320     {
321         player->setSpeed( ratio: slider->getValue());
322         if (player->isPlaying()) {
323             diskArt.setRotationSpeed(slider->getValue());
324         }
325     }
```

speedSlider event handler - DeckGUI.cpp

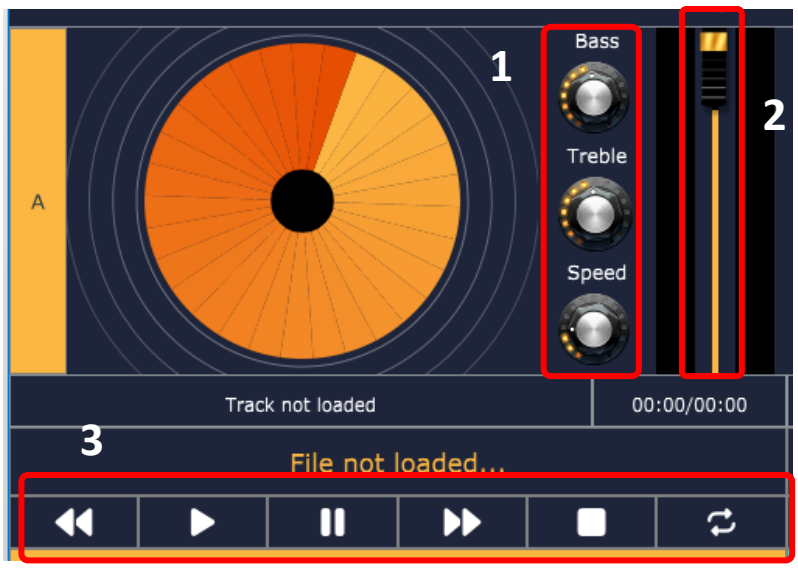
```
void DJAudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0 || ratio > 100.0)
    {
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::endl;
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

speedSlider function to update the speed of the audio - 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 have added several custom graphics in the decks 1) Custom rotary slider 2) Vertical volume slider 3) Multiple playback buttons which include backward, play, pause, forward stop and loop.



For the rotary slider and vertical slider, I have used a custom look and feel for which I have used PNG images, specifically rotary sliders the PNG image is a vertical strip of 101 images which is generated by using Knobman software.

Inside the look and feel of the rotary slider, the it finds correct frame (image) by using the rotation of the rotary slider and draws the image. The vertical slider uses a thumb PNG image and draws two rectangles one with the player theme color and the other with the black color. As per the slider value, the thumb updates its position and the colored rectangle updates its height.

```
void MyRotarySliderLookAndFeel::drawRotarySlider(juce::Graphics& g, int x, int y, int width, int height, float sliderPos,
    const float rotaryStartAngle, const float rotaryEndAngle, juce::Slider& slider)
{
    if (knob.isValid())
    {
        const double rotation = (slider.getValue() - slider.getMinimum()) / (slider.getMaximum() - slider.getMinimum());
        const int frames = knob.getHeight() / knob.getWidth();
        const int frameId = (int)ceil( x: rotation * ((double)frames - 1.0));
        const float padding = 5.0f;
        const float radius = jmin( a: width / 2.0f - padding,  b: height / 2.0f - padding);
        const float centerX = x + width * 0.5f;
        const float centerY = y + height * 0.5f;

        const float rx = centerX - radius - 1.0f;
        const float ry = centerY - radius;

        g.drawImage(knob,  destX: (int)rx,  destY: (int)ry,  destWidth: 2 * (int)radius,  destHeight: 2 * (int)radius,
            sourceX: 0,  sourceY: frameId * knob.getWidth(),  sourceWidth: knob.getWidth(),  sourceHeight: knob.getWidth());
    }

    // if the knob image is not valid, shows a text instead of the knob
    else
    {
        static const float textPercent = 0.35f;
        Rectangle<float> text_bounds( initialX: 1.0f + width * (1.0f - textPercent) / 2.0f,  initialY: 0.5f * height, width * textPercent, 0.5f *
          height);

        g.setColour(Colours::white);

        g.drawFittedText(String( text: "No Image"),  area: text_bounds.getSmallestIntegerContainer(),
            Justification::horizontallyCentred | Justification::centred,  maximumNumberOfLines: 1);
    }
}
```

