



**UNIVERSITY
OF LONDON**

CM2005: OBJECT ORIENTED PROGRAMMING

Student Name: Lawrence Ho Sheng Jin
Date Submitted: 15th March 2021
Degree Title: Computer Science
Local Institution: Singapore Institute of Management
Student ID: 10205927

Table of Contents

<i>Introduction of Audio App:</i>	<i>3</i>
<i>R4: Implementation of a Complete Custom GUI</i>	<i>6</i>
<i>R1: Basic Functionality</i>	<i>18</i>
<i>R2: Custom Deck Control Implementation</i>	<i>26</i>
<i>R3: Music Library Implementation</i>	<i>41</i>
<i>References</i>	<i>50</i>

Introduction of Audio App:

The audio app I have created assumes the following layout:



It has two audio player decks to control basic functions such as playing/stopping audio playback, filters that modify the audio, and a library to load, delete or save audio tracks. Details about respective functions will be further elaborated through the report as per the requirement sequence.

A brief explanation of the classes involved and their purpose:

1. Main: Runs the app
2. MainComponent: Creates objects of every necessary class

Most classes take in a reference of another object, and the constructor declarations can be found in 'MainComponent.h'.

Object creation in 'MainComponent.h'

```
private:
//=====

AudioFormatManager formatManager;
AudioThumbnailCache thumbCache{100};

/* audio player logic */
/* takes in reference to formatManager to create AudioReaderSource */
DJAudioplayer player1{&formatManager};
DJAudioplayer player2{&formatManager};
```

```

/* deckGUI and playlistComponent referencing each other */
/* takes in reference to player for audio playback */
/* takes in reference to formatManager and thumbCache for creating WaveFormDisplay object */
/* takes in reference to playlistComponent to pushback to TrackEntry vector whenever a file is loaded directly
into the deck */
DeckGUI deckGUI1{&player1, &formatManager, &thumbCache, &playlistComponent};
DeckGUI deckGUI2{&player2, &formatManager, &thumbCache, &playlistComponent};

/* playlist aka library */
/* takes in reference to deckGUI for audio playback and controls */
PlaylistComponent playlistComponent{&deckGUI1, &deckGUI2};

/* mixer source for two audioSources from two deckGUIs */
MixerAudioSource mixerSource;

/* Tooltips for ImageButtons on mouse hover */
TooltipWindow tip{this, 700};

/* REV filter GUI */
RevFilterGUI revFilterGUI{&revFilter};
/* reverb audio filter */
/* takes in reference to mixerSource to wrap filter around it */
RevFilter revFilter{&mixerSource};

JUICE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MainComponent)
};

```

- | | |
|-----------------------|--|
| 3. DJAudioPlayer: | Logic of playing audio from audio file |
| 4. DeckGUI: | Component for playing audio file |
| 5. WaveformDisplay: | Component for audio waveform based on audio source |
| 6. PlaylistComponent: | Contains logic and component aspect of the audio library. |
| 7. TrackEntry: | Class that has variables to store audio file data |
| 8. CSVManager: | In charge of reading/writing from vector to '.txt' file or vice versa. |
| 9. Filter: | Logic of audio filters that are applied to audio sources |
| 10. FilterGUI: | Component for filters |
| 11. OtherLookAndFeel: | In charge of custom settings regarding appearance of sliders/buttons |
| 12. Init: | Contains helper functions for initialising buttons/sliders/labels. |

The library is able to insert audio files, load into the DeckGUIs, and save the inserted audio files such that the user can load the very same files upon restarting the app. The audio files data will be

displayed in a tabular format, and there is a search bar allowing users to search for their desired track within the library.

The 2 audio filters alters the audio source differently. One of the audio filter (IIRFilter) alters the audio source from respective audio players, whereas the other filter (Reverb Filter) alters the mixer source that contains both audio sources from the players. Users are able to control the parameters/variables pertaining to the filters using the sliders.

Note that in the the 'Filter' and 'FilterGUI' set of files, there are 2 Filter classes in each set respectively.

