The OtoDecks which I had created is made up of 2 separate components which are the DeckGUI and PlaylistComponent. The following (Figure 1) shows the layout of DeckGUI. In MainComponent.h, I had called 2 objects of DeckGUI which are deckGUI1 and deckGUI2. They are then made visible using the addAndMakeVisible method in the MainComponent constructor function and their sizes are set in the resized function of MainComponent.cpp.

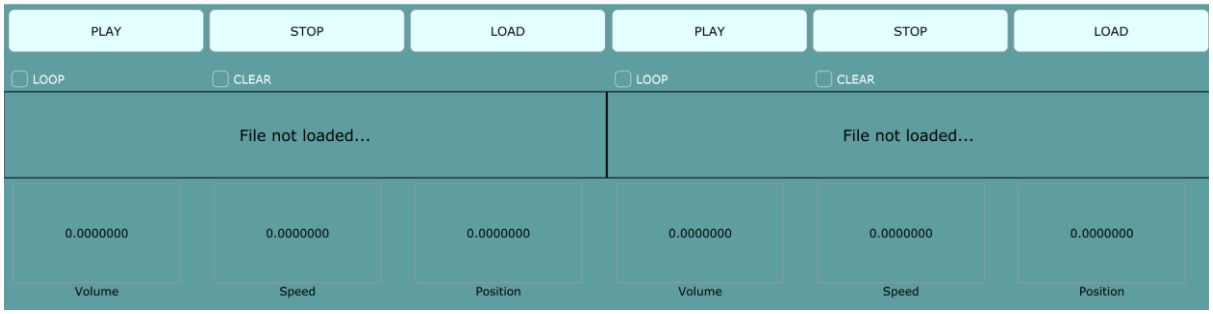


Fig 1. DeckGUI component

After the DeckGUI has been created, buttons are added into the deck, which are the 'PLAY', 'STOP' and 'LOAD' buttons. They are created using the 'TextButton' class in DeckGUI.h and placeholder text are written at the same time, as shown in Fig 2 below.

```
TextButton playButton{ "PLAY" };
TextButton stopButton{ "STOP" };
TextButton loadButton{ "LOAD" };
```

Fig 2. TextButton Class

Just like DeckGUI, they are made visible using the addAndMakeVisible method and an addListener event is added to the button so that if the buttons were to be clicked, it will trigger an event. The design of the button was done in the paint() function of DeckGUI while the position was set-up in the resized()function. The volume, speed and position of the waveform of the current track is done using the 'Slider' class while the label for each slider was done using the 'Label' class, shown in Fig 3 below.

```
Slider volSlider;
Slider speedSlider;
Slider posSlider;

Label volume;
Label speed;
Label position;
```

Fig 3. Slider and Label Classes

The image (Fig 4) below shows how the waveforms should appear within the bounds of 'waveFormDisplay' when the 'LOAD' button for respective decks have been clicked and file was chosen or when file is dropped into the bounds.

