Table of Contents

Introduction:

Otodecks has evolved into a cutting-edge DJ application, offering users a unique and improved experience in music mixing. We strive to improve it into a platform that merges creativity, functionality, and user-friendly design. In this report, I will delve into the features I have meticulously crafted to enhance the capabilities and user interaction within Otodecks.

Basic Requirements:

R1A: Loading audio files into player

```
37  void DJAudioPlayer::loadURL(URL audioURL)
38  {
39      auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
40      if (reader != nullptr) // good file!
41      {
42          std::unique_ptr<AudioFormatReaderSource> newSource(new AudioFormatReaderSource(reader, true));
43          transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
44          readerSource.reset(newSource.release());
45      }
46  }
```

Figure 1A.1

One of the main features of our DJ application is loadURL, which enables users to import their favourite audio tracks with ease. This technique is important to the DJing experience since it allow users to load audio files from a given URL, as seen in Figure 1.1, loadURL is specifically made to take an audio URL argument (line 37) that specifies the location of the file. It then uses "AudioFormatManager" (line 39) to create an audio reader that is specific to the URL that is supplied. After this audio reader is successfully created, we validate the file (line 40) to make sure the audio that was imported is appropriate for further processing. A critical phase in this process involves the instantiation of an "AudioFormatReaderSource," (line 42) which contains the audio reader that was previously constructed. This source helps channel audio data into the application's audio transport system.. Essentially, loadURL is essential to the streamlined audio track import process, demonstrating our dedication to giving users a dynamic and user-friendly DJing experience.

```
150  bool DeckGUI::isInterestedInFileDrag(const StringArray& files)
151  {
152      DBG("DeckGUI::isInterestedInFileDrag");
153      return true;
154  }
155
156  void DeckGUI::filesDropped(const StringArray& files, int x, int y)
157  {
158      DBG("DeckGUI::filesDropped");
159      if (files.size() == 1)
160      {
161          // Load the URL into the player and waveform display
162          juce::URL musicUrl = juce::URL{ juce::File{files[0]} };
163          loadMusicFileToApplication(musicUrl);
164      }
165  }
```

Figure 1A.2

```
210  // Load an audio file from a URL as input into the player and waveform display for visualization
211  void DeckGUI::loadMusicFileToApplication(const juce::URL& musicUrl)
212  {
213      player->loadURL(musicUrl);
214      waveformDisplay.loadURL(musicUrl);
215
216      // Update the musicNameLabel content
217      updateLabels(musicUrl);
218  }
```

Figure 1A.3

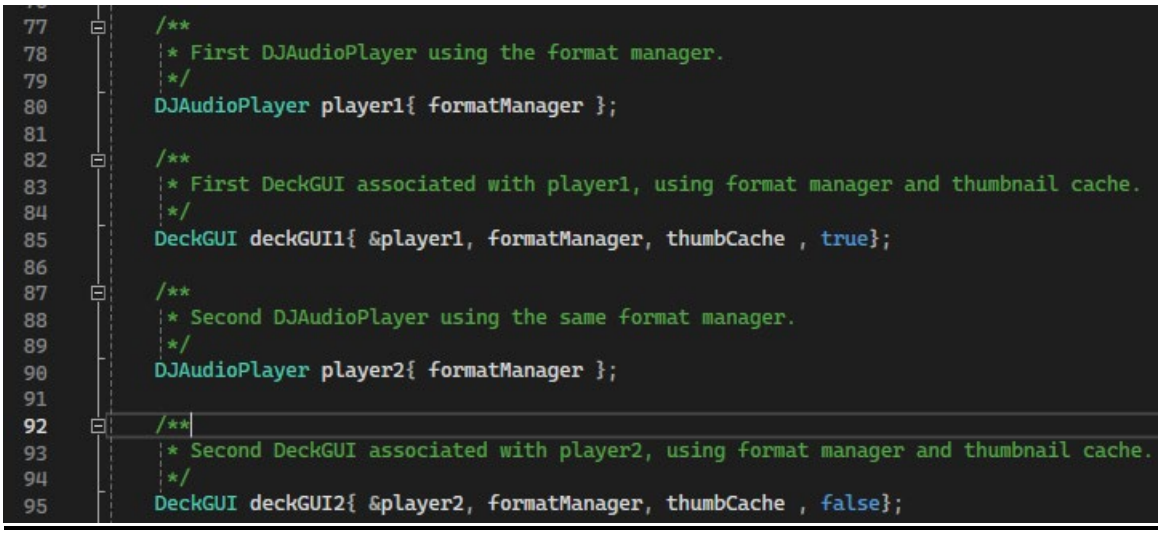
The isInterestedInFileDrag function (line 150) indicates file drag events is true, thus allowing file drag to load the music. Having loadURL and filesDropped (line 156) empowers users to effortlessly incorporate audio files into the DJ application through a simple drag-and-drop action. Specifically, when a user drops a single file onto the application, filesDropped orchestrates a sequence of actions to ensure a streamlined integration process. The initial step involves the transformation of the file path into a JUCE URL object, encapsulating the precise location of the dropped audio file. This URL serves as a comprehensive representation of the audio file's source, preparing it for subsequent processing within the application. Following this, the loadURL function is invoked on both the player and waveformDisplay objects in the loadMusicFileToApplication function (line 211).

```
116     // Load file
117     if (button == &loadButton) {
118         DBG("Load button was clicked.");
119         auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
120         fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
121             {
122                 auto musicFile = chooser.getResult();
123                 if (musicFile.exists()) {
124                     DBG("Chosen file exists: " << musicFile.getFullPathName());
125 
126                     // Construct a URL from the music file
127                     juce::URL musicUrl(musicFile);
128                     DBG("Constructed URL: " << musicUrl.toString(true));
129 
130                     // Load the URL into the player and waveform display
131                     player->loadURL(musicUrl);
132                     waveformDisplay.loadURL(musicUrl);
133 
134                     // Update the musicNameLabel content
135                     updateLabels(musicUrl);
136 
137                     DBG("Music Name Label Text: " << musicNameLabel.getText());
138                 }
139             });
140     }
```

Figure 1A.4

Finally, figure 1A.4 shows the loadButton within the buttonClicked function. The code begins by checking if the clicked button corresponds to the designated "loadButton.". Upon detecting the "loadButton" click, a file chooser dialog is launched asynchronously. This dialog empowers users to navigate and select an audio file from their system. The callback function within the asynchronous file chooser launch executes once the user selects a file. This ensures a responsive and non-blocking user interface. Then, the chosen file's existence is validated to ensure that subsequent actions are performed only when a valid file is selected. Following which, a JUCE URL object (musicUrl) is constructed using the selected audio file. This URL serves as a versatile representation of the audio file's source. The loadURL function is then invoked on both the audio player (player) and waveform display (waveformDisplay). This action loads the selected audio file into the associated components, preparing it for playback.

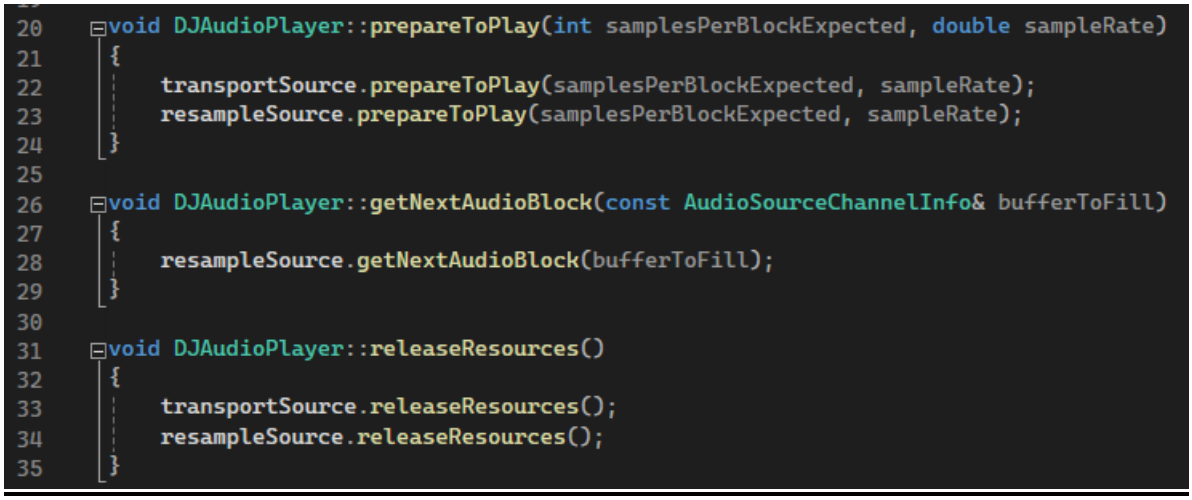
R1B: Able to play two tracks concurrently



```
77      /**
78       * First DJAudioPlayer using the format manager.
79       */
80      DJAudioPlayer player1{ formatManager };
81
82      /**
83       * First DeckGUI associated with player1, using format manager and thumbnail cache.
84       */
85      DeckGUI deckGUI1{ &player1, formatManager, thumbCache , true};
86
87      /**
88       * Second DJAudioPlayer using the same format manager.
89       */
90      DJAudioPlayer player2{ formatManager };
91
92      /**
93       * Second DeckGUI associated with player2, using format manager and thumbnail cache.
94       */
95      DeckGUI deckGUI2{ &player2, formatManager, thumbCache , false};
```

Figure 1B.1

To play two tracks concurrently, I have created two instances of DJAudioPlayer, player1, and player2, and also created two instances of DeckGUI, deckGUI1, and deckGUI2. These are responsible for handling audio playback. These DeckGUI instances are initialized with pointers to their respective DJAudioPlayer instances (&player1 and &player2), the formatManager for handling audio file formats, a thumbCache for thumbnail caching, and a boolean indicating whether it's associated with the first or second deck. Having two separate instances of DJAudioPlayer and DeckGUI allows us to play two audio tracks concurrently. Each DJAudioPlayer is associated with a DeckGUI, allowing for independent control and visualization of audio playback for each track.



```
20  void DJAudioPlayer::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
21  {
22      transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
23      resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
24  }
25
26  void DJAudioPlayer::getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
27  {
28      resampleSource.getNextAudioBlock(bufferToFill);
29  }
30
31  void DJAudioPlayer::releaseResources()
32  {
33      transportSource.releaseResources();
34      resampleSource.releaseResources();
35  }
```

Figure 1B.2

Following which, we have the **prepareToPlay** method, which is called by the audio device manager to allow the AudioSource to initialize and prepare for playback. In this implementation, it calls prepareToPlay on both the transportSource (line 22) and resampleSource (line 23). The parameters samplesPerBlockExpected and sampleRate is provided by the audio device manager and are used to configure the audio processing. The **getNextAudioBlock** method is called by the audio device manager to request the next block of audio samples for playback, it delegates the task to the resampleSource, which is responsible for handling the actual playback and resampling of the audio. The **releaseResources** method is called when the audio device manager is shutting down or switching to a different audio format. It calls releaseResources on both the transportSource and resampleSource to release any audio resources they may have acquired during playback. They work together to provide audio playback capabilities.