***'Filter' contains IIR_Filter and RevFilter classes.*

***'FilterGUI' contains IIRFilterGUI and RevFilterGUI classes.*

I will be going through R4 requirements first, before going through the rest of the requirements R1-3. Given the overview description of R4, I believe it will give a better introduction to the layout and components available in my audio app.

R4: Implementation of a Complete Custom GUI

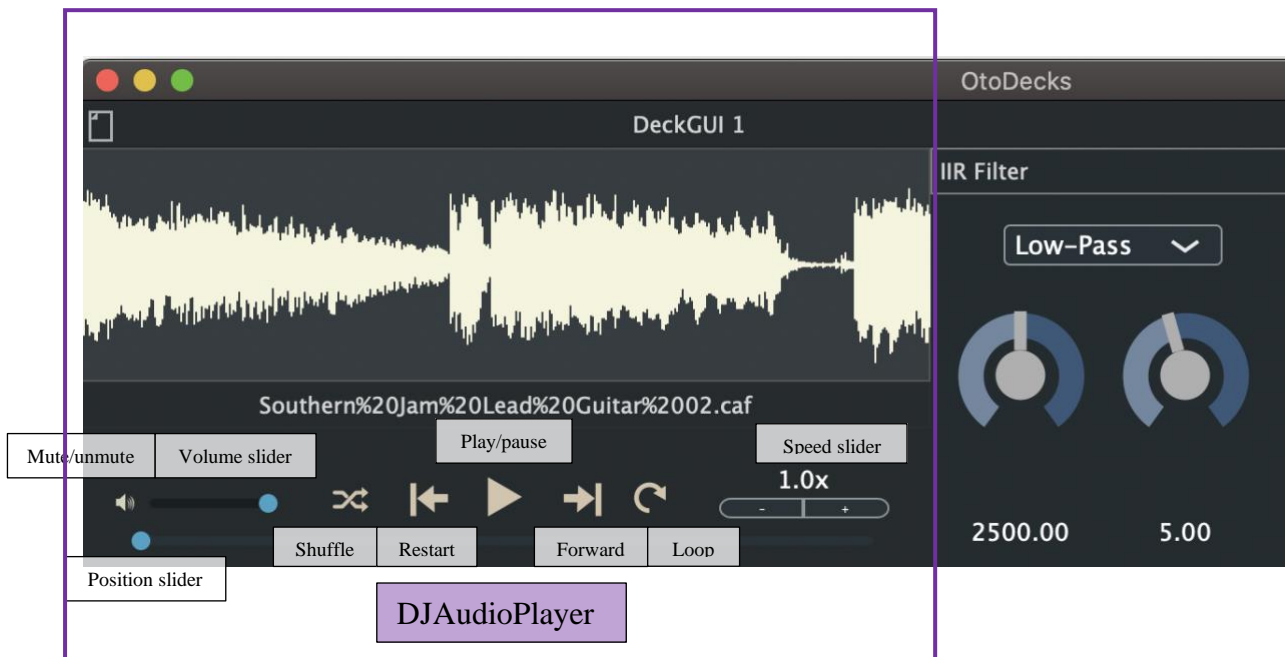
- R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls

As seen in the layout shown in the introduction, the different components are segregated as shown below:



In the deckGUI, there are two parts to it. The DJAudioPlayer object (player), and the IIRFilterGUI Component. Within the player, there are additional controls that I have added in. Whereas in the IIRFilterGUI object, there are sliders and a filter selection drag-down menu to adjust the filter variables and type of IIR filter to be used.

