

# Object-Oriented Programming

## End-Term Coursework

### Report

#### Table of Contents

<b>R1: All the basic functionality shown in class</b>	<b>3</b>
R1A: Loading audio files into audio players	3
R1B: can play two or more tracks	4
R1C: can mix the tracks by varying each of their volumes	5
R1D: can speed up and slow down tracks	6
<b>R2: Custom deck control Component with custom graphics</b>	<b>7</b>
R2A: Component has custom graphics implemented in a paint function	7
R2B: Component enables the user to control the playback of a deck somehow	11
<b>R3: Implementation of a music library component</b>	<b>20</b>
R3A: Component allows the user to add files to their library	20
R3B: Component parses and displays meta data such as filename and song length	24
R3C: Component allows the user to search for files	26
R3D: Component allows the user to load files from the library into a deck	27
R3E: The music library persists so that it is restored when the user exits then restarts the application	29
<b>R4: Implementation of a complete custom GUI</b>	<b>31</b>
<b>Reference for images</b>	<b>33</b>

## Otodecks DJ Application

The interface is divided into two main sections: the top section for deck control and the bottom section for track management.

**Deck 1 (Left):** Displays a waveform at the top. Below it, the track name "stomper\_reggae\_bit" is shown with a progress bar at 00:16 | 01:28. Controls include a speed slider (2.0x), a volume slider (mute), and buttons for <<, PAUSE, and >>. A central crossfader is located between the two decks.

**Deck 2 (Right):** Displays "Waveform not loaded" and "No Track Loaded". Controls include a speed slider (2.0x), a volume slider (mute), and buttons for <<, PLAY, and >>.

**Track Management (Bottom):** Features a search bar "Enter Search Keyword", a status "10 TRACKS ADDED", and a "Remove All" button. A table lists tracks with columns for Track Title, Length, and PLAY IN (DECK 1, DECK 2, and a status X).

Track Title	Length	PLAY IN	PLAY IN	
bleep_10	00:18	DECK 1	DECK 2	X
electro_smash	02:14	DECK 1	DECK 2	X
fast_melody_regular_drums	01:24	DECK 1	DECK 2	X
fast_melody_thing	01:20	DECK 1	DECK 2	X
hard	00:12	DECK 1	DECK 2	X
ms20_improvisation	07:42	DECK 1	DECK 2	X
selection1	01:19	DECK 1	DECK 2	X
soft	00:13	DECK 1	DECK 2	X
stomper1	02:44	DECK 1	DECK 2	X
twindrive	06:50	DECK 1	DECK 2	X

**Queued Tracks:** A list of tracks with a clear button (X) next to each:

- fast\_melody\_regular\_drums
- electro\_smash
- bleep\_10
- ms20\_improvisation

Buttons at the bottom right: "Clear Queue" and "Play Queue".

## R1: All the basic functionality shown in class

### R1A: Loading audio files into audio players

#### 1. Clicking the “LOAD” button.

Clicking the “LOAD” button on Deck 1 or 2 will open a dialog box for the user to choose a file from their local machine. The URL of the selected file is then passed into **WaveformDisplay::loadURL()** and **DJAudioPlayer::loadURL()** to load the player and the waveform display respectively.

**WaveformDisplay::loadURL()** - **WaveformDisplay.cpp**, line 62

**DJAudioPlayer::loadURL()** - **DJAudioPlayer.cpp**, line 40

```
// if the load button is clicked
if (button == &loadButton) {
    DBG("load button pressed");
    FileChooser chooser{ "Select a file to play..." };

    if (chooser.browseForFileToOpen()) {
        player->loadURL(URL{ chooser.getResult() });
        waveformdisplay.loadURL(URL{ chooser.getResult() });
    }
}
```

(DeckGUI.cpp - line 242)



(Load Button in DeckGUI)

#### 2. Drag and drop file into Deck 1 or Deck 2

A file can also be loaded by dragging and dropping into either Deck 1 or Deck 2. In **DeckGUI::filesDropped()**, when a file is dropped into either Deck 1 or 2, the URL of the file is passed into **DJAudioPlayer ::loadURL()** and **WaveformDisplay::loadURL()** to load the player and waveform display respectively.

```
/* Determines whether the component is interested in the set of files being dragged in */
bool DeckGUI::isInterestedInFileDrag(const StringArray& /*files*/) {
    return true;
}

/* Processing of the files dropped onto this component */
void DeckGUI::filesDropped(const StringArray& files, int /*x*/, int /*y*/) {
    // only drop 1 file at a time
    if (files.size() == 1) {
        player->loadURL(URL{ File{files[0]} });
        waveformdisplay.loadURL(URL{ File{files[0]} });

        setNameAndLength(files[0]);
    }
}
```

(DeckGUI.cpp - line 341)

## R1B: can play two or more tracks

After loading a file into Deck 1 and/or 2, the player can be started by clicking on the “PLAY” button. If the play button on both Deck 1 and 2 are clicked, both files will play at the same time.

```
/* Determine what action to take when a button is clicked */  
void DeckGUI::buttonClicked(Button* button) {  
    // if the play/pause button is clicked  
    if (button == &playpauseButton) {  
        // if the play button is clicked, play the track and set  
        if (!player->isPlaying) {  
            DBG("play button pressed");  
            player->start();  
        }  
    }  
}
```

*(DeckGUI.cpp - line 204: start player if play button is clicked)*

By adding player 1 and 2 as an input source to the MixerAudioSource as shown below, it allows both players to play a file at the same time.

```
/* Prepare the audio source for playing. */  
void MainComponent::prepareToPlay(int samplesPerBlockExpected, double sampleRate) {  
    // This function will be called when the audio device is started, or when  
    // its settings (i.e. sample rate, block size, etc) are changed.  
  
    // add player1 and player2 as an input source to mixer  
    mixerSource.addInputSource(&player1, false);  
    mixerSource.addInputSource(&player2, false);  
  
    // call prepareToPlay() on each input source  
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);  
    player2.prepareToPlay(samplesPerBlockExpected, sampleRate);  
}
```

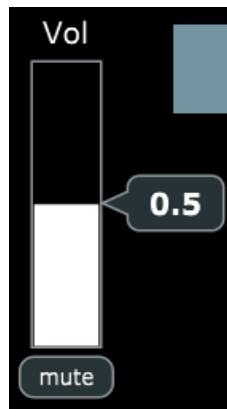
*(MainComponent.cpp - line 38)*



*(Play Button)*

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

The volume of both decks can be varied by moving the volume slider up and down. Hovering a cursor over the volume slider will display the current volume slider value.



*(volume slider showing current value on mouse over)*

The volume slider has a range of 0 to 1, with 0.5 being the default value, and each increment being 0.1.

```
// Volume Slider
volSlider.setRange(0.0, 1.0, 0.1);
volSlider.setValue(0.5); // initial value of 0.5
volSlider.setSliderStyle(Slider::SliderStyle::LinearBarVertical); // set the slider style to LinearBarVertical
volSlider.setTextBoxStyle(Slider::NoTextBox, false, 0, 0); // remove text box inside the slider
volSlider.setPopupDisplayEnabled(true, true, this); // a pop-up showing the current value
```

*(DeckGUI.cpp - line 59)*

Whenever the slider's value changes, the **DeckGUI::sliderValueChanged()** function is called, and the gain is set to the current value of the volume slider.

```
/* Determine what action to take when the value of a slider is changed */
void DeckGUI::sliderValueChanged(Slider* slider) {
    // if slider moved and mute button is 'off'
    if ((slider == &volSlider) && (muteButton.getToggleState() == false)) {
        player->setGain(slider->getValue());
    }
}
```

*(DeckGUI.cpp - line 324)*

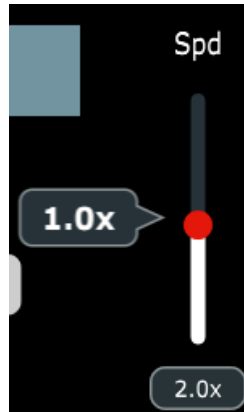
**DJAudioPlayer::setGain()** function sets the gain value to the value passed in.

```
/* Set the volume control */
void DJAudioPlayer::setGain(double gain) {
    if (gain < 0 || gain > 1.0) {
        DBG("DJAudioPlayer::setGain - gain should be between 0 and 1");
    }
    else {
        transportSource.setGain(gain);
    }
}
```

*(DJAudioPlayer.cpp - line 61)*

## R1D: can speed up and slow down tracks

The speed of the track can be controlled by moving the speed slider up and down. Hovering a cursor over the speed slider will display the current speed slider value.