Drawing the rotary knob image as per slider value - MyLookAndFeel.cpp

```
void MySliderLookAndFeel::drawLinearSlider(Graphics& g, int x, int y, int width, int height,
    float sliderPos, float minSliderPos, float maxSliderPos, const Slider::SliderStyle style, Slider& slider)
{
    const int lineThickness = 4;
    const int dialWidth = sliderThumbImage.getWidth();
    const int dialHeight = sliderThumbImage.getHeight();

    const float padding = 10.0f;
    const float destWidth = width;
    const float destHeight = height * 0.5;
    const float centerX = x + destWidth * 0.5f;
    const float centerY = y + destHeight * 0.5f - 8.0f;
    const float rx = centerX - destWidth / 2 - 1.0f;
    const float ry = centerY;

    // If the slider is for deck A then set the colour to orange else set it to cyan
    if (*side == String( text: "A"))
    {
        g.setColour(Colour( red: 252, green: 183, blue: 67));
    }
    else
    {
        g.setColour(Colour( red: 74, green: 244, blue: 210));
    }

    // Draw the slider with respective theme color
    g.fillRect( x: ((int)destWidth - lineThickness) / 2, y: (int)destHeight / 2 - 8.0, width: lineThickness, height);

    g.setColour(Colours::black);
    // draw black slider backround
    g.fillRect( x: ((int)destWidth - lineThickness) / 2, y: (int)destHeight / 2 - 8.0, width: lineThickness, height: (int)sliderPos);

    // draw the slider thumb
    g.drawImage(sliderThumbImage, destX: (int)rx, destY: sliderPos - (int)ry, destWidth: (int)destWidth, destHeight: (int)destHeight,
        sourceX: 0, sourceY: 0, sourceWidth: dialWidth, sourceHeight: dialHeight);
```

Drawing the vertical volume slider – MyLookAndFeel.cpp

Taking reference from the following GitHub repository, using knobs from g200kg.com gallery, and Knobman software to edit the knobs:

- https://github.com/remberg/juceCustomSliderSample

- https://www.g200kg.com/en/webknobman/gallery.php

For the playback button, I have changed its color and removed the round corners, the modifications are done through the custom look and feel as well.

```
void MyLookAndFeel::drawButtonBackground(juce::Graphics& g, juce::Button& button, const juce::Colour& backgroundColour,
    bool isMouseOverButton, bool isButtonDown)
{
    LookAndFeel_V4::drawButtonBackground( [&] g, [&] button, backgroundColour, shouldDrawButtonAsHighlighted: isMouseOverButton,

    g.setColour(backgroundColour);
    g.fillRect( rectangle: button.getLocalBounds());
    g.setColour(juce::Colours::grey);
    g.drawRect( rectangle: button.getLocalBounds(), lineThickness: 1);

    if (isButtonDown)
    {
        g.setColour(juce::Colours::grey); // Set background color when button is pressed
        g.fillRect( rectangle: button.getLocalBounds());
    }
}
```

Updating the button theme – MyLookAndFeel.cpp

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

The rotary slider, vertical slider, and playback buttons allow controlling the playing audio. And the waveform display allows the user to reposition the current playing track by clicking on the waveform display.

The rotary slider event triggers the setBass() and setTreble() function which modifies IIRCoefficients and updates the audio Bass and Treble. setSpeed() function changes the sampling ratio of the audio which causes the audio to change the speed.

```
120 void DJAudioPlayer::setBass(double bass)
121 {
122     if (bass < 0 || bass > 2)
123     {
124         DBG("DJAudioPlayer::setBass ratio should be between 0 and 2");
125     }
126     else
127     {
128         bassSource.setCoefficients(juce::IIRCoefficients::makeLowShelf(trackSampleRate, bassCutOffFreq, Q: 1,
              bass));
129     }
130 }
131
132 void DJAudioPlayer::setTreble(double treble)
133 {
134     if (treble < 0 || treble > 2)
135     {
136         DBG("DJAudioPlayer::setTreble ratio should be between 0 and 2");
137     }
138     else
139     {
140         trebleSource.setCoefficients(juce::IIRCoefficients::makeHighShelf(trackSampleRate, trebleCutOffFreq,
              Q: 1, treble));
141     }
142 }
143
144 void DJAudioPlayer::setSpeed(double ratio)
145 {
146     if (ratio < 0 || ratio > 100.0)
147     {
148         std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::endl;
149     }
150     else {
151         resampleSource.setResamplingRatio(ratio);
152     }
153 }
```

DJ Player's logic for bass, treble, speed – DJAudioPlayer.cpp

The playback button triggers start, stop, pause, backward, forward, and loop functions which allow the update of the transportSource as per the function of the buttons.