Visual representation of different functions in DeckGUI Component:



All the buttons, sliders and labels created in deckGUI are all ImageButtons. The images are all sourced from <http://simpleicon.com>, and are all initialised using helper functions in 'Init.cpp':

helper function to initialise sliders/buttons/labels in 'Init.cpp'

```
/* helper fn to initialise slider settings */
void Init::sliderOptions(Component *component, Slider *slider, Slider::Listener *listener, Slider::SliderStyle style,
Slider::TextEntryBoxPosition textPos, bool readOnly, int textBoxW, int textBoxH, double rangeStart, double
rangeEnd, double increment, LookAndFeel *otherLookAndFeel, Slider::ColourIds colourid, Colour colour)
{
    /* addAndMakeVisible
    addListener
    setSliderStyle
    setTextBoxStyle
    setColour
    setRange
    setLookAndFeel
    setColour */
    component->addAndMakeVisible(slider);
    slider->addListener(listener);
    slider->setSliderStyle(style);
    slider->setTextBoxStyle(textPos, readOnly, textBoxW, textBoxH);
    slider->setColour(slider->textBoxOutlineColourId, Colours::transparentWhite);
    slider->setRange(rangeStart, rangeEnd, increment);
    slider->setLookAndFeel(otherLookAndFeel);
}
```

```

        slider->setColour(colourid, colour);
    }

    /* helper fn to initialise label settings */
    void Init::labelOptions(Component *component, Label *label, String title, NotificationType notifType, Justification
justification, float fontSize, Label::ColourIds colourid, Colour colour)
    {
        /* addAndMakeVisible
        setText
        setJustificationType
        setFont
        setColour */
        component->addAndMakeVisible(label);
        label->setText(title, notifType);
        label->setJustificationType(justification);
        label->setFont(fontSize);
        label->setColour(colourid, colour);
    }

    /* helper fn to initialise button settings */
    void Init::buttonOptions(Component *component, Button *button, Button::Listener *listener, bool toggleOn, String
tooltip, float alpha)
    {
        /* addAndMakeVisible
        addListener
        setClickingTogglesState
        setTooltip
        setAlpha */
        component->addAndMakeVisible(button);
        button->addListener(listener);
        button->setClickingTogglesState(toggleOn);
        button->setTooltip(tooltip);
        button->setAlpha(alpha);
    }

```

And in the constructor declaration of deckGUI, the images for the buttons are created from local .png files and then set to the buttons. Sliders and labels are all initialised here as well using the Init helper functions.

DeckGUI constructor in 'DeckGUI.cpp'

```

DeckGUI::DeckGUI( DJAudioPlayer* _player,
    AudioFormatManager* formatManagerToUse,
    AudioThumbnailCache* cacheToUse,
    PlaylistComponent* _playlistComponent
)
: waveformDisplay(formatManagerToUse, cacheToUse),
  player(_player),
  playlistComponent(_playlistComponent),
  trackEntries(playlistComponent->getTrackEntries()),
  iirFilterGUI(&player->filter)
{
    /* create images for buttons */
    auto playImage = ImageCache::getFromMemory(BinaryData::play_png, BinaryData::play_pngSize);
    auto pauseImage = ImageCache::getFromMemory(BinaryData::pause_png, BinaryData::pause_pngSize);
    auto loadImage = ImageCache::getFromMemory(BinaryData::file_png, BinaryData::file_pngSize);
    auto ffwdImage = ImageCache::getFromMemory(BinaryData::fast_forward_png,
BinaryData::fast_forward_pngSize);
    auto frevImage = ImageCache::getFromMemory(BinaryData::fast_reverse_png,
BinaryData::fast_reverse_pngSize);
    auto loopImage = ImageCache::getFromMemory(BinaryData::loop_png, BinaryData::loop_pngSize);
    auto volumeImage = ImageCache::getFromMemory(BinaryData::volume_png, BinaryData::volume_pngSize);
    auto mutelImage = ImageCache::getFromMemory(BinaryData::mute_png, BinaryData::mute_pngSize);
    auto shuffleImage = ImageCache::getFromMemory(BinaryData::shuffle_png, BinaryData::shuffle_pngSize);

    /* set images to buttons */
    playButton.setImages(true, true, true, playImage, 1, Colours::blanchedalmond, Image(nullptr), 1,
Colours::transparentWhite, pauseImage, 1, Colours::beige);
    loadButton.setImages(true, true, true, loadImage, 1, Colours::lightgrey, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);
    ffwdButton.setImages(true, true, true, ffwdImage, 0.2, Colours::blanchedalmond, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);
    frevButton.setImages(true, true, true, frevImage, 1, Colours::blanchedalmond, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);
    loopButton.setImages(true, true, true, loopImage, 1, Colours::blanchedalmond, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);
    volumeButton.setImages(true, true, true, volumeImage, 1, Colours::lightyellow, Image(nullptr), 1,
Colours::transparentWhite, mutelImage, 1, Colours::transparentBlack);
    shuffleButton.setImages(true, true, true, shuffleImage, 1, Colours::blanchedalmond, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);