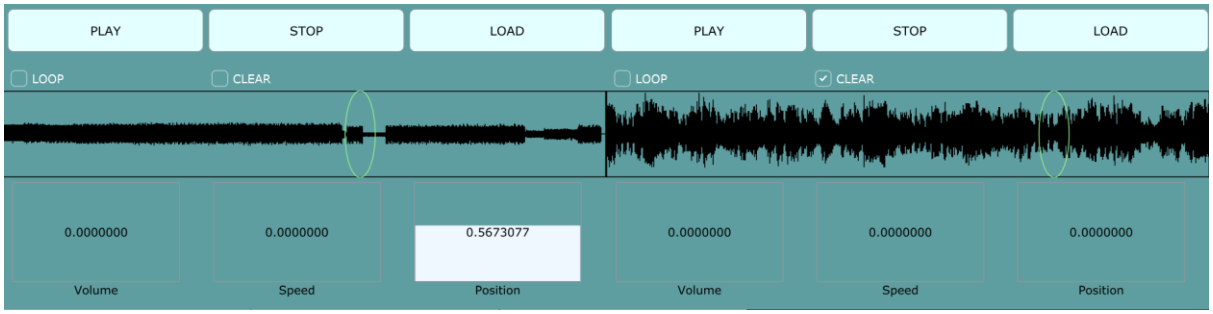


Fig 4. Waveform appearing when track has been loaded

As mentioned, there are 2 ways to upload the track onto the decks. One of which is to use the 'LOAD' button and manually select the file and the other way is to drag and drop the file from the file explorer onto the deck.

```
void DeckGUI::buttonClicked(Button* button)
{
    if(button == &playButton)
    {
        DBG("Play button was clicked");
        player->start();

    }
    if (button == &stopButton)
    {
        DBG("Stop button was clicked");
        player->stop();
    }
    if(button == &loadButton)
    {
        FileChooser chooser{ "Select a file..." };
        if (chooser.browseForFileToOpen())
        {
            player->loadURL(juce::URL{chooser.getResult()});
            waveformDisplay.loadURL(juce::URL{ chooser.getResult()});
        }
    }
}
```

Fig 5. Code snippet for play, stop and load buttons

The code snippet above (Fig 5) shows how the load button in DeckGUI works. When the button has been clicked, the 'FileChooser chooser {"Select a file…"}' runs hence displaying the file explorer. After which, a file can be selected due to the 'browseForFileToOpen ()' method. This then calls onto the 'player->loadURL(…)' where 'player' is a variable that stores a memory address of an object of the 'DJAudioPlayer' class and 'loadURL()' is a member function of 'DJAudioPlayer' class. Hence, 'player' is pointing to the 'loadURL ()' function on the 'DJAudioPlayer' object. The player variable is created as a private member of the 'DeckGUI' class in the header file.

However, to play the audio file, loadURL () is used to load the audio from a URL. It creates an 'AudioFormatReader' object from the URL and uses it to create an 'AudioFormatReaderSource' object that is set as the source for the 'transportSource' and the choice is converted to URL as an argument. An instance of 'WaveformDisplay' stated as 'waveformDisplay' is called as a private member of DeckGUI class. In Fig 5, after the track has been selected, it is converted into an URL object. The loadURL () function is then called on the 'waveformDisplay' object and passing the URL object (audio file) as an argument. This generates the waveform display of the audio data.

```
void DeckGUI::filesDropped(const StringArray &files, int x, int y)
{
    for(String filename:files)
    {
        if (files.size() == 1)
        {
            DBG("DeckGUI::filesDropped");
            URL fileURL = URL{ File{filename} };
            player->loadURL(fileURL);
            waveformDisplay.loadURL(URL{ File{files[0]} });
        }
    }
}
```

Fig 6. Code snippet for file drag and drop

The code snippet above (Fig 6) shows the code functionality when the track is dragged straight from the file explorer and dropped within the bounds of 'WaveformDisplay'. The 'files' parameter is a 'StringArray' object which contains the path of the file that was dropped. It then iterates over each file in the 'files' and if there is only 1 file, the file is loaded using 'player ->loadURL()' and the waveform is displayed using 'waveformDisplay.loadURL()'.

Above the waveformDisplay bounds, there are 2 toggle buttons which are the 'LOOP' and 'CLEAR' buttons. The toggle 'LOOP' button allows us to choose whether we wish for the track to continue playing. When the 'LOOP' button has been selected, it allows the track to go back to the beginning and continue playing until the 'STOP' button has been clicked or until the 'LOOP' button has been deselected. When the 'CLEAR' button has been selected, it removes the waveform. Hence, the application does not have to be restarted if we wish play a new track instead. As seen from below in Fig 8, I had created a 'clear ()' function in WaveformDisplay which removes the audioThumb, set 'fileLoaded' to be false, and to repaint the waveform (to make it disappear). Inside DeckGUI::buttonClicked (Button* button), if the toggle button selected was the 'CLEAR' button, it will call the 'clear()' function hence removing the waveform.

```
void DeckGUI::timerCallback()
{
    if (std::to_string(loopButton.getToggleState()) == "0")
    {
        if (player->getPositionRelative() >= 1)
        {
            player->setPositionRelative(0);
            player->stop();
        }
    }

    if (std::to_string(loopButton.getToggleState()) == "1")
    {
        if (player->getPositionRelative() >= 1)
        {
            player->setPositionRelative(0);
            player->start();
        }
    }
    waveformDisplay.setPositionRelative(player->getPositionRelative());
}
```

Fig 7. Code snippet responsible for 'LOOP' button

```
if(button == &clearButton)
{
    if(clearButton.getToggleState())
    {
        waveformDisplay.clear();
    }
}
```

```
void WaveformDisplay::clear()
{
    audioThumb.clear();
    fileLoaded = false;
    repaint();
}
```

