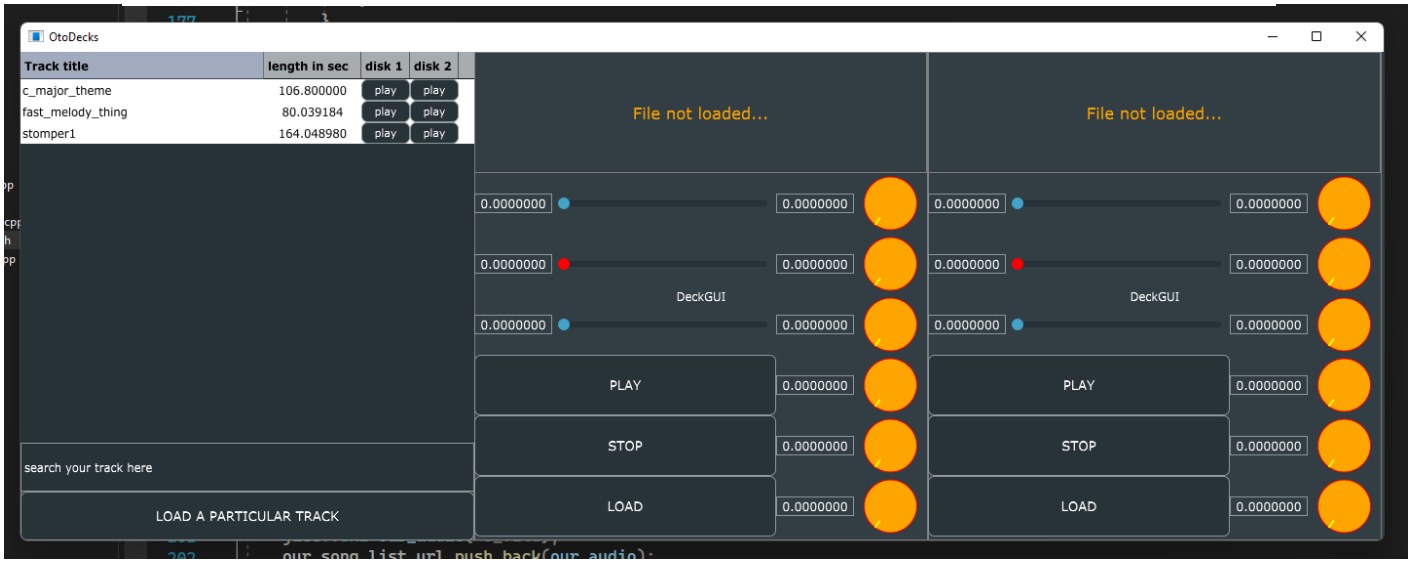
Name – Rishi Sankhla
Student No. – **200102874**

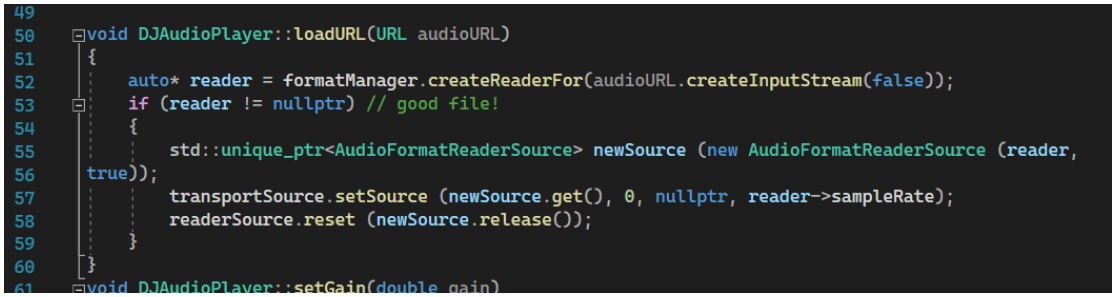
Object Oriented Programming Coursework for Endterm: Otodecks

