CM2005
Object Oriented Programming
End-term Assignment Report

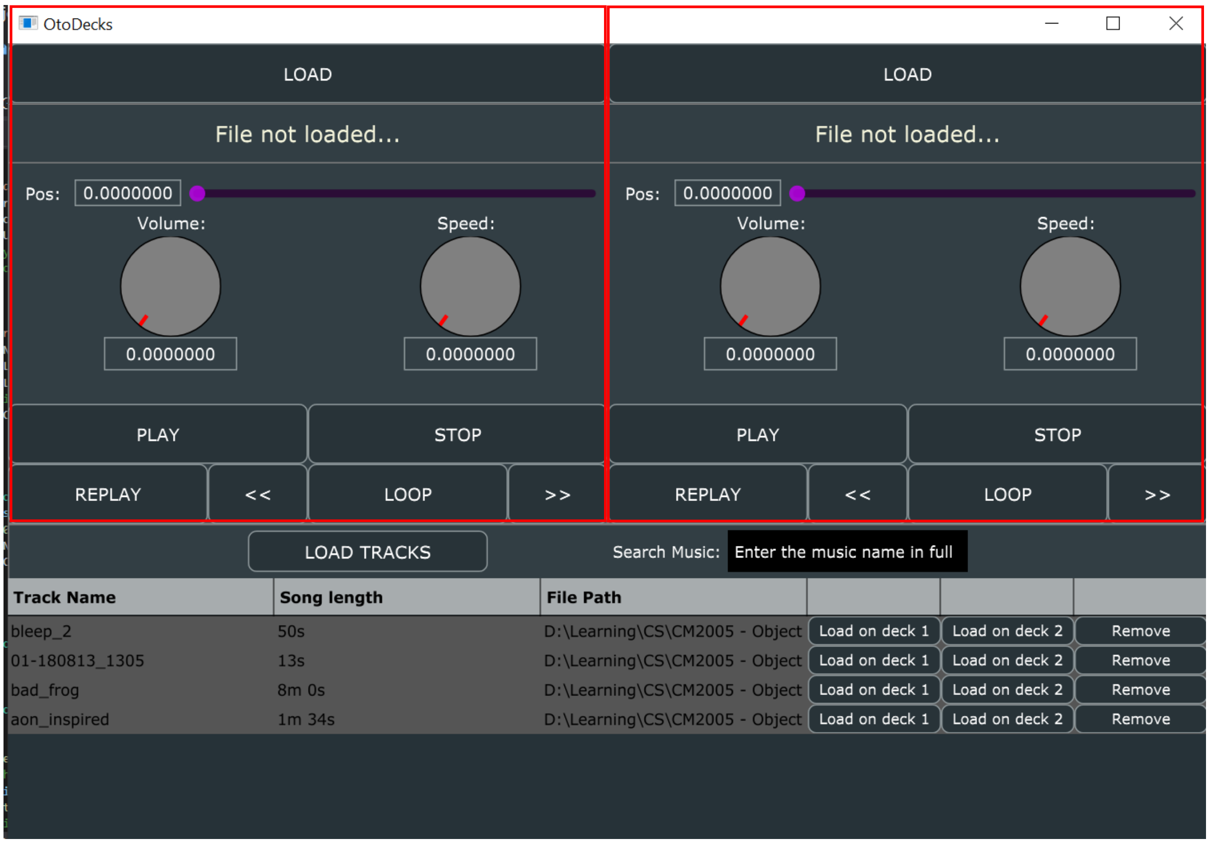
10211835

Kwang Kian Hui

Contents

R1: Basic Functionalities

R1A - can load audio files into audio players:



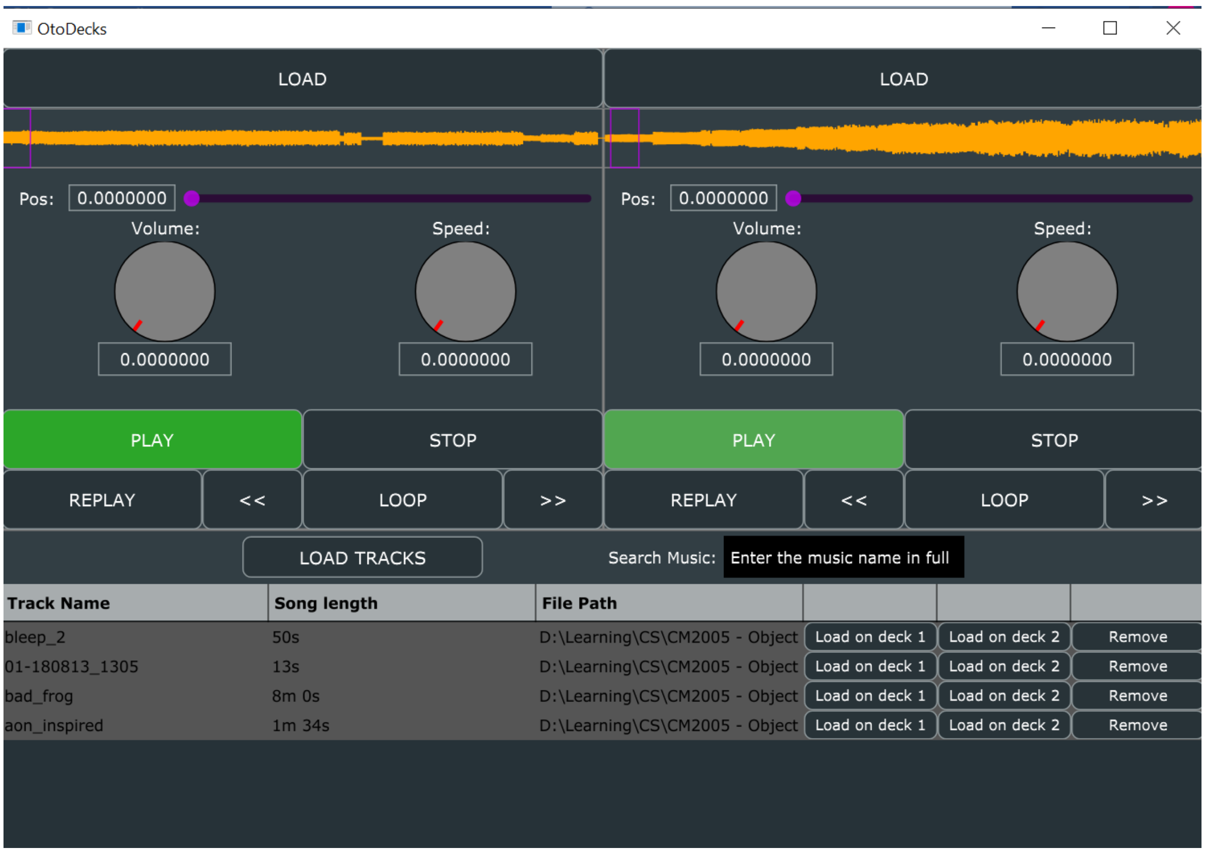
Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 1A.1 – OtoDecks

Users can drag a single audio file and drop it anywhere within the region of decks 1 or 2 highlighted in red for it to be loaded. If multiple files are dragged, nothing happens. The other way that the user can load audio files is by clicking the load button at the top of either decks.

R1A (achieved)

R1B - can play two or more tracks:



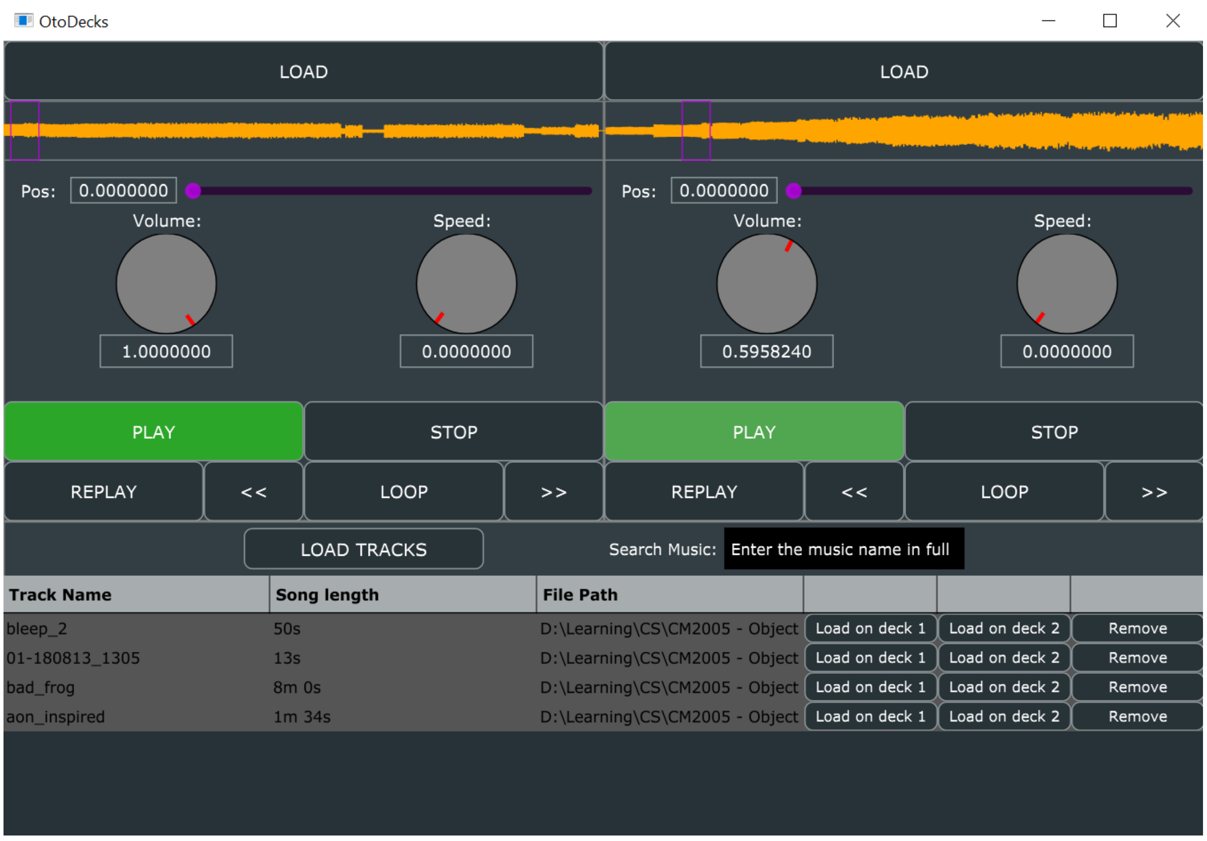
Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 1B.1 – OtoDecks

Two decks are present to play any tracks loaded in.

