Otter Decks

Object Oriented Programming Final Report

September 2022

Student Number: 210170537

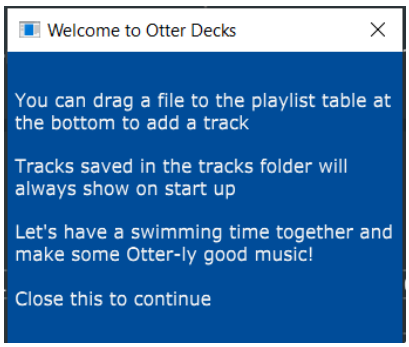
Contents

Introduction

In this report I will outline the changes that I have made to the basic DJ application Otodecks that was given as a starting point for the Object Oriented Programming module. First and foremost, I renamed my version of this application “Otter Decks”, and followed a subsequent “water” theme whereby the colours used in the application are all derivatives of a deep blue. The IDE used to create this application was Visual Studio 2022, the version of JUCE used was v7.0.0. The otter image used repeatedly in this project is from [1] Spreadshirt, 2022.

The final application looks as follows:

With a popup that looks as follows:



R1 Requirements

- R1A: can load audio files into audio players ✓

This has been achieved. This can be done by either directly dropping a file in the DeckGUI or by dropping the file in the PlaylistComponent. Once the file has been dropped the subsequent methods are called to prepare the audio file to play. It is important to note that all of this is being done with the file's "URL". This is why a vector is used to store URLs and another vector is used to store trackTitles.

This implementation can be seen in the DeckGUI.cpp file on lines 233-247:

```
233 bool DeckGUI::isInterestedInFileDrag(const juce::StringArray& files)
234 {
235
236     return true;
237 }
238 void DeckGUI::filesDropped(const juce::StringArray& files, int x, int y)
239 {
240
241     if (files.size() == 1)
242     {
243         player->loadURL(juce::URL{ juce::File{files[0]} });
244         waveformDisplay->loadURL(juce::URL{ juce::File{files[0]} });
245         header->setHeader( title: juce::File{ files[0] } .getFileName().toStdString(), player->getSongLength(), header->getRowIndex());
246     }
247 }
```

This implementation can also be seen in the PlaylistComponent.cpp file on lines 381-399:

```
381 bool PlaylistComponent::isInterestedInFileDrag(const juce::StringArray& files) {
382     return true;
383 }
384
385 void PlaylistComponent::filesDropped(const juce::StringArray& files, int x, int y) {
386     //load up multiple files dropped in the playlist component
387     if (files.size() >= 1)
388     {
389         for (int i = 0; i < files.size(); i++) {
390             trackTitles.push_back( _val: juce::File{ files[i] } .getFileNameWithoutExtension());
391             trackURLs.push_back(juce::URL{ juce::File{files[i]} });
392         }
393         tableComponent.updateContent();
394
395         //make sure the track data is always up to date
396         header1->setTrackData( trackURLs);
397         header2->setTrackData( trackURLs);
398     }
399 }
```

Files can also be loaded by choosing either "Left Deck" or "Right Deck" on the Playlist Component, this assigns the track to the corresponding deck and chooses the file URL based on its position in the playlist. This implementation is seen in PlaylistComponent.cpp on lines 350-362 under the buttonClicked method:

```
349         //Load song on left deck
350         if (idStr.at( _off: 0) == '4') {
351             player1->loadURL(juce::URL{ trackURLs[trackIndex] });
352             waveformDisplay1->loadURL(juce::URL{ trackURLs[trackIndex] });
353             header1->setHeader( _title: trackTitles[trackIndex], player1->getSongLength(),trackIndex);
354             header1->setTrackData( trackURLs);
355         }
356
357         //load song on right deck
358         if (idStr.at( _off: 0) == '5') {
359             player2->loadURL(juce::URL{ trackURLs[trackIndex] });
360             waveformDisplay2->loadURL(juce::URL{ trackURLs[trackIndex] });
361             header2->setHeader( _title: trackTitles[trackIndex], player2->getSongLength(),trackIndex);
362             header2->setTrackData( trackURLs);
363         }
```

- R1B: can play two or more tracks ✓

This has been achieved. There are 2 instances of the DJAudioPlayer class that are shown side by side. These players can then play music at the same time. This can be seen in the screenshots of the full implementation of Otter Decks on page 1. The two players are instantiated in the MainComponent.cpp file at lines 41-63 in the methods prepareToPlay, getNextAudioBlock and releaseResources:

```
41 void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
42 {
43     player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
44     player2.prepareToPlay(samplesPerBlockExpected, sampleRate);
45
46     mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
47
48     mixerSource.addInputSource(&player1, deleteWhenRemoved: false);
49     mixerSource.addInputSource(&player2, deleteWhenRemoved: false);
50
51 }
52
53 void MainComponent::getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill)
54 {
55     mixerSource.getNextAudioBlock(bufferToFill);
56 }
57
58 void MainComponent::releaseResources()
59 {
60     player1.releaseResources();
61     player2.releaseResources();
62     mixerSource.releaseResources();
63 }
64
65 //==============================================================================
```