*(Speed Slider showing current value on mouse over)*

The speed slider has a range of 0.1 to 2, with 1 being the default value, and each increment being 0.1. The range starts from 0.1 instead of 0 as the program crashes when the speed slider's value is 0.

```
// Speed Slider
speedSlider.setRange(0.1, 2.0, 0.1); // program crashes when speed slider hits 0. So the range starts from 0.1
speedSlider.setValue(1.0); // initial value of 1.0
speedSlider.setSliderStyle(Slider::SliderStyle::LinearVertical); // set the slider style to LinearVertical
speedSlider.setTextBoxStyle(Slider::NoTextBox, false, 0, 0); // remove text box
speedSlider.setPopupDisplayEnabled(true, true, this); // a pop-up showing the current value
speedSlider.setTextValueSuffix("x"); // add a suffix to the pop-up value
```

*(DeckGUI.cpp - line 70)*

Whenever the slider's value changes, the **DeckGUI::sliderValueChanged()** function is called, and the speed is set to the current value of the speed slider using the **DJAudioPlayer::setSpeed()** function.

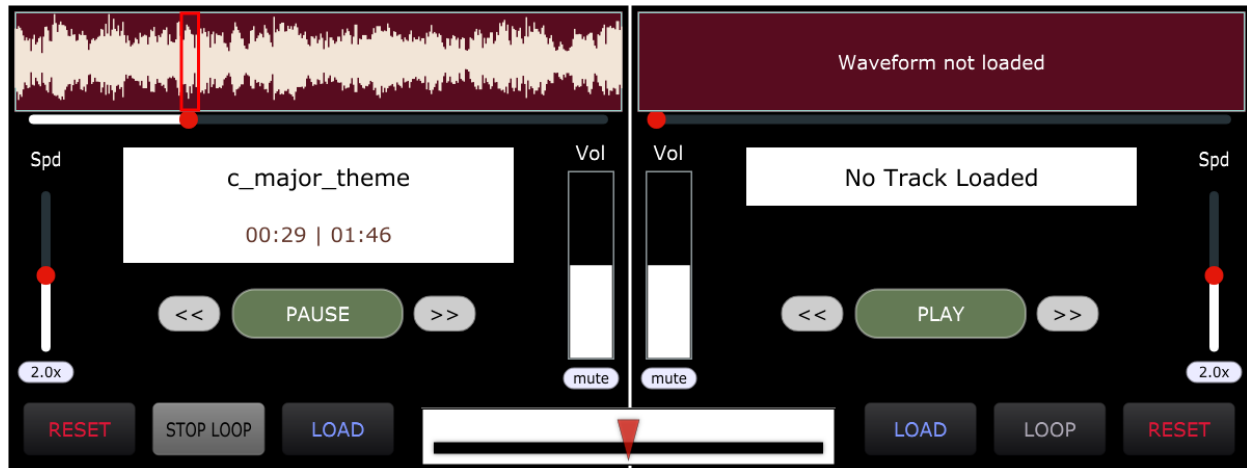
```
// if slider moved and 2x button is 'off'
if ((slider == &speedSlider) && (twoTimesButton.getToggleState() == false)) {
    player->setSpeed(slider->getValue());
}
```

*(DeckGUI.cpp - line 331)*

```
/* Set the speed control */
void DJAudioPlayer::setSpeed(double ratio) {
    if (ratio < 0 || ratio > 2.0) {
        DBG("DJAudioPlayer::setSpeed - ratio should be between 0 and 2");
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

*(DJAudioPlayer.cpp - line 71: Setting the speed of the audio playback)*

## R2: Custom deck control Component with custom graphics



(Left: Deck 1 playing the file “c\_major\_theme”, Right: Deck 2 without any file loaded yet)

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

- Colour changes and different versions of JUCE LookAndFeel used

Overall, I went with the theme of red and black, with a different text colour for every button so that they can be told apart easily. This is achieved in the **DeckGUI::paint()** function.

The colour and look of all buttons and sliders are changed using the **juce::Component::setColour()** function to set the colour of each component. The colours are in Hex format. Different versions of JUCE LookAndFeel are also used to get a different button style.

```
/* Drawing of the component */
void DeckGUI::paint(Graphics& g) {
    g.fillAll(Colour{ 0xFF000000 }); // background colour
    g.setColour(Colour{ 0xFFFFFFFF });
    g.drawRect(getLocalBounds(), 1); // draw an outline around the component
    g.setFont(14.0f);

    // set colour of the buttons
    playpauseButton.setColour(ComboBox::outlineColourId, Colour(0xFFFFFFFF)); // button outline colour
    playpauseButton.setColour(TextButton::buttonColourId, Colour(0xFF317A00)); // button background colour
    playpauseButton.setColour(TextButton::textColourOffId, Colour(0xFFFFFFFF)); // button text colour
    playpauseButton.setLookAndFeel(&LookAndFeel_V1);
}
```

(DeckGUI.cpp - line 99: setting colour of background and button)

```
// for using different versions of LookAndFeel
LookAndFeel_V1 LookAndFeel_V1;
LookAndFeel_V3 LookAndFeel_V3;
```

(DeckGUI.h - line 192: declaring different LookAndFeel versions)

```
// set colour of slider track and thumb
getLookAndFeel().setColour(Slider::thumbColourId, Colour(0xffE3170A));
getLookAndFeel().setColour(Slider::trackColourId, Colour(0xffffffff));
```

(DeckGUI.cpp - line 146: changing colour of the thumb and track of sliders)

- Name and length of file is displayed once a track is loaded into the deck.

DeckGUI displays “No Track Loaded” if no file has been loaded into the deck. Once a file is loaded either through the “LOAD” button or drag-and-drop, the name and length of the loaded file is displayed in DeckGUI (refer to screenshot above).

The **DeckGUI::setNameAndLength()** function is called whenever a file is loaded into Deck 1 or 2. This function initialises the TrackName String with the name of the file, and TrackLength String with the length of the file. The **DeckGUI::lengthInString()** function is used to format the length of the file from seconds into minutes:seconds format.