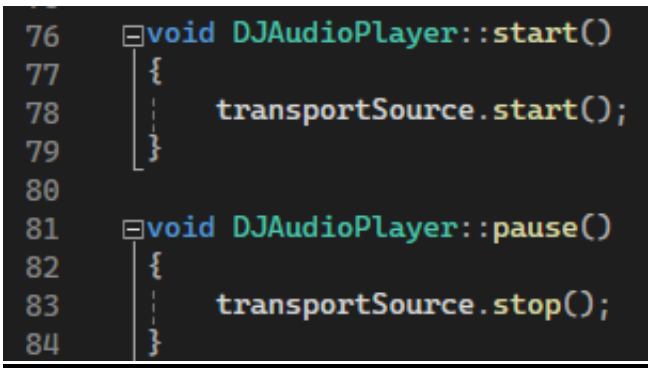


Figure 1B.3

Lastly, the implementation includes two methods, specifically start (at line 76) and pause (at line 81). The start method is utilized to commence the playback of an audio track using the transportSource from the current position. On the other hand, the pause method is invoked to cease the playback of an audio track. It halts the transportSource, pausing the playback at the current position and allowing for later resumption from the same point.

These methods contribute to the audio player's playback control mechanism. By incorporating the start and pause functionality, users or the application logic gain the capability to manage the playback state of the audio tracks associated with the DJAudioPlayer. With the existence of two created players, concurrent playback of two music tracks is achievable.

R1C: Tracks mixing by varying each of their volumes

```
47  void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
48  {
49
50      // ************
51      mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
52      // ************
53      mixerSource.addInputSource(&player1, false);
54      mixerSource.addInputSource(&player2, false);
55  }
56
57  void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
58  {
59      mixerSource.getNextAudioBlock(bufferToFill);
60  }
61
62  void MainComponent::releaseResources()
63  {
64      // This will be called when the audio device stops, or when it is being
65      // restarted due to a setting change.
66      player1.releaseResources();
67      player2.releaseResources();
68      mixerSource.releaseResources();
69      // Clear memory when stopped
70      mixerSource.removeAllInputs();
71  }
```

Figure 1C.1

Following the establishment of the two players as illustrated in Figure 1B.1, the **prepareToPlay** method in the MainComponent is invoked to ready the audio sources for playback. The mixerSource is prepared for playback by executing its prepareToPlay method with the provided parameters. Both instances of DJAudioPlayer (player1 and player2) are introduced as input sources to the mixer using mixerSource.addInputSource(&player1, false) and mixerSource.addInputSource(&player2, false). The second parameter (false) in addInputSource implies that the input sources are not employed as references, permitting independent control of their volumes.

The **getNextAudioBlock** method is summoned by the audio device manager to request the next block of mixed audio samples for playback. The mixerSource is entrusted with the responsibility of delivering the mixed audio block by invoking its getNextAudioBlock method with the provided bufferToFill.

Subsequently, the **releaseResource** method is invoked when the audio device is halting or undergoing a restart due to a setting alteration. It liberates resources for both player1 and player2 by triggering their respective releaseResources methods. The mixerSource is also discharged using its releaseResources method, and all input sources are eradicated via mixerSource.removeAllInputs() to clear memory.

```
48  void DJAudioPlayer::setGain(double gain)
49  {
50      if (gain < 0 || gain > 1.0)
51      {
52          DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
53      }
54      else
55      {
56          transportSource.setGain(gain);
57      }
58  }
```

Figure 1C.2

In figure 1C.2, the **setGain** method takes a double parameter gain, which is expected to be between 0 and 1, inclusive. The if condition (line 50) checks whether the provided gain is within the valid range. If the gain is outside the valid range, a debug message is printed indicating that the gain should be between 0 and 1. If the gain is within the valid range, the transportSource.setGain(gain) method is called to set the volume of the audio track. This method provides a mechanism to control the volume of the associated audio track by adjusting its gain. The valid gain range is typically between 0 (silent) and 1 (full volume). If an invalid gain is provided, a debug message is printed for debugging purposes.

```
131 void DeckGUI::sliderValueChanged(Slider* slider) {
132     if (slider == &volSlider) {
133         DBG("vol slider moved." << volSlider.getValue());
134         player->setGain(slider->getValue());
135     }
```

Figure 1C.3

In the **sliderValueChanged** function, if the volume slider undergoes a change, a debug message (DBG) is generated, indicating the movement of the volume slider and displaying its current value. The line player->setGain(slider->getValue()) implies that the setGain method of a player is invoked, utilizing the new value of the volume slider. This signifies an adjustment in the volume of the corresponding audio track based on the slider's current value. With the incorporation of these code segments in Figures 1B.1, 1C.1, 1C.2, and 1C.3, users gain the capability to mix tracks by modifying their volumes.

R1D: Speeding and slowing down the tracks

```
60    // changed the ratio to be <= 0 to prevent breakpoint activation
61    void DJAudioPlayer::setSpeed(double ratio)
62    {
63        if (ratio <= 0 || ratio > 5.0) {
64            DBG("DJAudioPlayer::setSpeed ratio should be between 0 and 5");
65        }
66        else {
67            resampleSource.setResamplingRatio(ratio);
68        }
69    }
```

Figure 1D.1

To speed up and slow down the track, a set speed method is created. It first take in a double parameter ratio, and set the speed to range of 0 (exclusive) (if speed is 0 then song would be idle) to 5 (inclusive). If the ratio is outside the valid range, a debug message is printed, indicating that the ratio should be between 0 and 5. If the ratio is within the valid range, the resampling ratio would be adjusted, thereby changing the playback speed of the audio track. This allow us to speed up the song to 5 times its original or slowing it down.

```
131   void DeckGUI::sliderValueChanged(Slider* slider) {
132       if (slider == &volSlider) {
133           DBG("vol slider moved." << volSlider.getValue());
134           player->setGain(slider->getValue());
135       }
136       if (slider == &speedSlider) {
137           DBG("speed slider moved." << speedSlider.getValue());
138           player->setSpeed(slider->getValue());
139       }
```

Figure 1D.2

With the setSpeed method in Figure 1D.1, the speed slider is implemented to adjust the song speed. Within the sliderValueChanged method in deckGUI, if the speedSlider is moved, there would be a debugging message stating the value moved, and the player would set the speed by getting the new value from the slider. Thus, with these methods, we are able to set the speed of the track we want by using the speedSlider.

R2A: GUI layout is significantly different

Figure 2A.1

As shown in Figure 2A.1, I have changed the position slider to a disk slider and adjustable on the waveform display, moreover, I changed the design of play pause and upload button.

WaveForm display:

Figure 2A.2

```
32  void WaveformDisplay::paint (juce::Graphics& g)
33  {
34      g.fillAll(getLookAndFeel().findColour(
35          juce::ResizableWindow::backgroundColourId));   // clear the background
36
37      g.setColour(juce::Colours::grey);
38      g.drawRect(getLocalBounds(), 1);   // draw an outline around the component
39
40      if (fileLoaded)
41      {
42          if (isDeck1)
43              g.setColour(Colours::orange);
44          else
45              g.setColour(Colours::deepskyblue);
46
47          audioThumb.drawChannel(g, getLocalBounds(), 0,
48              audioThumb.getTotalLength(), 0, 1.0f);
49      }
50      else
51      {
52          if (isDeck1)
53              g.setColour(Colours::orange.withAlpha(0.5f));
54          else
55              g.setColour(Colours::deepskyblue.withAlpha(0.7f).brighter());
56
57          g.setFont(Font(20.0f).withTypefaceStyle("Bold"));
58
59          g.drawText("Load Music Files :3", getLocalBounds(),
60              juce::Justification::centred, true);   // draw some placeholder text
61      }
62  }
```

Figure 2A.3

```cpp
void DeckGUI::resized()
{
    float height = getHeight() * 0.1;
    float width = getWidth() * 0.2;

    musicNameLabel.setBounds(0, height *0.25, getWidth(), height);

    // set where the waveformdisplay is located
    posSlider.setBounds(0, height * 1.5, getWidth(), height * 2);
    waveformDisplay.setBounds(0, height * 1.5, getWidth(), height * 2);

    if (isDeck1)
    {
        djSlider.setBounds(width * 2.45, height * 3.9, width * 2.15, width * 2.15);
        loadBtn.setBounds(width * 0.7, height * 7.6, height, height);
        playBtn.setBounds(width * 1.2, height * 7.3, height * 1.5, height * 1.5);
        pauseBtn.setBounds(width * 1.9, height * 7.6, height, height);
        loopToggleBtn.setBounds(width * 1.1, height * 8.9, width, height);
        volSlider.setBounds(width * 0.1, height * 3.75, width * 0.4, height * 6);
        speedSlider.setBounds(width * 0.88, height * 4.5, width * 1.2, width * 0.8);
    }
    else
    {
        djSlider.setBounds(width * 0.4, height * 3.9, width * 2.15, width * 2.15);
        loadBtn.setBounds(width * 2.7, height * 7.6, height, height);
        playBtn.setBounds(width * 3.2, height * 7.3, height*1.5, height*1.5);
        pauseBtn.setBounds(width * 3.9, height * 7.6, height, height);
        loopToggleBtn.setBounds(width * 3.1, height * 8.9, width, height);
        volSlider.setBounds(width * 4.5, height * 3.75, width * 0.4, height * 6);
        speedSlider.setBounds(width * 2.88, height * 4.5, width*1.2, width *0.8);
    }
}
```

Figure 2A.4

```cpp
void DeckGUI::sliderValueChanged(Slider* slider) {
    if (slider == &volSlider) {
        DBG("vol slider moved." << volSlider.getValue());
        player->setGain(slider->getValue());
    }
    if (slider == &speedSlider) {
        DBG("speed slider moved." << speedSlider.getValue());
        player->setSpeed(slider->getValue());
    }
    if (slider == &posSlider) {
        DBG("pos slider moved." << posSlider.getValue());
        player->setPositionRelative(slider->getValue());
    }
    if (slider == &djSlider) {
        player->setPositionRelative(slider->getValue());
    }
}
```

Figure 2A.5

```
92    // Set the playback position based on a relative value
93    void DJAudioPlayer::setPositionRelative(double pos)
94    {
95        if (pos < 0 || pos > 1.0)
96        {
97            // Print a warning if the relative position is out of bounds
98            std::cout << "DJAudioPlayer::setPositionRelative: Relative pos should be between 0 and 1." << std::endl;
99        }
100       else
101       {
102           // Convert relative position to absolute seconds and set the position
103           double posInSecs = transportSource.getLengthInSeconds() * pos;
104           setPosition(posInSecs);
105       }
106   }
```

Figure 2A.6

For the visualization of loaded files in the waveform display, I have implemented distinct color schemes to convey information to the user. If a file is loaded for Deck 1 (line 42), the drawing color is set to orange, while for Deck 2 (line 44), it is set to deepskyblue. In scenarios where no file is loaded, I've employed translucent colors for added visual cues. For Deck 1, a translucent orange color is used, providing a subtle indication, and for Deck 2, a translucent, brighter deep sky blue color is chosen. In such cases, a prompt is also outputted to inform users that no music is loaded into the deck (see Figure 2A.1). Within the resized function, a deliberate decision was made to overlap the posSlider (line 68) and the waveformDisplay (line 69). As seen in Figure 2A.5, the posSlider make use of setPositionRelative in Figure 2A.6 to change the playback position.