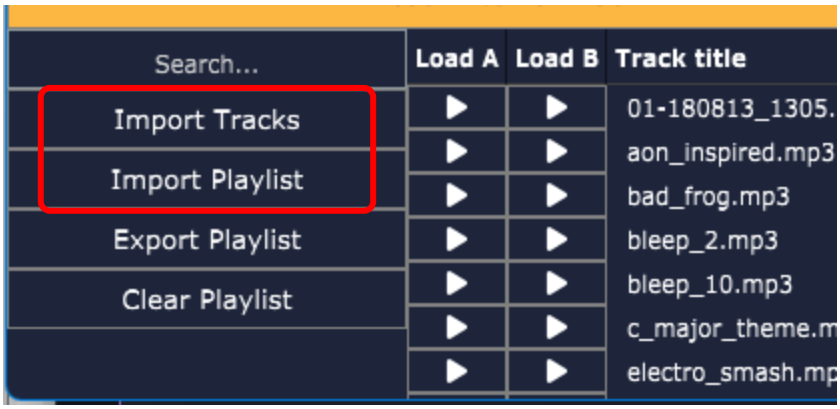
```
172 void DJAudioPlayer::start()
173 {
174     transportSource.start();
175 }
176
177 void DJAudioPlayer::stop()
178 {
179     transportSource.stop();
180     transportSource.setPosition(0);
181 }
182
183 void DJAudioPlayer::backward()
184 {
185     auto currentPos :double = transportSource.getCurrentPosition();
186     if(currentPos < 5)
187     {
188         transportSource.setPosition(0);
189     } else
190     {
191     transportSource.setPosition(currentPos - 5); // 5 seconds back from current position
192     }
193 }
194
195 void DJAudioPlayer::forward()
196 {
197     auto currentPos :double = transportSource.getCurrentPosition();
198     auto totalSec :double = transportSource.getLengthInSeconds();
199     if (totalSec - currentPos < 5)
200     {
201         transportSource.setPosition(totalSec);
202     }
203     else
204     {
205         transportSource.setPosition(currentPos + 5); // 5 seconds forward from current posi
206     }
207 }
208
209 void DJAudioPlayer::pause()
210 {
211     transportSource.stop();
212 }
213
214 void DJAudioPlayer::loop()
215 {
216     if (readerSource->isLooping())
```

Playback buttons logic – DJAudioPlayer.cpp

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

PlaylistComponent.cpp “Import Tracks” and “Import Playlist” buttons allow to addition of tracks to the playlist.



The “Import Tracks” button triggers the importTracksToPlaylist() function which opens File choose after selecting the music tracks, the tracks get added to playlistArr and the playlist gets updated.

```
256 void PlaylistComponent::importTracksToPlaylist()
257 {
258     auto flags = FileBrowserComponent::canSelectMultipleItems;
259     musicTracksChooser.launchAsync(flags, [this](const FileChooser& chooser)->
260     {
261         auto file :Array<File> = chooser.getResults();
262         playlistArr.addArray(file);
263         tableComponent.updateContent();
264         tableComponent.repaint();
265         tableComponent.selectRow( rowNumber: 0);
266     });
267 }
```

Importing the tracks to the playlist – PlaylistComponent.cpp

“Import Playlist” allows the user to import previously exported playlist .txt file which includes audio track paths. By doing a while loop on the input stream, it adds each file in the playlistArr array.

```
321 void PlaylistComponent::importExportedPlaylist()
322 {
323     auto flags = FileBrowserComponent::canSelectMultipleItems;
324     importPlaylist.launchAsync(flags, [this](const FileChooser& chooser)->void
325     {
326         auto result :URL = chooser.getURLResult();
327         auto name :String = result.isEmpty() ? String()
328                           : (result.isLocalFile() ? result.getLocalFile().getFullPathName()
329                           : result.toString( includeGetParameters: true));
330 
331         if (!result.isEmpty())
332         {
333             autoImportDefaultPlaylist( result.getLocalFile().getFullPathName());
334         }
335     });
336 }
```

Importing the previously exported .txt playlist file – PlaylistComponent.cpp

```
338 void PlaylistComponent::autoImportDefaultPlaylist(String path)
339 {
340     File file(path);
341 
342     if (!file.existsAsFile())
343         return;  // file doesn't exist
344 
345     juce::FileInputStream inputStream(file); // [2]
346 
347     if (!inputStream.openedOk())
348         return;  // failed to open
349 
350     while (!inputStream.isExhausted()) // [3]
351     {
352         auto line :String = inputStream.readNextLine();
353         playlistArr.add( newElement: File(line));
354     }
355 
356     // Update the table component
357     tableComponent.updateContent();
358     tableComponent.repaint();
359 }
```

importing the .txt file's tracks to the playlist – PlaylistComponent.cpp

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

Load A	Load B	Track title	Format	Time
▶	▶	01-180813_1305.mp3	MP3 file	00:13
▶	▶	aon_inspired.mp3	MP3 file	01:34
▶	▶	bad_frog.mp3	MP3 file	08:00
▶	▶	bleep_2.mp3	MP3 file	00:50
▶	▶	bleep_10.mp3	MP3 file	00:18
▶	▶	c_major_theme.mp3	MP3 file	01:46
▶	▶	electro_smash.mp3	MP3 file	02:14