```
// Stores track information to be displayed
String TrackName;
String TrackLength;
Label name;
Label length;
```

(DeckGUI.h - line 165)

```
/* Gets the name and length of the file passed in to display
void DeckGUI::setNameAndLength(File file) {
    TrackName = (File{ file }.getFileNameWithoutExtension());
    TrackLength = lengthInString(player->getLengthInSeconds());
    isLoaded = true;
```

(DeckGUI.cpp - line 410)

```
String DeckGUI::lengthInString(double time) {
    int totalTime = (int)time;

    String mins{ totalTime / 60 };
    String secs{ totalTime % 60 };

    // if mins is 1 digit, add a leading zero
    if (mins.length() == 1)
        mins = "0" + mins;

    // if secs is 1 digit, add a leading zero
    if (secs.length() == 1)
        secs = "0" + secs;

    return mins + ':' + secs;;
```

(DeckGUI.cpp - line 419)

The track name and length is displayed using **Label::setText()** in **DeckGUI::paint()** function to set the text of the name and length label to the value stored in trackName and TrackLength respectively.

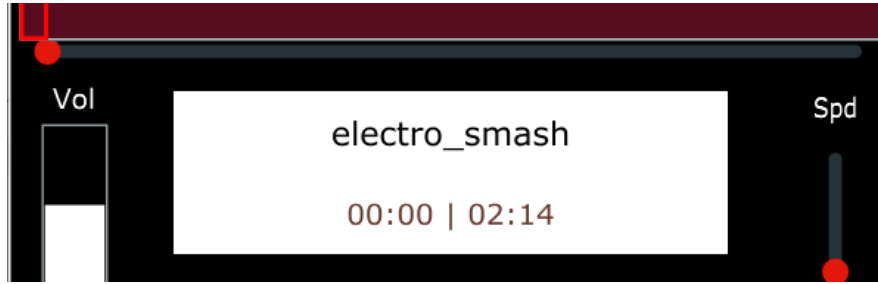
```
// display track name and length if a track is loaded into Deck
if (isLoaded) {
    name.setText(TrackName, dontSendNotification);
    length.setText(movingTrackLength + " | " + TrackLength, dontSendNotification);
    length.setColour(Label::backgroundColourId, Colour(0xff7293A0));
}

else {
    name.setText("No Track Loaded", dontSendNotification);
}
```

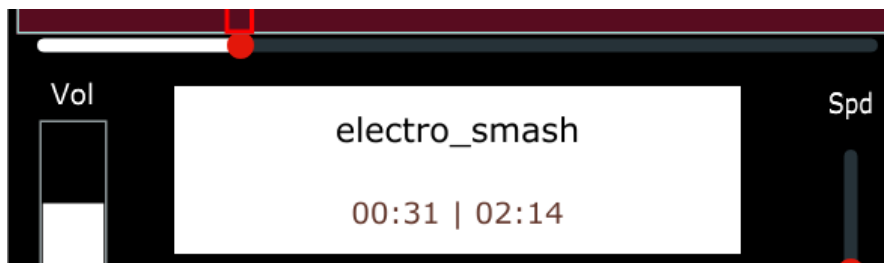
(DeckGUI.cpp - line 150)



- The current play position of the file is updated and displayed as it plays



*(Moving Track length | Track length - when file is first loaded into Deck)*



*(while the file is playing, with the current play position at 00:31 seconds)*

In **DeckGUI::timerCallback()**, the `movingTrackLength` is set to the current position of the player, but formatted into minutes:seconds format (using **DeckGUI::lengthInString()** function discussed previously). This is done every 500 milliseconds to consistently update the value to the current play position of the file.

```
// stores the current play position of the track
String movingTrackLength;
Label movingLength;
```

*(DeckGUI.h - line 171)*

```
// set interval to every 500 millisecond
startTimer(500);
```

*(DeckGUI.cpp - line 91)*

```
356  /* Callback routine that gets called according to the interval set */
357  void DeckGUI::timerCallback() {
358      // update waveformdisplay, posSlider, and movingTrackLength
359      waveformdisplay.setPositionRelative(player->getPositionRelative());
360      posSlider.setValue(player->getPositionRelative());
361      movingTrackLength = lengthInString(player->getPosition());
```

*(DeckGUI.cpp - line 361: setting movingTrackLength to the current position every 500ms)*

- “PLAY” and “STOP” button combined into a single “PLAYPAUSE” button.



*(playpause button showing initial text “play”)*



*(button text changed to “pause” after clicking “play”)*

When the “PLAY” button is clicked, the player will be started and the button text will be changed to “PAUSE” (see screenshot above). Clicking on the now “PAUSE” button will stop the player, and the button text will be changed to “PLAY” again.

```
// if the playpause button is clicked
if (button == &playpauseButton) {
    // if the play button is clicked, play the track and set the button text to "PAUSE"
    if (!player->isPlaying) {
        DBG("play button pressed");
        player->start();
        playpauseButton.setButtonText("PAUSE");
    }

    // if the "PAUSE" button is clicked, stop the track and set the button text to "PLAY"
    else {
        DBG("pause button pressed");
        player->stop();
        playpauseButton.setButtonText("PLAY");
    }
}
```

*(DeckGUI.cpp - line 206)*

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

I have implemented five additional components to control the playback of a deck. They are reset, loop, skip front and back, mute, two times speed, crossfader and queue.

- **RESET**

When the reset button is clicked, the position of the player is set back to the beginning (position 0) of the track.

```
// if the reset button is clicked
if (button == &resetButton) {
    DBG("reset button pressed");

    // if track was playing when reset button was clicked,
    // immediately play the track after resetting the track
    if (player->isPlaying) {
        player->setPosition(0);
        player->start();
    }

    // if track was not playing when reset button was clicked,
    // keep the track paused after resetting
    else {
        player->setPosition(0);
        player->stop();
    }
}
```

*(DeckGUI.cpp - line 223)*



*(reset button)*

- LOOP

In the **DeckGUI::buttonClicked()** function, when the loop button is clicked, the toggle state of the button will be checked. The **juce::Button::getToggleState()** function returns true if the button is toggled 'On' and returns false if the button is toggled 'Off'. If the button is "On", the `isLooping` boolean variable is set to true, and false otherwise.

```
// determines whether the loop button is 'on' or 'off'
bool isLooping;
```

*(DeckGUI.h - line 184)*

```
// if the loop button is clicked
if (button == &loopButton) {
    DBG("loopButton pressed");

    // if loop button is 'On'
    if (loopButton.getToggleState()) {
        isLooping = true;
        loopButton.setButtonText("STOP LOOP");
    }

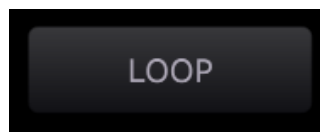
    else {
        isLooping = false;
        loopButton.setButtonText("LOOP");
    }
}
```

*(DeckGUI.cpp - line 286)*

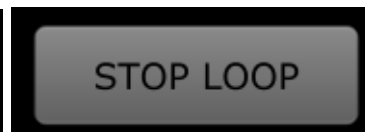
The `isLooping` boolean variable is used in **DeckGUI::timerCallback()**, where every 500 milliseconds, it is checked if the looping button is toggled "on". When the track reaches the end and the loop button is "on", the position of the player is set to the beginning of the track and the track is played again.

```
// replay the track if the loop button is 'on' when the track reaches the end
if (isLooping) {
    if (player->getPosition() ≥ player->getLengthInSeconds()) {
        player->setPosition(0);
        player->start();
    }
}
```

*(DeckGUI.cpp - line 364)*



*(Loop button 'Off')*



*(Loop button 'On')*

- **Skip front and back (5 seconds)**

The “skip front” button skips to 5 seconds after the current play position, while the “skip back” button skips to 5 seconds before the current play position

In the **DeckGUI::buttonClicked()** function, if the “skip front” button is clicked, the position of the player is set to 5 seconds after the current position. Similarly, if the “skip back” button is clicked, the position of the player is set to 5 seconds before the current position.

**DJAudioPlayer::setPosition()** - **DJAudioPlayer.cpp, line 81**  
**DJAudioPlayer::getPosition()** - **DJAudioPlayer.cpp, line 118**

```
// if >> button is clicked
if (button == &skipFrontButton) {
    DBG("skipFrontButton pressed");

    // set new position to +5 secs
    player->setPosition(player->getPosition() + 5.0);
}

// if << button is clicked
if (button == &skipBackButton) {
    DBG("skipBackButton pressed");

    // get new position (-5 secs of current pos)
    double pos{ player->getPosition() - 5.0 };

    // set new position if it is not lesser than 0
    if (!pos <= 0) {
        player->setPosition(pos);
    }
}
```

*(DeckGUI.cpp - line 302)*



*(Skip back button)*

*(Skip front button)*

- **MUTE**

The mute button is located under the volume slider.

In **DeckGUI::buttonClicked()** function, if the mute button is toggled 'On', the gain of the player will be set to 0. This will mute the volume while the track continues playing until the mute button is pressed again which will set the toggle state back to 'Off'.

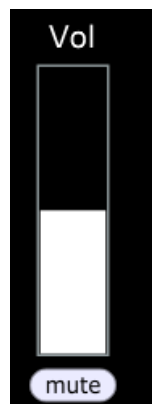
**DJAudioPlayer::setGain()** - **DJAudioPlayer.cpp**, line 61