(R1B Achieved)

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



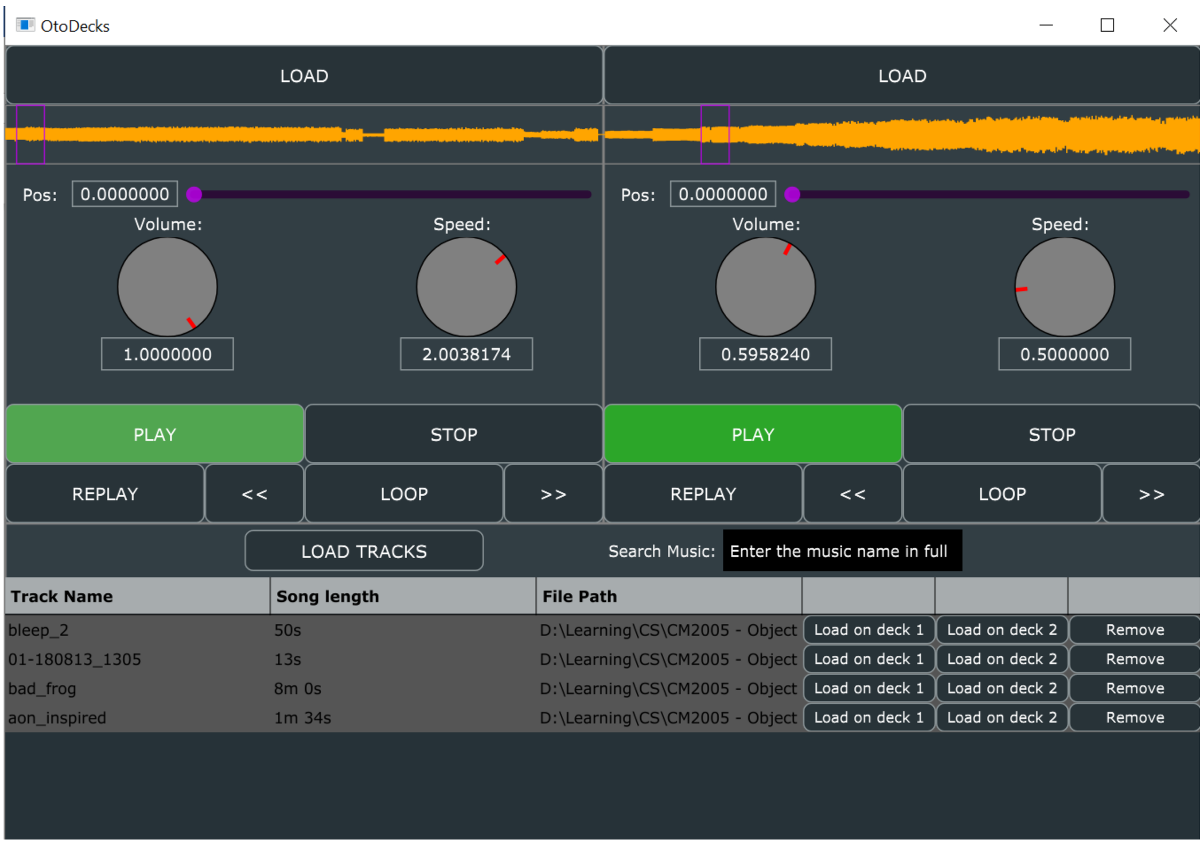
Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 1C.1 – OtoDecks

When both audio files are playing at the same time, the MixerAudioSource mixes the output of both audios together to allow both songs to be played at the same time. The slider to control the volume for both tracks has been converted to a rotary slider where the value can be changed by clicking and dragging along the ellipse or by editing the value in the textbox below. The value ranges from 0 to 1.

(R1C Achieved)

R1D - can speed up and slow down the tracks:



Pos: 0.0000000 Volume: 1.0000000 Speed: 2.0038174

Pos: 0.0000000 Volume: 0.5958240 Speed: 0.5000000

Search Music: Enter the music name in full

Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

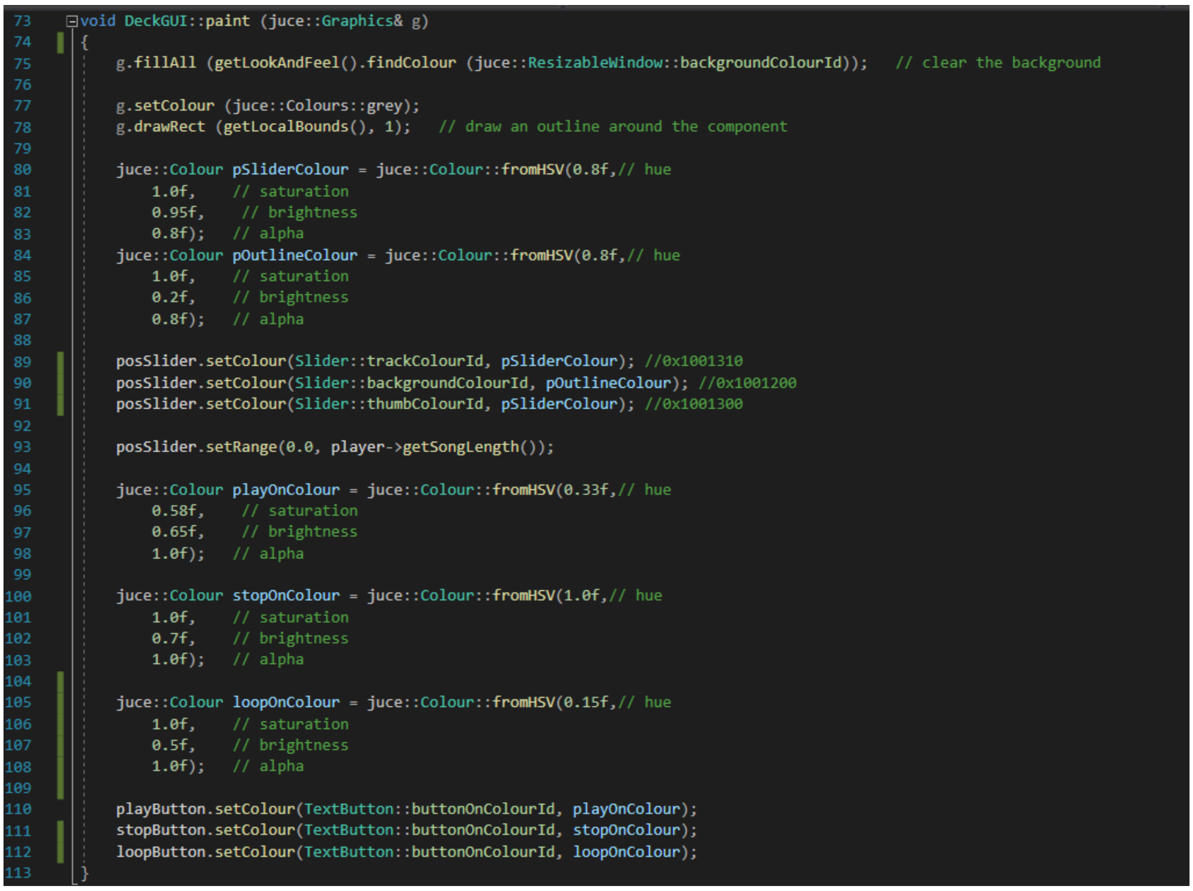
Figure 1D.1 – OtoDecks

The slider to control the speed for both tracks has been converted to a rotary slider where the value can be changed by clicking and dragging along the ellipse or by editing the value in the textbox below. The value ranges from 0 to 3.

(R1D Achieved)

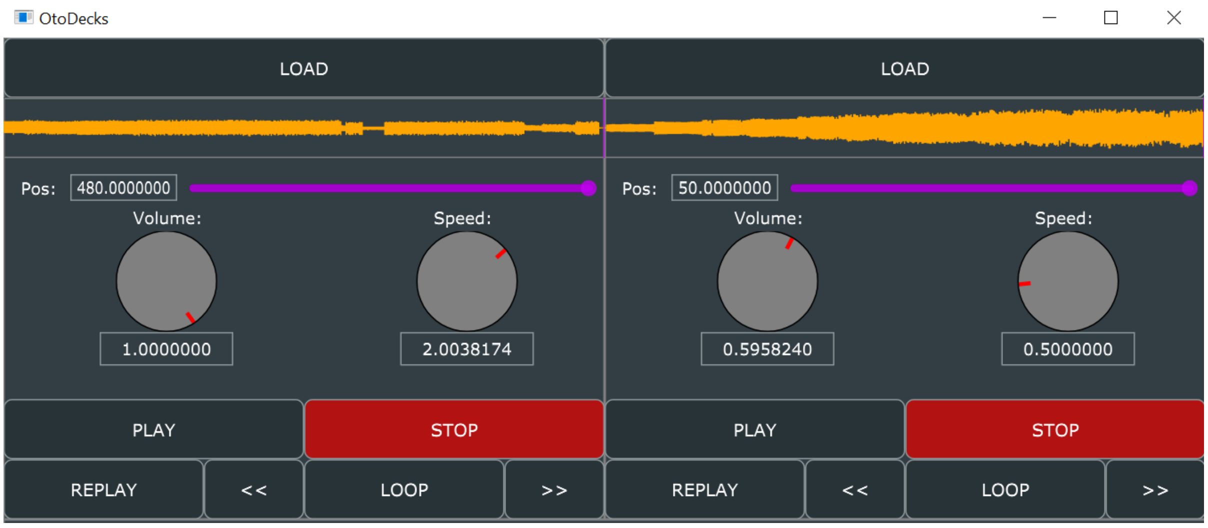
R2: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/ start

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



```cpp
void DeckGUI::paint (juce::Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));   // clear the background

    g.setColour (juce::Colours::grey);
    g.drawRect (getLocalBounds(), 1);   // draw an outline around the component

    juce::Colour pSliderColour = juce::Colour::fromHSV(0.8f,// hue
        1.0f,    // saturation
        0.95f,    // brightness
        0.8f);   // alpha
    juce::Colour pOutlineColour = juce::Colour::fromHSV(0.8f,// hue
        1.0f,    // saturation
        0.2f,    // brightness
        0.8f);   // alpha

    posSlider.setColour(Slider::trackColourId, pSliderColour); //0x1001310
    posSlider.setColour(Slider::backgroundColourId, pOutlineColour); //0x1001200
    posSlider.setColour(Slider::thumbColourId, pSliderColour); //0x1001300

    posSlider.setRange(0.0, player->getSongLength());

    juce::Colour playOnColour = juce::Colour::fromHSV(0.33f,// hue
        0.58f,    // saturation
        0.65f,    // brightness
        1.0f);   // alpha

    juce::Colour stopOnColour = juce::Colour::fromHSV(1.0f,// hue
        1.0f,    // saturation
        0.7f,    // brightness
        1.0f);   // alpha

    juce::Colour loopOnColour = juce::Colour::fromHSV(0.15f,// hue
        1.0f,    // saturation
        0.5f,    // brightness
        1.0f);   // alpha

    playButton.setColour(TextButton::buttonOnColourId, playOnColour);
    stopButton.setColour(TextButton::buttonOnColourId, stopOnColour);
    loopButton.setColour(TextButton::buttonOnColourId, loopOnColour);
}
```

Figure 2A.1 – DeckGUI.cpp (line 73 – 113)



Pos: 480.0000000
Volume: 1.0000000
Speed: 2.0038174

Pos: 50.0000000
Volume: 0.5958240
Speed: 0.5000000

Figure 2A.2 – OtoDecks

2 colours of pSliderColour and pOutlineColour are created using the HSV colour. We can use the HSL colour picker online to identify the types of colour we want and convert them into 0.0f to 1.0f values for the hue, saturation, brightness and alpha.

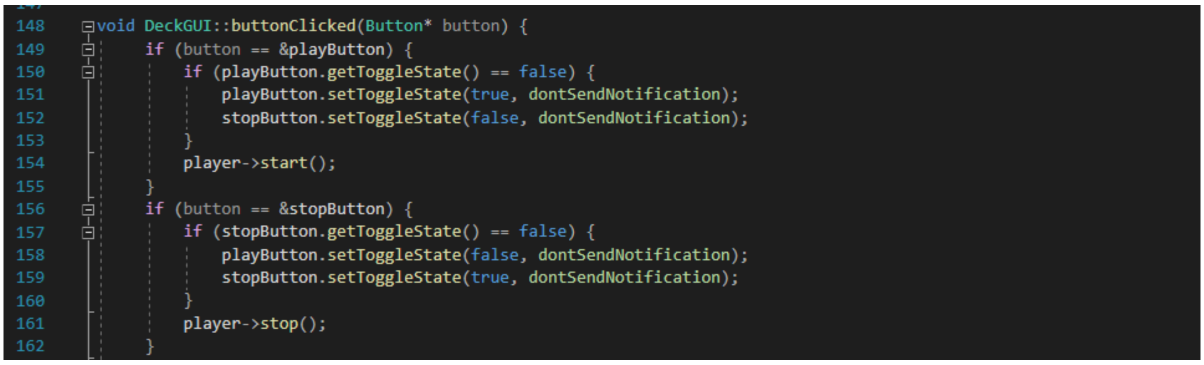
posSlider adopts those colours by using setColour function and passing the lookAndFeel colour id of the parts that we want to colour, followed by the colour itself.