Sliders:

```
14  //==============================================================================
15  DeckGUI::DeckGUI(DJAudioPlayer* _player,
16                   AudioFormatManager & formatManagerToUse,
17                   AudioThumbnailCache & cacheToUse,
18                   bool isDeck1)
19      : player(_player),
20        waveformDisplay(formatManagerToUse, cacheToUse, isDeck1),
21        isDeck1(isDeck1)
22  {
23      // Output the music name and waveform
24      initializeComponents();
25
26      // Output the buttons
27      initializeButtons();
28
29      // slider
30      initializeSliders();
31
32      // configure vol speed position and dj slider
33      configureSlider(volSlider, 0.0, 1.0, 0.5, Slider::LinearBarVertical, Slider::NoTextBox, true, 0.5);
34      configureSlider(speedSlider, 0.0, 5.0, 1.0, Slider::Rotary, Slider::TextBoxBelow, true, 1.0);
35      configureSlider(posSlider, 0.0, 1.0, 0.0, Slider::LinearBar, Slider::NoTextBox, false, 0.0);
36      configureSlider(djSlider, 0.0, 1.0, 0.0, Slider::Rotary, Slider::NoTextBox, false, 0.0);
37
38      // Listeners for the buttons
39      addAndMakeVisible(loopToggleBtn);
40      loopToggleBtn.addListener(this);
41
42      startTimer(100);
43  }
```

Figure 2A.7

```
271 void DeckGUI::initializeSliders()
272 {
273     // Volume Slider
274     addAndMakeVisible(volSlider);
275
276
277     // Speed Slider
278     addAndMakeVisible(speedSlider);
279     speedSlider.setNumDecimalPlacesToDisplay(2);
280
281     // Position Slider
282     addAndMakeVisible(posSlider);
283     posSlider.setColour(Slider::trackColourId, Colours::grey.withAlpha(0.3f));
284
285     // DJ Slider
286     addAndMakeVisible(djSlider);
287     djSlider.setLookAndFeel(&otherLookAndFeel);
288 }
289
290 void DeckGUI::configureSlider(Slider& slider, double minValue, double maxValue, double startValue,
291     Slider::SliderStyle style, Slider::TextEntryBoxPosition textBoxPos,
292     bool doubleClickReturnValue, double interval)
293 {
294     slider.setRange(minValue, maxValue);
295     slider.setValue(startValue);
296     slider.setSliderStyle(style);
297     slider.setTextBoxStyle(textBoxPos, false, 60, 20);
298     slider.setDoubleClickReturnValue(doubleClickReturnValue, interval);
299     slider.addListener(this);
300 }
```

Figure 2A.8

In figure 2A.4, 2A.5, 2A.6, it shows how my sliders are created. Within the constructor of DeckGUI, I have a initializeSliders function and a configureSlider function used by vol,

speed, pos and dj slider. For the initializeSliders function (Figure 2A.8), it initializes and sets up four sliders (volSlider, speedSlider, posSlider, and djSlider) within the DeckGUI component, each with its own specific properties. For volume slider, I set it to a LinearBarVertical, speed and dj slider set to a rotary slider and position slider as a linearBar.

Then, for configureSliders (line 290), it helps set the range of the slider from minValue to maxValue, set the initial value of the slider to startValue, set the visual style of the slider using the provided Slider::SliderStyle, set the style of the text box associated with the slider, specifies the position of the text box and an uneditable text box.The last two parameters (60 and 20) specify the width and height of the text box. With the setDoubleClickReturnValue(doubleClickReturnValue, interval) it return slider to initial value. Lastly, I add a listener to the sliders. Moreover, for the resized function (Figure 2A.4), I adjusted their respective position for the sliders in deck 1 and deck 2 and I set them to be mirroring each other.

Rotary Sliders:

Figure 2A.9

```
20  void OtherLookAndFeel::drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos,
21      float rotaryStartAngle, float rotaryEndAngle, Slider& slider)
22  {
23      auto angle = MathConstants<float>::pi;
24      slider.setRotaryParameters(angle, (angle * 2) + angle, true);
25
26
27      float dia = jmin(width, height);
28      float rad = dia * 0.5;
29      float focusX = x + width * 0.5;
30      float focusY = y + height * 0.5;
31      float ang = rotaryStartAngle + (sliderPos * (rotaryEndAngle - rotaryStartAngle));
32
33      AffineTransform rot = AffineTransform::rotation(ang).translated(focusX, focusY);
34
35      auto djImg = ImageCache::getFromMemory(BinaryData::disc_png, BinaryData::disc_pngSize);
36
37      g.addTransform(AffineTransform::rotation(rotAngle, focusX, focusY));
38
39      g.drawImageWithin(djImg, x, y, width, height, RectanglePlacement::centred, false);
40  }
41
42  void OtherLookAndFeel::rotateDisc(double ang)
43  {
44      rotAngle = ang;
45  }
```

Figure 2A.10

In figure 2A.7 and 2A.8, I created the djSlider and make used of OtherLookAndFeel to make a disc slider. In figure 2A.10, I created drawRotarySlider in the OtherLookAndFeel constructor. This method is an override of a virtual function from the LookAndFeel class and is called when a rotary slider needs to be drawn. First, I used auto angle = MathConstants<float>::pi to set angle to the constant value of pi. Following which, I set the rotary parameters. Then I calculate the diameter (line 27), radius (line 28), x-coordinate (line 29), y-coordinate (line 30), and the current angle (line 31) of the slider based on its position.

After that, I create an AffineTransform for rotation and translation based on the calculated angle. Finally, I use ImageCache::getFromMemory(BinaryData::disc_png, BinaryData::disc_pngSize) to load the disc image from memory and add a rotation transformation to the graphics context based on the rotAngle, which is separated from the slider's rotation. With drawImageWithin, I draw the disc image within the specified rectangular area. For the rotateDisc function, it updates the rotation angle of the disc. As seen in Figure 2A.5, the djSlider also make use of setPositionRelative in Figure 2A.6 to change the playback position.

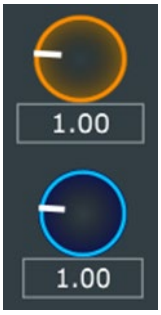
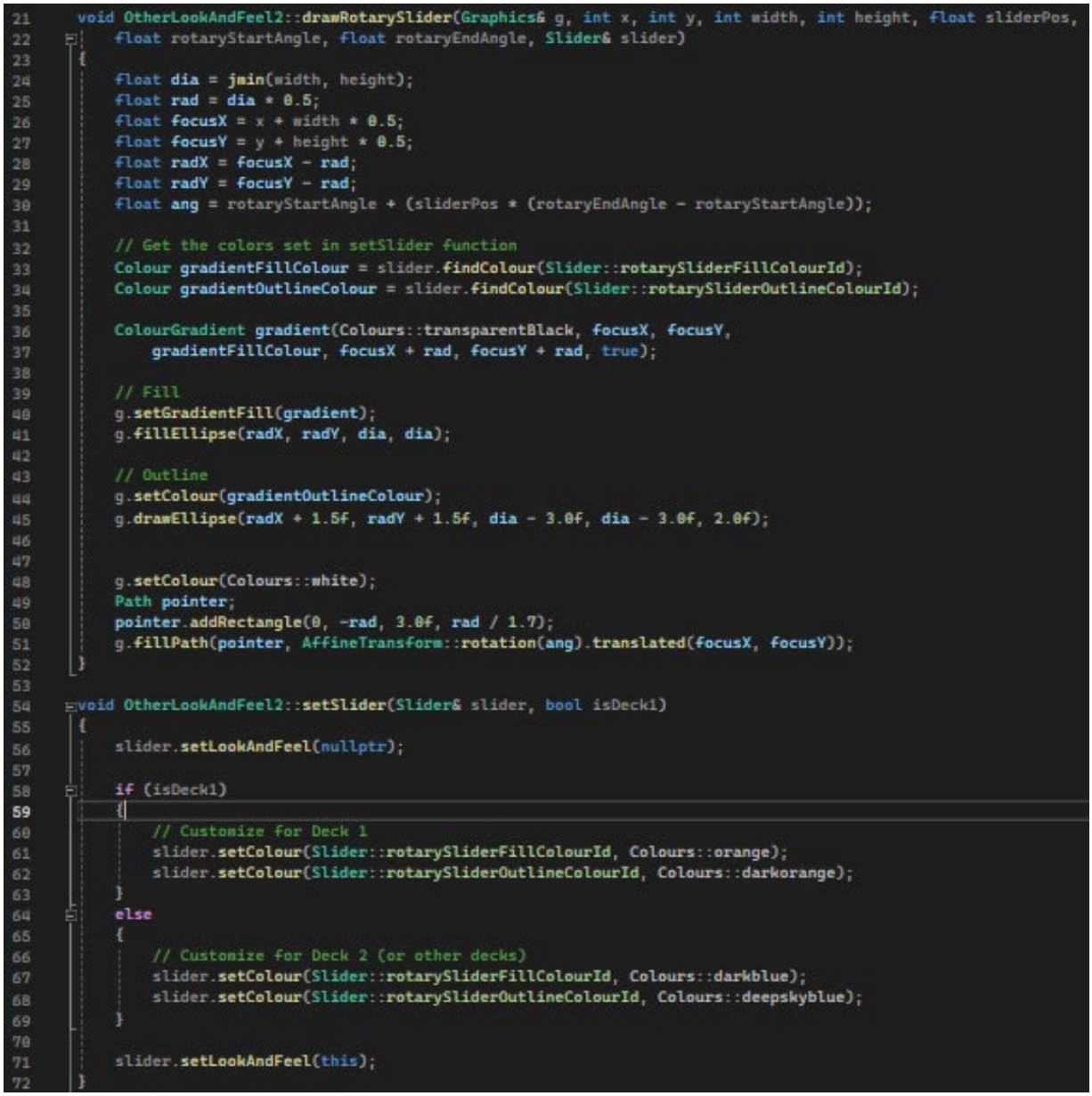