```
// if the mute button is clicked
if (button == &muteButton) {
    DBG("mute button pressed");

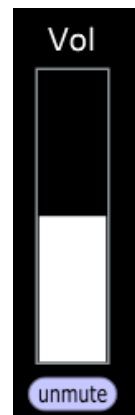
    // if mute button is 'On'
    if (muteButton.getToggleState()) {
        player->setGain(0); // set volume to 0
        muteButton.setButtonText("unmute");
    }

    else {
        muteButton.setButtonText("mute");
        player->setGain(volSlider.getValue());
    }
}
```

(DeckGUI.cpp - line 254)



(mute button 'Off')



(mute button 'On')

- **2x Speed**

The two times button is located under the speed slider

In **DeckGUI::buttonClicked()** function, if the 2x button is toggled 'On', the speed of the player is set to 2. When the button is toggled 'Off', the speed of the player will be set back to the original speed before the 2x button was clicked.

(For example, if the current speed value is 0.7 and I click the 2x button, the speed value is changed to 2. If I click the 2x button again, the value will be changed back to 0.7.)

```
// if the 2x button is clicked
if (button == &twoTimesButton) {
    DBG("2x button pressed");
    // set speed to 2 if the twotimes button is 'On'
    if (twoTimesButton.getToggleState()) {
        player->setSpeed(2);
        twoTimesButton.setButtonText(speedSlider.getTextFromValue(speedSlider.getValue())); // set text to the speed slider value
    }

    // set twoTimesButton text back to "2.0x"
    else {
        twoTimesButton.setButtonText("2.0x");
        player->setSpeed(speedSlider.getValue());
    }
}
```

*(DeckGUI.cpp - line 270)*



*(2x button 'Off')*

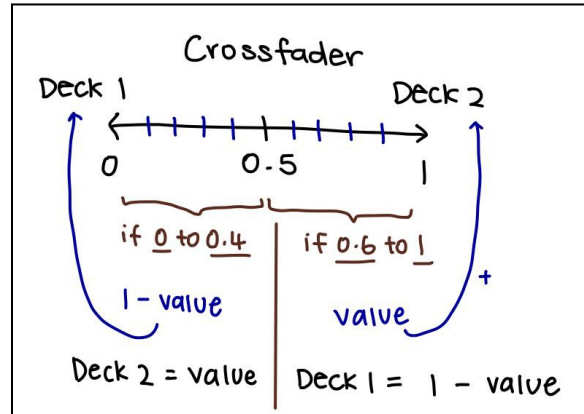


*(2x button 'On')*

- **Crossfader**

The crossfader is a slider which goes across the two decks. This crossfader slider allows the track in both decks to fade in or out by moving the slider left and right. Moving the slider towards the left will allow the track playing in deck 1 to fade in, while deck 2 fades out. Similarly, moving the slider towards the right will allow the track in deck 2 to fade in, while deck 1 fades out.

To better visualise how this should work, I drew the diagram below. The Gain of Deck 1 and Deck 2 should be adjusted according to the value of the crossfader slider.



A Crossfader class is created for this crossfader component. When the crossfader slider has been moved, **Crossfader::sliderValueChanged()** is called and the gain value of the player in Deck 1 and 2 is adjusted.

```
/* Determine what action to take when the value of a slider is changed */
void Crossfader::sliderValueChanged(Slider* slider) {
    if (slider == &crossfadeSlider) {
        // 0.5 is the neutral value between the 2 players
        // if crossfadeSlider value is lesser than 0.5, deckGUI1 should get louder
        if (slider->getValue() < 0.5) {
            deckGUI1->player->setGain(1 - slider->getValue());
            deckGUI2->player->setGain(slider->getValue());
        }

        // if crossfadeSlider value is more than 0.5, deckGUI2 should get louder
        else if (slider->getValue() > 0.5) {
            deckGUI2->player->setGain(slider->getValue());
            deckGUI1->player->setGain(1 - slider->getValue());
        }
    }
}
```

(Crossfader.cpp - line 51)

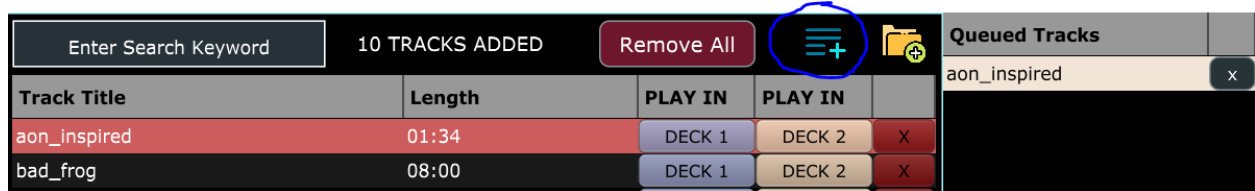


(crossfader slider)



- **Queue Component**

The queue component can be found on the right of the playlist component. Tracks can be added to the queue by selecting a row in the library(selected row highlighted in red), and clicking on the “add to queue” icon circled blue below. The track title of the selected row is then displayed in the Queued Tracks table. In DeckGUI, when the loaded track in Deck 1 or 2 has finished playing, the first queued track will be loaded and played automatically.



A vector of custom class TrackInfo is declared as a private member in QueueComponent.h to store information of tracks added into the queue. When the “add to queue” icon is clicked, the information of the selected row is added into the queuedTracks vector.

```
// vector storing information of queued tracks
std::vector<TrackInfo> queuedTracks;
```

*(QueueComponent.h - line 138)*

```
else if (button == &addToQueue) {
    DBG("PlaylistComponent::buttonClicked addToQueue button clicked");

    // if at least 1 row is selected
    if (tableComponent.getNumSelectedRows() > 0) {
        int selectedRow = tableComponent.getSelectedRow(); // get id of selected row

        // add the file of the selected row into queuedTracks vector
        queueComponent->queuedTracks.push_back(trackInfo[selectedRow].file);
        queueComponent->queueTable.updateContent();
    }
}
```

*(PlaylistComponent.cpp - line 225)*

The file information stored in queuedTracks is then painted in the cell of the first column.

```
/* drawing of information inside each cell */
void QueueComponent::paintCell(Graphics& g, int rowNumber, int columnId, int width, int height, bool rowIsSelected) {
    if (rowNumber < getNumRows()) {
        if (columnId == 1) {
            g.drawText(queuedTracks[rowNumber].trackTitle, 2, 0, width - 4, height, Justification::centredLeft, true);
        }
    }
}
```

*(QueueComponent.cpp - line 70)*

In the **DeckGUI::timerCallback()** function, after a file is loaded into deck 1 or 2 and player is started, when the player reaches the end of the file, checking is done to see if the “Play Queue” button in the Queue Component is toggled ‘On’. If it is toggled ‘On’, the track at the very top of the queue (added first) will be loaded and played in the deck. Once a track from the queue has been loaded into either deck, it will be deleted from the queue. However, if no file is loaded into any of the two decks when the “Play Queue” button is ‘On’, the queued track will be loaded into whichever deck that is empty.