Fig 8. Code snippets responsible for 'CLEAR' button

Below the waveformDisplay bound, there are 3 sliders which are ‘volume’, ‘speed’ and ‘position’ slider. On the waveform, there is a green ellipse which represents the audioThumb and its position can be shifted using the position slider(‘posSlider’) hence allowing us to play the section of the track that we want, as seen in Fig 9. At the same time, there speed slider (‘speedSlider’) which is responsible for how fast the track plays and the volume slider (‘volSlider) is responsible for how loud the track plays. The sliders can be adjusted and tracks can be played for both decks at the same time.

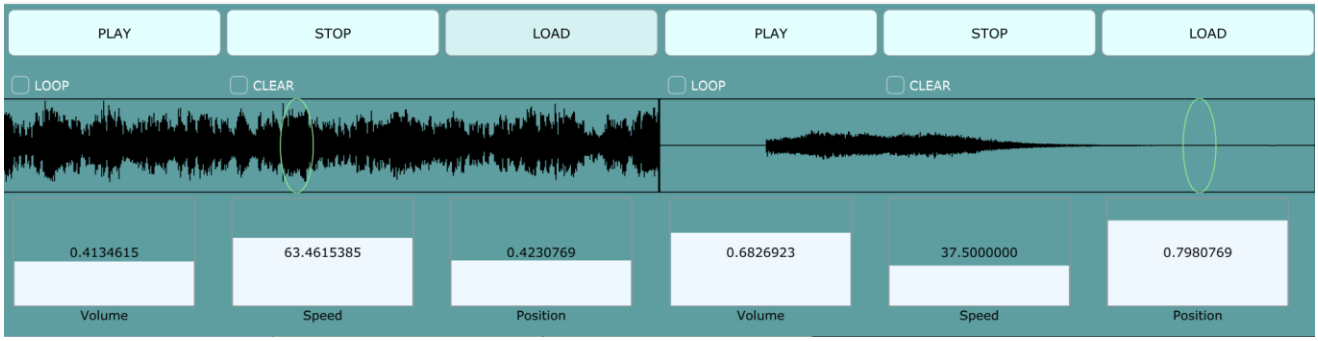


Fig 9. Waveforms with effects of the sliders

The ‘sliderValueChanged ()’ function in DeckGUI is responsible for the effects when the sliders are being adjusted, as shown in Fig 9 below. For all 3 sliders, we make use of the ‘player’ variable of the ‘DJAudioPlayer’ class. It is used to point to different functions such as ‘setGain ()’, ‘setSpeed ()’ and ‘setPositionRelative ()’.

```
void DeckGUI::sliderValueChanged(Slider* slider)
{
    if(slider == &volSlider)
    {

        player->setGain(slider->getValue());
    }
    if(slider == &speedSlider)
    {

        player->setSpeed(slider->getValue());
    }
    if(slider == &posSlider)
    {
        player->setPositionRelative(slider->getValue());
    }
}
```

Fig 10. Code snippet of sliderValueChanged

Next, there is the PlaylistComponent which is used to display the tracks on the music library (Fig 11). It is used to display tables of tracks and to do this, I have used the TableListBox but it makes use of the TableListBoxModel class to allow access to data for display. Afterwhich, another class named PlaylistComponent is used to wrap TableListBox to provide the graphical user interface hence PlaylistComponent class will have to be added to MainComponent header for the interface to appear as shown in Fig 12.

Fig 11. PlaylistComponent

```
PlaylistComponent playlistComponent{&deckGUI1, &deckGUI2};
```