Requirements	Completed or not
R1: The application should contain all the basic functionality shown in	✓
R1A: can load audio files into audio players	✓
R1B: can play two or more tracks	✓
R1C: can mix the tracks by varying each of their volumes	✓
R1D: can speed up and slow down the tracks	✓
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	✓
R2B: Component enables the user to control the playback of a deck	✓
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	✓
R3B: Component parses and displays meta data such as filename and song length	✓
R3C: Component allows the user to search for files	✓
R3D: Component allows the user to load files from the library into a deck	✓
R3E: The music library persists so that it is restored when the user exits then restarts the application	✓
R4: Implementation of a complete custom GUI	✓
R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls	✓
R4B: GUI layout includes the custom Component from R2	✓
R4C: GUI layout includes the music library component fro R3	✓



R1A: yes user can directly load audio file into the deck, through load button. As we can see in the screenshot below.

```
143     }
144     if (button == &loadButton)
145     {
146         auto fileChooserFlags =
147         FileBrowserComponent::canSelectFiles;
148         fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
149         {
150             player->loadURL(URL{chooser.getResult()});
151             // and now the waveformDisplay as well
152             waveformDisplay.loadURL(URL{chooser.getResult()});
153         });
154     }
155 }
156
```

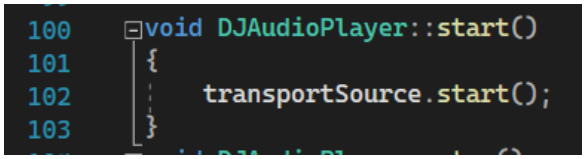


```
49
50  void DJAudioPlayer::loadURL(URL audioURL)
51  {
52      auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
53      if (reader != nullptr) // good file!
54      {
55          std::unique_ptr<AudioFormatReaderSource> newSource (new AudioFormatReaderSource (reader,
56  true));
57          transportSource.setSource (newSource.get(), 0, nullptr, reader->sampleRate);
58          readerSource.reset (newSource.release());
59      }
60  }
61  void DJAudioPlayer::setGain(double gain)
```

Here we can see in the deckGUI.cpp file that Whenever user clicks the load button it opens the file browser component and pass the URL link to the appropriate player deck.

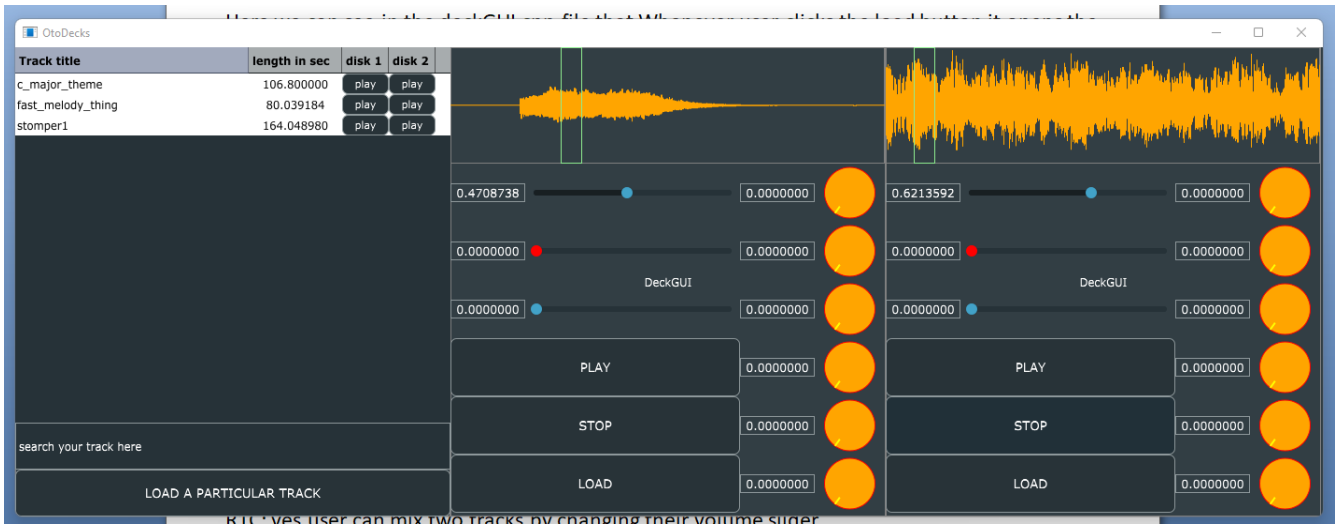
R1B: yes user can play two or more tracks, by clicking the play button. It activates the start function of the appropriate player DJAudiofile class. Which calls the start function of transportSource layer. (refer below screenshots)

```
130
131 void DeckGUI::buttonClicked(Button* button)
132 {
133     if (button == &playButton)
134     {
135         std::cout << "Play button was clicked " << std::endl;
136         player->start();
137     }
138     if (button == &stopButton)
```



```
100 void DJAudioPlayer::start()
101 {
102     transportSource.start();
103 }
```