Figure 2A.11



```
void OtherLookAndFeel2::drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos,
    float rotaryStartAngle, float rotaryEndAngle, Slider& slider)
{
    float dia = jmin(width, height);
    float rad = dia * 0.5;
    float focusX = x + width * 0.5;
    float focusY = y + height * 0.5;
    float radX = focusX - rad;
    float radY = focusY - rad;
    float ang = rotaryStartAngle + (sliderPos * (rotaryEndAngle - rotaryStartAngle));

    // Get the colors set in setSlider function
    Colour gradientFillColour = slider.findColour(Slider::rotarySliderFillColourId);
    Colour gradientOutlineColour = slider.findColour(Slider::rotarySliderOutlineColourId);

    ColourGradient gradient(Colours::transparentBlack, focusX, focusY,
        gradientFillColour, focusX + rad, focusY + rad, true);

    // Fill
    g.setGradientFill(gradient);
    g.fillEllipse(radX, radY, dia, dia);

    // Outline
    g.setColour(gradientOutlineColour);
    g.drawEllipse(radX + 1.5f, radY + 1.5f, dia - 3.0f, dia - 3.0f, 2.0f);


    g.setColour(Colours::white);
    Path pointer;
    pointer.addRectangle(0, -rad, 3.0f, rad / 1.7);
    g.fillPath(pointer, AffineTransform::rotation(ang).translated(focusX, focusY));
}

void OtherLookAndFeel2::setSlider(Slider& slider, bool isDeck1)
{
    slider.setLookAndFeel(nullptr);

    if (isDeck1)
    {
        // Customize for Deck 1
        slider.setColour(Slider::rotarySliderFillColourId, Colours::orange);
        slider.setColour(Slider::rotarySliderOutlineColourId, Colours::darkorange);
    }
    else
    {
        // Customize for Deck 2 (or other decks)
        slider.setColour(Slider::rotarySliderFillColourId, Colours::darkblue);
        slider.setColour(Slider::rotarySliderOutlineColourId, Colours::deepskyblue);
    }

    slider.setLookAndFeel(this);
}
```

Figure 2A.12

For my speed slider, I created drawRotarySlider and setSlider functions in the OtherLookAndFeel2. In the drawRotarySlider function I firstly calculate the dimensions and position of the speed slider, following which, I get the colors set in setSlider function, create a gradient for the rotary slider fill, an outline for the slider, and lastly a pointer to indicate the change of the slider. Then for setSlider function, it set the colour for the slider and its outline for deck1 and deck 2 respectively. In figure 2A.4 and figure 2A.5, the speedSlider is initialised and the slider would make use of the setSpeed function in 1D.1 to adjust the speed of the music.

Play, pause, load button:

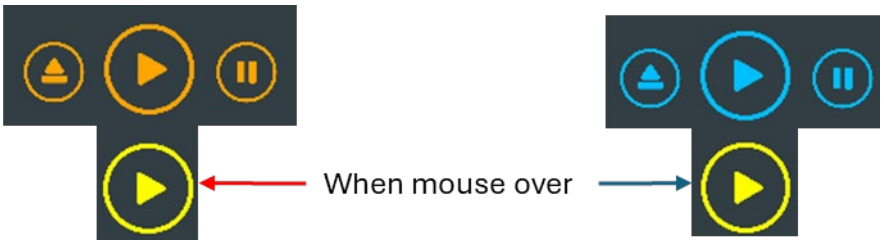


Figure 2A.13

For the play, pause and load button, I set their location by using setBounds (Figure 2A.4) for them respectively.

```
237  void DeckGUI::initializeButtons()
238  {
239      Image playImage = ImageCache::getFromMemory(BinaryData::play_png, BinaryData::play_pngSize);
240      Image pauseImage = ImageCache::getFromMemory(BinaryData::pause_png, BinaryData::pause_pngSize);
241      Image loadImage = ImageCache::getFromMemory(BinaryData::upload_png, BinaryData::upload_pngSize);
242
243      addAndMakeVisible(playBtn);
244      addAndMakeVisible(pauseBtn);
245      addAndMakeVisible(loadBtn);
246
247      playBtn.addListener(this);
248      pauseBtn.addListener(this);
249      loadBtn.addListener(this);
250
251      playBtn.setImages(true, true, true, playImage, 1.0f, isDeck1 ? Colours::orange : Colours::deepskyblue,
252          playImage, 1.0f, Colours::yellow,
253          playImage, 1.0f, Colours::orange);
254
255      pauseBtn.setImages(true, true, true, pauseImage, 1.0f, isDeck1 ? Colours::orange : Colours::deepskyblue,
256          pauseImage, 1.0f, Colours::yellow,
257          pauseImage, 1.0f, Colours::orange);
258
259      loadBtn.setImages(true, true, true, loadImage, 1.0f, isDeck1 ? Colours::orange : Colours::deepskyblue,
260          loadImage, 1.0f, Colours::yellow,
261          loadImage, 1.0f, Colours::orange);
262
263      musicNameLabel.setColour(Label::textColourId, isDeck1 ? Colours::orange : Colours::deepskyblue);
264      volSlider.setColour(Slider::trackColourId, isDeck1 ? Colours::orange : Colours::deepskyblue);
265
266      otherLookAndFeel2.setSlider(speedSlider, isDeck1);
267      speedLabel.setColour(Label::textColourId, isDeck1 ? Colours::orange : Colours::deepskyblue);
268      loopToggleBtn.setColour(ToggleButton::textColourId, isDeck1 ? Colours::orange : Colours::deepskyblue);
269  }
```

Figure 2A.14

In the initializeButtons funtion, three images (playImage, pauseImage, loadImage) are loaded from memory. These images represent the play, pause, and load buttons. I make them visible

by usiing addAndMakeVisible. Then, DeckGUI is set as a listener for each button where it will handle button click events. For each button (playBtn, pauseBtn, loadBtn), the setImages method is used to set different images for different button states (normal, mouse-over, and mouse-down). The colors for these images are configured based on whether it is for Deck1 or Deck2 using the isDeck1 flag. If it's for Deck1, the color is set to orange; otherwise, it's set to deepskyblue.

R2B: GUI code has at least one event listener that is not original

Loop toggle button:

Figure 2B.1

```
168   // Added music looping feature
169   void DeckGUI::timerCallback()
170   {
171       // Update the waveform display position
172       double currentPos = player->getPositionRelative();
173
174       if (currentPos > 0.0 && currentPos < 1.0)
175       {
176           waveformDisplay.setPositionRelative(currentPos);
177           posSlider.setValue(currentPos);
178           djSlider.setValue(currentPos);
179
180           auto angle = currentPos * 360.0;
181
182           otherLookAndFeel.rotateDisc(angle);
183
184           repaint();
185       }
186       // Check if the audio playback has reached the end
187       if (player->getPositionRelative() >= 1)
188       {
189           // Set position back to the start
190           player->setPositionRelative(0);
191
192           // Check if loop toggle button is enabled
193           bool isLoopEnabled = loopToggleBtn.getToggleState();
194           if (isLoopEnabled)
195           {
196               DBG("Loop button is enabled.");
197               // Start playing the audio file
198               player->start();
199           }
200           else
201           {
202               DBG("Loop button is disabled.");
203               // Stop the audio file
204               player->pause();
205           }
206       }
207
208   }
```

Figure 2B.2

For the event listener that is not the original, I have implemented the loop toggle button and the disc slider which is mentioned at the additional features part. In Figure 2A.7, it shows how I initilise the loopToggleBtn. Figure 2A.4 is the code on how I set its position for deck1 and 2 respectively. In figure 2B.2, it is shown how my loop toggle button works (line 193), if the loop button is enabled, it would start playing the file, else it would pause if the song have reach the end.

R3: New feature inspired by a real DJ program.

Figure 3.1

Source: https://youdj.online/

YOUDJ:

YouDJ is a user-friendly DJ software designed for both beginners and experienced users. With features like AutoSync and AutoBPM, it assists users in seamless music mixing, even without prior DJing knowledge. The software offers essential DJ tools such as virtual turntables, sound effects, a sampler with 80 built-in samples, and realistic vinyl scratching. Advanced features include auto beat synchronization, keylock, seamless loops, playlist automix, hotcues, a 3-band equalizer, and more. YouDJ comes preloaded with 3000 songs from emerging artists, and users can play music from various sources, including YouTube, local MP3 files, Beatport, Beatsource, Google Drive, Dropbox, and iTunes. Available on desktop, mobile, and tablets, YouDJ is the only DJ software accessible on all platforms, including iOS and Android.