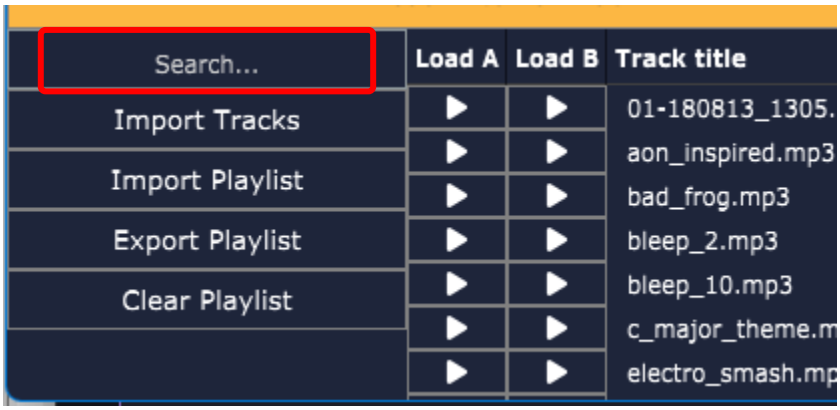
formatManager.createReaderFor() function reads each track's details and gets the track name and calculates the total length in seconds. The seconds are converted in hh:mm:ss format using the utilities component's convertSecTohhmmssFormat. I have also tried to get the artist name of the track file but found that juce is unable to get the metadata of mp3 files so, I replace the artist with 'file format'.

```
140     if (columnId == 3)
141     {
142         g.drawText(tracksArr[rowNumber].getFileName(), x: 10, y: 0, width, height,
                Justification::centredLeft);
143     }
144     if (columnId == 4 || columnId == 5)
145     {
146         AudioFormatManager formatManager;
147         formatManager.registerBasicFormats();
148         ScopedPointer<AudioFormatReader> reader = formatManager.createReaderFor(tracksArr[rowNumber]);
149
150         // ...
158
159         if (columnId == 4  && reader)
160         {
161             //Juce is not able to read the metadata of the mp3 file so I am using 'file format' instead of
                  'artist'.
162             g.drawText(reader->getFormatName(), x: 0, y: 0, width, height,  Justification::centredLeft);
163         }
164
165         if (columnId == 5  && reader)
166         {
167             int seconds = reader->lengthInSamples / reader->sampleRate;
168             String time = utils.convertSecTohhmmssFormat(seconds);
169             g.drawText(time, x: 0, y: 0, width, height,  Justification::centredLeft);
170         }
171     }
```

Parsing the metadata of the tracks – PlaylistComponent.cpp

R3C: Component allows the user to search for files

The top left search input field allows to filter the tracks in the playlist.



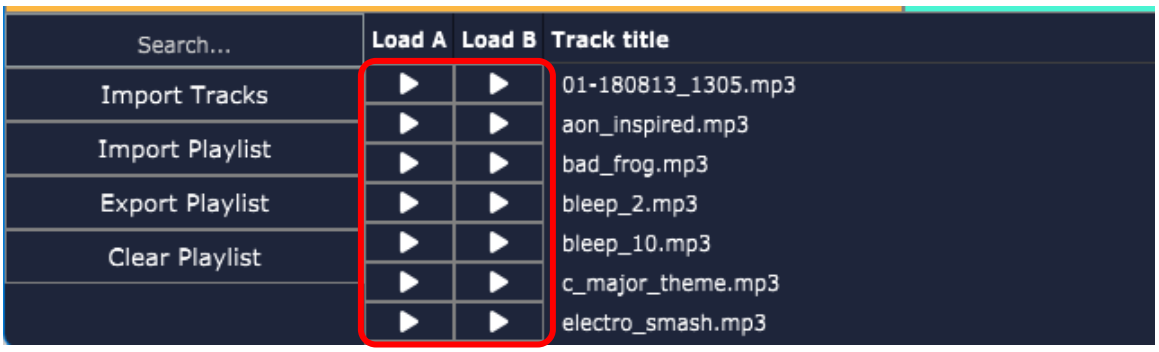
The search input triggers searchTrackInPlaylist() on each change in the search input. Inside the search function first, it clears the previously filtered playlist then find the matching tracks and adds them to filteredPlaylistArr.

```
382 void PlaylistComponent::searchTrackInPlaylist(String textString)
383 {
384
385     // Clear the filtered playlist array
386     filteredPlaylistArr.clear();
387
388     // Iterate through the playlist array and add the matching tracks to the filtered playlist array
389     for (int i = 0; i < playlistArr.size(); i++)
390     {
391         if (playlistArr[i].getFileNameWithoutExtension().containsIgnoreCase( textString))
392         {
393             filteredPlaylistArr.add(playlistArr[i]);
394         }
395     }
396
397     // Update the playlistArrIndex to reflect the new indices of the filtered tracks
398     playlistArrIndex.clear();
399     for (int i = 0; i < filteredPlaylistArr.size(); i++)
400     {
401         int index = playlistArr.indexOf(filteredPlaylistArr[i]);
402         if (index != -1)
403         {
404             playlistArrIndex.add(index);
405         }
406     }
407
408     // Update the table to display the filtered playlist
409     tableComponent.updateContent();
410     tableComponent.repaint();
411     tableComponent.selectRow( rowNumber: 0);
412 }
```

Searching for matching tracks in the playlist – PlaylistComponent.cpp

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

The “Load A” and “Load B” buttons allow the user to load tracks in the specific decks.