There is a compiled list of all the lookAndFeel colour id done by a user “nonchain” in a forum which can be found in this link: https://forum.juce.com/t/here-is-a-compiled-list-of-all-lookandfeel-colour-id-s-for-all-juce-widgets/14747.

We can also use Slider::trackColourId instead of entering the whole id itself. As seen in figure 2A.1 and 2A.2, this changes the colour of the position slider.

The posSlider is previously set to have a range of 0 to 1 but will update to have the range of 0 to the length of the song as seen in line 93 in figure 2A.1. This cannot be done at the start as there will not be any tracks in the player which will result in an error due to failure to retrieve getSongLength() of an empty player.

The play, stop and loop button are given the buttonOnColourId so that when the state of the button is toggled, it changes colour to their set colours of playOnColour, stopOnColour and loopOnColour.



```
148  void DeckGUI::buttonClicked(Button* button) {
149      if (button == &playButton) {
150          if (playButton.getToggleState() == false) {
151              playButton.setToggleState(true, dontSendNotification);
152              stopButton.setToggleState(false, dontSendNotification);
153          }
154          player->start();
155      }
156      if (button == &stopButton) {
157          if (stopButton.getToggleState() == false) {
158              playButton.setToggleState(false, dontSendNotification);
159              stopButton.setToggleState(true, dontSendNotification);
160          }
161          player->stop();
162      }
```

Figure 2A.3 – DeckGUI.cpp (line 148 – 162)

In the buttonClicked() function, whenever the playButton is clicked, we will change both the playButton and stopButton toggle states. If the toggle state of the buttons has not been clicked or toggled, we will change its toggle state. Once the state has been changed, we will start or stop the player based on which button is clicked. This changed state will allow them to be coloured green for the playButton and red for the stopButton.

(R2A Achieved)

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

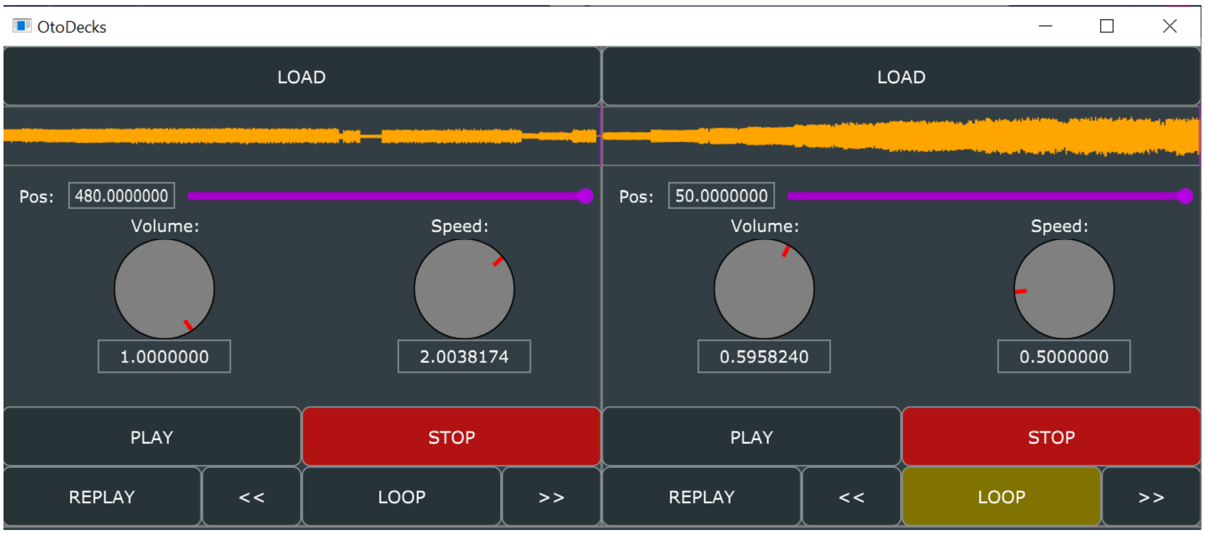
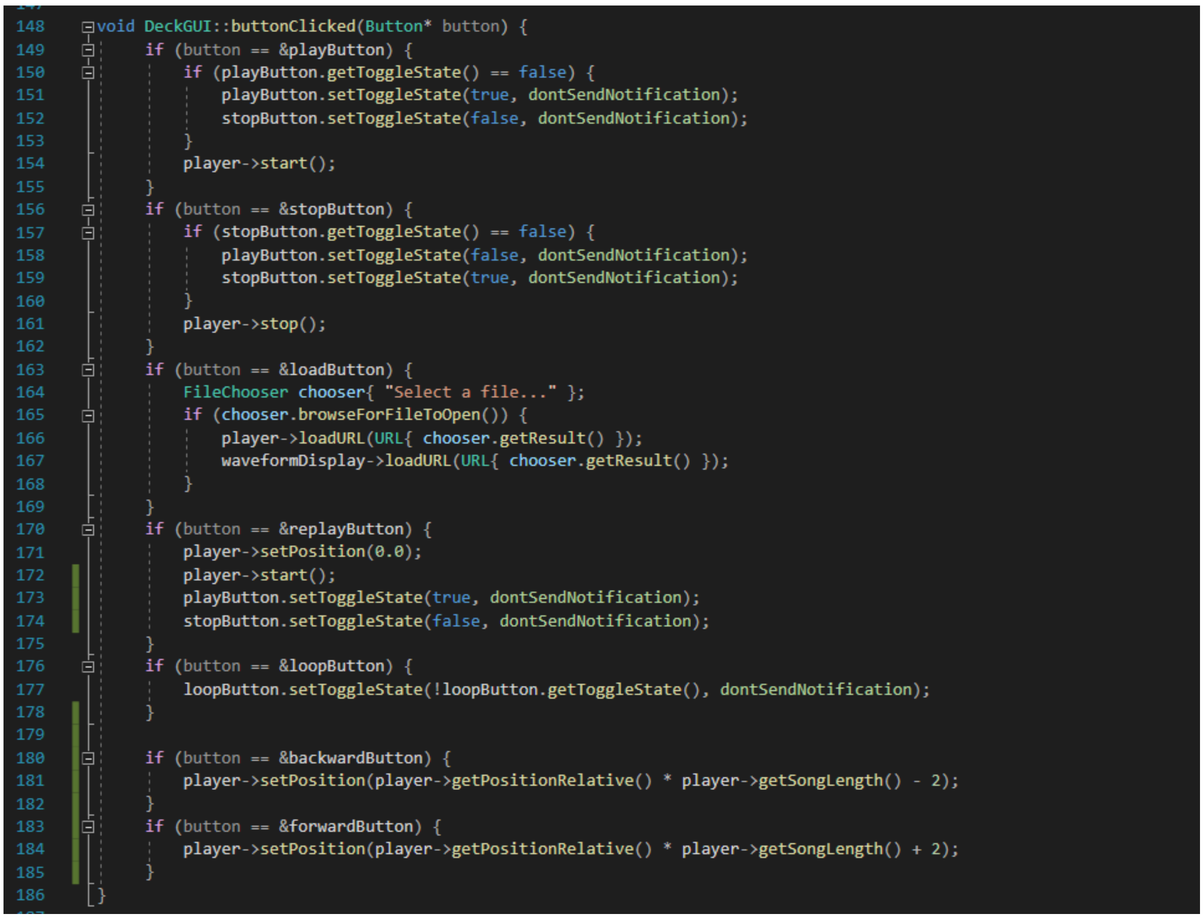


Figure 2B.1 – OtoDecks



```
void DeckGUI::buttonClicked(Button* button) {
    if (button == &playButton) {
        if (playButton.getToggleState() == false) {
            playButton.setToggleState(true, dontSendNotification);
            stopButton.setToggleState(false, dontSendNotification);
        }
        player->start();
    }
    if (button == &stopButton) {
        if (stopButton.getToggleState() == false) {
            playButton.setToggleState(false, dontSendNotification);
            stopButton.setToggleState(true, dontSendNotification);
        }
        player->stop();
    }
    if (button == &loadButton) {
        FileChooser chooser{ "Select a file..." };
        if (chooser.browseForFileToOpen()) {
            player->loadURL(URL{ chooser.getResult() });
            waveformDisplay->loadURL(URL{ chooser.getResult() });
        }
    }
    if (button == &replayButton) {
        player->setPosition(0.0);
        player->start();
        playButton.setToggleState(true, dontSendNotification);
        stopButton.setToggleState(false, dontSendNotification);
    }
    if (button == &loopButton) {
        loopButton.setToggleState(!loopButton.getToggleState(), dontSendNotification);
    }

    if (button == &backwardButton) {
        player->setPosition(player->getPositionRelative() * player->getSongLength() - 2);
    }
    if (button == &forwardButton) {
        player->setPosition(player->getPositionRelative() * player->getSongLength() + 2);
    }
}
```

Figure 2B.2 – DeckGUI.cpp (line 148 – 186)

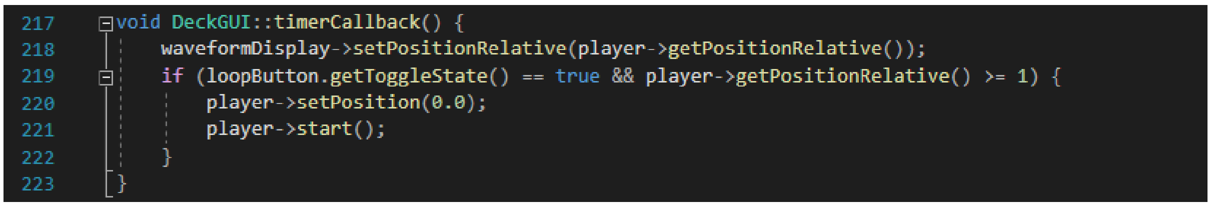
Users have 4 additional control options in OtoDecks on top of play and stop button. When the track ends, clicking play does not allow the deck to replay the song again and thus, I have added a replay button.

I have also included the loop button which can be toggled on or off which allows the song to replay itself when the song ends.

As seen in the buttonClicked() function, when the replay button is clicked, the player position will be set to 0.0. This will set the player to play the song from the start. I have added a start() function call as the user may have stopped the song before clicking the replay button and simply setting the position to 0.0 will not start the player.

However, we could also not include the replay button and write an if check inside the buttonClicked function for the play button, to check if the song has ended when the button was clicked and set the position to 0.0 to replay the song if true.

The loopButton simply toggles the state of the loopButton from true to false or vice versa.



```
217  void DeckGUI::timerCallback() {
218      waveformDisplay->setPositionRelative(player->getPositionRelative());
219      if (loopButton.getToggleState() == true && player->getPositionRelative() >= 1) {
220          player->setPosition(0.0);
221          player->start();
222      }
223  }
```

Figure 2B.3 – DeckGUI.cpp (line 217 – 223)

In the timerCallback() function, we will check if the loopButton is toggled and the relative position of the player is greater than or equal to 1. The player relative position will move from 0 to 1 when the player is running. However, the relative position does not end exactly at 1 and thus, we will check if its greater than 1.