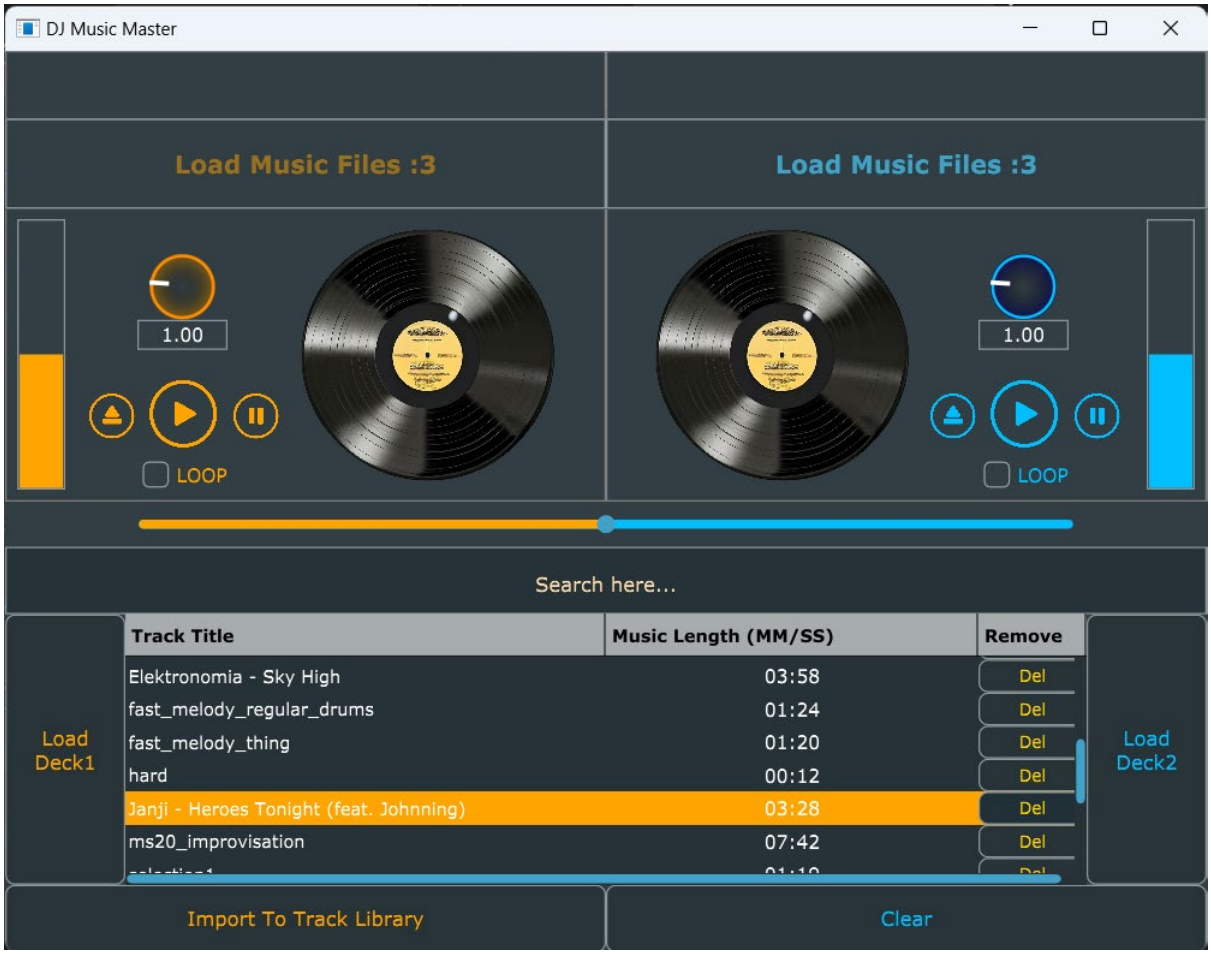


Figure 3.2

Figure 3.2 is my dj application where some of the ideas of the features come from Figure 3.1.

The reference of my online DJ program is YouDJ where it have some of the features I like and this is where most of my ideas of the features came from. I referred to this and created a column of music length, disc slider, crossfader, some similarities in colours and the song title display.

Feature 1: Crossfader

Figure 3.3

For my additional features, I firstly created the crossfade slider as seen in Figure 3.3. The crossfader is a norm in almost all the dj applications. It help DJs seamlessly alternates between two or more audio sources.

```
90   // Setting the crossfader
91   void MainComponent::setupCrossFadeSlider()
92   {
93       addAndMakeVisible(crossFadeSlider);
94       crossFadeSlider.setSliderStyle(Slider::LinearHorizontal);
95       crossFadeSlider.setTextBoxStyle(Slider::NoTextBox, false, 0, 0);
96       crossFadeSlider.setRange(0.0, 1.0, 0.01); // Range from 0 to 1 with a step of 0.01
97       crossFadeSlider.setValue(0.5); // Set initial value to center position
98       crossFadeSlider.setDoubleClickReturnValue(true, 0.5);
99       crossFadeSlider.setColour(Slider::trackColourId, juce::Colours::orange);
100      crossFadeSlider.setColour(Slider::backgroundColourId, juce::Colours::deepskyblue);
101      crossFadeSlider.addListener(this);
102  }
103
104  // Handling slider value changes
105  void MainComponent::sliderValueChanged(Slider* slider)
106  {
107      DBG("Slider Value: " << slider->getValue());
108
109      // Adjust volume levels based on slider position
110      float sliderValue = slider->getValue();
111      double gain1 = 1.0 - sliderValue; // Gain for deck 1
112      double gain2 = sliderValue;       // Gain for deck 2
113
114      player1.setGain(gain1);
115      player2.setGain(gain2);
116  }
```

Figure 3.4

In Figure 3.4, it shows how I create the crossfader. In the setupCrossFaderSlider method, I created the crossFadeSlider (line 93). The crossFadeSlider is instantiated with a linear horizontal style (line 94). The textBoxStyle is set to NoTextBox (line 95). Range of slider from 0.0 to 1.0 with a step of 0.01 (line 96) and a default center of 0.5 (line 97). Also I make use of setDoubleClickReturnValue for the slider to go back to its default 0.5 by double clicking (line 98). Lastly, a listener is added for slider value changes.

For thesliderValueChanged method (line 105), it would be activated when the slider is moved, after that, the debugger would output the changed value (line 107) and we are able to adjust

the volume based on the slider position. Then, the double gain 1 and 2 are created to set the volume. Lastly, we make use of the setGain function in Figure 1C.2 for players in deckGUI 1 and 2 respectively.

Feature 2: PlaylistComponent: Music length

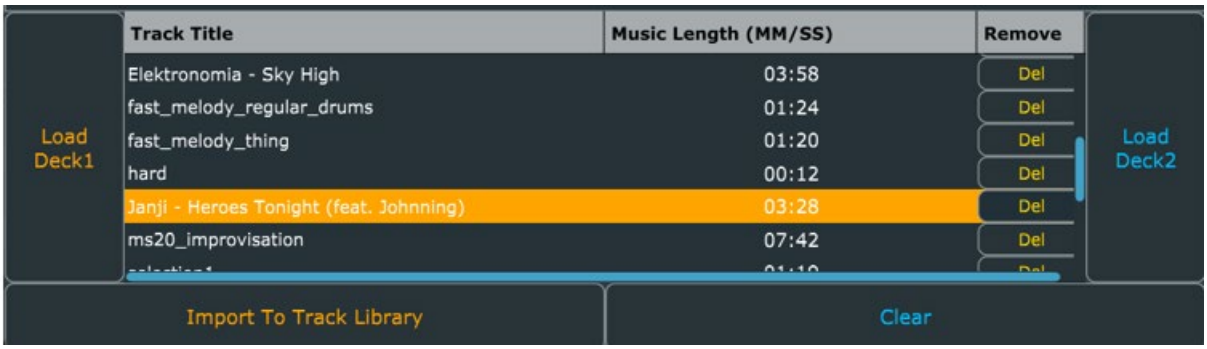


Figure 3.5

```
58      // Configure columns for the table component
59      tableComponent.getHeader().addColumn("Track Title", 1, 320);
60      tableComponent.getHeader().addColumn("Music Length (MM/SS)", 2, 150);
61      tableComponent.getHeader().addColumn("Remove", 3, 150);
62
63      // Set the model for the table component
64      tableComponent.setModel(this);
65
66      // Make the table component visible
67      addAndMakeVisible(tableComponent);
```

```
148 juce::Component* PlaylistComponent::refreshComponentForCell(int row,
149     int columnId,
150     bool isRowSelected,
151     juce::Component* componentToUpdate)
152 {
153     if (columnId == 2)
154     {
155         // Check if the rowNumber is within the valid range
156         if (row >= 0 && row < soundTrack.size())
157         {
158             // Get the music URL for the current row
159             juce::URL musicURL = soundTrack[row].MusicUrl;
160
161             // Use the modified getMusicLength function to get the minutes and seconds
162             std::pair<int, int> musicLength = getMusicLength(musicURL);
163
164             int minutes = musicLength.first;
165             int seconds = musicLength.second;
166
167             // Attempt to cast the existing componentToUpdate to a Label
168             juce::Label* durationLabel = dynamic_cast<juce::Label*>(componentToUpdate);
169
170             if (durationLabel == nullptr)
171             {
172                 // If componentToUpdate is not a Label, create a new Label
173                 durationLabel = new juce::Label{};
174                 durationLabel->setJustificationType(juce::Justification::centred);
175                 durationLabel->setColour(juce::Label::textColourId, juce::Colours::white);
176                 durationLabel->setColour(juce::Label::backgroundColourId, juce::Colours::transparentWhite);
177
178                 // Set the label as the componentToUpdate
179                 componentToUpdate = durationLabel;
180             }
181
182             // Set the label's text to the minutes and seconds
183             durationLabel->setText(juce::String::formatted("%02d:%02d", minutes, seconds),
184                 juce::dontSendNotification); // tell the system not to trigger any listeners or observers associated with the change
185         }
186     }
```

Figure 3.6

Figure 3.7

```
397  // make use of url to fet the minutes and seconds of each files
398  std::pair<int, int> PlaylistComponent::getMusicLength(const juce::URL& musicURL)
399  {
400      // Create an AudioFormatReader for each audio track URL
401      std::unique_ptr<juce::AudioFormatReader> reader(formatManager.createReaderFor(musicURL.createInputStream(false)));
402
403      if (reader != nullptr) {
404          // Get the length of the audio track in seconds
405          double audioLengthInSeconds = reader->lengthInSamples / static_cast<double>(reader->sampleRate);
406
407          // Convert to minutes and seconds
408          int minutes = static_cast<int>(audioLengthInSeconds / 60);
409          int seconds = static_cast<int>(audioLengthInSeconds - minutes * 60);
410
411          // Return a pair of minutes and seconds
412          return std::make_pair(minutes, seconds);
413      }
414      else {
415          // Return default value if an error occurs
416          return std::make_pair(0, 0);
417      }
418  }
```

Figure 3.8

For the PlaylistComponent I added a Music length column (line 60) where it would find the length of the music in minutes and seconds. In Figure 3.6, I added a column of music length so that we would be able to display length of music in minutes and seconds at that column. To be able to display the length, I created the getMusicLength function (Figure 3.8 line 398) where it make use of the musicURL to calculate the music length. Firstly, I created an reader for each musicURL by using the std::unique_ptr constructor. Next, it check if the reader is a null pointer (line 403) to ensure reader is a valid object before accessing members. If reader is not null, it calculates the music length by lengthInSamples which gives the total number of samples in the audio track divide by sampleRate that gives the number of samples per second. This gives the total duration of the audio track in seconds. Then, I convert them to whole minutes and seconds as a integer. Finally the duration is returned as a pair in minutes and seconds. However, if reader is null, they would return 0 and 0 which would be displayed as 0 minutes and 0 seconds.