Then within the DJAudioPlayer class, the start() and stop() methods use the AudioTransportSource to start and stop the track playing on each player on lines 52-59:

```
52 void DJAudioPlayer::play()
53 {
54     transportSource.start();
55 }
56 void DJAudioPlayer::stop()
57 {
58     transportSource.stop();
59 }
```

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

This has been achieved. The volume slider can be changed from the DeckGUI class. There is a listener called sliderValueChanged that detects when the slider has been moved for each deck. This then calls the DJAudioPlayer class method setGain and changes the volume. Since the two DJAudioPlayers have been already instantiated as shown above, the change in volume is specific to each DJAudioPlayer. This volume slider listener is in the DeckGUI.cpp file on line 218-221:

```
216 void DeckGUI::sliderValueChanged(juce::Slider* slider)
217 {
218     if (slider == &volSlider)
219     {
220         player->setGain(slider->getValue());
221     }
222     if (slider == &speedSlider)
223     {
224         player->setSpeed(ratio: slider->getValue());
225     }
226     if (slider == &posSlider)
227     {
228         player->setPositionRelative(pos: slider->getValue());
229     }
230 }
```

The setGain() method then uses the AudioTransportSource to set the gain of the track to whatever value was chosen from the volume slider. This can be seen on lines 89-96 in the DJAudioPlayer.cpp file:

```
87 void DJAudioPlayer::setGain(double gain)
88 {
89     if(gain < 0 || gain > 1.0)
90     {
91         DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
92
93     }else
94     {
95         transportSource.setGain(gain);
96     }
97
98 }
```

- R1D: can speed up and slow down tracks ✓

  This has been achieved. The speed slider can be changed from the DeckGUI class. There is a listener called sliderValueChanged that detects when the slider has been moved for each deck. This then calls the DJAudioPlayer class method setSpeed and changes the speed of the tracks. I also changed the maximum value of the speed slider to be 20 because anything more than this just sounded like noise to me. The slider listener is implemented in the DeckGUI.cpp file on line 222-226:

```
216 void DeckGUI::sliderValueChanged(juce::Slider* slider)
217 {
218     if (slider == &volSlider)
219     {
220         player->setGain(slider->getValue());
221     }
222     if (slider == &speedSlider)
223     {
224         player->setSpeed(ratio: slider->getValue());
225     }
226     if (slider == &posSlider)
227     {
228         player->setPositionRelative(pos: slider->getValue());
229     }
230 }
```

The setSpeed() method then uses the ResamplingAudioSource class to set the speed of the track to whatever value was chosen from the speed slider. This can be seen on lines 100-111 in the DJAudioPlayer.cpp file:

```
100 void DJAudioPlayer::setSpeed(double ratio)
101 {
102     if (ratio < 0 || ratio > 100)
103     {
104         DBG("DJAudioPlayer::setSpeed ratio should be between 0 and 100");
105
106     }
107     else
108     {
109         resampleSource.setResamplingRatio(ratio);
110     }
111 }
```

R2 Requirements

- R2A: Component has custom graphics implemented in paint function ✓

  This has been achieved. There are various examples of custom graphics being used throughout the application, but some examples are described here. The paint function in the DeckGUI class has several functions to change the colour of buttons on hover, the slider colour has been changed and the colour of several other items in this class have been changed, the paint method contains many visual changes in the DeckGUI class:

```
85  void DeckGUI::paint (juce::Graphics& g)
86  {
87      //grey background
88      g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));
89
90      // border color
91      g.setColour (juce::Colours::grey);
92
93      // draw an outline around the component
94      g.drawRect (rectangle: getLocalBounds(), lineThickness: 1);
95
96      //text color
97      g.setColour (juce::Colours::white);
98      g.setFont (14.0f);
99
100     //Change slider knob color
101     getLookAndFeel().setColour(juce::Slider::thumbColourId, juce::Colour(red: 0, green: 102, blue: 204));
102
103     //Button hover color change
104     if (playButton.isOver())
105     {
106         playButton.setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 102, blue: 204));
107     }
108     else
109     {
110         playButton.setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 51, blue: 102));
111     }
112
113     if (prevButton.isOver())
114     {
115         prevButton.setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 102, blue: 204));
116     }
117     else
118     {
119         prevButton.setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 51, blue: 102));
120     }
121     if (nextButton.isOver())
122     {
123         nextButton.setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 102, blue: 204));
124     }
```