```
// play songs from the queue when the player reaches end of track
if (player->getPosition() ≥ player->getLengthInSeconds()) {
    if (queueComponent->playQueueButton.getToggleState() && queueComponent->queuedTracks.size() > 0) {

        // the first item in queuedTracks vector
        File file = queueComponent->queuedTracks[0].file;
        player->loadURL(URL{ file });
        waveformdisplay.loadURL(URL{ file });
        setNameAndLength(file);

        player->setPosition(0);
        player->start();
        playpauseButton.setText("PAUSE");

        // erase the first item in the vector (queuedTracks[0])
        queueComponent->queuedTracks.erase(queueComponent->queuedTracks.begin());
        queueComponent->queueTable.updateContent();
    }
}
```

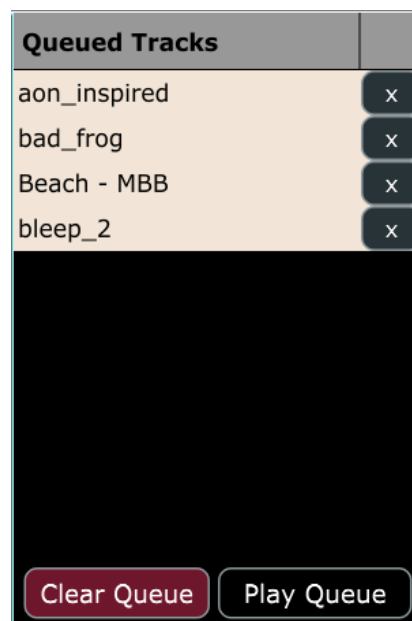
*(DeckGUI.cpp - line 373: loading of file from queue into the deck)*



*(Toggle 'Off')*



*(Toggled 'On')*



*(Queue Component with 4 tracks added to the queue)*

- **Queue Component (Remove & Clear Queue)**

The queue component also has a 'remove' and 'clear queue' component.

The 'remove' component allows the user to remove one track from the queue by clicking on the button marked with an 'x' beside the name of each track in the queue. This is done in the **QueueComponent::buttonClicked()** function where if any of the 'x' button is clicked, the id of the row that is clicked is used to remove the track from the queuedTracks vector.

```
// get the component's ID as an int
int id = stoi(button->getComponentID().toString());

// delete the item stored in queuedTracks[id]
if (button->getText() == "x") {
    DBG("QueueComponent::buttonClicked - delete button clicked");
    queuedTracks.erase(queuedTracks.begin() + id);
    queueTable.updateContent();
}
```

*(QueueComponent.cpp - line 115)*

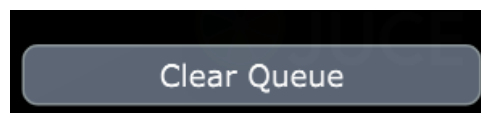
The 'clear queue' component is similar to 'remove' except it removes all the tracks from the queue. This is done by using the **vector::clear()** function to clear the queuedTracks vector.

```
// clear the queue if "Clear Queue" button clicked
if (button == &clearQueue) {
    DBG("QueueComponent::buttonClicked - clear queue button clicked");
    if (!queuedTracks.empty()) {
        queuedTracks.clear();
        queueTable.updateContent();
    }
}
```

*(QueueComponent.cpp - line 96)*



Queued Tracks	
Beach - MBB	x
bleep_2	x
Dreams - Bensound	x
fast_melody_regular_drums	x

*(All queued tracks with an 'x' beside each track)*



*(button to clear all tracks from the queue)*

### R3: Implementation of a music library component

Enter Search Keyword	10 TRACKS ADDED	Remove All		
Track Title	Length	PLAY IN	PLAY IN	
aon_inspired	01:34	DECK 1	DECK 2	X
bad_frog	08:00	DECK 1	DECK 2	X
bleep_2	00:50	DECK 1	DECK 2	X
bleep_10	00:18	DECK 1	DECK 2	X
c_major_theme	01:46	DECK 1	DECK 2	X
electro_smash	02:14	DECK 1	DECK 2	X
fast_melody_regular_drums	01:24	DECK 1	DECK 2	X
soft	00:13	DECK 1	DECK 2	X
stomper_reggae_bit	01:28	DECK 1	DECK 2	X
stomper1	02:44	DECK 1	DECK 2	X

*(music library component)*

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

In order to store information of files added into the library, I created a TrackInfo class which takes in a file, and initialises the member variables using values extracted from the file passed in.

```
/* Initialises the other variables with the file passed in */
TrackInfo::TrackInfo(File _file)
    : file(_file),
      trackTitle(_file.getFileNameWithoutExtension().toStdString()), // file name without the extension
      trackLength(getTrackLength(_file)) // length of the track in minutes:seconds format
{
```

*(TrackInfo.cpp - line 14)*

In PlaylistComponent.h, a vector of TrackInfo Class is created to store information of tracks loaded into the music library.

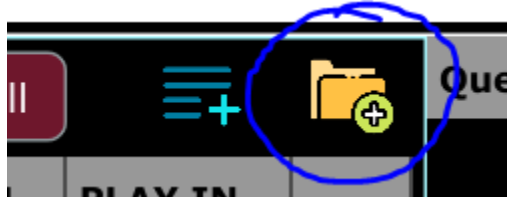
```
// vector for storing and retrieving track information
std::vector<TrackInfo> trackInfo;
```

*(PlaylistComponent.h - line 228)*

In order for information to be displayed, files have to be added first. Users can add files into the library through two different ways.

### 1. Clicking on the “Load to Library” button.

Below shows the “load to library” icon which is an image button. This is done by loading an image from an image file, and setting up the images to draw in different states.



*(“load to library” icon circled in blue)*

```
// load image from image file
Image loadIcon = ImageCache::getFromMemory(BinaryData::load_icon_png, BinaryData::load_icon_pngSize);
```

*(PlaylistComponent.cpp - line 23)*

```
// set drawing of image in different states
loadToLibrary.setImages(true, true, true,
                        loadIcon, 1.0f, Colours::transparentBlack, // normal
                        loadIcon, 1.0f, Colour(0xFFBBBBBF),        // on mouse over
                        loadIcon, 0.5f, Colours::transparentBlack); // when clicked
```

*(PlaylistComponent.cpp - line 27)*

When the “Load to Library” icon is clicked, the **PlaylistComponent::addToLibrary()** function is called. In the **addToLibrary()** function, a dialog box is opened for the user to choose one or more files to add into the library. The file name of the file(s) selected is then stored into the trackInfo vector so that it can be displayed in the music library.

```
/* Determine what action to take when a button is clicked */
void PlaylistComponent::buttonClicked(Button* button) {
    if (button == &loadToLibrary) {
        DBG("PlaylistComponent::buttonClicked loadToPlaylist button clicked");
        addToLibrary();
    }
}
```

*(PlaylistComponent.cpp - line 214)*

```
void PlaylistComponent::addToLibrary() {
    FileChooser chooser{ "Select file(s) to add into library..." };

    // able to select one or multiple files
    if (chooser.browseForMultipleFilesToOpen()) {
        for (File& file : chooser.getResults()) {
```

*(PlaylistComponent.cpp - line 293)*

Checking is also done to ensure that the file that the user wants to add does not already exist in the library. This is done by comparing the file path of the file(s) stored in the trackInfo vector, with the file path of the track the user wants to add. If the file path does not already exist, the file is added into the trackInfo vector.

```
// compare name of chosen file with the names stored in trackTitle to determine if it already exists
if (!trackInfo.empty()) {
    for (TrackInfo& track : trackInfo) {
        if (track.file == file)
            exists = true;
    }
}

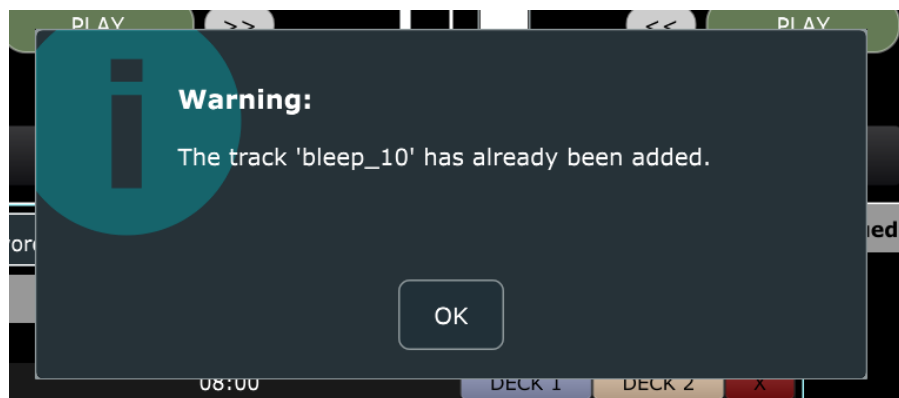
// add file into trackInfo if the file does not already exist
if (!exists) {
    trackInfo.push_back(file);
}
```

*(PlaylistComponent.cpp - line 303)*

If the file already exists in the library, a message box will be displayed.