After returning the duration, if column id is 2 (which is the music length) in the refreshComponentForCell, and if the row number is within the valid range of the soundTrack, it will get the musicURL for the current row. Then, I would get the music length from getMusicLength with the repective musicURL and setting first and second music length as minutes and seconds. After that, I attempts to cast componentToUpdate to a juce::Label using dynamic_cast. If the cast is successful (durationLabel != nullptr), it updates the existing label; otherwise, it creates a new juce::Label and configures the label's justification and colors. Lastly, I set the label's text to the formatted minutes and seconds.

Feature 3: Clear button

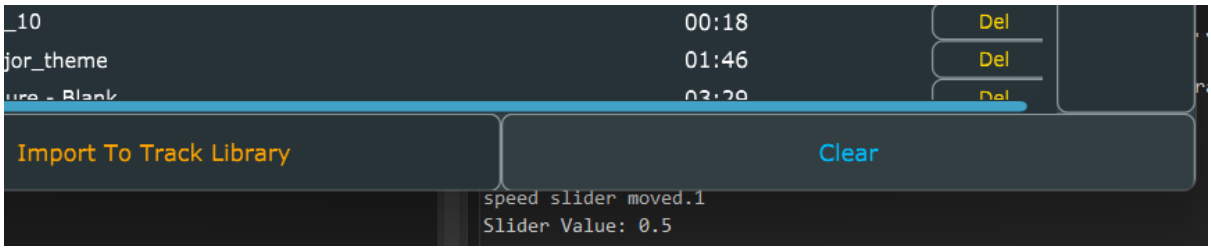


Figure 3.9

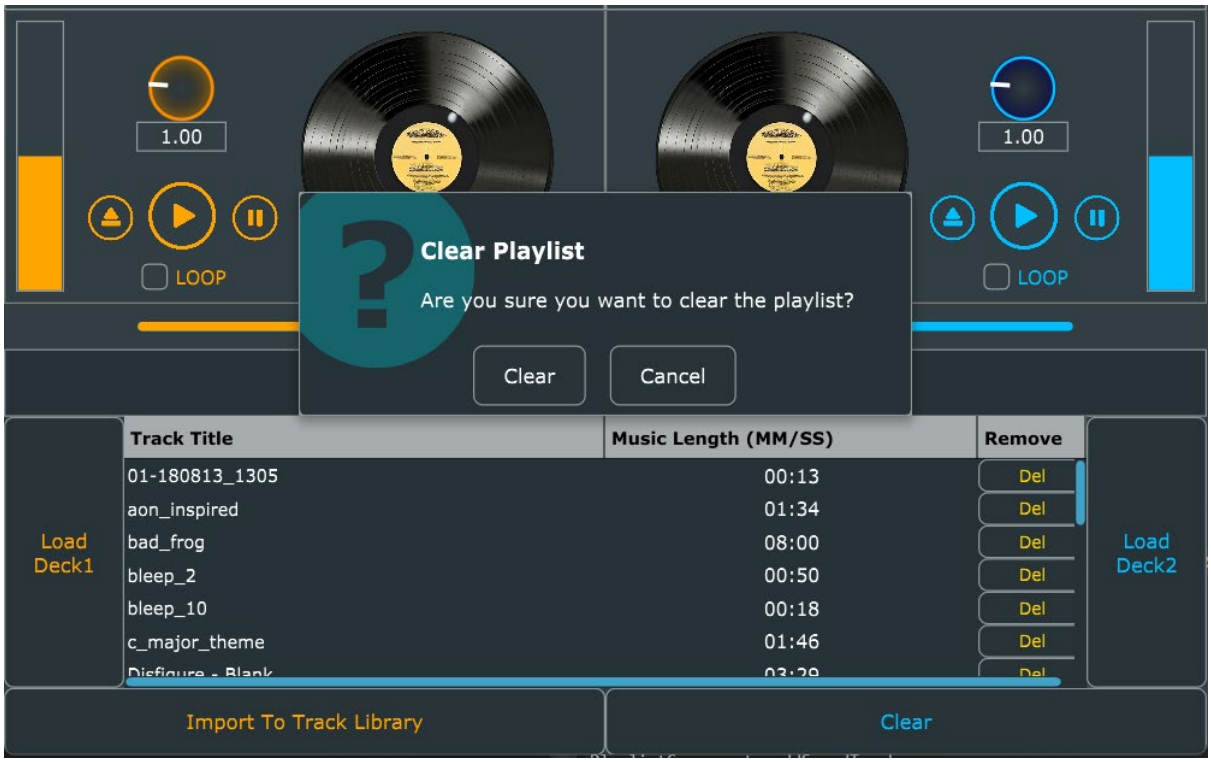


Figure 3.10

I have also implemented a Clear playlist button which would clear all the music within the playlist component if pressed, which would benefit the user when they want to clear the whole playlist. When the clear button is clicked, there would be a pop up box to confirm clearing the songs (Figure 3.10).

```
244     else if (button == &clearPlaylistBtn)
245     {
246         DBG("PlaylistComponent::buttonClicked - Clear Playlist button was clicked");
247
248         // Display a confirmation dialog before clearing the playlist
249         juce::AlertWindow confirmDelete("Clear Playlist", "Are you sure you want to clear the playlist?", juce::AlertWindow::QuestionIcon);
250         confirmDelete.setUsingNativeTitleBar(true);
251         confirmDelete.addButton("Clear", 1); // If the "Clear" button is clicked, returns 1
252         confirmDelete.addButton("Cancel", 0); // If the "Cancel" button is clicked, returns 0
253
254         // define JUCE_MODAL_LOOPS_PERMITTED=1 at "defined at preprocessor defination"
255         // For runModalLoop()
256         // Run the modal loop to wait for user input in the confirmation dialog
257         int result = confirmDelete.runModalLoop();
258
259         if (result == 1) // "Clear" button clicked
260         {
261             // Clear the playlist if confirmed
262             clearPlaylist();
263         }
264     }
```

Figure 3.11

In Figure 3.11, it shows the what will happen if the clearPlaylistBtn is clicked within the buttonClicked method. Firstly, the debugger would output that the button was clicked. Next, I make use of the juce AlertWindows to output what message I want to display on the alert/confirmation window. If the clear button is clicked on the alert window, it would return 1 and if cancel is clicked, it would return 0. If Clear is clicked and 1 is passed, it would run the clearPlaylist() function. Nothing would be done when Cancel is clicked. To add on, for me to used the runModalLoop() I have to define JUCE_MODAL_LOOPS_PERMITTED=1 at preprocessor defination as it is removed from the current juce library.

```
442 void PlaylistComponent::clearPlaylist()
443 {
444     // Clear the playlist by removing all sound tracks
445     soundTrack.clear();
446
447     // Clear the associated text file
448     std::ofstream playlistFile("CurrentPlaylist.txt", std::ios::trunc);
449
450     // Update the table after clearing the playlist
451     tableComponent.updateContent();
452 }
```

Figure 3.12

In Figure 3.12, I set the soundTrack.clear() to clear the whole playlist and update the tableComponent.

Feature 4: Save Current Playlist to txt file

This is a feature which I added, when users close the dj app with musicFiles still within the playList, it would write the musicName and musicURL onto a txt file where when the app is loaded the next time, the previous musics would be reloaded into the playlistComponent.

```
454  void PlaylistComponent::savePlaylistToFile()
455  {
456      try {
457          // Open the text file CurrentPlaylist.txt for writing
458          std::ofstream playlistFile("CurrentPlaylist.txt");
459
460          // Check if the file is open for writing
461          if (!playlistFile.is_open()) {
462              DBG("Error: Could not open playlist file for writing.");
463              return;
464          }
465
466          // Iterate through soundTrack and write each track's information to the file
467          for (const auto& track : soundTrack) {
468              playlistFile << track.MusicName << "," << track.MusicUrl << std::endl;
469          }
470
471          // Close the file
472          playlistFile.close();
473      }
474      catch (const std::exception& e) {
475          DBG("Exception caught while saving playlist: " << e.what());
476      }
477  }
```

Figure 3.13

I created the savePlaylistToFile function to write the loaded MusicName and MusicURL onto the txt file. Firstly, I used std::ofstream to get the file named CurrentPlaylist for writing, if it is not avaliable, it would create a empty txt. Next, if the txt file is currently open, it would output debugging message that the file is open and unavaliable for writing and would return. I also created a for loop to iterate through soundTrack and write the available music information onto the file where after writing, close the file.

```
76   PlaylistComponent::~PlaylistComponent()
77   {
78       // Save the current playlist state before closing the application
79       savePlaylistToFile();
80   }
```

Figure 3.14

The savePlaylistToFile method is placed in the destructor of PlaylistComponent, so then when the application stops, the information would be written onto the txt file.

```
469  void PlaylistComponent::readExistingPlaylistData()
470  {
471      try {
472          // Open the text file CurrentPlaylist.txt for reading
473          std::ifstream file("CurrentPlaylist.txt");
474          std::string str;
475
476          // Check if the file is open for reading
477          if (!file.is_open()) {
478              DBG("Error: Could not open playlist file for reading.");
479              return;
480          }
481
482          // Read each line and add it to soundTrack
483          while (std::getline(file, str)) {
484              std::istringstream iss(str);
485              std::string musicName, musicUrl;
486
487              // Extract MusicName and MusicUrl from each line
488              if (std::getline(std::getline(iss, musicName, ','), musicUrl)) {
489                  // Debug print for verification
490                  DBG(musicName);
491                  DBG(musicUrl);
492
493                  // Add a new SoundTrack to soundTrack
494                  soundTrack.emplace_back(SoundTrack{ musicName, musicUrl });
495                  tableComponent.updateContent();
496              }
497          }
498
499          // Close the file
500          file.close();
501      }
502      catch (const std::exception& e) {
503          DBG("Exception caught while reading playlist: " << e.what());
504      }
505  }
```

Figure 3.15

After the saving playlist to file, I created the read existing playlist method, where it first open the CurrentPlaylist.txt for reading, it will reach each line within the txt and get the music into the SoundTrack vector by using the musicName and musicUrl (line 505) and update the tableComponent (line 506). After this it would close the file.

```
15  //==============================================================================
16  PlaylistComponent::PlaylistComponent(AudioFormatManager& _formatManager,
17                                       DeckGUI* _deckGUI1,
18                                       DeckGUI* _deckGUI2)
19      : formatManager(_formatManager),
20      deckGUI1(_deckGUI1),
21      deckGUI2(_deckGUI2)
22  {
23      formatManager.registerBasicFormats();
24
25      readExistingPlaylistData();
```

Figure 3.16

In the playlistComponent constructor, I have shifted the formatManager.registerBasicFormats() from mainComponent to PlaylistComponent constructor as the PlaylistComponent is created first. Then, it would make use of the readExistingPlaylistData which would update the playlist.