```

```

/* BUTTONS */
Init::buttonOptions(this, &playButton, this, true, "", 0.8f);
Init::buttonOptions(this, &loadButton, this, false, "Insert track to DeckGUI", 0.8f);
Init::buttonOptions(this, &ffwdButton, this, false, "Plays next track in playlist", 0.8f);
Init::buttonOptions(this, &frevButton, this, false, "Goes back to start of track", 0.8f);
Init::buttonOptions(this, &loopButton, this, true, "Loop current track", 0.8f);
Init::buttonOptions(this, &volumeButton, this, true, "", 0.8f);
Init::buttonOptions(this, &shuffleButton, this, true, "Enable shuffling", 0.8f);

/* LABELS */
Init::labelOptions(this, &deckIndex, "", NotificationType::dontSendNotification,
Justification::horizontallyCentred, 14.0f, deckIndex.outlineColourId, getLookAndFeel().findColour
(ResizableWindow::backgroundColourId).withMultipliedBrightness(0.6f));
Init::labelOptions(this, &trackName, "", NotificationType::dontSendNotification,
Justification::horizontallyCentred, 14.0f, deckIndex.outlineColourId, getLookAndFeel().findColour
(ResizableWindow::backgroundColourId).withMultipliedBrightness(0.6f));

//customise
trackName.setAlpha(0.8);
deckIndex.setAlpha(0.8);
deckIndex.toBack();

/* SLIDERS */
Init::sliderOptions(this, &volSlider, this, juce::Slider::LinearHorizontal, juce::Slider::NoTextBox, true, 0, 0, 0.0,
1.0, 0.01, nullptr, volSlider.textBoxTextColourId, Colours::lightgrey);
Init::sliderOptions(this, &speedSlider, this, juce::Slider::IncDecButtons, juce::Slider::TextBoxAbove, true, 50,
50, 0.0, 10.0, 0.5, nullptr, speedSlider.textBoxOutlineColourId, Colours::transparentWhite);
Init::sliderOptions(this, &posSlider, this, juce::Slider::LinearHorizontal, juce::Slider::NoTextBox, true, 0, 0, 0.0,
1.0, 0, nullptr, posSlider.trackColourId, Colours::lightblue.withBrightness(0.7));

/* MISC */
//value
volSlider.setValue(1.0);
speedSlider.setValue(1.0);
//tooltip
speedSlider.setTooltip("Adjust playback speed");
speedSlider.setTextBoxIsEditable(true);
//text
speedSlider.setTextValueSuffix("x");

```

```

/* WFD */
addAndMakeVisible(waveformDisplay);

/* start the timer thread */
startTimer(100); //parameter takes in milliseconds; how frequent u want to call this function

/* FilterGUI */
addAndMakeVisible(iirFilterGUI);
iirFilterGUI.toBack();
}

```

Moving on to the function and logic of the buttons/sliders:

Audio Player functions (BUTTONS):

- a. Mute/unmute
When the user clicks on it, it will set the gain of the audio to 0 or 1. Logic in AudioPlayer, graphical representation in deckGUI.
- b. Shuffle
If the button is toggled, when the forwardButton is clicked, get a random track from the playlist. Else, when the forwardButton is clicked, plays the next track in sequence.
- c. Reverse (restart)
Resets the position of the playback back to the start, and then plays the audio.
- d. Play/pause
Allows user to play/pause audio.
- e. Forward
Goes to the next track present in the playlistComponent. If not, goes back to the first track.
- f. Loop
If the button is toggled, set the audio to loop