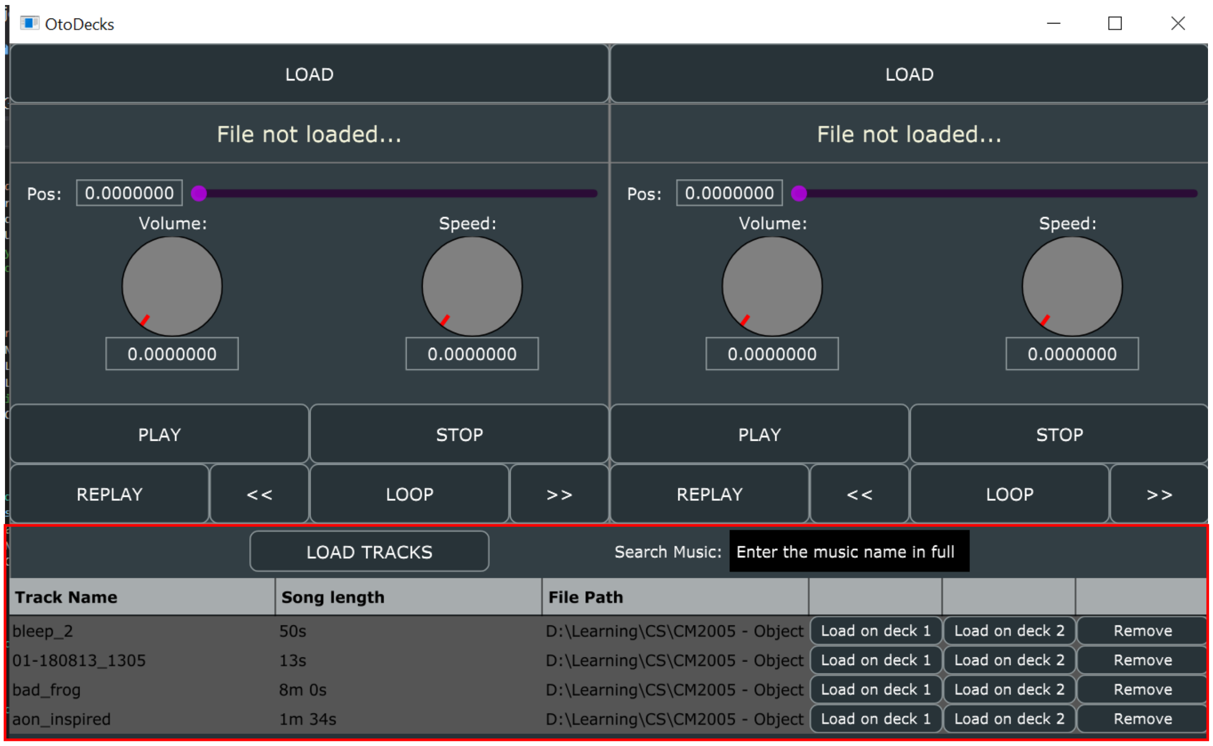
If true we will set the position to the starting position and start the player.

As seen in figure 2B.2 of line 180 to 185, I have also added the backward and forward button that moves the position of the player forward and backwards by 2 seconds. The relative position is retrieved and multiplied by the total song length to get the current position in seconds as there are not getPosition() function readily available and only the getPositionRelative() function. That position value is then subtracted or added by 2 before setting that position to the player.

(R2B Achieved)

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

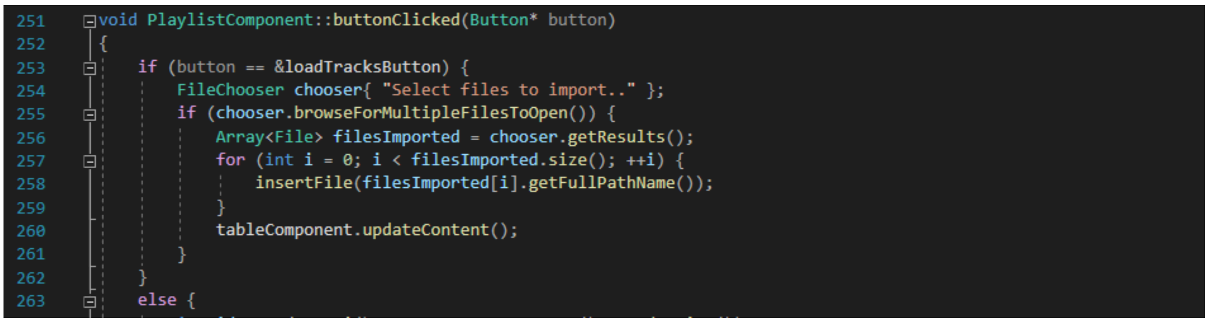
R3A: Component allows the user to add files to their library:



Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 3A.1 – OtoDecks

Users can add files to the library in 2 ways. The first would be to click the load tracks button and the other would be to drag 1 or more files and drop them in the area highlighted in red.



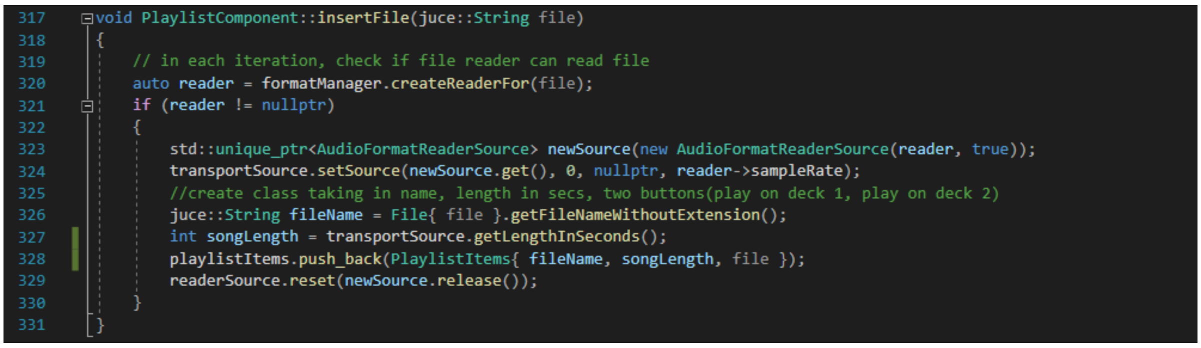
```
251  void PlaylistComponent::buttonClicked(Button* button)
252  {
253      if (button == &loadTracksButton) {
254          FileChooser chooser{ "Select files to import.." };
255          if (chooser.browseForMultipleFilesToOpen()) {
256              Array<File> filesImported = chooser.getResults();
257              for (int i = 0; i < filesImported.size(); ++i) {
258                  insertFile(filesImported[i].getFullPathName());
259              }
260              tableComponent.updateContent();
261          }
262      }
263      else {
```

Figure 3A.2 – PlaylistComponent.cpp (line 251 – 263)

In the buttonClicked() function, if the loadTrackButton was clicked, it will create a FileChooser that prompts user to select the files that they want to import/add into the library. We use the browseForMultipleFilesToOpen() to allow users to add 1 or more files into the library.

We can get the files selected by using the getResults() instead of getResult(), which returns the files selected in an Array<File>. The next would be to iterate through the files and call the function insertFile() to insert the file into our vectors that store the file name, file length and file path.

The insertFile() takes in a juce::String as its parameter and so, we have to convert the file of the current iteration to a juce::String by using the getFullPathName() function.



```cpp
void PlaylistComponent::insertFile(juce::String file)
{
    // in each iteration, check if file reader can read file
    auto reader = formatManager.createReaderFor(file);
    if (reader != nullptr)
    {
        std::unique_ptr<AudioFormatReaderSource> newSource(new AudioFormatReaderSource(reader, true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        //create class taking in name, length in secs, two buttons(play on deck 1, play on deck 2)
        juce::String fileName = File{ file }.getFileNameWithoutExtension();
        int songLength = transportSource.getLengthInSeconds();
        playlistItems.push_back(PlaylistItems{ fileName, songLength, file });
        readerSource.reset(newSource.release());
    }
}
```

Figure 3A.3 – PlaylistComponent.cpp (line 317 – 331)

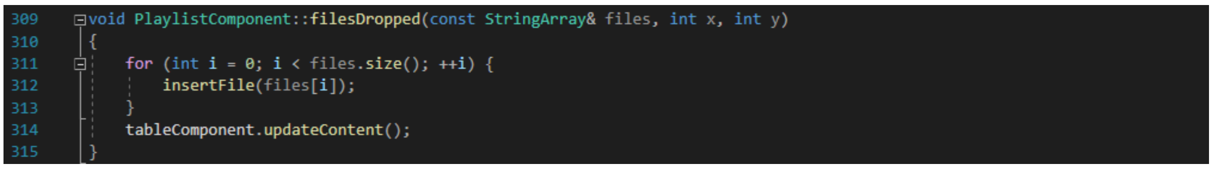
In the insertFile() function, we will create a reader for the file passed in and check if it is successful. If the reading fails, it will return the nullptr and we will do nothing. If successful, we will create a new AudioFormatReaderSource before setting it to the transportSource. The main purpose of this is so that we can retrieve the song length later on using the transportSource.

The file name is retrieved without the extensions and initialised to a new juce::String variable fileName. We then use the getLengthInSeconds() function available to the AudioTransportSource class and initialise the value to int variable songLength.

A PlaylistItems class object is created and pushed into the playlistItems vector of <PlaylistItems>. PlaylistItems class variables are (juce::String filename, int songLength, juce::String filePath).

file variable does not need to be converted it is the correct variable type of juce::String.

Once all of this is completed, the insertFile() function exits and proceeds to the next iteration in the filesImported array. Once the loop is exited, we will update the content of the tableComponent as this will redraw the application and any changes in the list will be seen.



```cpp
void PlaylistComponent::filesDropped(const StringArray& files, int x, int y)
{
    for (int i = 0; i < files.size(); ++i) {
        insertFile(files[i]);
    }
    tableComponent.updateContent();
}
```

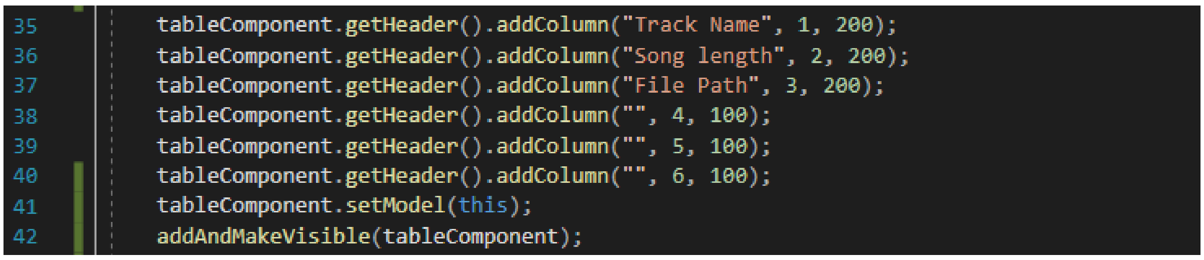
Figure 3A.4 – PlaylistComponent.cpp (line 309 – 315)

The filesDropped() function handles the event where user drops the files within the red highlighted region. Similar to buttonClicked(), we will iterate through the files that are made available as its parameters and call the insertFile() function.

However, since the files array is in the juce::String format, we can pass it to the insertFile() function immediately without conversion. The remaining steps are the same as previously explained.