Moreover, from Figure 3.12 clearPlaylist function, line 448 I created the ofstream for the txt file to clear the file when we empty the playlist. With all these codes, I am able to keep the previously loaded playlist without needing to find the location of music files.

Feature 5: Import to playlist and load to deck 1 or deck 2

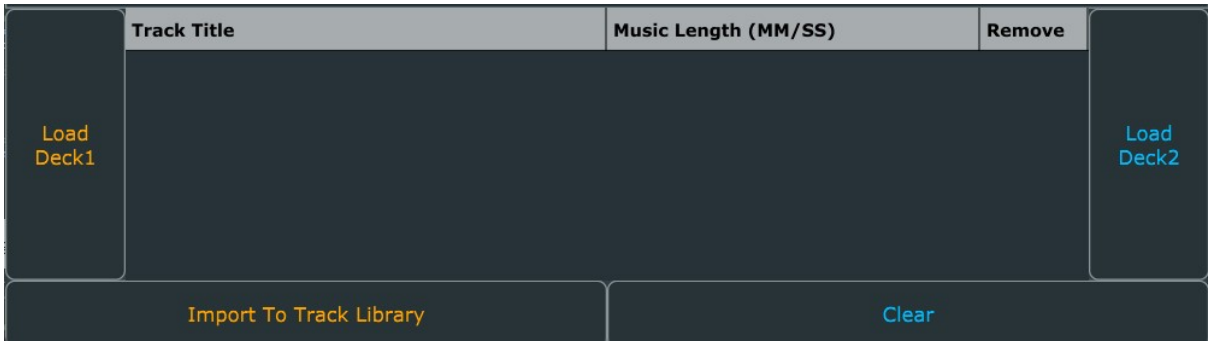


Figure 3.17

Now, I would be explaining on my importing to playlist and load to deck 1 or deck 2 features, when I clicked the Import button, I would be able to load multiple files by loading or dragging and load to the respective decks.

```
273  // Convert the music file into music name and url and add into the SoundTrack
274  void PlaylistComponent::addSoundTrack(const juce::File& musicFile)
275  {
276      DBG("PlaylistComponent::addSoundTrack");
277
278      // Get the music name and URL of the of the file.
279      juce::String musicName = musicFile.getFileNameWithoutExtension();
280      juce::String musicUrl = juce::URL{ musicFile }.toString(false);
281
282      // Check if the musicName is already in the playlist.
283      if (matchingMusicTitle(musicName))
284      {
285          DBG(musicName << " is already loaded");
286          return;  // Exit early if the track is already in the playlist.
287      }
288      // Create a new SoundTrack object and add it to the playlist.
289      SoundTrack newTrack{ musicName, musicUrl };
290      soundTrack.push_back(newTrack);
291
292      // Update the table content.
293      tableComponent.updateContent();
294  }
```

Figure 3.18

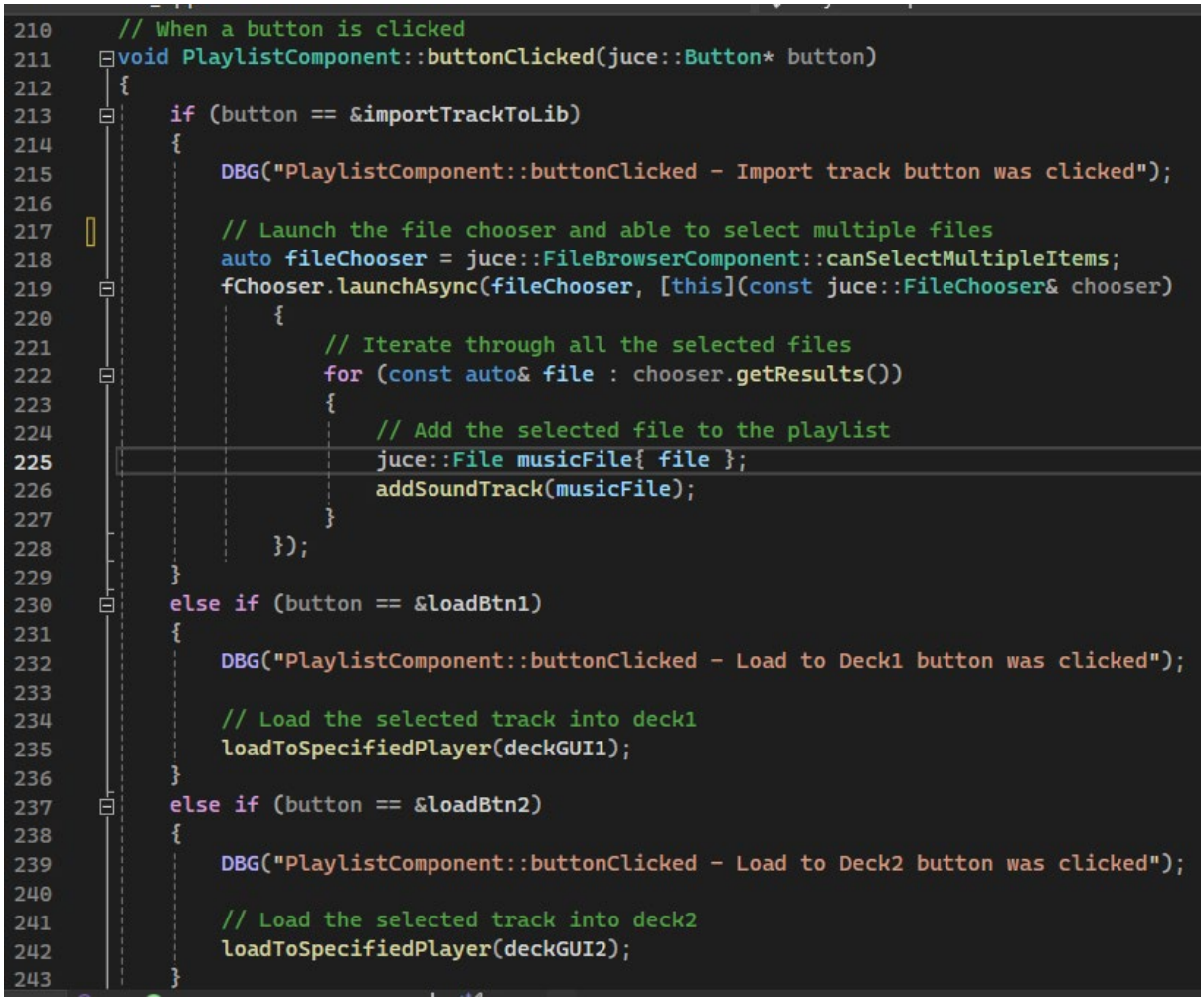
Initially, the addSoundTrack function retrieves the musicName and musicUrl from the File . It get the musicName by employing getFileNameWithoutExtension(). Subsequently, it generates the track's URL using juce::URL{ musicFile }.toString(false), with the false parameter indicating the usage of only the relative path, excluding the full path. The function then verifies if a track with an identical title is already present in the playlist via the

matchingMusicTitle function. If the track is not a duplicate, it constructs a new SoundTrack object with the acquired title and URL, appending the new track to the soundTrack vector. Finally, the function invokes tableComponent.updateContent() to dynamically refresh the table with the updated playlist. In essence, this function ensures the addition of tracks to the playlist without duplication while simultaneously maintaining the playlist table's up-to-date status.

```
305     // Load track into a specified player
306     void PlaylistComponent::loadToSpecifiedPlayer(DeckGUI* deckGUI)
307     {
308         DBG("PlaylistComponent::loadInPlayer");
309
310         try
311         {
312             // Get the selected row from the table component.
313             std::optional<int> selectedRow = tableComponent.getSelectedRow();
314
315             // Check if no row is selected
316             if (!selectedRow.has_value())
317             {
318                 throw std::runtime_error("Error: No row selected.");
319             }
320
321             // Check if the selected row is within the valid range of soundTrack.
322             if (selectedRow.value() < 0 || selectedRow.value() >= soundTrack.size())
323             {
324                 throw std::out_of_range("Error: Selected row is out of bounds.");
325             }
326             // Load the selected track into the deck.
327             deckGUI->loadMusicFileToApplication(soundTrack[selectedRow.value()].MusicUrl);
328         }
329         catch (const std::exception& e)
330         {
331             // Handle exceptions and print error messages.
332             DBG("Exception caught: " << e.what());
333         }
334     }
```

Figure 3.19

Also, I have the loadToSpecificPlayer function where it would take a pointer to a DeckGUI object (deckGUI) as a parameter. It is responsible for loading a selected track into the specified DeckGUI. Then I retrieve the selected row index from the tableComponent. It returns an optional integer to handle the case when no row is selected. If no exceptions are thrown, it loads the selected track into the specified DeckGUI by calling the loadMusicFileToApplication function (Figure 1A.3) of the DeckGUI object.



```
210     // When a button is clicked
211     void PlaylistComponent::buttonClicked(juce::Button* button)
212     {
213         if (button == &importTrackToLib)
214         {
215             DBG("PlaylistComponent::buttonClicked - Import track button was clicked");
216
217             // Launch the file chooser and able to select multiple files
218             auto fileChooser = juce::FileBrowserComponent::canSelectMultipleItems;
219             fChooser.launchAsync(fileChooser, [this](const juce::FileChooser& chooser)
220                 {
221                     // Iterate through all the selected files
222                     for (const auto& file : chooser.getResults())
223                     {
224                         // Add the selected file to the playlist
225                         juce::File musicFile{ file };
226                         addSoundTrack(musicFile);
227                     }
228                 });
229         }
230         else if (button == &loadBtn1)
231         {
232             DBG("PlaylistComponent::buttonClicked - Load to Deck1 button was clicked");
233
234             // Load the selected track into deck1
235             loadToSpecifiedPlayer(deckGUI1);
236         }
237         else if (button == &loadBtn2)
238         {
239             DBG("PlaylistComponent::buttonClicked - Load to Deck2 button was clicked");
240
241             // Load the selected track into deck2
242             loadToSpecifiedPlayer(deckGUI2);
243         }
```

Figure 3.20

Within the buttonClicked function (line 211), when the importTrackToLib is clicked, it launch the file chooser and allows users to select multiple music files, then for all the files added, it would iterate through all the added files and add them into the playlist by using the addSoundTrack function.

Which these functions, I am able to load the tracks I wanted into the playlist and the respective decks or players.