For loading the tracks it uses the buttonClicked() event function as the function gets triggered it checks if the button has buttonText as “LOAD_DECK_1” or “LOAD_DECK_2”. If it is correct then it loads the track using the loadTrackToDeck() function.

```
429     else if (button->getButtonText() == "LOAD_DECK_1")
430     {
431         DBG("LOAD_DECK_1");
432         deckGUI1->loadTrackToDeck(playlistArr[id]);
433 
434         // ...
437 
438         // ...
445     }
446 
447     else if (button->getButtonText() == "LOAD_DECK_2")
448     {
449         DBG("LOAD_DECK_2");
450         deckGUI2->loadTrackToDeck(playlistArr[id]);
451 
452         // ...
455 
456         // ...
463     }
```

Load track button event handler – PlaylistComponent.cpp

```
354 void DeckGUI::loadTrackToDeck(File file)
355 {
356     player->loadURL(URL{ file });
357     waveformDisplay.loadURL(URL{ file });
358     trackTitleTxt.setText( newText: URL::removeEscapeChars(player->getTrackDetails()[0]));
359     trackDurationTxt.setText(player->getTrackDetails()[1]);
360     player->start();
361 
362     // Start disk rotation
363     if (player->isPlaying()) {
364         diskArt.setRotationSpeed(speedSlider.getValue());
365     }
366 }
```

Loading the track to the deck – DeckGUI.cpp

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

Yes, the playlist persists the previous time-loaded tracks by importing the playlist.txt file at the start of the application by using autoImportDefaultPlaylist() and exports the currently loaded track as playlist.txt file at the time of closing of the application by using the autoExportDefaultPlaylist() function.

```
338 void PlaylistComponent::autoImportDefaultPlaylist(String path)
339 {
340     File file(path);
341 
342     if (!file.existsAsFile())
343         return;  // file doesn't exist
344 
345     juce::FileInputStream inputStream(file); // [2]
346 
347     if (!inputStream.openedOk())
348         return;  // failed to open
349 
350     while (!inputStream.isExhausted()) // [3]
351     {
352         auto line :String  = inputStream.readNextLine();
353         playlistArr.add( newElement: File(line));
354     }
355 
356     // Update the table component
357     tableComponent.updateContent();
358     tableComponent.repaint();
359 }
```

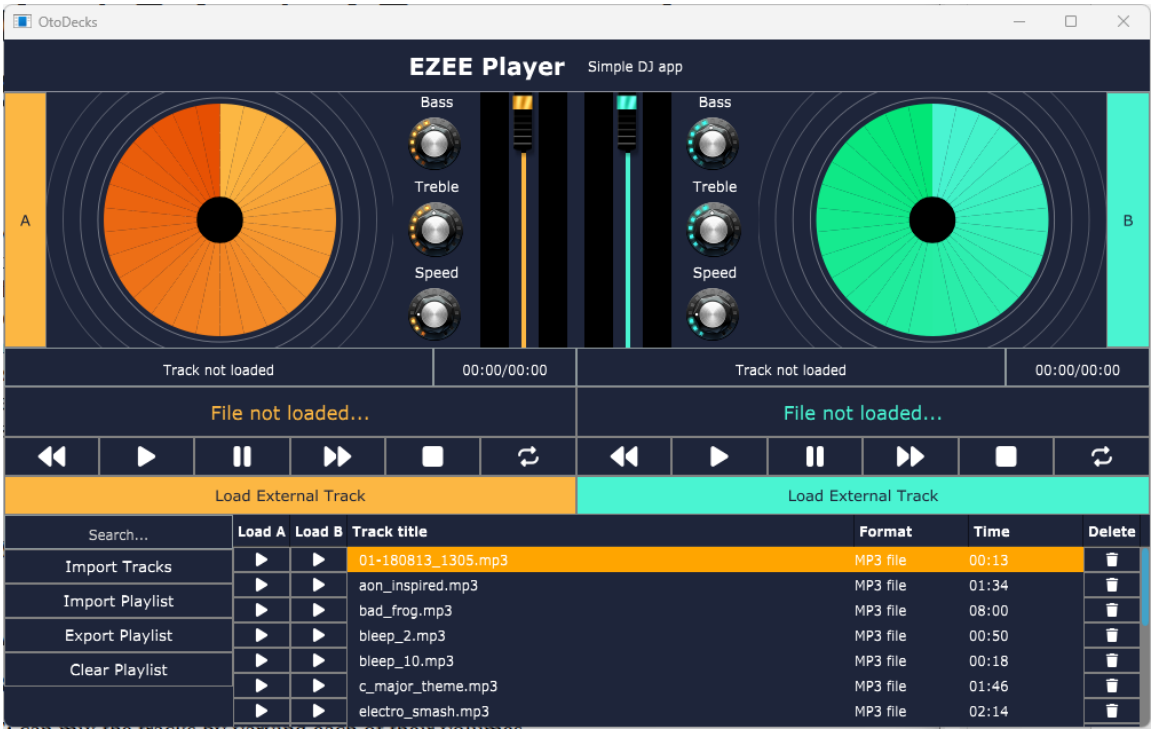
Automatically import the default playlist playlist.txt file – PlaylistComponent.cpp

```
304 void PlaylistComponent::autoExportDefaultPlaylist(String path)
305 {
306     auto file = juce::File::getCurrentWorkingDirectory().getChildFile( "playlist.txt");
307     FileOutputStream output(file);
308 
309     if (output.openedOk())
310     {
311         output.setPosition(0);  // default would append
312         output.truncate();
313 
314         // Append the playlist to the playlist.txt
315         for (auto file  : playlistArr) {
316             output.writeText(file.getFullPathName()  + "\n", asUTF16: false, writeUTF16ByteOrderMark: false,
                    lineEndings: "");
317         }
318     }
319 }
```