(R3A Achieved)

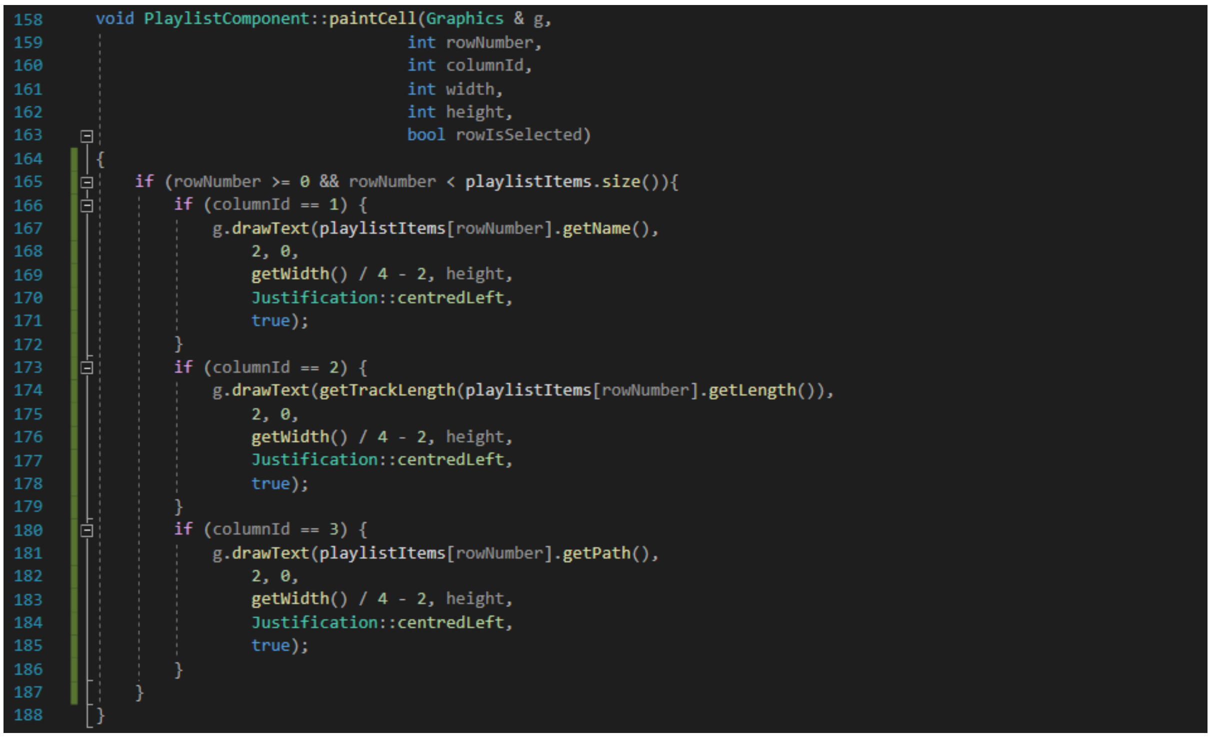
R3B: Component parses and displays meta data such as filename and song length:



```
35        tableComponent.getHeader().addColumn("Track Name", 1, 200);
36        tableComponent.getHeader().addColumn("Song length", 2, 200);
37        tableComponent.getHeader().addColumn("File Path", 3, 200);
38        tableComponent.getHeader().addColumn("", 4, 100);
39        tableComponent.getHeader().addColumn("", 5, 100);
40        tableComponent.getHeader().addColumn("", 6, 100);
41        tableComponent.setModel(this);
42        addAndMakeVisible(tableComponent);
```

Figure 3B.1 – PlaylistComponent.cpp (line 35 – 42)

A TableListBox class variable called tableComponent is created. It is assigned a total of 6 columns. The first 3 columns will display the file name, song length and the file path. Each column is set to take up the width of 200 while the last 3 columns will take the width of 100.



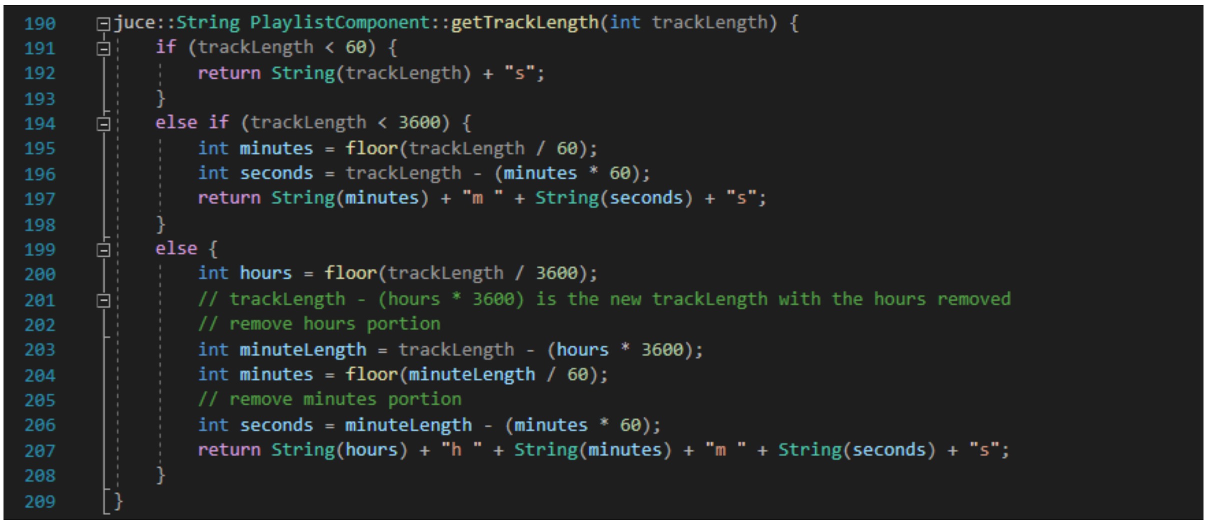
```
158  void PlaylistComponent::paintCell(Graphics & g,
159                                    int rowNumber,
160                                    int columnId,
161                                    int width,
162                                    int height,
163                                    bool rowIsSelected)
164  {
165      if (rowNumber >= 0 && rowNumber < playlistItems.size()){
166          if (columnId == 1) {
167              g.drawText(playlistItems[rowNumber].getName(),
168                  2, 0,
169                  getWidth() / 4 - 2, height,
170                  Justification::centredLeft,
171                  true);
172          }
173          if (columnId == 2) {
174              g.drawText(getTrackLength(playlistItems[rowNumber].getLength()),
175                  2, 0,
176                  getWidth() / 4 - 2, height,
177                  Justification::centredLeft,
178                  true);
179          }
180          if (columnId == 3) {
181              g.drawText(playlistItems[rowNumber].getPath(),
182                  2, 0,
183                  getWidth() / 4 - 2, height,
184                  Justification::centredLeft,
185                  true);
186          }
187      }
188  }
```

Figure 3B.2 – PlaylistComponent.cpp (line 158 – 188)

The paintCell() function of the TableListBox is overridden and will draw the data based on the rowNumber and columnId. We first check if the rowNumber is greater than or equal to 0 and less than the respective vector size.

The reason for checking if the rowNumber is greater than or equal to 0 is that during some of my testing, I have found that the rowNumber may go below 0 and causes an out-of-range error as we attempt to get the negative rowNumber element index of each vectors.

Next, we check the columnId to see which should be drawn. Based on the columnId, we retrieve the respective data by using the functions in the PlaylistItems class to retrieve the fileName, fileLength or filePath.



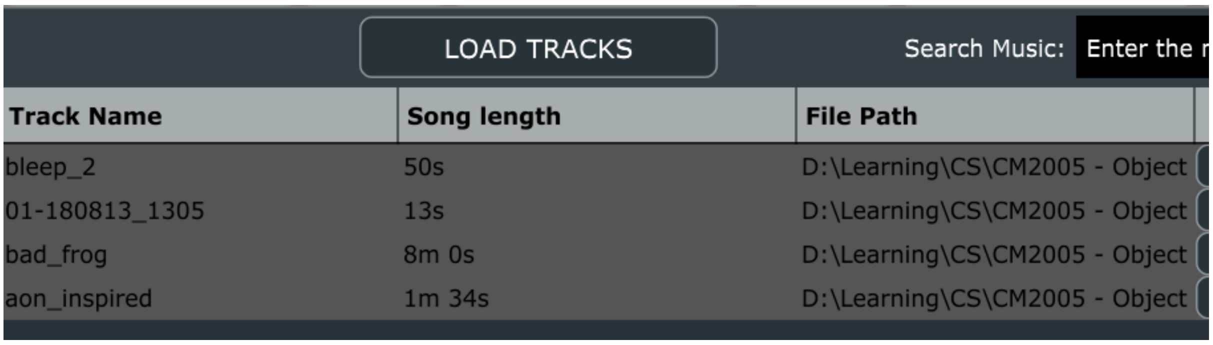
```
juce::String PlaylistComponent::getTrackLength(int trackLength) {
    if (trackLength < 60) {
        return String(trackLength) + "s";
    }
    else if (trackLength < 3600) {
        int minutes = floor(trackLength / 60);
        int seconds = trackLength - (minutes * 60);
        return String(minutes) + "m " + String(seconds) + "s";
    }
    else {
        int hours = floor(trackLength / 3600);
        // trackLength - (hours * 3600) is the new trackLength with the hours removed
        // remove hours portion
        int minuteLength = trackLength - (hours * 3600);
        int minutes = floor(minuteLength / 60);
        // remove minutes portion
        int seconds = minuteLength - (minutes * 60);
        return String(hours) + "h " + String(minutes) + "m " + String(seconds) + "s";
    }
}
```

Figure 3B.3 – PlaylistComponent.cpp (line 190 – 209)

When drawing the text for song length column, the getTrackLength() function will be called which converts the track length in seconds to a juce::String in "-h -m -s" format. This function is done by first checking if the trackLength is less than 1 minute(60s). If true we will return the trackLength with an additional "s" to represent second.

Else, we will check if trackLength is less than 1 hour(3600s). We will first divide the trackLength by 60 and apply the floor() function to get the quotient. We then subtract trackLength by the number of minutes * 60 or in other words, total number of seconds for those minutes. This will give us the remaining seconds.

Lastly, we will retrieve the hours value similar to how the minute value is retrieved but we will have to perform the flooring and division twice to get both hours and minutes before we can get the seconds value.



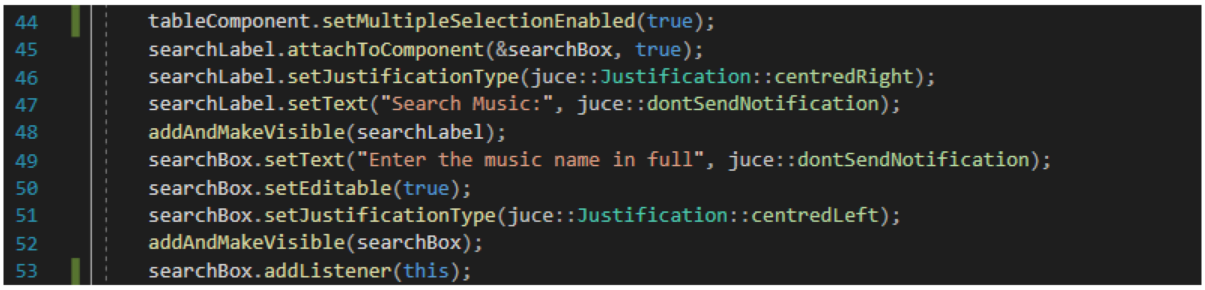
Track Name	Song length	File Path
bleep_2	50s	D:\Learning\CS\CM2005 - Object
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object

Figure 3B.4 – OtoDecks

The result of the tableComponent is as shown in figure 3B.4.