```
// display message box if track name already exists in library
else {
    AlertWindow::showMessageBox(AlertWindow::AlertIconType::InfoIcon,
        "Warning:", // title
        "The track '" + file.getFileNameWithoutExtension() + "' has already been added.", // message
        "OK", // button text
        nullptr);
}
```

*(PlaylistComponent.cpp - line 317)*



*(Message box displayed when adding duplicate file into music library)*

## 2. Drag and Drop multiple files into the library

Files can also be added into the library by dragging and dropping one or more files into the library. When a file is dragged above the library, the **isInterestedInFileDrag()** function is called. It returns true to indicate interest in receiving the file(s).

```
/* Determines whether the component is interested in the set of files being dragged in */
bool PlaylistComponent::isInterestedInFileDrag(const StringArray&) {
    return true;
}
```

*(PlaylistComponent.cpp - line 331)*

Once the file(s) are dropped into the library, the **filesDropped()** function is called. The same checking is done as per above to ensure that the file(s) dropped does not already exist in the library. The same message box will be displayed if the file already exists in the library. If the file path does not already exist, the file is added into the trackInfo vector.

```
/* Processing of the files dropped onto this component */
void PlaylistComponent::filesDropped(const StringArray& files, int, int) {
    // one or more files dropped
    for (String file : files) {
        File droppedFile{ file };

        // track does not exist = false, already exists = true
        bool exists = false;

        // compare name of chosen file with the names stored in trackTitle to determine if it already exists
        if (!trackInfo.empty()) {
            for (TrackInfo& line : trackInfo) {
                if (line.file == droppedFile)
                    exists = true;
            }
        }

        // add track name and length into respective vectors if track does not already exist
        if (!exists) {
            trackInfo.push_back(droppedFile);
        }
    }
}
```

*(PlaylistComponent.cpp - line 336)*

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

When files are first added into the library either through the “Load to Library” button or drag and drop, the information passed in is the full path of the file (For example, *file:///C%3A/Desktop/Tracks/bleep\_2.mp3*). This file is then added into the trackInfo vector of custom class TrackInfo which is R3A.

In the TrackInfo class, the juce function **juce::File::getFileNameWithoutExtension()** is used to get just the name of the file. This information is then used to initialise the trackTitle member variable.

```
trackTitle(_file.getFileNameWithoutExtension().toStdString())
```

*(TrackInfo.cpp - line 16)*

The **TrackInfo::getTrackLength()** function is created to get the length of the file in minutes:seconds format. This is done by creating a temporary reader to get the length of the file in seconds using the formula total number of samples in the audio stream divided by the sample rate of the stream. This will return the length of the file in seconds. The length in seconds is divided by 60 to get the minutes, and the modulo operation is used for getting the seconds (remainder after dividing by 60). Both are converted into std::string format. The function then returns the length as a std::string in minutes:seconds format. This value is then used to initialise the trackLength member variable.

```
trackLength(getTrackLength(_file)) // length of the track in minutes:seconds format
```

*(TrackInfo.cpp - line 17)*

```
/* Calculates the length of the track and returns it in minutes:seconds format */
std::string TrackInfo::getTrackLength(File file) {
    AudioFormatManager formatManager;
    formatManager.registerBasicFormats();

    // create temporary reader for the file
    AudioFormatReader* reader = formatManager.createReaderFor(file);

    if (reader) {
        // get the length(in seconds) of the track (lengthInSamples/sampleRate)
        int lengthInSecs = reader->lengthInSamples / reader->sampleRate;
        delete reader; // delete the reader after using

        std::string mins = std::to_string(lengthInSecs / 60); // minutes in string
        std::string secs = std::to_string(lengthInSecs % 60); // seconds in string (the remainder)

        // if secs is single digit, add a leading zero
        if (secs.length() == 1)
            secs = "0" + secs;

        // if mins is single digit, add a leading zero
        if (mins.length() == 1)
            mins = "0" + mins;

        // return in minutes:seconds format
        return mins + ":" + secs;
    }
}
```

*(TrackInfo.cpp - line 22: getting length of track in minutes:seconds format)*



With the `trackInfo` vector created in `PlaylistComponent`, the title and length of all files in the vector can be obtained using `trackInfo[index].trackTitle` or `trackInfo[index].trackLength`.

In the `PlaylistComponent::paintCell()` function, `juce::Graphics::drawText()` is used to draw the track title and length onto the table. Track titles are drawn on the first column, while track lengths are drawn on the second column.

```
g.drawText(trackInfo[rowNumber].trackTitle, 2, 0, width - 4, height, Justification::centredLeft, true);
```

*(PlaylistComponent.cpp - line 147: Drawing of track title)*

```
g.drawText(trackInfo[rowNumber].trackLength, 2, 0, width - 4, height, Justification::centredLeft, true);
```

*(PlaylistComponent.cpp - line 156: Drawing of track length)*

Track Title	Length
aon_inspired	01:34
bad_frog	08:00
bleep_10	00:18
c_major_theme	01:46
electro_smash	02:14
fast_melody_regular_drums	01:24

*(File name and length displayed in the playlist component)*

### R3C: Component allows the user to search for files

To achieve this, a search box is created for the user to key in a keyword.

```
// search box to search for a track
TextEditor searchBox;
```

*(PlaylistComponent.h - line 247)*

In the constructor of PlaylistComponent, a lambda expression is used to call the **PlaylistComponent::search()** function on every text change in the search box.

```
// lambda expression for calling search function on every text change in the search box
searchBox.onTextChanged = [this] {
    search(searchBox.getText());
};
```

*(PlaylistComponent.cpp - line 63)*

In the **search()** function, a vector of TrackInfo class is used to store the tracks that match the search keyword. The trackInfo vector is iterated over and the title of each track stored in the vector is compared with the keyword input by the user. If the title contains the keyword, that file will be added into the searchResult vector and displayed as a search result.

```
/* Search for tracks within the library */
void PlaylistComponent::search(String keyword) {
    // clear vector before adding new search results
    searchResult.clear();

    // compare keyword to track titles of all added tracks to find matches
    if (keyword.isNotEmpty()) {
        for (TrackInfo& track : trackInfo) {
            if (String{ track.trackTitle }.containsIgnoreCase(keyword)) {
                searchResult.push_back(track.file);
            }
        }
    }
```

*(PlaylistComponent.cpp - line 422)*

Enter Search Keyword		10 TRACKS ADDED	
Track Title		Length	
aon_inspired		01:34	
bad_frog		08:00	
bleep_2		00:50	
bleep_10		00:18	
c_major_theme		01:46	
electro_smash		02:14	
fast_melody_regular_drums		01:24	
soft		00:13	
stomper_reggae_bit		01:28	
stomper1		02:44	

*(empty search box)*

sto		2 RESULTS FOUND	
Track Title		Length	
stomper_reggae_bit		01:28	
stomper1		02:44	

*(displaying search results for keyword "sto")*

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

There are two ways users can load files from the library into Deck 1 or 2.

### 1. “DECK 1” and “DECK 2” button

Clicking on the “DECK 1” or “DECK 2” button will load the file into Deck 1 and 2 respectively. The deck on the left is Deck 1 and the deck on the right is Deck 2.

Track Title	Length	PLAY IN	PLAY IN	
aon_inspired	01:34	DECK 1	DECK 2	X
bad_frog	08:00	DECK 1	DECK 2	X
bleep_2	00:50	DECK 1	DECK 2	X

This is done by having 2 instances of DeckGUI class, where deckGUI1 represents Deck 1, while deckGUI2 represents Deck 2. In PlaylistComponent.cpp, when the “DECK 1” button is clicked, the URL of the row of file where the button has been clicked is loaded into the player in Deck 1. The same is done if the “DECK 2” button is clicked but loaded into the player in Deck 2.