R1C: yes user can mix two tracks by changing their volume slider. Whenever slider value is changed it call the set gain function in DJAudiofile class, As we can see in the below screenshots, which then calls the set gain function of transportSource layer.




```
61  void DJAudioPlayer::setGain(double gain)
62  {
63      if (gain < 0 || gain > 1.0)
64      {
65          std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl;
66      }
67      else {
68          transportSource.setGain(gain);
69      }
70
71  }
```

R1D: yes user can speed up or slow down the track using second slider(the one with red color). Here we first call the setspeed function of DJAudiofile class in deckGUI.cpp file, which in the call the setresamplingratio function at resamplesource layer. (refer below screenshot)

```
71  }
72  void DJAudioPlayer::setSpeed(double ratio)
73  {
74      if (ratio < 0 || ratio > 100.0)
75      {
76          std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::endl;
77      }
78      else {
79          resampleSource.setResamplingRatio(ratio);
80      }
81  }
82  void DJAudioPlayer::setPosition(double posInSecs)
```

R2A: yes I have implemented custom graphic of my custom component(i.e reverb parameters knobs). I studied the Juce library documentation(reference: https://docs.juce.com/master/tutorial_look_and_feel_customisation.html) and created one more class inside DeckGUI file naming otherlookandfeel, it inherits the juce lookandfeel class.

Then after creating the object of it, inside our DeckGUI.h file, I passed it as a parameter of setlookandfeel function of slider. And also change the slider style to rotary. (check out line 28 in below screenshots)

```cpp
//crerated a new class, by studing juce documentation website.
//reference:  /** https://docs.juce.com/master/tutorial_look_and_feel_customisation.html */
class OtherLookAndFeel : public juce::LookAndFeel_V4
{
public:

    //setting up the constructor function
    OtherLookAndFeel()
    {
        setColour(juce::Slider::thumbColourId, juce::Colours::red);
    }

    //public overriding drawRotarySlider to draw our rotating sliders
    void drawRotarySlider(juce::Graphics& g, int x, int y, int width, int height, float sliderPos,
        const float rotaryStartAngle, const float rotaryEndAngle, juce::Slider&) override
    {
        auto radius = (float)juce::jmin(width / 2, height / 2) - 4.0f;
        auto centreX = (float)x + (float)width * 0.5f;
        auto centreY = (float)y + (float)height * 0.5f;
        auto rx = centreX - radius;
        auto ry = centreY - radius;
        auto rw = radius * 2.0f;
        auto angle = rotaryStartAngle + sliderPos * (rotaryEndAngle - rotaryStartAngle);

        // fill
        g.setColour(juce::Colours::orange);
        g.fillEllipse(rx, ry, rw, rw);

        // outline
        g.setColour(juce::Colours::red);
        g.drawEllipse(rx, ry, rw, rw, 1.0f);

        juce::Path p;
        auto pointerLength = radius * 0.33f;
        auto pointerThickness = 2.0f;
        p.addRectangle(-pointerThickness * 0.5f, -radius, pointerThickness, pointerLength);
        p.applyTransform(juce::AffineTransform::rotation(angle).translated(centreX, centreY));

        // pointer
        g.setColour(juce::Colours::yellow);
        g.fillPath(p);
```