The logic for these additional button functions all resides in the buttonClicked() function of deckGUI:

```

void DeckGUI::buttonClicked(Button* button)
{
    //reset search to display trackEntries (NOT the search results vector)
    playlistComponent->trackFound = false;
}

```

```

if (button == &playButton) //if button same as memory address of ____
{
    //starts the audio player if button set to true, pauses if false
    //if playButton toggle state = true (pause img)
    if(button->getToggleState())
        player->start();
    else //false
        player->stop();
}

//load file into deckGUI and playlist
if (button == &loadButton)
{
    FileChooser chooser{"Select a file..."};
    if (chooser.browseForFileToOpen())
    {
        URL audioURL = URL{chooser.getResult()};
        player->loadURL(audioURL);
        waveformDisplay.loadURL(audioURL);
        trackName.setText(audioURL.getFileName(), dontSendNotification);
        //adding to trackEntries vector and updating tableComponent in playlistComponent
        playlistComponent->addToTrackEntries(audioURL, player->getTrackDuration());
        // playlistComponent->printTrackEntries(); //tester code
    }
}

//reverse button (go back to start)
if (button == &frevButton)
{
    player->stop();
    player->setPosition(0);
    player->start();
    //change state of playButton to display pause img
    playButton.setToggleState(true, dontSendNotification);
}

//forward button (go to next track in trackEntries)
if (button == &ffwdButton)
{
    std::cout<<"shuffle: "<<shuffle<<std::endl;
    //does nothing if empty, error handler
    if(trackEntries->size()==0)
        return;
}

```

```

//stops the player, gets the URL of the next track, loads it, and then plays it
player->stop();

//get the next track based on the trackCounter index in trackEntries
URL audioURL = playlistComponent->getNextTrack(shuffle);
player->loadURL(audioURL);
waveformDisplay.loadURL(audioURL);

//change the trackName to new loaded track
trackName.setText(audioURL.getFileName(), dontSendNotification);
player->start();

//reset the playButton state
playButton.setToggleState(true, dontSendNotification);
}

//loop track
if(button == &loopButton)
{
    //if loopButton toggled on, audio will be looped
    player->setLoop(button->getToggleState());
//    std::cout<<"shuffle state:: "<<shuffleButton.getToggleState()<<std::endl;

}

//mute/unmute button
if(button == &volumeButton)
{
    //if button is clicked (true) -> change button image to mute image and adjust gain value according to slider
    value
    if(button->getToggleState())
    {
        player->setGain(0);
        volSlider.setValue(0);
    }

    //else if button state is (false), change button image to unmute image and adjust gain value according to
    slider value
    else
    {
        player->setGain(1);
        volSlider.setValue(1);
    }
}

//sets shuffle condition

```

```

if(button == &shuffleButton)
{
    if(button->getToggleState())
        shuffle = true;
    else //false
        shuffle = false;

    // std::cout<<"shuffle state:: "<<button->getToggleState()<<std::endl;
}
}

```

Audio Player functions (SLIDERS):

a. Volume Slider

Sets the gain of audio source according to slider value.

b. Position Slider

Sets the audio playback position according relative to the slider value. This slider moves along as the audio plays as well, showing the user where the current playback location is along the slider.

c. Speed Slider

Increases/decreases speed of playback.