Feature 6: Display music title when music is loaded to deck

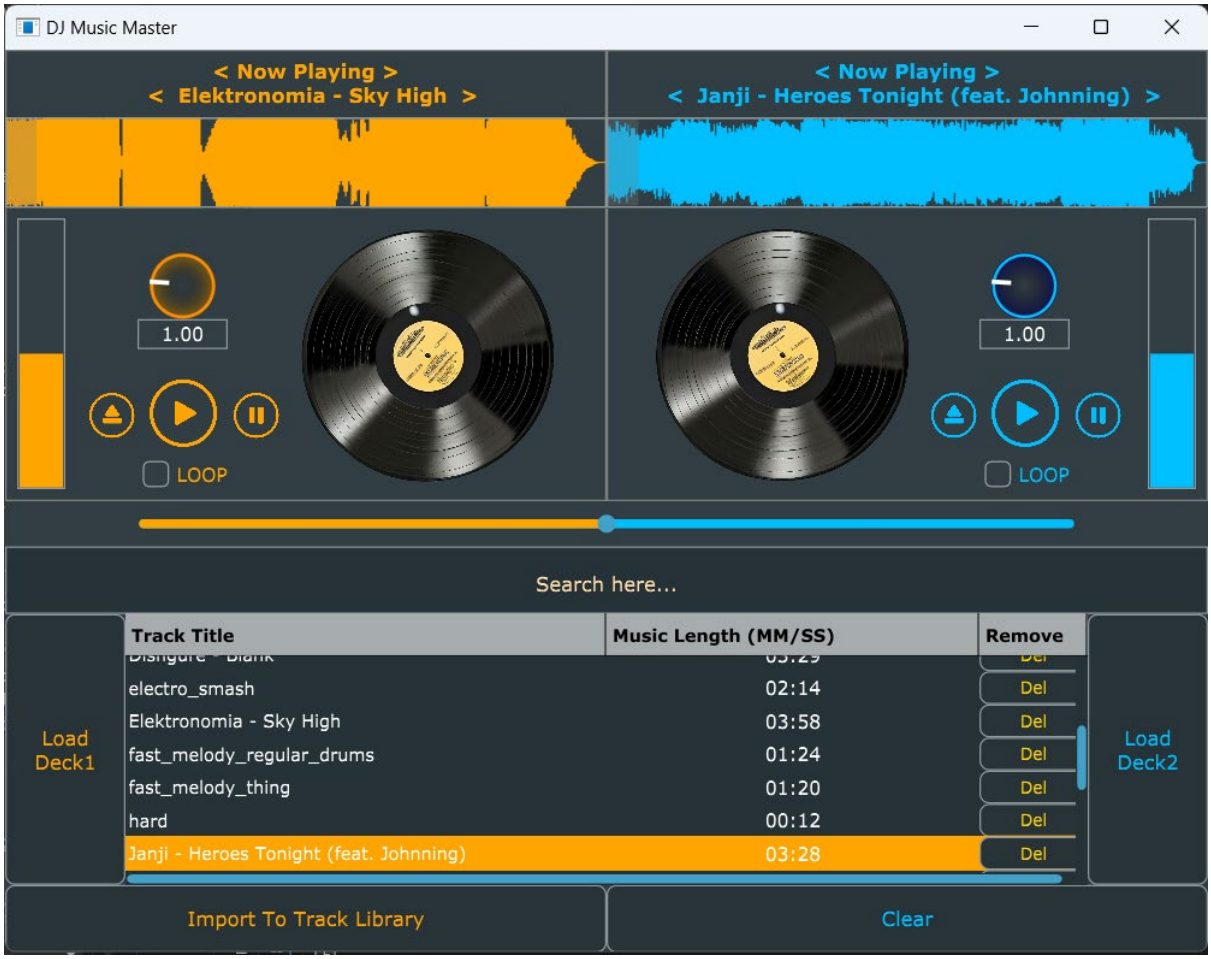


Figure 3.21

The feature I have is displaying the song title when the respective musics are being loaded into the decks.

```
220  void DeckGUI::updateLabels(const juce::URL& musicUrl)
221  {
222      // Update the musicNameLabel content
223      juce::File musicFile(musicUrl.getLocalFile());
224      String musicName = musicFile.getFileNameWithoutExtension();
225      musicNameLabel.setText("< Now Playing >\n  <  " + musicName + "  >", dontSendNotification);
226  }
```

Figure 3.22

Firstly I created the update label function where it takes in musicUrl as a parameter and use juce::File object from the juce::URL to represent the local file. Then, it extracts the name of the music file (excluding the extension) using getFileNameWithoutExtension. With this I would be able to output the music name whenever this function is called like in loadMusicFileToApplication() (Figure 1A.3) and Figure 1A.4.

Feature 7: Searchbox

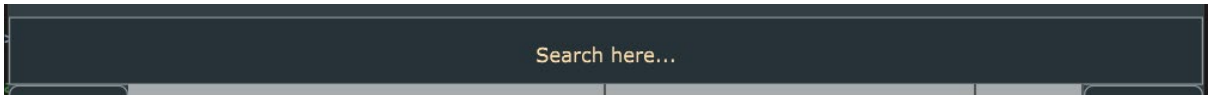


Figure 3.23

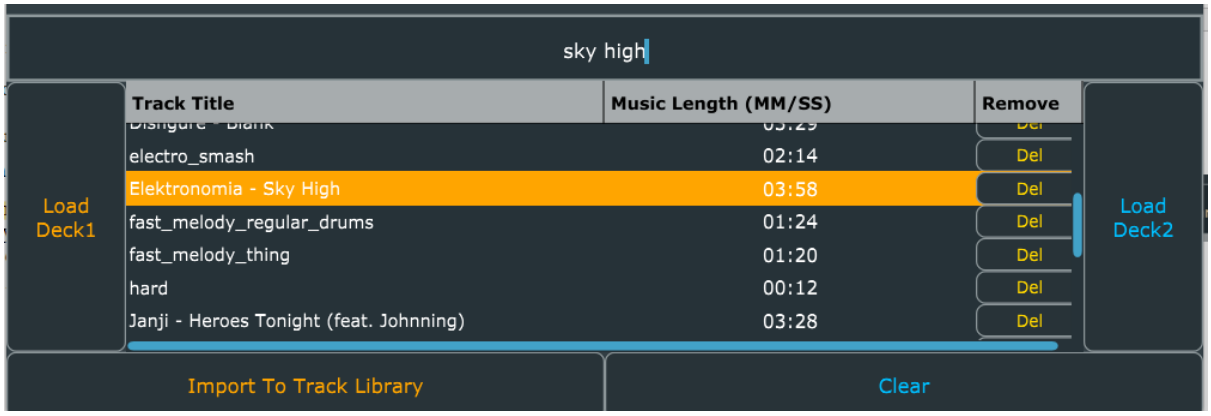


Figure 3.23 Continued

My 8th feature is a searchbox where I am able to search for a song with part of its name by pressing enter key, and does not matter if they are upper or lower case.

```cpp
bool PlaylistComponent::matchingMusicTitle(const juce::String& MusicName) const
{
    // Store the titles of imported audio tracks in a set
    std::unordered_set<juce::String> musicTitles;

    // Populate with existing track titles
    for (const auto& music : soundTrack)
    {
        musicTitles.insert(music.MusicName);
    }

    // Check if the imported music title exists
    return musicTitles.find(MusicName) != musicTitles.end();
}

// It updates the search results in the searhbox based on the search input
void PlaylistComponent::textEditorReturnKeyPressed(juce::TextEditor& editor)
{
    // Handle the "Enter" key press
    searchBoxInput = editor.getText().toLowerCase();  // Convert search input to lowercase
    int selectedRow = -1;  // Initialize selected row index

    // Check if the search bar contains any input
    if (!searchBoxInput.isEmpty())
    {
        // Iterate through the tracks in the playlist to find a matching track.
        for (int i = 0; i < soundTrack.size(); ++i)
        {
            // Convert track name to lowercase for case-insensitive comparison
            juce::String trackNameLowerCase = soundTrack[i].MusicName.toLowerCase();

            // Check if the track name contains the lowercase search input
            if (trackNameLowerCase.contains(searchBoxInput))
            {
                selectedRow = i;  // Record the index of the matching track
                break;  // Exit the loop when a match is found.
            }
        }
    }

    // Select the row if a match is found, otherwise deselect all rows.
    if (selectedRow != -1)
    {
        tableComponent.selectRow(selectedRow);  // Select the matching row in the table
    }
    else
    {
        tableComponent.deselectAllRows();  // Deselect all rows if no match is found
    }

    // Update the table to display the search results.
    tableComponent.updateContent();  // Refresh the table to reflect the selected/deselected rows
}
```

Figure 3.24

The matchingMusicTitle boolean is responsible for checking whether a music title already exists in the playlist. It uses a set to store the titles of imported audio tracks and checks if a given MusicName exists in this set. It initializes a set named musicTitles to store unique music titles. Then, tt populates the set with existing track titles from the soundTrack vector.

Lastly, it uses the find function of the set to check if the given MusicName exists in the set. The function returns true if the title exists, and false otherwise.

The textEditorReturnKeyPressed function handles the enter key pressed in the searchbox. Firstly, it gets the search input from the text editor, converts it to lowercase, and stores it in searchBoxInput (line 355). Next, it iterates through the tracks in the playlist (line 362), comparing the lowercase version of track names with the lowercase search input. If a match is found (line 368), it records the index of the matching track (line 370). Then, it selects the row if a match is found (line 379), and if no match is found, it deselects all rows in the table (line 383). Lastly, it calls tableComponent.updateContent() to refresh the table, reflecting the selected/deselected rows based on the search results (line 387).

This allow users to search the specific music title that they want to find in the playlist instead of looking through all the songs by scrolling.

Summary:

In conclusion, I have successfully designed and implemented a feature-rich DJ application for my OOP project that provides a seamless and intuitive experience for users. From loading and managing music files to dynamically controlling playback and applying crossfades, my application empowers DJs with the tools they need to curate immersive musical experiences. The application facilitates efficient playlist management, allowing users to import, organize, and save playlists effortlessly. The ability to load tracks into specific decks, complete with real-time updates, enhances the DJ's control over the mixing experience. The crossfader functionality enables smooth transitions between tracks, enhancing the fluidity and cohesion of DJ sets. The flexibility to adjust crossfade parameters in real-time provides DJs with creative freedom in shaping the auditory journey. The inclusion of a search functionality in the playlist simplifies track discovery, making it easy for DJs to find and load tracks on-the-fly. The responsive table display ensures clear visibility of track information and simplifies selection. Overall, I had a joyful experience working on this project.

References:

Disc image: https://pngtree.com/freepng/retro-record-song-for-dj-and-music-3d-set_14575774.html

Play pause and load button: https://pngtree.com/freepng/play-the-pause-button_6083047.html