No issues found

Output

```cpp
    addAndMakeVisible(playButton);
    addAndMakeVisible(stopButton);
    addAndMakeVisible(loadButton);
    otherLookAndFeel.setColour(juce::Slider::thumbColourId, juce::Colours::red);
    speedSlider.setLookAndFeel(&otherLookAndFeel);

    reverb_1.setSliderStyle(juce::Slider::Rotary);
    reverb_2.setSliderStyle(juce::Slider::Rotary);
    reverb_3.setSliderStyle(juce::Slider::Rotary);
    reverb_4.setSliderStyle(juce::Slider::Rotary);
    reverb_5.setSliderStyle(juce::Slider::Rotary);
    reverb_6.setSliderStyle(juce::Slider::Rotary);
    reverb_1.setLookAndFeel(&otherLookAndFeel);
    reverb_2.setLookAndFeel(&otherLookAndFeel);
    reverb_3.setLookAndFeel(&otherLookAndFeel);
    reverb_4.setLookAndFeel(&otherLookAndFeel);
    reverb_5.setLookAndFeel(&otherLookAndFeel);
    reverb_6.setLookAndFeel(&otherLookAndFeel);
    addAndMakeVisible(volSlider);
    addAndMakeVisible(speedSlider);
    addAndMakeVisible(posSlider);
```

R2B: As we can see In above screenshots my custom deck controls for user are reverb parameter knobs, with the help of them user can not only mix the song effects but can also set the individual 6 arguments of reverb, which are:

```
our_r_parameter.dryLevel = 0.4f;
our_r_parameter.roomSize = 0.5f;
our_r_parameter.damping = 0.5f;
our_r_parameter.wetLevel = 0.0f;
our_r_parameter.width = 1.0f;
our_r_parameter.freezeMode = 0.0f;
```

to implement it I first created `juce::ReverbAudioSource our_reverb{ &resampleSource, false}` and `juce::Reverb::Parameters our_r_parameter;` object in the DJAudiofile.h file, now we can change the reverb parameters with the help of `our_r_parameter`, and the finally pass it to setparameters

function of `ReverbAudioSource`, here `ReverbAudioSource` replaces the sample layer of our DJ software.

Example function of setting room size reverb-parameter in DJAudiofile.cpp file:

```
137
138  void DJAudioPlayer::setting_room(float r_level)
139  {
140      if (r_level < 0 || r_level > 1.0)
141      {
142          DBG("please enter between 0 to 1");
143      }
144      else {
145          DBG("room changed");
146          our_r_parameter.roomSize = r_level;
147          our_reverb.setParameters(our_r_parameter);
148      }
149  }
150
```

R3A: yes the load button in our playlist allows the user to add any song to our playlist. If user clicks this button, we perform two major task, first it calls the import_library function which open up file browser and load tracks to our playlist by passing the results to select_file function, secondly it updates our table component so that our table reload the data again. (check out below screenshots)

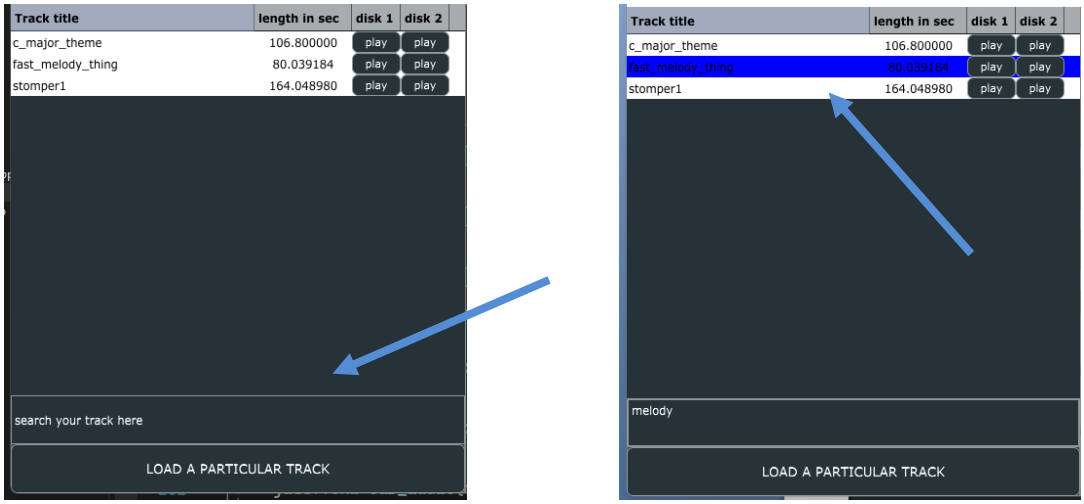
```
158  void PlaylistComponent::buttonClicked(Button* button)
159  {
160      if(button == &load_Button)
161      {
162          std::cout << "Play button was clicked " << std::endl;
163          import_library();
164          tableComponent.updateContent();
165          DBG(button->getComponentID());
166      }
167      else {
```

```
188
189  void PlaylistComponent::import_library()
190  {
191      FileChooser chooser{"Select a file..."};
192      if (chooser.browseForFileToOpen())
193      {
194          select_file(chooser.getResult());
195      }
196  }
197
198  void PlaylistComponent::select_file(juce::File o_file)
199  {
200      trackTitles.push_back(o_file.getFileNameWithoutExtension().toStdString());
201      juce::URL our_audio{ o_file};
202      our_song_list_url.push_back(our_audio);
203      tracklength.push_back(std::to_string(return_song_length(our_audio)));
204      DBG(o_file.getFileNameWithoutExtension().toStdString());
205  }
206
```