(R3B Achieved)

R3C: Component allows the user to search for files:



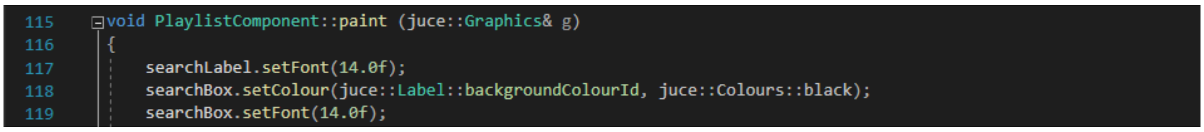
```
44      tableComponent.setMultipleSelectionEnabled(true);
45      searchLabel.attachToComponent(&searchBox, true);
46      searchLabel.setJustificationType(juce::Justification::centredRight);
47      searchLabel.setText("Search Music:", juce::dontSendNotification);
48      addAndMakeVisible(searchLabel);
49      searchBox.setText("Enter the music name in full", juce::dontSendNotification);
50      searchBox.setEditable(true);
51      searchBox.setJustificationType(juce::Justification::centredLeft);
52      addAndMakeVisible(searchBox);
53      searchBox.addListener(this);
```

Figure 3C.1 – PlaylistComponent.cpp (line 44 – 53)

We first have to enable the tableComponent to accept multiple row selections so that any files that matches the search text will be selected. searchLabel and searchBox are both variables of Label class. searchLabel will act as the label for the search bar and the searchBox will allow users to enter the song name to be searched.

searchLabel is first attached to searchBox, then given a justification to the right and the text of “Search Music:”.

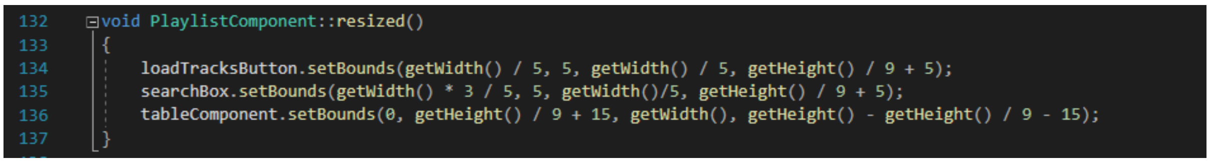
Similar to searchLabel, the preset text and justification type is given to searchBox but searchBox is set to be editable which allows user to edit the text.



```
115  void PlaylistComponent::paint (juce::Graphics& g)
116  {
117      searchLabel.setFont(14.0f);
118      searchBox.setColour(juce::Label::backgroundColourId, juce::Colours::black);
119      searchBox.setFont(14.0f);
```

Figure 3C.2 – PlaylistComponent.cpp (line 115 – 119)

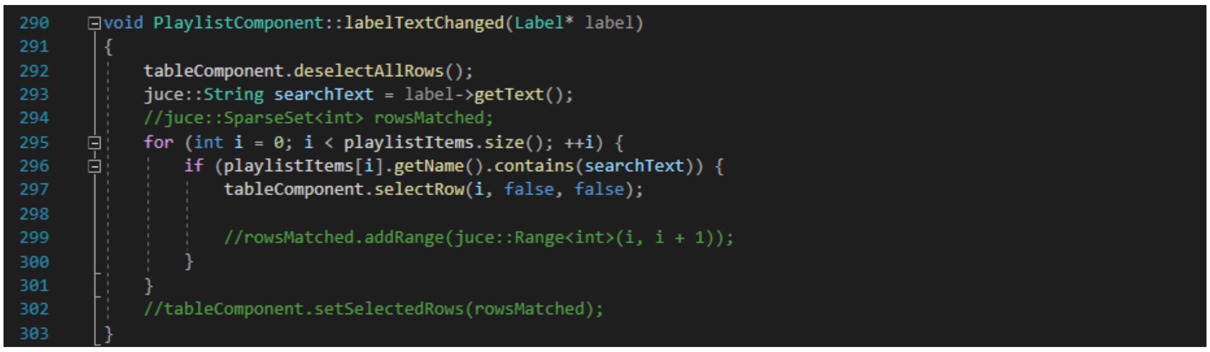
Both searchLabel and searchBox fonts are both set to 14.0f. Background colour of searchBox is changed to black so that the search box can be easily distinguished.



```cpp
void PlaylistComponent::resized()
{
    loadTracksButton.setBounds(getWidth() / 5, 5, getWidth() / 5, getHeight() / 9 + 5);
    searchBox.setBounds(getWidth() * 3 / 5, 5, getWidth()/5, getHeight() / 9 + 5);
    tableComponent.setBounds(0, getHeight() / 9 + 15, getWidth(), getHeight() - getHeight() / 9 - 15);
}
```

Figure 3C.3 – PlaylistComponent.cpp (line 132 – 137)

The searchBox is set to the same height as loadTracksButton() but of a different width to take up position on the right of the loadTracksButton.



```cpp
void PlaylistComponent::labelTextChanged(Label* label)
{
    tableComponent.deselectAllRows();
    juce::String searchText = label->getText();
    //juce::SparseSet<int> rowsMatched;
    for (int i = 0; i < playlistItems.size(); ++i) {
        if (playlistItems[i].getName().contains(searchText)) {
            tableComponent.selectRow(i, false, false);

            //rowsMatched.addRange(juce::Range<int>(i, i + 1));
        }
    }
    //tableComponent.setSelectedRows(rowsMatched);
}
```

Figure 3C.4 – PlaylistComponent.cpp (line 290 – 303)

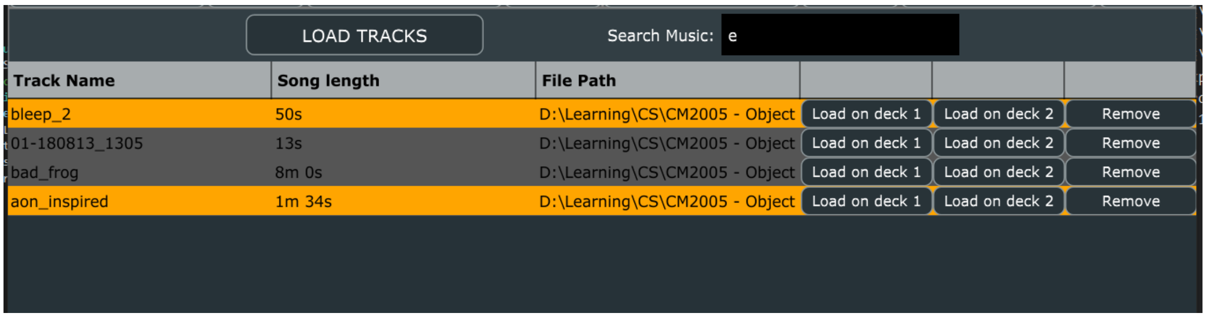
labelTextChanged() is called whenever the user edits the labelBox and confirms the changes. Any rows selected currently will first be deselected before proceeding.

The new value entered into the searchBox can be retrieved by simply using label->getText() as it is the only Label that is able to be edited, calling this function.

We will then iterate through the playlistItems to check if any track name contains the searched text. If true, we will select the row of index i while setting dontScrollToShowThisRow and deselectOthersFirst to false. This will stop the application from scrolling to the last matched row and also prevent any deselection of rows.

If we do not set the tableComponent to enable multiple row selections, changing dontScrollToShowThisRow and deselectOthersFirst to false would not work. Another option is to use SpareSet<int> to record the range of rows that contains the search text by adding a range starting from index i to i + 1. A range of (3, 7) equates to (3, 4, 5, 6).

After completing the loop, we can simply setSelectedRows() and passing the range in as its parameter. This will select all the rows inside that range.



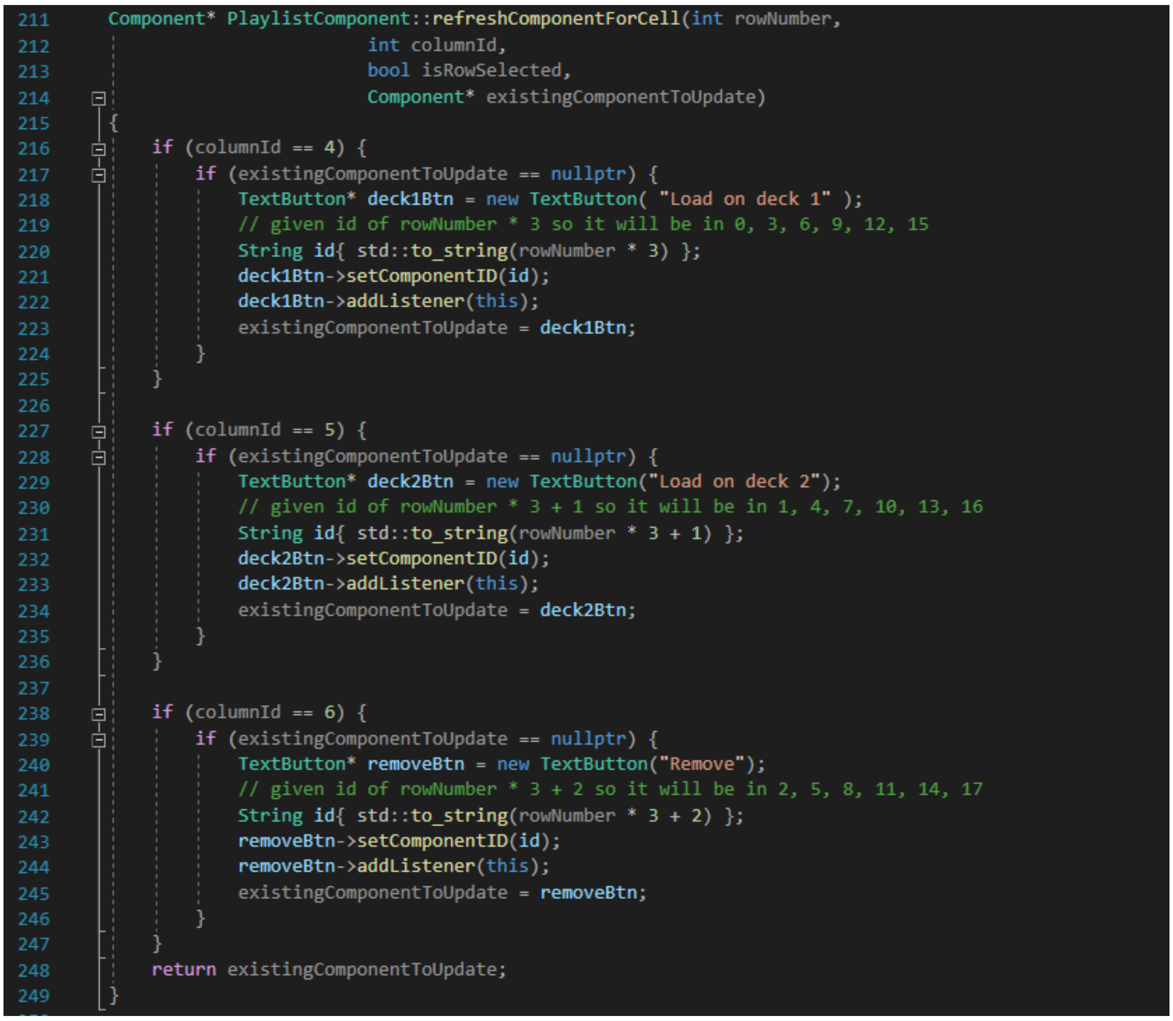
Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 3C.5 – OtoDecks

When searching for tracks with e in it, “bleep_2” and “aon_inspired” will be selected.