The logic for these additional button functions all resides in the buttonClicked() function of deckGUI:

```

void DeckGUI::sliderValueChanged (Slider *slider)
{
    if (slider == &volSlider)
    {
        //as long as volume > 0, volumeButton state = false (unmuted)
        if(slider->getValue()>slider->getMinimum())
        {
            volumeButton.setToggleState(false, dontSendNotification);
        }

        //adjusts volume of audio playback depending on slider value
        player->setGain(slider->getValue());
    }

    if (slider == &speedSlider)

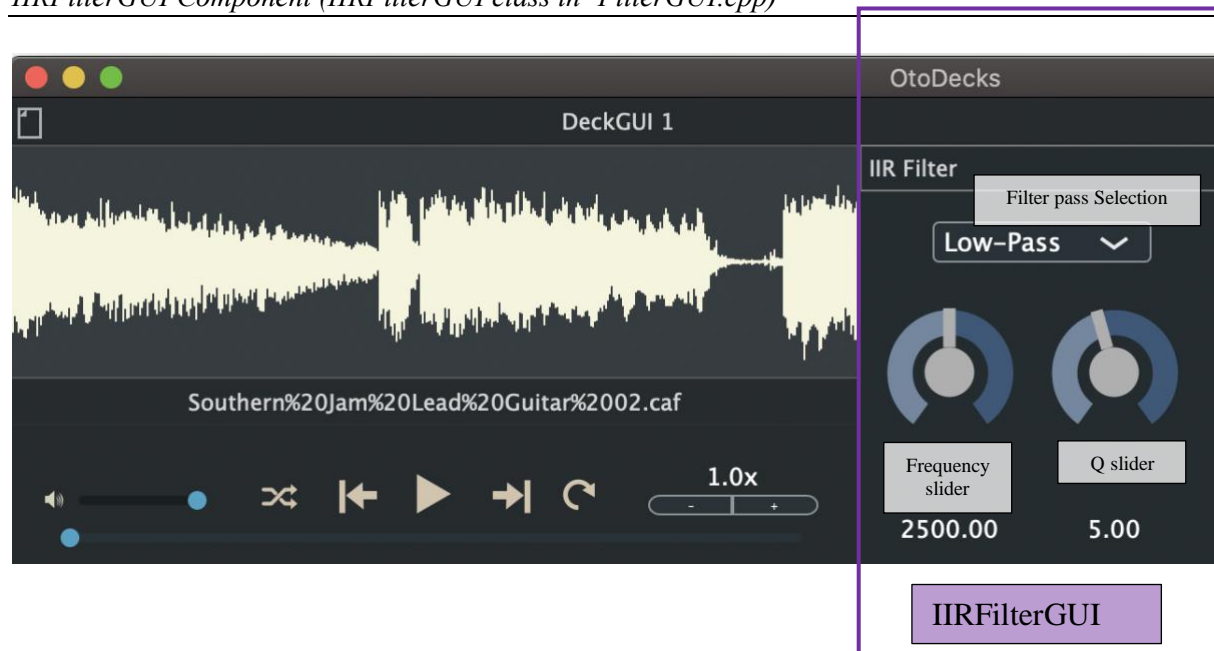
```

```
{
    //set the speed of playback based on slider value
    player->setSpeed(slider->getValue());
}

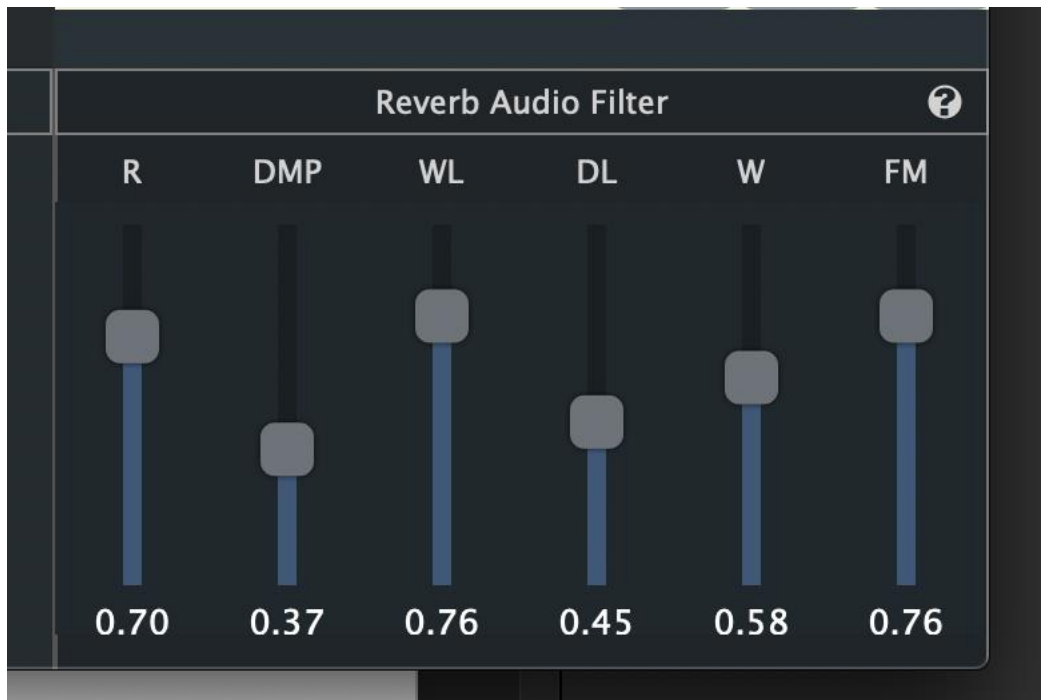
if (slider == &posSlider)
{
    if(slider->getValue() == slider->getMaximum())
    {
        //if reach end of track, reset playButton
        playButton.setToggleState(false, dontSendNotification);
    }
    player->setPositionRelative(slider->getValue());
}
}
```