Another good example where this is done is in the extra class LogoHeader, where an image is retrieved from local storage and is formatted to fit into the top space of the application within the class LogoHeader. This can be seen in the LogoHeader.cpp file on lines 24-48:

```
void LogoHeader::paint (juce::Graphics& g)
{
    //background dark blue
    g.fillAll (juce::Colour(red: 204, green: 229, blue: 255));

    //outline dark blue
    g.setColour (juce::Colour(red: 0, green: 51, blue: 102));
    g.drawRect (rectangle: getLocalBounds(), lineThickness: 3);

    //colour of box with logo light blue
    g.setColour (juce::Colour(red: 294, green: 229, blue: 255));
    g.setFont (14.0f);

    //get Logo from local file

    //image was sourced from
    ////https://www.spreadshirt.ie/shop/design/otter+with+headphones+gift+funny+birthday+sticker-D5d4187b1b264a1173d740552?sellable=R43wBOoEzZSeAr5ywvMb-1459-215
    //text was made in Microsoft Word and then cut to create logo
    logo = juce::ImageFileFormat::loadFrom(juce::File::getCurrentWorkingDirectory().getChildFile("OtterDecks.JPG"));

    //rescale logo to fit in header
    logo = logo.rescaled(newWidth: 360, newHeight: 94, juce::Graphics::mediumResamplingQuality);

    //draw logo
    g.drawImageAt(logo, topLeftX: 300, topLeftY: 3, fillAlphaChannelWithCurrentBrush: false);
}
```

The TrackHeader class also uses a custom paint method which allows a textbox to be styled in a visually appealing way to display the track names when they are assigned to a deck. This implementation can be seen in the TrackHeader.cpp file on lines 32-45:

```
void TrackHeader::paint (juce::Graphics& g)
{
    //formating for title text
    titleBox = displayTitle;
    title.setBoundingBox(titleBox);

    g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));    // clear the background

    //Filling the background with a dark blue and a light blue inner box
    g.fillAll(juce::Colour(red: 204, green: 229, blue: 255));
    g.setColour(juce::Colour(red: 0, green: 51, blue: 102));

    // draw an outline around the component
    g.drawRect(rectangle: getLocalBounds(), lineThickness: 3);
}
```

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

This has been achieved. The application has two main functions that control playback differently to the usual start and stop. These are the next/prev track buttons and the loop radio button. The “nextTrack” and “previousTrack” methods allow the user to skip to the next track on the playlist or go to the previous track on the playlist.

The DeckGUI class has the next and previous buttons which have buttonListeners attached to them, when the button next or previous is clicked the following code will run which is found in DeckGUI.cpp under the buttonClicked method on lines 174-197:

```
if (button == &nextButton)
{
    //track vector stored in header
    const juce::URL nextTrack = header->getNextTrack();

    if (nextTrack.isEmpty() == false)
    {
        //load track
        player->loadURL(juce::URL{ nextTrack });
        waveformDisplay->loadURL(juce::URL{ nextTrack });
        //change title text
        header->setHeader(_title: nextTrack.getFileName().toStdString(), player->getSongLength(), header->getRowIndex());
    }
}

if (button == &prevButton)
{
    const juce::URL prevTrack = header->getPrevTrack();
    if ( prevTrack.isEmpty() == false)
    {
        player->loadURL(juce::URL{ prevTrack });
        waveformDisplay->loadURL(juce::URL{ prevTrack });
        header->setHeader(_title: prevTrack.getFileName().toStdString(), player->getSongLength(), header->getRowIndex());
    }

}
```

This will firstly call the new class TrackHeader.cpp which was created to display the track titles and also hold the array of Track URLs so that they can be accessed easily from other parts of the program. The code to get the previous or next track is shown in file TrackHeader.cpp on lines 75-117, it accessed the stored vector and calculated the next or previous track while also checking if the vector has been deleted so that it can return a null URL which is caught from the calling program, this prevents the program from crashing when the playlist is empty:

```
75  juce::URL TrackHeader::getNextTrack()
76  {
77      rowIndex = rowIndex + 1;
78      const int last_element = trackURLs.size() - 1; //find the last element of the vector
79
80      if (rowIndex > last_element)
81      {
82          rowIndex = 0; //if we go past the last element rollback to the first element
83      }
84
85      //if the playlist has been emptied then return a nul URL so suitable error handling can happen
86      //this stops the program from crashing
87      if (trackURLs.empty() == false)
88      {
89          return trackURLs[rowIndex];
90      }
91      else
92      {
93          juce::URL juceURL = "";
94          return juceURL;
95      }
96
97  }
98  juce::URL TrackHeader::getPrevTrack()
99  {
100     rowIndex = rowIndex - 1;
101
102     const int last_element = trackURLs.size() - 1; //find the last element of the vector
103
104     if (rowIndex == -1)
105     {
106         rowIndex = last_element; //if go back past first element then restart from beginning
107     }
108
109     if (trackURLs.empty() == false)
110     {
111         return trackURLs[rowIndex];
112     }
113     else
114     {
115         juce::URL juceURL = "";
116         return juceURL;
117     }
118 }
```

Then the DJAudioPlayer class can take the returned URL and subsequently play the audio as per usual.