```
// to load tracks into deckGUI1 & deckGUI2
DeckGUI* deckGUI1;
DeckGUI* deckGUI2;
```

*(PlaylistComponent.h - line 243)*

```
// id of the row where button clicked
int id = stoi(button->getComponentID().toString());

// load track into Deck 1 player if "DECK 1" button is pressed
if (button->getButtonText() == "DECK 1") {
    DBG("PlaylistComponent::buttonClicked DECK 1 button clicked");

    if (searchBox.isEmpty()) {
        deckGUI1->player->loadURL(URL{ trackInfo[id].file });
        deckGUI1->waveformdisplay.loadURL(URL{ trackInfo[id].file });
        deckGUI1->waveformdisplay.setPositionRelative(deckGUI1->player->getPositionRelative());

        deckGUI1->setNameAndLength(File{ URL(URL{ trackInfo[id].file }).getLocalFile() });
    }
}
```

*(PlaylistComponent.cpp - line 240: Loading file into Deck 1 player)*

## 2. Drag and drop from library to deck

Files can also be loaded into the deck by dragging and dropping from the library into the deck. This is done by making the `PlaylistComponent` a `DragAndDropContainer` and implementing `PlaylistComponent::getDragSourceDescription()` function. This function returns details which the target of the drag and drop will need, which in this case is the URL of the file.

```
class PlaylistComponent : public Component,
                          public TableListBoxModel,
                          public Button::Listener,
                          public FileDragAndDropTarget,
                          public TextEditor::Listener,
                          public DragAndDropContainer
```

*(PlaylistComponent.h - line 25)*

```
/* To allow any track added in the library to be dragged and dropped into deckGUI1 or deckGUI2 */
var PlaylistComponent::getDragSourceDescription(const SparseSet<int>& selectedRow) {
    String details;

    // return the URL of the row that is dragged (in String)
    // use trackInfo vector if searchBox is empty and searchResult vector otherwise
    URL row = searchBox.isEmpty() ? URL{ trackInfo[selectedRow[0]].file } : URL{ searchResult[selectedRow[0]].file };
    details << row.toString(false) << " ";

    return details;
}
```

*(PlaylistComponent.cpp - line 468)*

In `DeckGUI.cpp`, the `DeckGUI::isInterestedInDragSource()` function returns true when a file is dragged in to indicate that it is interested in the file. `DeckGUI::itemDropped()` will then process the information passed in from `PlaylistComponent::getDragSourceDescription()`, which is the URL of the file. This URL is then loaded into the player.

```
/* Check whether the component is interested in the type of object being dragged in */
bool DeckGUI::isInterestedInDragSource(const SourceDetails& /*dragSourceDetails*/) {
    return true;
}

/* Processes the item dropped in */
void DeckGUI::itemDropped(const SourceDetails& dragSourceDetails) {
    URL trackURL = URL{ dragSourceDetails.description.toString() };

    player->loadURL(trackURL);
    waveformdisplay.loadURL(trackURL);

    setNameAndLength(File{ trackURL.getLocalFile() });
}
```

*(DeckGUI.cpp - line 395)*

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

The **PlaylistComponent::saveLibrary()** function is called in the destructor to allow the library to be saved whenever the application is closed. This is done by creating a .txt file to store the file path of all the files that are currently added into the library.

```
/* Persist the library by storing the file path in a .txt file */  
void PlaylistComponent::saveLibrary() {  
    // create a .txt file to store information of tracks added into library  
    std::ofstream playlist("library.txt");  
  
    // store the full track path into the .txt file  
    for (TrackInfo& track : trackInfo) {  
        playlist << track.file.getFullPathName() << '\n';  
    }  
}
```

*(PlaylistComponent.cpp - line 438)*

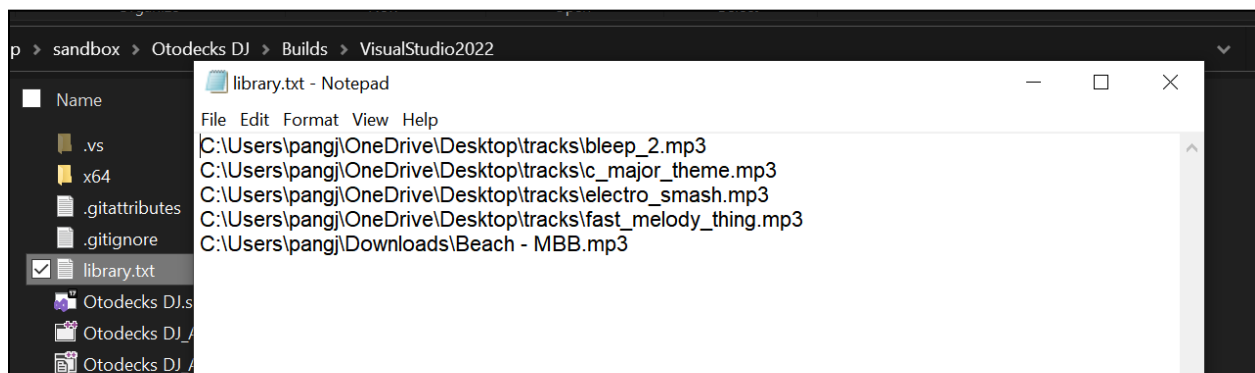
```
PlaylistComponent::~~PlaylistComponent() {  
    // save tracks that are currently in the library whenever destructor is called  
    saveLibrary();  
}
```

*(PlaylistComponent.cpp - line 71: calling the function in the destructor)*

If I were to add the songs as shown in Figure A below into the library and close the application, the library.txt file is created and updated to look like what is shown in Figure B.

Track Title	Length	PLAY IN	PLAY IN	
bleep_2	00:50	DECK 1	DECK 2	X
c_major_theme	01:46	DECK 1	DECK 2	X
electro_smash	02:14	DECK 1	DECK 2	X
fast_melody_thing	01:20	DECK 1	DECK 2	X
Beach - MBB	02:02	DECK 1	DECK 2	X

*(Figure A)*



*(Figure B)*

Upon opening the application, the **PlaylistComponent::addSavedLibrary()** function is called in the constructor to open and read the .txt file containing the file paths. The file path is then added to the trackInfo vector to be displayed in the music library. This will allow the music library to have the same files that have been added into the library when the application was last closed.