(R3C Achieved)

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



```cpp
Component* PlaylistComponent::refreshComponentForCell(int rowNumber,
                        int columnId,
                        bool isRowSelected,
                        Component* existingComponentToUpdate)
{
    if (columnId == 4) {
        if (existingComponentToUpdate == nullptr) {
            TextButton* deck1Btn = new TextButton( "Load on deck 1" );
            // given id of rowNumber * 3 so it will be in 0, 3, 6, 9, 12, 15
            String id{ std::to_string(rowNumber * 3) };
            deck1Btn->setComponentID(id);
            deck1Btn->addListener(this);
            existingComponentToUpdate = deck1Btn;
        }
    }

    if (columnId == 5) {
        if (existingComponentToUpdate == nullptr) {
            TextButton* deck2Btn = new TextButton("Load on deck 2");
            // given id of rowNumber * 3 + 1 so it will be in 1, 4, 7, 10, 13, 16
            String id{ std::to_string(rowNumber * 3 + 1) };
            deck2Btn->setComponentID(id);
            deck2Btn->addListener(this);
            existingComponentToUpdate = deck2Btn;
        }
    }

    if (columnId == 6) {
        if (existingComponentToUpdate == nullptr) {
            TextButton* removeBtn = new TextButton("Remove");
            // given id of rowNumber * 3 + 2 so it will be in 2, 5, 8, 11, 14, 17
            String id{ std::to_string(rowNumber * 3 + 2) };
            removeBtn->setComponentID(id);
            removeBtn->addListener(this);
            existingComponentToUpdate = removeBtn;
        }
    }
    return existingComponentToUpdate;
}
```

Figure 3D.1 – PlaylistComponent.cpp (line 211 – 249)

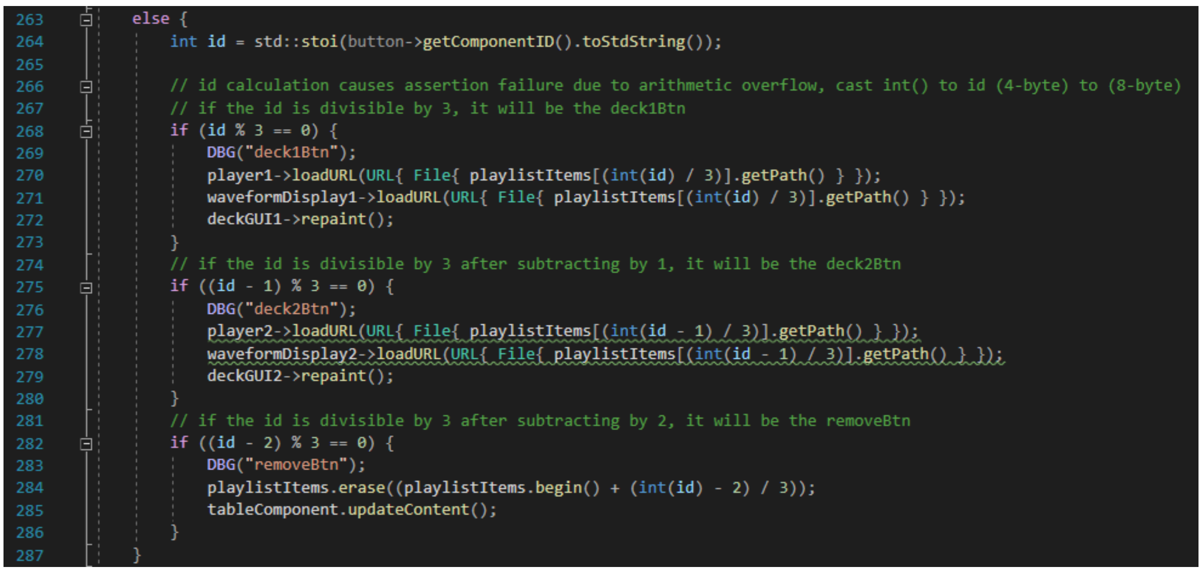
As seen previously in figure 3B.1, there are 3 additional columns that are added into the tableComponent with columnIds of 4, 5 and 6. In the refreshComponentForCell() function, we will be giving each column a different button with a unique component id based on which column it is.

The button in the first column will be given an id of rowNumber * 3. As the rowNumber starts with 0, buttons in this column will have an id starting from 0, 3, 6, 9 and so on.

The button in the second column will be given an id of rowNumber * 3 + 1. Buttons in this column will have an id starting from 1, 4, 7, 10 and so on.

The button in the third column will be given an id of rowNumber * 3 + 2. Buttons in this column will have an id starting from 2, 5, 8, 11 and so on.

Whenever there are any changes to the items inside the tableComponent, the component ids will be reassigned for the components.



```cpp
    else {
        int id = std::stoi(button->getComponentID().toStdString());

        // id calculation causes assertion failure due to arithmetic overflow, cast int() to id (4-byte) to (8-byte)
        // if the id is divisible by 3, it will be the deck1Btn
        if (id % 3 == 0) {
            DBG("deck1Btn");
            player1->loadURL(URL{ File{ playlistItems[(int(id) / 3)].getPath() } });
            waveformDisplay1->loadURL(URL{ File{ playlistItems[(int(id) / 3)].getPath() } });
            deckGUI1->repaint();
        }
        // if the id is divisible by 3 after subtracting by 1, it will be the deck2Btn
        if ((id - 1) % 3 == 0) {
            DBG("deck2Btn");
            player2->loadURL(URL{ File{ playlistItems[(int(id - 1) / 3)].getPath() } });
            waveformDisplay2->loadURL(URL{ File{ playlistItems[(int(id - 1) / 3)].getPath() } });
            deckGUI2->repaint();
        }
        // if the id is divisible by 3 after subtracting by 2, it will be the removeBtn
        if ((id - 2) % 3 == 0) {
            DBG("removeBtn");
            playlistItems.erase((playlistItems.begin() + (int(id) - 2) / 3));
            tableComponent.updateContent();
        }
    }
```

Figure 3D.2 – PlaylistComponent.cpp (line 261– 287)

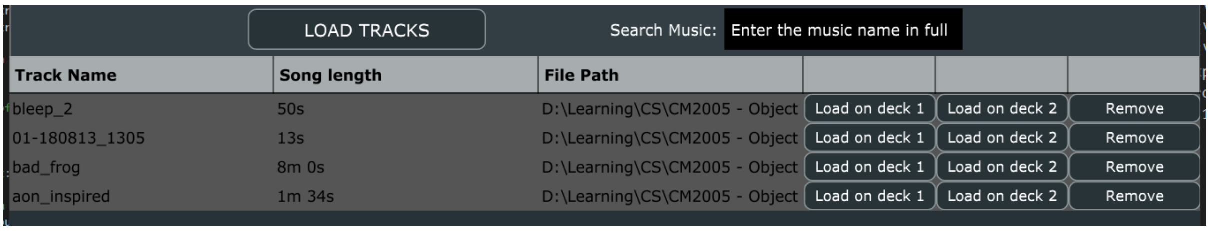
In the buttonClicked() function, it first checks if the button clicked is the loadTracksButton. In the case that either of these 3 buttons are clicked, that condition will be false and the else{} codes will run instead, as seen above in figure 3D.2.

We first retrieve the int id by getting the component id from the button before converting it to std::string so that it can be converted to int, since juce::String cannot be converted to integer.

The integer id is then used to check if it is divisible by 3. If true, the load track to deck 1 button is clicked. If false, we will check if the id is divisible by 3 after subtracting it by 1. If true, the load track to deck 2 button is clicked. If false, we will check if the id is divisible by 3 after subtracting it by 2. If true, the remove button is clicked.

For both load to deck 1 and 2 buttons, we will call the respective player pointers loadURL function to load the track into the respective players followed by calling the respective waveform display pointers loadURL function which will update the respective waveform displays. We will next repaint the respective decks' GUIs.

The remove button can be done by using the erase() function with the index of vector.begin() + index. Index can be calculated by subtracting the id by 2 before dividing it by 3 and the vector to be used is playlistItems.

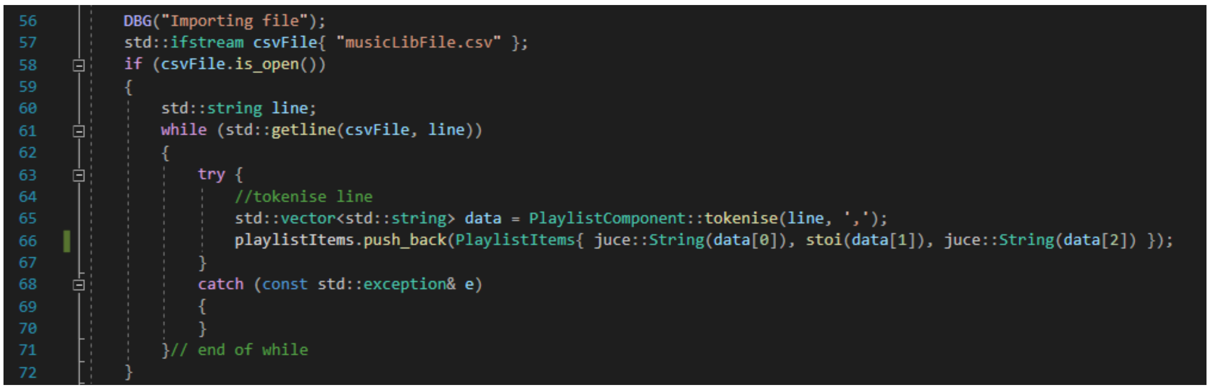


Track Name	Song length	File Path			
bleep_2	50s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
01-180813_1305	13s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
bad_frog	8m 0s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove
aon_inspired	1m 34s	D:\Learning\CS\CM2005 - Object	Load on deck 1	Load on deck 2	Remove

Figure 3D.3 – OrderBook.cpp (line 1124 – 1133)

(R3D Achieved)

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



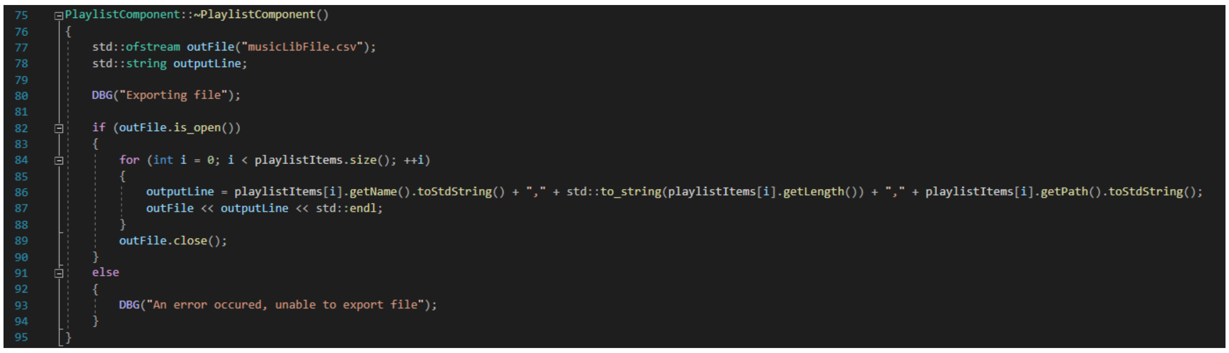
```
    DBG("Importing file");
    std::ifstream csvFile{ "musicLibFile.csv" };
    if (csvFile.is_open())
    {
        std::string line;
        while (std::getline(csvFile, line))
        {
            try {
                //tokenise line
                std::vector<std::string> data = PlaylistComponent::tokenise(line, ',');
                playlistItems.push_back(PlaylistItems{ juce::String(data[0]), stoi(data[1]), juce::String(data[2]) });
            }
            catch (const std::exception& e)
            {
            }
        }// end of while
    }
```

Figure 3E.1 – PlaylistComponent.cpp (line 56 – 72)

In the PlaylistComponent constructor, we will attempt to open a csv file of file name “musicLibFile.csv”. If founds, we will loop through each line of the csv file and call the tokenise() function that will separate the string values by the delimiter of ‘,’.