The other custom code addition to enhance playback is a loop radio button that has been added to the DeckGUI.cpp file which allows the user to make the track go into a loop. The main logic for this is in the DJAudioPlayer class whereby the PositionableAudioSource class has been inherited and the pure virtual functions have been implemented in the .cpp file. This implementation can be seen in the DJAudioPlayer.cpp file on lines 136-179:

```cpp
//Pure virtual functions implemented below because inheritence from PositionableAudioSource class
void DJAudioPlayer::setNextReadPosition(juce::int64 newPosition)
{

}

juce::int64 DJAudioPlayer::getNextReadPosition() const
{
    return readerSource->getNextReadPosition();
}

juce::int64 DJAudioPlayer::getTotalLength() const
{
    return readerSource->getTotalLength();
}

//Returning true if it is looping, or false if it is not
bool DJAudioPlayer::isLooping() const
{
    return false;
}

//Setting the playback to loop
void DJAudioPlayer::setLoop()
{
    //if the readerSource is not a null pointer, the readerSource is set to loop
    //using the function from the PositionableAudioSource class

    if (readerSource != nullptr)
    {
        readerSource->setLooping(shouldLoop: true);
    }
}

//Setting the playback not to loop
void DJAudioPlayer::unsetLoop()
{
    //if the readerSource is not a null pointer, the readerSource is set to not loop
    //using the function from the PositionableAudioSource class
    if (readerSource != nullptr)
    {
        readerSource->setLooping(shouldLoop: false);
    }
}
```

This then subsequently calls the method within the inherited class AudioFormatReaderSource and changes the looping parameters depending on the state of the loop radio button. This state of the loop button is set in the DeckGUI.cpp file on lines 201-212 under the method buttonClicked:

```cpp
    if (button == &loopButton)
    {

        if (loopButton.getToggleState() == true)
        {
            player->setLoop();
        }
        else
        {
            player->unsetLoop();
        }
    }

}
```

R3 Requirements

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

This has been achieved. The user can add files to the library by either dragging or dropping the files into the component or by reading a playlist textfile that has been created from a previous session. The drag and drop functionality can be found in the file PlaylistComponent.cpp on lines 385-414:

```
385 void PlaylistComponent::filesDropped(const juce::StringArray& files, int x, int y) {
386     //load up multiple files dropped in the playlist component
387     if (files.size() >= 1)
388     {
389         for (int i = 0; i < files.size(); i++) {
390             trackTitles.push_back( _Val: juce::File{ files[i] } .getFileNameWithoutExtension());
391             trackURLs.push_back(juce::URL{ juce::File{files[i]} });
392         }
393         tableComponent.updateContent();
394 
395         //make sure the track data is always up to date
396         header1->setTrackData( & trackURLs);
397         header2->setTrackData( & trackURLs);
398     }
399 }
400
```

This also subsequently updates the vector containing the trackURLs within the TrackHeader.cpp file using the setTrackData method.Once files have been added they can be swapped out for different files using the “Change File” button on each row. This gives the user the file chooser dialog box where they can choose a file from their library and then can swap out a file in-place with a new file in the playlist. This can be found in the PlaylistComponent.cpp file under the method buttonClicked on lines 323-346:

```
323     if (button != &saveButton && button != &newButton && button != &loadButton) {
324         //get the ID of the button clicked
325         std::string idStr = button->getComponentID().toStdString();
326         int id = std::stoi(idStr);
327          trackIndex = std::stoi( _Str: idStr.substr( _Off: 1));
328 
329         //change file
330         if (idStr.at( _Off: 0) == '3') {
331             //file selector pops up
332             auto fileChooserFlags = juce::FileBrowserComponent::canSelectFiles;
333             std::string idStr = button->getComponentID().toStdString();
334             fChooser.launchAsync(fileChooserFlags, & [this](const juce::FileChooser& chooser) ->void
335                 {
336                     const auto chosenFile = chooser.getResult();
337 
338                     if (chosenFile.exists()) {
339                         trackTitles[trackIndex] = chooser.getResult().getFileNameWithoutExtension();
340                         trackURLs[trackIndex] = chooser.getURLResult();
341                         tableComponent.updateContent();
342                         tableComponent.repaintRow(trackIndex);
343                         header1->setTrackData( & trackURLs);
344                         header2->setTrackData( & trackURLs);
345                     }
346                 });
347         }
```