R3B: once we pass the music file to select_file function, it adds the song title, song length and song URL to 3 different vectors. So that in the end we can call updateContent() function of tableComponent to reload the table. To calculate song length, I have created a separate function which passes the song file to a new DJAudiofile instance, to get the length in seconds. (refer below screenshot)

```
double PlaylistComponent::return_song_length(juce::URL our_audio)
{
    player_extra_DJAudioPlayer->loadURL(our_audio);
    double seconds{ player_extra_DJAudioPlayer->return_current_length() };
    return seconds;
}
```

R3C: just able the load button I have added a juce text editor to search the song, which perform liner search in the vector containing song title. And then selects the appropriate row.




```
29      addAndMakeVisible(search_area);
30      load_Button.addListener(this);
31      search_area.addListener(this);
32      search_area.setTextToShowWhenEmpty("search your track here",juce::Colours::white);
33      search_area.onReturnKey = [this] {
34          search_the_track(search_area.getText());
35      };
```

As we can see in the above screenshot of playlistcomponent.cpp file, Once user hits the enter key, text is passed on to search_the_track function, which then perform liner search on our tracktitle vector and finally selects the appropriate row according to the index number. (refer below screenshot of the code)

```
241
242 void PlaylistComponent::search_the_track(juce::String our_text)
243 {
244     if (our_text != "")
245     {
246         int c = 0;
247         for (auto& element : trackTitles)
248         {
249             std::size_t found = element.find(our_text.toStdString());
250             if (found != std::string::npos)
251                 tableComponent.selectRow(c);
252             c++;
253         }
254     }
255     else
256     {
257         tableComponent.deselectAllRows();
258     }
259 }
260
```

R3D: in front of every track there are two buttons, through which user can load any song to our decks. This was possible with the help of juce library refreshcomponentforcell function and buttonclicked function.

```
125  Component* PlaylistComponent::refreshComponentForCell(
126      int rowNumber,
127      int columnId,
128      bool isRowSelected,
129      Component* existingComponentToUpdate)
130  {
131      if (columnId == 2)
132      {
133          if (existingComponentToUpdate == nullptr)
134          {
135              TextButton* btn = new TextButton("play");
136              String id{ std::to_string(c_1) };
137              btn->setComponentID(id);
138              btn->addListener(this);
139              existingComponentToUpdate = btn;
140              c_1 = c_1 + 1;
141          }
142      }
143      if (columnId == 3)
144      {
145          if (existingComponentToUpdate == nullptr)
146          {
147              TextButton* btn = new TextButton("play");
148              String id{ std::to_string(c_1) };
149              btn->setComponentID(id);
150              btn->addListener(this);
151              existingComponentToUpdate = btn;
152              c_1 = c_1 + 1;
153          }
154      }
155      return existingComponentToUpdate;
156  }
157
```

```
158  void PlaylistComponent::buttonClicked(Button* button)
159  {
160      if(button == &load_Button)
161      {
162          std::cout << "Play button was clicked " << std::endl;
163          import_library();
164          tableComponent.updateContent();
165          DBG(button->getComponentID());
166      }
167      else {
168          int id = std::stoi(button->getComponentID().toStdString());
169          if (id % 2 == 0) {
170              DBG("tt");
171              if (id == 0) {
172                  deck_1->loadthecurrentfile(our_song_list_url[id]);
173              }
174              else {
175                  deck_1->loadthecurrentfile(our_song_list_url[id/2]);
176              }
177          }
178          else {
179              if (id == 1) {
180                  deck_2->loadthecurrentfile(our_song_list_url[id-1]);
181              }
182              else {
183                  deck_2->loadthecurrentfile(our_song_list_url[(id - 1)/2]);
184              }
185          }
186      }
187  }
188
```

Here we are dynamically creating buttons then assigning them index number as ID (refer refreshcomponentforcell function) and then finally adding button listener to them.