Automatically export the tracks to default playlist.txt file – PlaylistComponent.cpp

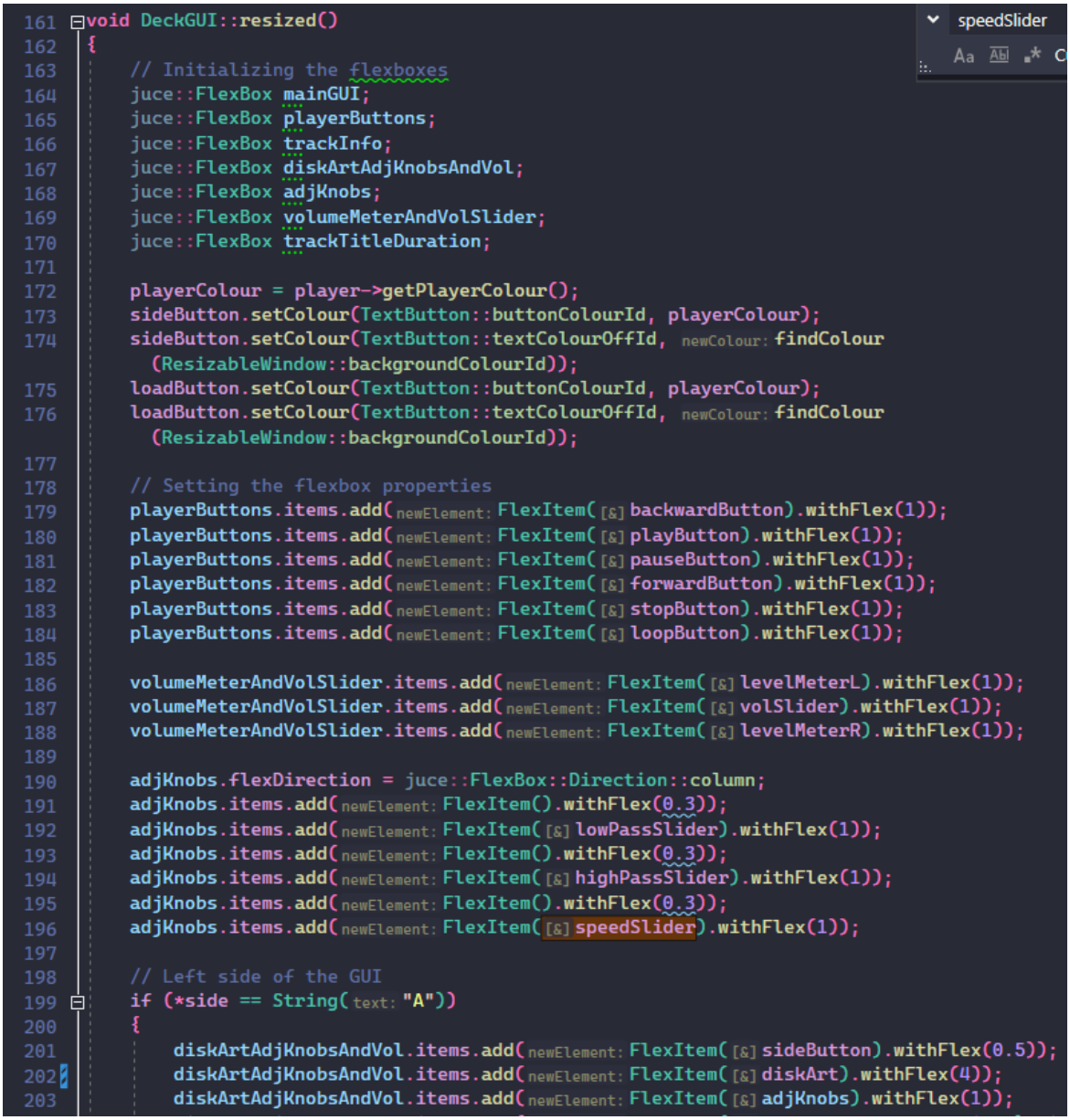
R4: Implementation of a complete custom GUI

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



I have significantly modified the GUI of the DeckGUI. I have added rotary sliders, a rotating disk, deck name (A/B), track title and duration, playback buttons, volume level meters, and a volume slider.