A playlist file can also be read from a previous session and the details of the playlist is parsed out of a textfile. This can be seen in the buttonClicked method in the PlaylistComponent.cpp file on lines 284-321:

```
283     //load a playlist that has alraedy been created
284     if (button == &loadButton) {
285         trackURLs.clear();
286         trackTitles.clear();
287         tableComponent.updateContent();
288         auto fileChooserFlags = juce::FileBrowserComponent::canSelectFiles;
289         fChooser.launchAsync(fileChooserFlags, [this](const juce::FileChooser& chooser)->void
290             {
291                 const auto chosenFile = chooser.getResult();
292                 //check if the file exists
293                 if (chosenFile.exists())
294                 {
295                     //get a long string of what is inside of the file
296                     std::ifstream playlistLoad{ _Str:chooser.getResult().getFileName().toStdString() };
297                     //declare a line to hold our value per line
298                     std::string line;
299                     //have a small vector to seperate the data
300                     std::vector<std::string> fileLineData;
301                     if (playlistLoad.is_open()) {
302                         while (std::getline([&]playlistLoad, [&]line)) {
303                             try {
304                                 //tokenising our data
305                                 fileLineData = tokenise(line, separator:',');
306                                 //push to relevant arrays
307                                 trackURLs.push_back(juce::URL{ fileLineData[0] });
308                                 trackTitles.push_back(fileLineData[1]);
309                             }
310                             catch (const std::exception& e) {
311                                 DBG("CSVReader::readCSV bad data");
312                             }
313                         }
314                     }
315                     //keep our data up to date
316                     tableComponent.updateContent();
317                     header1->setTrackData(trackURLs);
318                     header2->setTrackData(trackURLs);
319                 }
320             });
321     }
```

This essentially uses the tokenise logic found in file PlaylistComponent.cpp in lines 451-465:

```
451 std::vector<std::string> PlaylistComponent::tokenise(std::string csvLine, char separator)
452 {
453     //tokenize the data from the file
454     std::vector<std::string> tokens;
455     signed int first = csvLine.find_first_not_of(separator, 0), end;
456     std::string token;
457     do {
458         end = csvLine.find_first_of(separator, first);
459         if (first == csvLine.length() || first == end) break;
460         if (end >= 0) token = csvLine.substr(first, end - first);
461         else token = csvLine.substr(first, csvLine.length() - first);
462         tokens.push_back(token);
463         first = end + 1;
464     } while (end > 0);
465     return tokens;
466 }
```

To parse through the text file and extract the track URL and the track titles and subsequently update the corresponding vectors to contain these new tracks and then display them in the PlaylistComponent table.

- R3B: Component parses and displays meta data such as filename and song length ✓
  This has been achieved. The user can see the song title and the song length. The song title is simply extracted from the trackTitles vector that is consistently being maintained using the rowIndex (of the table) to iterate through the vetor. This is called in the paintCell method of

the PlaylistComponent class and the code for this is found on lines 139-156 in the PlaylistComponent.cpp file:

```
139 void PlaylistComponent::paintCell(juce::Graphics& g, int rowNumber, int columnId, int width, int height, bool rowIsSelected)
140 {
141     //uses the temporary player to load the URLs of the tracks
142     tempPlayer->loadURL( file: juce::URL(trackURLs[rowNumber]));
143     if (columnId == 1)
144     {
145         g.drawText(trackTitles[rowNumber], x: 2,
146         y: 0, width - 4, height,
147         juce::Justification::centredLeft, useEllipsesIfTooBig: true);
148     }
149
150     if (columnId == 2)
151     {
152         //access the getSongLength function from the DJAudioPlayer class
153         g.drawText(tempPlayer->getSongLength(), x: 2,
154         y: 0, width - 4, height,
155         juce::Justification::centredLeft, useEllipsesIfTooBig: true);
156     }
157
158 }
```

The getSongLength method is called from the DJAudioPlayer class and is implemented in the DJAudioPlayer.cpp file on lines 118-134:

```
118 juce::String DJAudioPlayer::getSongLength()
119 {
120     double minutesLength{ transportSource.getLengthInSeconds() / 60 };
121
122     //return value of getLengthInSeconds is in seconds
123     //take the decimal and the integer parts
124     double decimal, integer;
125     decimal = modf(minutesLength, &integer);
126
127     //convert to mins and secs
128     std::string min{ std::to_string((int)integer) };
129     std::string sec{ std::to_string((int)round( x: decimal * 60)) };
130
131     juce::String songLength{ min + " min : " + sec + " sec" };
132
133     return songLength;
134 }
```

This method uses the AudioTransportSource class to get the length of the track in seconds and then proceeds to convert this into a formatted minute and second's string to display on the row of the playlist and on the header.

- R3C: Component allows the user to search for files ✓

This is achieved. This is done in the PlaylistComponent.cpp file on lines 401 – 414:

```
401 void PlaylistComponent::search(juce::String inputtext) {
402     //search for a song
403     if (inputtext == "") {
404         tableComponent.deselectAllRows();
405     }
406     else {
407         //iterate through the trackTitles
408         for (int i = 0; i < trackTitles.size(); i++) {
409             if (trackTitles[i].contains( inputtext)) {
410                 tableComponent.selectRow(i, dontScrollToShowThisRow: false, deselectOthersFirst: false);
411             }
412         }
413     }
414 }
```

The logic here iterates over the TrackTitles vector using the input text to compare the name to each item within the vector, and when a hit is found the row is selected. It uses the function contains() to loosly compare the search string with the track titles.