Fig 12. Adding PlaylistComponent to MainComponent

To upload the tracks onto the library, I have implemented a 'Load into library' button which was created using a TextButton class, in the header file, followed by 'addAndMakeVisible' and 'addListener' method in the constructor function. To adjust the location and size of the button, I have placed it within the resized() function. The following code (Fig 13) is responsible for what happens when the 'libraryLoad' button has been clicked.

```
void PlaylistComponent::buttonClicked(Button* button)
{
    if(button == &libraryLoad)
    {
        FileChooser choosers{ "Select a file to add to the library" };
        if(choosers.browseForMultipleFilesToOpen())
        {
            myFiles = choosers.getResults(); //store the results in an array of files
            for(int i=0; i<myFiles.size(); ++i) //looping over the array of files
            {
                if(trackTitles.size() != 0 || i==0) //if there are no tracks in the vector or index is equal to 0
                {
                    for(int j=0; j<trackTitles.size(); ++j) //loop through the vector of tracks
                    {
                        if(myFiles[i].getFileName().toStdString() == trackTitles[j]) //if the choice is equal to what is already in the vector
                        {
                            DBG("duplicate");
                            d_index = i; //take note of the index of the duplicated file
                            break;
                        }
                    }
                    if(i != d_index) //if it is not equal
                    {
                        fileName = URL{ myFiles[i] }.getFileName(); //creates a URL of the choices, get the name
                        fileNames.add(fileName); //store the names in fileNames
                        AudioFormatReader* reader = formatManager.createReaderFor(myFiles[i]);
                        duration = reader->lengthInSamples / reader->sampleRate; //read the length of the file
                        trackDurations.add(duration); //store the length of the files in trackDurations
                        trackTitles.push_back(myFiles[i].getFileName().toStdString());
                    }
                }
            }
            DBG("table update");
            tableComponent.updateContent();
        }
```

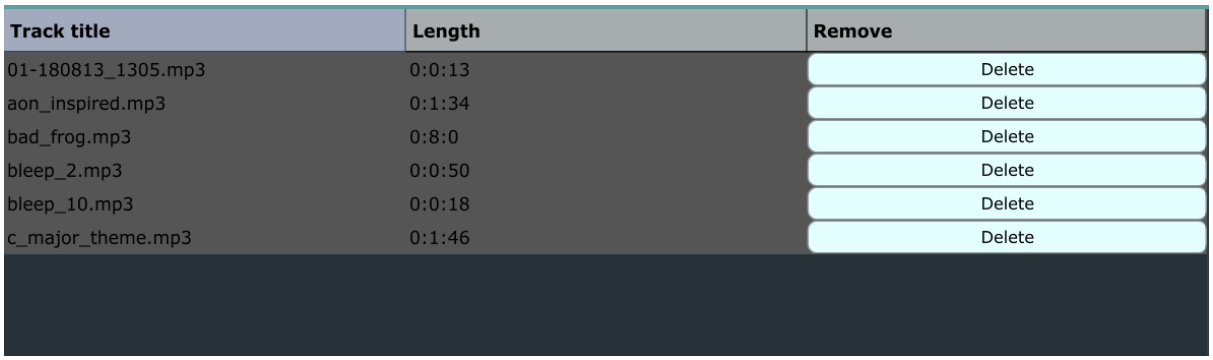
Fig 13. Functionality of libraryLoad button

In the above code, it states that if the button has been clicked, the file explorer opens and allows the user to choose more than 1 file. The selected files are stored in 'myFiles', which is an array of files. It is then being compared with 'trackTitles' which is a vector of std::string. If the track which I had selected is already in the trackTitles vector, the code breaks. If not, a URL of the selected file is taken and added to an array of strings called 'fileNames'. The selected file is then converted to 'std::string' and pushed into trackTitles vector and the table is updated using 'tableComponent.updateContent()'. As I had allowed users to choose for multiple tracks, multiple tracks can be added at once, as shown below(Fig 14).

As seen from Fig 11, there are 3 columns in the music library: ‘Track title’, ‘Length’ and ‘Remove’. The ‘Length’ column is used to show the duration of the track whilst the ‘Remove’ column is meant to create the ‘delete’ buttons to remove specific tracks.

Inside the ‘buttonClicked ()’ function, I had created an AudioFormatReader to read the selected audioFile. The duration of each track is then calculated by dividing the total number of samples by the number of samples per second and it is stored in a ‘duration’ variable with a double as its data type. ‘duration’ is then stored in an array of double called ‘trackDurations’. The conversion of duration to “hour, minute, seconds” format was done in the paintCell () function of PlaylistComponent class, as shown in Fig 15.

Next, we have the ‘Remove’ header which is responsible to delete the track using the delete button. The delete function was created in the refreshComponentForCell () of PlaylistComponent class hence, reducing the lines of code that is needed to create multiple buttons. The functionality of the button was also created in the same function as stated in Figure 16, “btn3->onClick = [this] {deleteTrack ();}”.