I have used font awesome icons for the buttons and for the layout I have used a modern flexbox style which is easy to understand and easy to adjust the size.



```
161 void DeckGUI::resized()
162 {
163     // Initializing the flexboxes
164     juce::FlexBox mainGUI;
165     juce::FlexBox playerButtons;
166     juce::FlexBox trackInfo;
167     juce::FlexBox diskArtAdjKnobsAndVol;
168     juce::FlexBox adjKnobs;
169     juce::FlexBox volumeMeterAndVolSlider;
170     juce::FlexBox trackTitleDuration;
171
172     playerColour = player->getPlayerColour();
173     sideButton.setColour(TextButton::buttonColourId, playerColour);
174     sideButton.setColour(TextButton::textColourOffId, newColour: findColour
          (ResizableWindow::backgroundColourId));
175     loadButton.setColour(TextButton::buttonColourId, playerColour);
176     loadButton.setColour(TextButton::textColourOffId, newColour: findColour
          (ResizableWindow::backgroundColourId));
177
178     // Setting the flexbox properties
179     playerButtons.items.add(newElement: FlexItem([&] backwardButton).withFlex(1));
180     playerButtons.items.add(newElement: FlexItem([&] playButton).withFlex(1));
181     playerButtons.items.add(newElement: FlexItem([&] pauseButton).withFlex(1));
182     playerButtons.items.add(newElement: FlexItem([&] forwardButton).withFlex(1));
183     playerButtons.items.add(newElement: FlexItem([&] stopButton).withFlex(1));
184     playerButtons.items.add(newElement: FlexItem([&] loopButton).withFlex(1));
185
186     volumeMeterAndVolSlider.items.add(newElement: FlexItem([&] levelMeterL).withFlex(1));
187     volumeMeterAndVolSlider.items.add(newElement: FlexItem([&] volSlider).withFlex(1));
188     volumeMeterAndVolSlider.items.add(newElement: FlexItem([&] levelMeterR).withFlex(1));
189
190     adjKnobs.flexDirection = juce::FlexBox::Direction::column;
191     adjKnobs.items.add(newElement: FlexItem().withFlex(0.3));
192     adjKnobs.items.add(newElement: FlexItem([&] lowPassSlider).withFlex(1));
193     adjKnobs.items.add(newElement: FlexItem().withFlex(0.3));
194     adjKnobs.items.add(newElement: FlexItem([&] highPassSlider).withFlex(1));
195     adjKnobs.items.add(newElement: FlexItem().withFlex(0.3));
196     adjKnobs.items.add(newElement: FlexItem([&] speedSlider).withFlex(1));
197
198     // Left side of the GUI
199     if (*side == String(text: "A"))
200     {
201         diskArtAdjKnobsAndVol.items.add(newElement: FlexItem([&] sideButton).withFlex(0.5));
202         diskArtAdjKnobsAndVol.items.add(newElement: FlexItem([&] diskArt).withFlex(4));
203         diskArtAdjKnobsAndVol.items.add(newElement: FlexItem([&] adjKnobs).withFlex(1));
```