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

  This is achieved. The user can assign each row item to either the “left deck” or the “right deck” and the subsequent deck is updated with the corresponding track details. This code can be found in the method refreshComponentForCell in the file PlaylistComponenet.cpp on lines 160-195:

```
181     if (columnId == 4) {
182         if (existingComponentToUpdate == nullptr) {
183             juce::TextButton* btn = new juce::TextButton{ "Left Deck" };
184             juce::String rowid{ std::to_string(rowNumber) };
185             juce::String colid{ std::to_string(columnId) };
186             btn->setComponentID(colid + rowid);
187             btn->addListener(this);
188             if (btn->isOver())
189             {
190                 btn->setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 102, blue: 204));
191             }
192             else
193             {
194                 btn->setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 51, blue: 102));
195             }
196             existingComponentToUpdate = btn;
197             //make sure the track data is always up to date
198
199         }
200     }
201     if (columnId == 5) {
202         if (existingComponentToUpdate == nullptr) {
203             juce::TextButton* btn = new juce::TextButton{ "Right Deck" };
204             juce::String rowid{ std::to_string(rowNumber) };
205             juce::String colid{ std::to_string(columnId) };
206             btn->setComponentID(colid + rowid);
207             btn->addListener(this);
208             if (btn->isOver())
209             {
210                 btn->setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 102, blue: 204));
211             }
212             else
213             {
214                 btn->setColour(juce::TextButton::buttonColourId, newColour: juce::Colour(red: 0, green: 51, blue: 102));
215             }
216             existingComponentToUpdate = btn;
217
218         }
219     }
```

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

  This has been achieved. This functionality has been in two ways:

  1. There is a tracks folder that is locally stored under OtoDecks->Builds->VisualStudio2022->Tracks. This folder is read on startup and populates the table automatically. This code is under the getTracks method in the file PlaylistComponenet.cpp on line 468-489:

```cpp
void PlaylistComponent::getTracks()
{
    //load up tracks on start up from the tracks folder
    const juce::String folderPath(juce::File::getCurrentWorkingDirectory().getFullPathName() + "/Tracks");
    const juce::File trackFolder(folderPath);
    //check that folder exists
    if (trackFolder.isDirectory())
    {
        juce::DirectoryIterator iter(juce::File(folderPath), recursive: true);

        while (iter.next())
        {
            const juce::File fileFound(iter.getFile());
            juce::String title = fileFound.getFileName();
            trackTitles.push_back(_Val: fileFound.getFileNameWithoutExtension());
            trackURLs.push_back(juce::URL{ fileFound });
        }

        tableComponent.updateContent();
        tableComponent.resized();
    }
}
```

This code essentially retrieves the folder and iterates through it to find all the files that are inside. It then takes the file and pushes it to the trackURLs vector and then populates the trackTitles vector with the file name only.

2. The user can save their playlist by clicking the “Save Playlist” button. This creates a textfile with each line containing the URL of the track that was in the playlist and the track title separated by a comma. The name of this textfile is the date and time. This is then later read by the “Read Playlist” button which was already referenced in **R3A**. This code is on the buttonClicked method PlaylistComponent.cpp in the method buttonClicked on line 247-284:

```cpp
//save to playlist
if (button == &saveButton) {
    //get time and date details to create a .txt file
    auto now:time_t = std::time(0);
    std::string dt = ctime(&now);
    tm* ltm = localtime(&now);
    std::string yr = std::to_string(_Val:1900 + ltm->tm_year);
    std::string mon = std::to_string(_Val:1 + ltm->tm_mon);
    std::string mday = std::to_string(ltm->tm_mday);
    std::string hs = std::to_string(ltm->tm_hour);
    std::string mins = std::to_string(ltm->tm_min);
    std::string secs = std::to_string(ltm->tm_sec);

    //create the text file and get it ready to edit
    playlistSave.open(_Str:"playlist_" + yr + "_" + mon + "_" + mday + "_" + hs + "_" + mins + "_" + secs + ".txt", _Mode:std::ios::out | std::ios::app);
    //populate the textfile with our URl and track title separated by a comma
    for (int i = 0; i < trackURLs.size(); i++) {
        juceUrl = trackURLs[i].toString(includeGetParameters:false);
        stringUrl = juceUrl.toStdString();
        //change URl to a std::string
        filenameString = trackTitles[i].toStdString();
        //change filename to std::string
        playlistSave << stringUrl << "," << filenameString << std::endl;
    }
}
```