Track title	Length	Remove
01-180813_1305.mp3	0:0:13	Delete
aon_inspired.mp3	0:1:34	Delete
bad_frog.mp3	0:8:0	Delete
bleep_2.mp3	0:0:50	Delete
bleep_10.mp3	0:0:18	Delete
c_major_theme.mp3	0:1:46	Delete

Fig 14. Loaded tracks into music library

```
void PlaylistComponent::paintCell(Graphics & g,
                                  int rowNumber,
                                  int columnId,
                                  int width,
                                  int height,
                                  bool rowIsSelected)
{
    if(columnId == 0)
    {
        g.drawText(fileNames[rowNumber],
            2, 0,
            width - 4, height,
            Justification::centredLeft,
            true);
    }

    if(columnId == 1)
    {
        int final_Duration = trackDurations[rowNumber];
        int hours = final_Duration / 3600;
        int min = (final_Duration % 3600) / 60;
        int sec = (final_Duration % 60);
        String durationStr = String(hours) + ":" + String(min) + ":" + String(sec);
        g.drawText(durationStr, 2, 0,
            width - 4, height,
            Justification::centredLeft,
            true);
    }
}
```

Fig 15. Converting duration to proper format

```
Component* PlaylistComponent::refreshComponentForCell(int rowNumber,
                                                       int columnId,
                                                       bool isRowSelected,
                                                       Component *existingComponentToUpdate)
{
    if(columnId == 2)
    {
        if(existingComponentToUpdate == nullptr)
        {
            TextButton* btn3 = new TextButton("Delete");
            String id{std::to_string(rowNumber)};
            btn3->setComponentID(id);
            btn3->addListener(this);
            existingComponentToUpdate = btn3;
            btn3->onClick = [this] {deleteTrack(); };
        }
    }
    return existingComponentToUpdate;
}
```

Fig 16. Creating delete button

As seen from Fig 11, beside the libraryLoad button, there are 2 other buttons namely deck 1 and 2. They are also created in the same way as the libraryLoad button. These 2 buttons are responsible for loading the files from the library to the respective decks, hence, it is necessary to call DeckGUI in PlaylistComponent. Therefore, 2 separate pointer variables are created to hold the memory addresses of DeckGUI objects.

```
DeckGUI* deckGUI1;
DeckGUI* deckGUI2;
```

Fig 17. Creating pointer variables to DeckGUI

In filesDropped and buttonClicked functions of DeckGUI, we had called 'player->loadURL ()' and 'waveformDisplay.loadURL ()' hence, I created another function to store these 2 lines of codes so that they can be easily loaded to play the audio and display the waveform.

```
void DeckGUI::loadedDeck(URL audioURL)
{
    DBG("Loaded deck called");
    player->loadURL(audioURL);
    waveformDisplay.loadURL(audioURL);
}
```

Fig 18. New function in DeckGUI

```
void PlaylistComponent::loadInDeck(DeckGUI* deckGUI)
{
   for(int i=0; i<myFiles.size(); ++i)
   {
       int loadRow=  tableComponent.getSelectedRow() ;
       if (loadRow != -1)
       {
           DBG("deck1load");
           deckGUI->loadedDeck(URL{ File{myFiles[i] }});
           DBG("DECKGUI LOADINDECK");
       }
   }
}
```

Fig 19. 'loadInDeck' function in PlaylistComponent

In PlaylistComponent, a function is created to load the files in DeckGUI as seen in Fig 19. It loops the files which were chosen and if one of the rows is selected, it is passed into 'loadedDeck ()' function which converts the file to URL audiofile. This function is also called when either of the buttons (deck1/deck2) are clicked.

From Fig 11, below the 3 buttons there is a rectangular box which is the ‘searchPlaylist’. This was created using the TextEditor class. The method to make it visible is the same as how the buttons were created. From Fig 19, we can see that we created a variable to store the text written inside ‘searchPlaylist’. The filesNames, which is an array of strings storing the files that was chosen, is looped through and checked with the word(input) that was return. If there is a match, the row will be colored in beige.

```
void  PlaylistComponent::textEditorTextChanged(juce::TextEditor& editor)
{

    input = searchPlaylist.getText();

    for(int i=0; i<fileNames.size(); ++i)
    {
        if(fileNames[i].containsWholeWordIgnoreCase(input) && input != " ")
        {
            tableComponent.selectRow(i, false, true);
        }
    }
}
```

Fig 20. searchPlaylist functionality