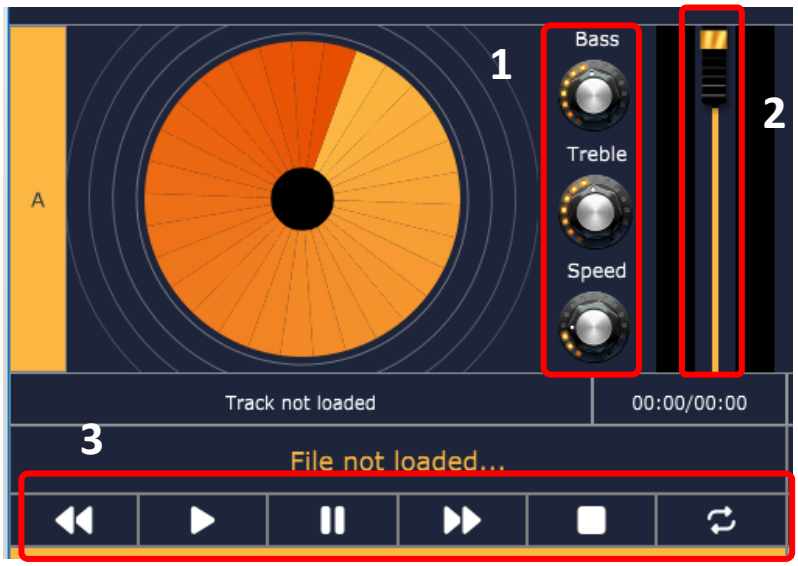
Flexbox structure of the DeckGUI – DeckGUI.cpp

```
{
    // Import svg button SVGs
    auto playSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::play_solid_svg,
      numBytes: BinaryData::play_solid_svgSize);
    auto backwardSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::backward_solid_svg,
      numBytes: BinaryData::backward_solid_svgSize);
    auto forwardSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::forward_solid_svg,
      numBytes: BinaryData::forward_solid_svgSize);
    auto pauseSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::pause_solid_svg,
      numBytes: BinaryData::pause_solid_svgSize);
    auto stopSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::stop_solid_svg,
      numBytes: BinaryData::stop_solid_svgSize);
    auto loopSvg :unique_ptr<Drawable> = Drawable::createFromImageData(BinaryData::repeat_solid_svg,
      numBytes: BinaryData::repeat_solid_svgSize);

    // Setting up the buttons text and images
    playButton.setImages(normalImage: playSvg.get());
    stopButton.setImages(normalImage: stopSvg.get());
    pauseButton.setImages(normalImage: pauseSvg.get());
    backwardButton.setImages(normalImage: backwardSvg.get());
    forwardButton.setImages(normalImage: forwardSvg.get());
    loopButton.setClickingTogglesState(shouldAutoToggleOnClick: true);
    loopButton.setImages(normalImage: loopSvg.get());

    sideButton.setButtonText(*side);
```

Importing and setting svg images to the playback buttons – DeckGUI.cpp

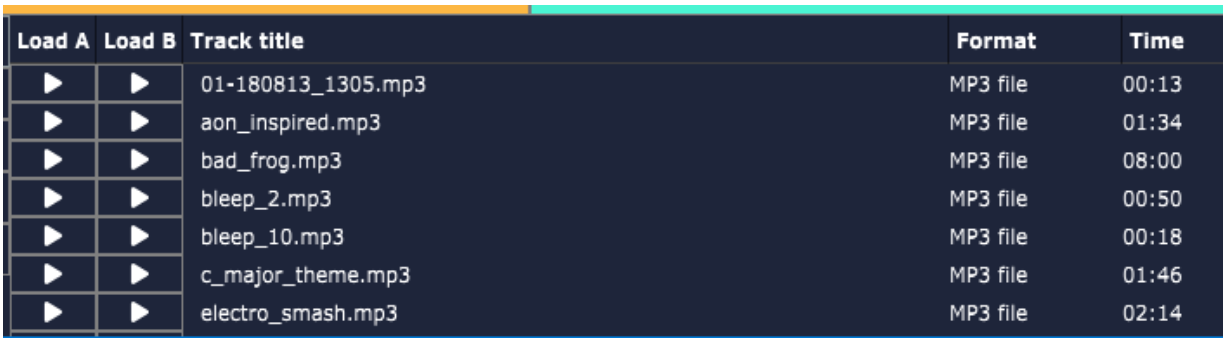
R4B: GUI layout includes the custom Component from R2

Yes, Rotary sliders, vertical volume sliders, and playback controllers are included in the GUI layout. A more detailed explanation is given in the R2.



R4C: GUI layout includes the music library component from R3

Yes, the DJ application includes a playlist library at the bottom of the application which uses the table list box class from the juce. A more detailed explanation is given in R3.



Load A	Load B	Track title	Format	Time
▶	▶	01-180813_1305.mp3	MP3 file	00:13
▶	▶	aon_inspired.mp3	MP3 file	01:34
▶	▶	bad_frog.mp3	MP3 file	08:00
▶	▶	bleep_2.mp3	MP3 file	00:50
▶	▶	bleep_10.mp3	MP3 file	00:18
▶	▶	c_major_theme.mp3	MP3 file	01:46
▶	▶	electro_smash.mp3	MP3 file	02:14

References:

[1] JUCE framework. (2023, January 12). JUCE. https://juce.com/

[2] Font Awesome. (n.d.). Font Awesome. https://fontawesome.com/

[3] Remberg. (n.d.). GitHub - remberg/juceCustomSliderSample: Simple juce custom slider example using png files loading from file. GitHub. https://github.com/remberg/juceCustomSliderSample

[4] KnobGallery. (n.d.). https://www.g200kg.com. https://www.g200kg.com/en/webknobman/gallery.php

[5] KnobMan, g200kg Music & Software. (n.d.). G200kg Music & Software. https://www.g200kg.com/

[6] Royalty Free Music Download - Pixabay. (n.d.). https://pixabay.com/music/