In buttonclicked function we first extracts the dynamic button ID and pass on to respected deck to play the particular track (we are selecting song url from the vector according to our ID).

R3E: the music playlist automatically saves all the current track data into a note pad file and then reload it back when application opens up again. And for this mechanism I created two functions, i.e load_the_track (called inside the constructor of playlistcomponent.cpp) and save_our_data (called inside the destructor of playlistcomponent.cpp).

So in save_our_data function we first create the our_loaded_track_data.txt file, add all 3 vectors (track title, track URL and track length) data in it, then finally close the file.

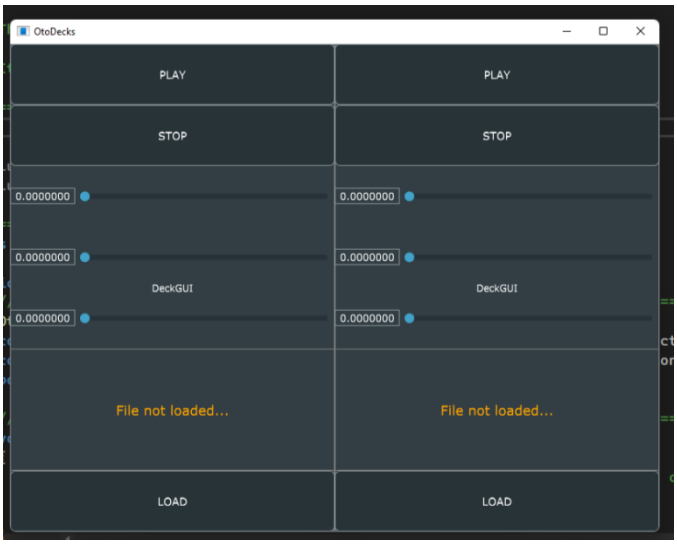
Whereas, in the load_the_track function we read our our_loaded_track_data.txt file and copy it back to our vectors. And finally calling updatecontent function to reload our table. (refer below screenshots)

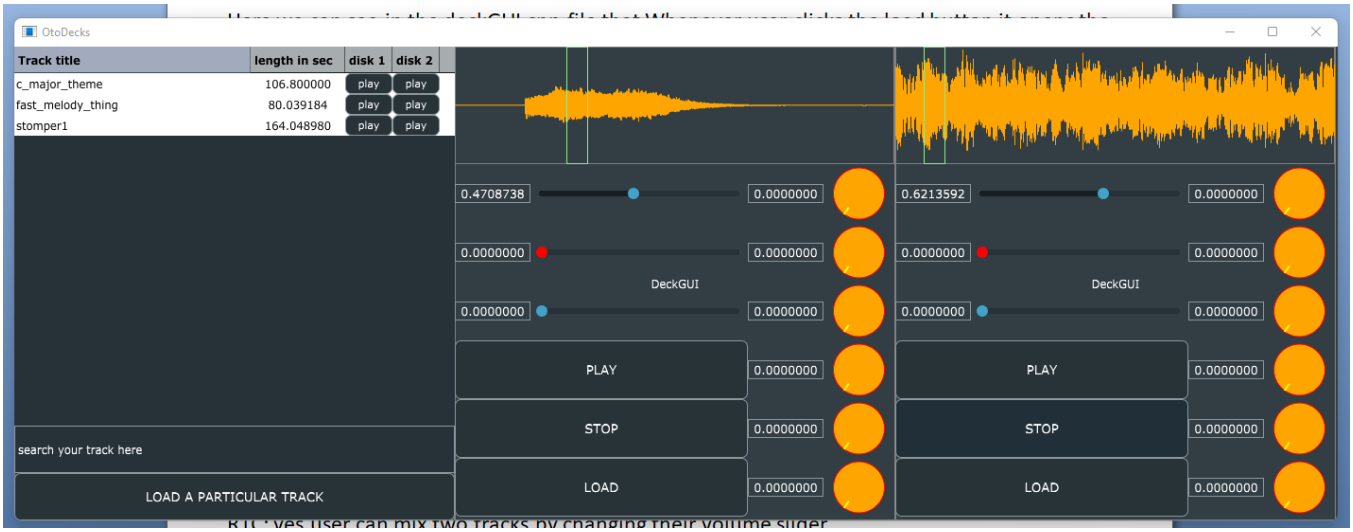
```cpp
void PlaylistComponent::load_the_track()
{
    int c = 0;
    std::ifstream input("our_loaded_track_data.txt");
    for (std::string line; getline(input, line); )
    {
        if (c % 3 == 0) {
            trackTitles.push_back(line);
        }
        if(c % 3 == 1) {
            tracklength.push_back(line);
        }
        if (c % 3 == 2) {
            juce::URL our_audio{ line };
            our_song_list_url.push_back(our_audio);
        }
        c++;
    }
    tableComponent.updateContent();

}
```

```cpp
void PlaylistComponent::save_our_data()
{
    int c = 0;
    file.open("our_loaded_track_data.txt", std::ios_base::out);
    for (auto& element : trackTitles)
    {
        if (c < tracklength.size()-1)
        {
            file << element;
            file << std::endl;
            file << tracklength[c];
            file << std::endl;
            file << our_song_list_url[c].toString(false);
            file << std::endl;
        }
        else
        {
            file << element;
            file << std::endl;
            file << tracklength[c];
            file << std::endl;
            file << our_song_list_url[c].toString(false);
        }
        c++;
    }
    file.close();
}
```