We will create a PlaylistItems object with the tokenised data, converting them into the correct data type at the same time. The tokenised data vector holds the trackName, trackLength and trackPath in the respective order for its index. The trackName (data[0]) and trackPath(data[2]) will have to be converted to juce::String from std::string while the trackLength(data[1]) will have to be converted back to integer.

If file is not found, no preloaded tracks will be displayed in the tableComponent.



```cpp
PlaylistComponent::~PlaylistComponent()
{
    std::ofstream outFile("musicLibFile.csv");
    std::string outputLine;

    DBG("Exporting file");

    if (outFile.is_open())
    {
        for (int i = 0; i < playlistItems.size(); ++i)
        {
            outputLine = playlistItems[i].getName().toStdString() + "," + std::to_string(playlistItems[i].getLength()) + "," + playlistItems[i].getPath().toStdString();
            outFile << outputLine << std::endl;
        }
        outFile.close();
    }
    else
    {
        DBG("An error occured, unable to export file");
    }
}
```

Figure 3E.2 – PlaylistComponent.cpp (line 75 – 95)

In the PlaylistComponent deconstructor, we will designate an output file of “musicLibFile.csv” and check if it is open and ready for writing. We must check if the file is open as there may be a possibility that a csv file of that name is already opened, preventing the program from writing into it.

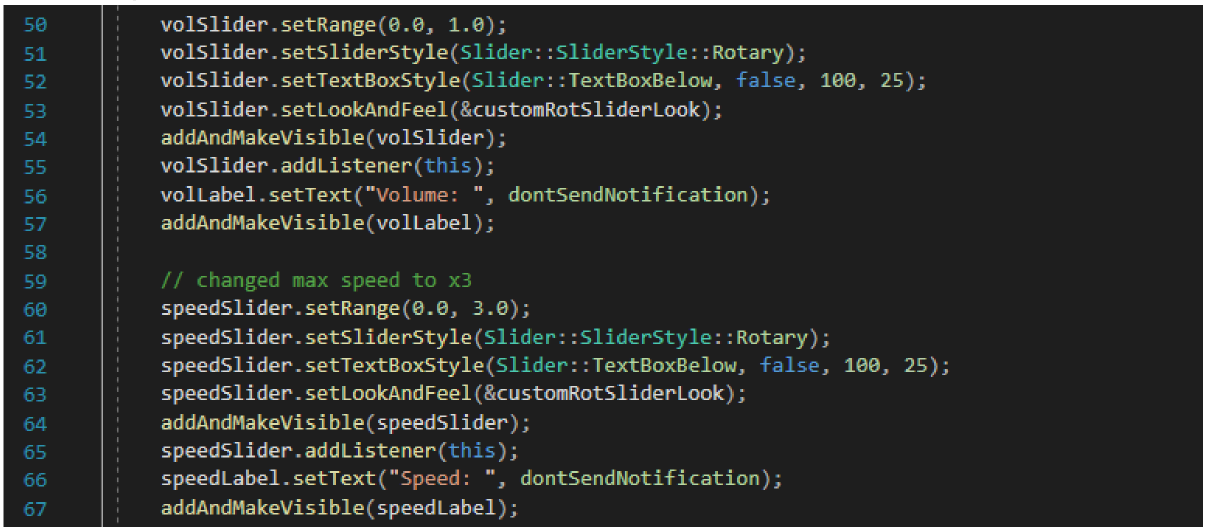
If file is open, we will iterate through the playlistItems and generate a std::string in the format “trackName,trackLength,trackPath” before writing it into the csv file line. After iterating through every existing track in the list, we will close the file.

When the user is exiting the application, the deconstructor will be called.

(R3E Achieved)

R4: Implementation of a complete custom GUI

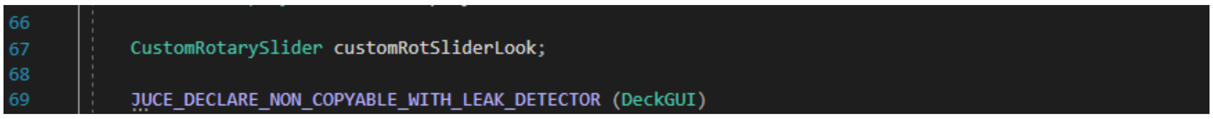
R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls:



```cpp
    volSlider.setRange(0.0, 1.0);
    volSlider.setSliderStyle(Slider::SliderStyle::Rotary);
    volSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 25);
    volSlider.setLookAndFeel(&customRotSliderLook);
    addAndMakeVisible(volSlider);
    volSlider.addListener(this);
    volLabel.setText("Volume: ", dontSendNotification);
    addAndMakeVisible(volLabel);

    // changed max speed to x3
    speedSlider.setRange(0.0, 3.0);
    speedSlider.setSliderStyle(Slider::SliderStyle::Rotary);
    speedSlider.setTextBoxStyle(Slider::TextBoxBelow, false, 100, 25);
    speedSlider.setLookAndFeel(&customRotSliderLook);
    addAndMakeVisible(speedSlider);
    speedSlider.addListener(this);
    speedLabel.setText("Speed: ", dontSendNotification);
    addAndMakeVisible(speedLabel);
```

Figure 4A.1 – DeckGUI.cpp (line 50 – 67)

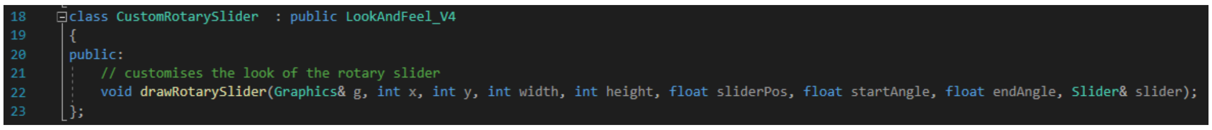
The volume and speed slider are first given a slider style of rotary. It is then customised by setting its look and feel with a pointer for a custom class of CustomRotarySlider object.



```
66
67        CustomRotarySlider customRotSliderLook;
68
69        JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (DeckGUI)
```

Figure 4A.2 – DeckGUI.h (line 66 – 69)

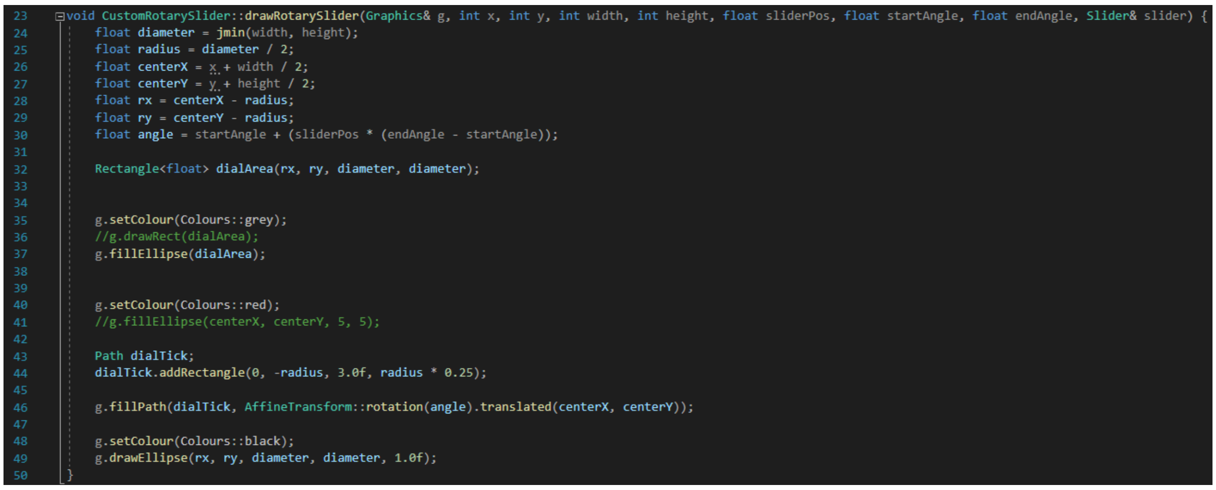
This CustomRotarySlider was created in the header.



```
18  class CustomRotarySlider  : public LookAndFeel_V4
19  {
20  public:
21      // customises the look of the rotary slider
22      void drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos, float startAngle, float endAngle, Slider& slider);
23  };
```

Figure 4A.3 – CustomRotarySlider.h (line 18 – 23)

CustomRotarySlider inherits the LookAndFeel_V4 class. The LookAndFeel_V4 class has various draw functions that allow us to customise the look of buttons, boxes, bars and more.



```
23  void CustomRotarySlider::drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos, float startAngle, float endAngle, Slider& slider) {
24      float diameter = jmin(width, height);
25      float radius = diameter / 2;
26      float centerX = x + width / 2;
27      float centerY = y + height / 2;
28      float rx = centerX - radius;
29      float ry = centerY - radius;
30      float angle = startAngle + (sliderPos * (endAngle - startAngle));
31
32      Rectangle<float> dialArea(rx, ry, diameter, diameter);
33
34
35      g.setColour(Colours::grey);
36      //g.drawRect(dialArea);
37      g.fillEllipse(dialArea);
38
39
40      g.setColour(Colours::red);
41      //g.fillEllipse(centerX, centerY, 5, 5);
42
43      Path dialTick;
44      dialTick.addRectangle(0, -radius, 3.0f, radius * 0.25);
45
46      g.fillPath(dialTick, AffineTransform::rotation(angle).translated(centerX, centerY));
47
48      g.setColour(Colours::black);
49      g.drawEllipse(rx, ry, diameter, diameter, 1.0f);
50  }
```

Figure 4A.4 – CustomRotarySlider.cpp (line 23 – 50)

The diameter is set to the smaller value between the width and height using the jmin() function. This prevents that slider from being oversized, pushing the label and textbox out of the set boundaries.

As defined in the class reference, x is the x coordinate of the top-left of the rectangle within which we should draw our rotary slider and y is the y coordinate of the top-left of the rectangle within which we should draw our rotary slider.

By subtracting centerX and centerY by the radius, we will get the coordinate which the dial tick of the slider should be positioned at. The angle of the dial tick is then calculated with the formula in line 30.

A grey rectangle is then used to fill the ellipse of the slider area.

In order to draw the dial tick, we will use Path to draw the rectangle and set it to the correct position by using fillPath() and base its rotation angle from the centre coordinates.

The outline of the slider is then drawn with a black colour ellipse.

Other customisations and extra playback controls are described in R2 and R3.

(R4A Achieved)

R4B: GUI layout includes the custom Component from R2:

As described and seen in R2, the sliders, waveform display and buttons are given its colours and various customised looks while still being part of the component.

(R4B Achieved)

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

As described and seen in R3, the library's features and various controls for each track rows allows user to control loading and deletion of tracks. Users can also search for tracks within the library and track data persists when exited and rerun again. All of this can be found in R3 library component.

(R4C Achieved)

Summary:

The features of OtoDecks application functions as it should and has achieved its requirements. However, a problem was found through the testing was that when the library has a certain number of tracks that makes the library scrollable, the component id of the tracks that are not on screen but in the list are not given. The id is only assigned when scrolled and it appears on screen.

The problem comes with how the id are assigned. Tracks that are leaving the screen no longer holds the component id while the new tracks that comes on screen may or may not replace those removed ids. However, those button components with removed ids will get back the same id as it was previously assigned.

This causes the problem of having multiple buttons with the same id. Whenever user tries to load any of the duplicate tracks into the deck, the first track that is ever assigned with that id will be played. Thus, as long as the library is not scrolled, the OtoDecks application works perfectly as it should.