```
/* Loads track by reading from the .txt file created in saveLibrary() function */
void PlaylistComponent::addSavedLibrary() {
    // open the .txt file containing saved tracks
    std::ifstream library{ "library.txt" };

    if (library.is_open()) {
        std::string line;

        while (getline(library, line, '\n')) {
            File file { line };
            trackInfo.push_back(file);
        }
    }

    tableComponent.updateContent();
    // close the .txt file
    library.close();
}
```

*(PlaylistComponent.cpp - line 449)*

```
// call function to restore library
addSavedLibrary();
```

*(PlaylistComponent.cpp - line 68)*

## **R4: Implementation of a complete custom GUI**

Figure A in the next page shows a screen capture of the GUI that was coded while following along with coursera lecture videos. Figure B shows the custom GUI which I have coded for this coursework.

### **(R4A) GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls**

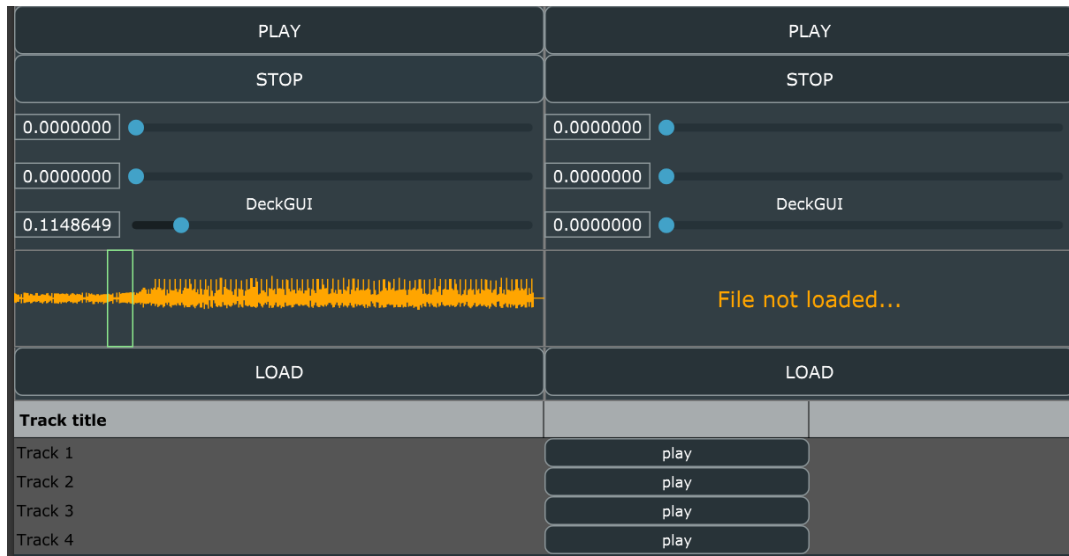
- The waveform display has been moved to the top of deck 1 and 2, with the position slider right below it for easy control.
- The components in Deck 1 and Deck 2 mirror each other.
- The name and length of the file is displayed after loading into a deck.
- Different LookAndFeel versions used to change button and slider style.

### **(R4B) GUI layout includes the custom Component from R2**

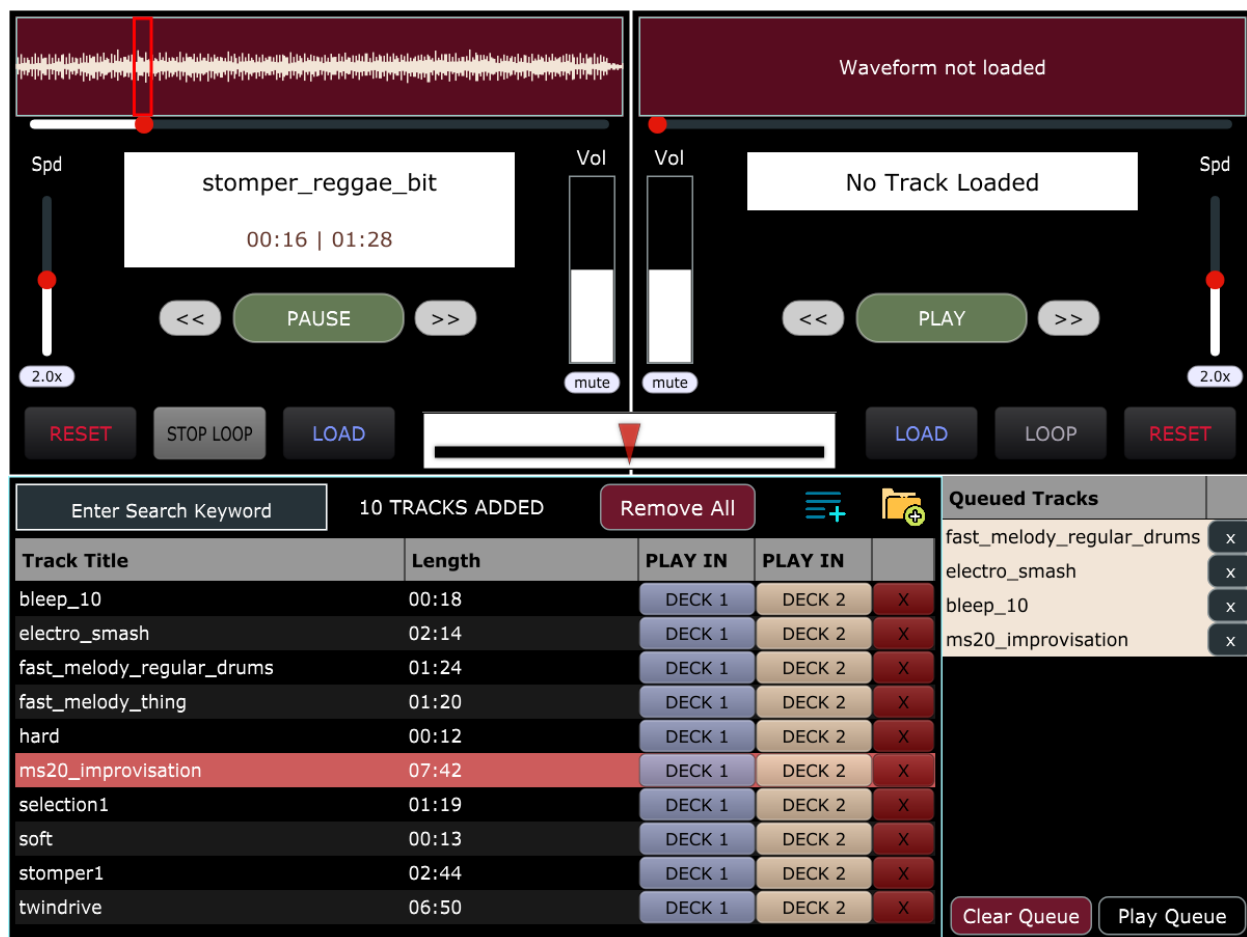
- Crossfader
- Loop
- Mute
- 2x
- Skip forward / backward
- Queue Component

### **(R4C) GUI layout includes the music library component from R3**

- Playlist Component located below the 2 decks, which allows loading of files into the library, removing files added, loading tracks from the library into deck 1 and deck 2, and the ability to search for tracks.



(Figure A: DeckGUI shown in class)

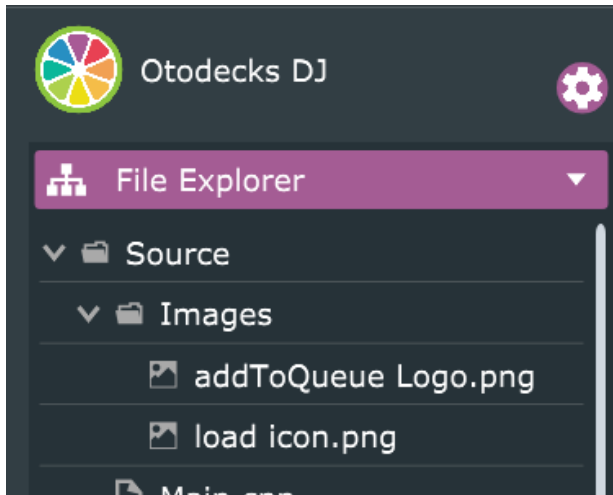


(Figure B: Custom GUI)



## Reference for images

Two images have been used in the application (in PlaylistComponent.cpp) which is loaded as image buttons. One for “Load to playlist” and one for the “Add to queue” image button. The two image(.png) files can be found in **Source > Images** in the submitted source code.



load icon:

[https://www.flaticon.com/free-icon/add\\_7039833?related\\_id=7039833](https://www.flaticon.com/free-icon/add_7039833?related_id=7039833)

add to queue icon:

[https://www.flaticon.com/free-icon/add-list\\_8094249?term=add%20to%20queue&page=1&position=21&page=1&position=21&related\\_id=8094249&origin=style](https://www.flaticon.com/free-icon/add-list_8094249?term=add%20to%20queue&page=1&position=21&page=1&position=21&related_id=8094249&origin=style)