R4A: as we can see from the below screenshots that there's a significant difference between DeckGUI shown in class and the one I made.





To change layout, I first reoriented my playlist to left in the maincomponent.cpp file and also changed the default screen size to 1400 width and 500 height.

```
107
108  void DeckGUI::resized()
109  {
110      double rowH = getHeight() / 8;
111      double rowW = getWidth() / 3;
112      waveformDisplay.setBounds(0, 0, rowW * 3, rowH * 2);
113
114      volSlider.setBounds(0, rowH * 2, rowW * 2, rowH);
115      speedSlider.setBounds(0, rowH * 3, rowW * 2, rowH);
116      posSlider.setBounds(0, rowH * 4, rowW * 2, rowH);
117
118      reverb_1.setBounds(rowW * 2, rowH * 2, rowW * 1, rowH);
119      reverb_2.setBounds(rowW * 2, rowH * 3, rowW * 1, rowH);
120      reverb_3.setBounds(rowW * 2, rowH * 4, rowW * 1, rowH);
121      reverb_4.setBounds(rowW * 2, rowH * 5, rowW * 1, rowH);
122      reverb_5.setBounds(rowW * 2, rowH * 6, rowW * 1, rowH);
123      reverb_6.setBounds(rowW * 2, rowH * 7, rowW * 1, rowH);
124
125      playButton.setBounds(0, rowH * 5, rowW * 2, rowH);
126      stopButton.setBounds(0, rowH * 6, rowW * 2, rowH);
127      loadButton.setBounds(0, rowH * 7, rowW * 2, rowH);
128
129  }
```

Then I rewrite both the resized function (one in DeckGUI and one in playlistcomponent) and also added the extension components in it, i.e reverb knobs, search button and load button for playlist.

R4B: as we can see in the above screenshot from DeckGUI file, my DJ app GUI includes the custom reverb parameters knobs. And with help of lookandfeel class I was able to implement custom graphics on it. By changing style to rotary and color to orange.

R4C: as we can see in the below screenshot from playlistcomponent file, my DJ app GUI contains the playlist as a component, which not only contain tablecomponent but also have the search editor & file load button.

```
61   void PlaylistComponent::resized()
62   {
63       // This method is where you should set the bounds of any child
64       // components that your component contains..
65       tableComponent.setBounds(0, 0, getWidth(), (getHeight()/10)*8);
66       search_area.setBounds(0, (getHeight() / 10) * 8, getWidth(), (getHeight() / 10) * 1);
67       load_Button.setBounds(0, (getHeight() / 10) * 9, getWidth(), (getHeight() / 10) * 1);
68   }
69
70   int PlaylistComponent::getNumRows()
```

I have changed the table dimensions, edited the paintrowbackground function and pain function to change the color of row when selected or deselected.