R4 Requirements

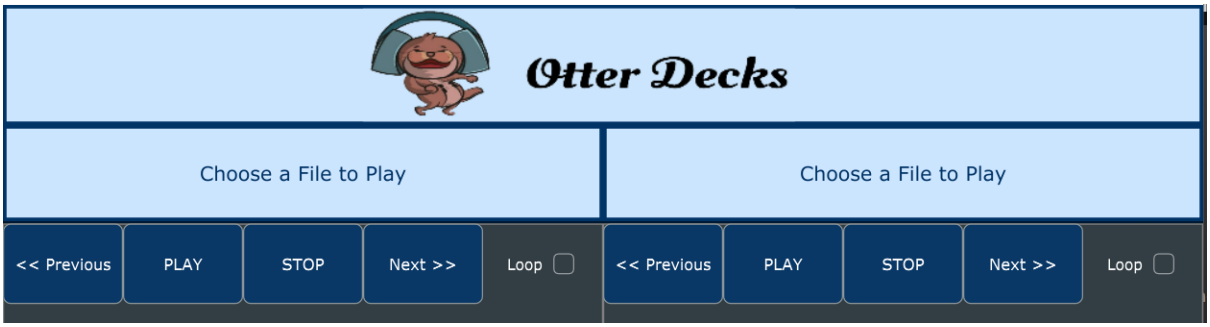
- R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls ✓

  This is achieved. There is a “Next”, “Previous” and loop function. The sliders have been changed to rotary sliders. There is two title classes: one with the logo, one with track titles and the Playlist component is significantly improved. There is also a popup that shows when the application is opened.

The code for the initial popup hasn't been shown yet in this report. This is a popup window that gives the users a brief introduction to OtterDecks to get them started. It is a juce Dialog Window and launches asynchronously with the application. This is contained in the PlaylistComponent.cpp file on lines 417-449:

```
416  //create a pop up on start up
417  void PlaylistComponent::popup() {
418      //message for popup
419      juce::String m;
420      m << "You can drag a file to the playlist table at the bottom to add a track" << juce::newLine;
421      m << juce::newLine;
422      m << "Tracks saved in the tracks folder will always show on start up" << juce::newLine;
423      m << juce::newLine;
424      m << "Let's have a swimming time together and make some Otter-ly good music!" << juce::newLine;
425      m << juce::newLine;
426      m << "Close this to continue" << juce::newLine;
427      juce::DialogWindow::LaunchOptions dialog;
428      auto* label = new juce::Label();
429      label->setText(m, juce::dontSendNotification);
430      label->setColour(juce::Label::textColourId, newColour: juce::Colours::whitesmoke);
431      dialog.content.setOwned(label);
432
433      //space for the text
434      juce::Rectangle<int> space(initialX: 0, initialY: 0, width: 280, height: 250);
435      dialog.content->setSize(space.getWidth(), space.getHeight());
436
437      //popup title
438      dialog.dialogTitle = "Welcome to Otter Decks";
439      dialog.dialogBackgroundColour = juce::Colour(red: 0, green: 76, blue: 153);
440      dialog.escapeKeyTriggersCloseButton = true;
441      dialog.useNativeTitleBar = true;
442      dialog.resizable = true;
443
444      //dialog window size
445      dialogWindow = dialog.launchAsync();
446      if (dialogWindow != nullptr) {
447          dialogWindow->centreWithSize(width: 270, height: 200);
448      }
449  }
```

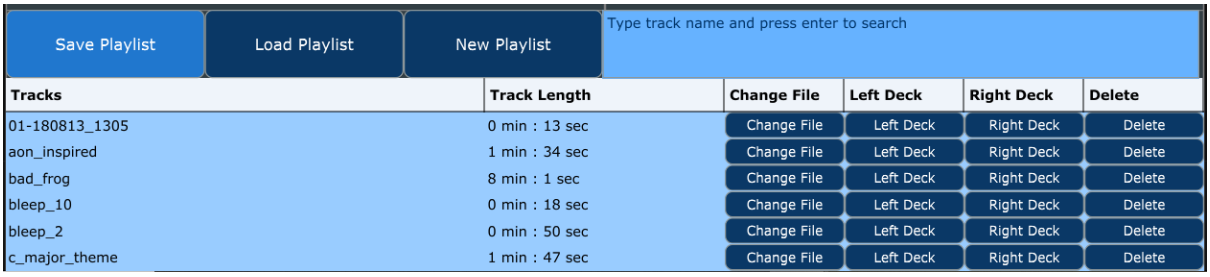
- R4B: GUI layout includes the custom Component from R2 ✓
  This is achieved. There is a "Next", "Previous" and loop function. These have all been described previously.



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

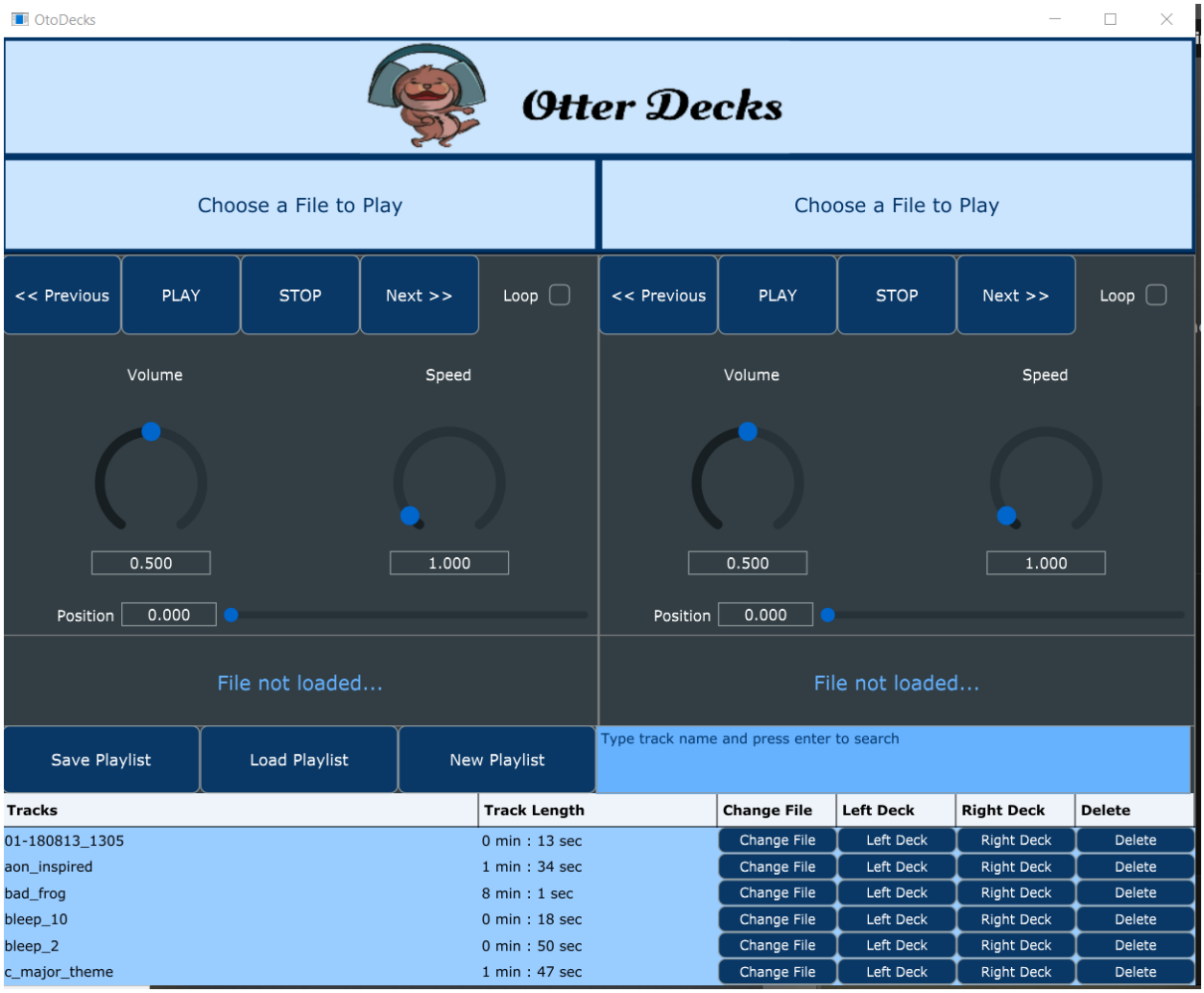
This is achieved. Most of these functionalities have already been explained previously. The delete button removes a row from the table, it also subsequently deletes the corresponding item from the TrackURLs and TrackTitles vectors. This can found on PlaylistComponent.cpp file on lines 365-373:

```
365         //delete song from playlist
366         if (idStr.at(_Off: 0) == '6') {
367             if (trackIndex <= trackURLs.size() - 1 && trackIndex <= trackTitles.size() - 1) {
368                 trackURLs.erase(_Where: trackURLs.begin() + trackIndex);
369                 trackTitles.erase(_Where: trackTitles.begin() + trackIndex);
370                 tableComponent.updateContent();
371                 //rebuff the track data
372                 header1->setTrackData(trackURLs);
373                 header2->setTrackData(trackURLs);
```



The “New Playlist” button removes all entries in the playlist allowing the user to start fresh. This can found on PlaylistComponent.cpp file in the method buttonClicked on lines 365-373:

```
273     if (button == &newButton) {
274         //clear the vectors
275         trackURLs.clear();
276         trackTitles.clear();
277         tableComponent.updateContent();
278         //update the header data
279         header1->setTrackData(trackURLs);
280         header2->setTrackData(trackURLs);
281     }
```



Bibliography

[1] 'OTTER WITH HEADPHONES GIFT FUNNY BIRTHDAY' STICKER | SPREADSHIRT

Spreadshirt. 2022. 'Otter with headphones gift funny birthday' Sticker | Spreadshirt. [online] Available at: <https://www.spreadshirt.ie/shop/design/otter+with+headphones+gift+funny+birthday+sticker-D5d4187b1b264a1173d740552?sellable=R43wBOoEzZSeAr5ywvMb-1459-215> [Accessed 1 September 2022].