- R4B: GUI layout includes the custom Component from R2

IIRFilterGUI Component (IIRFilterGUI class in 'FilterGUI.cpp')



RevFilterGUI Component (RevFilterGUI class in 'FilterGUI.cpp')



More details on the layout and logic of these two components will be further elaborated in R2.

- R4C: GUI layout includes the music library Component from R3

PlaylistComponent in 'PlaylistComponent.cpp'. Contains TableListBoxModel object tableComponent.

Playlist				
Search				
Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete
Southern%20Jam%20Le...	0:08	Load	Load	Delete
Megaton%20Pitch%20B...	0:07	Load	Load	Delete
Counter melody%20Gui...	0:07	Load	Load	Delete
Adele%20-%20Rolling%...	3:53	Load	Load	Delete
aon_inspired.mp3	1:34	Load	Load	Delete
c_major_theme.mp3	1:46	Load	Load	Delete
bleep_2.mp3	0:50	Load	Load	Delete

More details on the layout and logic of these two components will be further elaborated in R3.

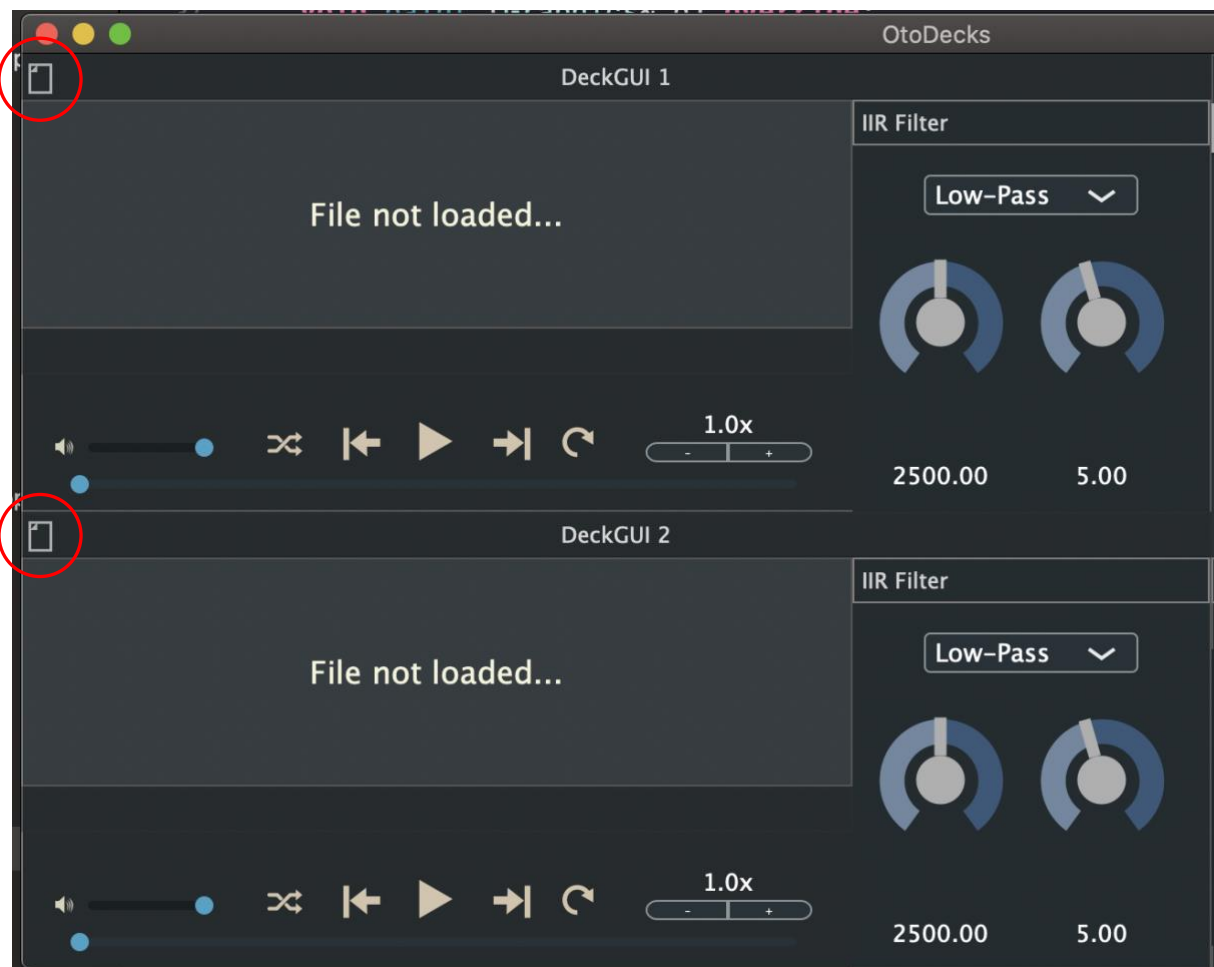
R1: Basic Functionality

- R1A: Can load audio files into audio players.

There are two ways to achieve this in my audio app.

1. Load the audio file directly into the deckGUI itself.

There is a load button located on the top left corner of respective deckGUIs, and by clicking on the button, it creates a FileChooser object which opens the file browser window for the user to select a file for input. Upon getting a valid file, the player then loads the track and other necessary details.



buttonClicked function in 'DeckGUI.cpp'

```
void DeckGUI::buttonClicked(Button* button)
{
    //reset search to display trackEntries (NOT the search results vector)
    playlistComponent->trackFound = false;

    //load file into deckGUI and playlist
```

```

if (button == &loadButton)
{
    FileChooser chooser{"Select a file..."};
    if (chooser.browseForFileToOpen())
    {
        URL audioURL = URL{chooser.getResult()};
        player->loadURL(audioURL);
        waveformDisplay.loadURL(audioURL);
        trackName.setText(audioURL.getFileName(), dontSendNotification);
        //adding to trackEntries vector and updating tableComponent in playlistComponent
        playlistComponent->addToTrackEntries(audioURL, player->getTrackDuration());
    }
}

```

2. Load into playlistComponent -> load into deckGUI.

It gets a little more complicated when loading a track into the deckGUI from the playlistComponent. Firstly, in the tableComponent (TableListBox) present in playlistComponent, I create load buttons in respective rows depending on the size of the trackEntries vector. This can be found in refreshComponentForCell(). I then assign a unique button id to each of these buttons by concatenating the rowID and colID.

refreshComponentForCell() in 'PlaylistComponent.cpp':

```

//override
Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int columnId, bool isRowSelected,
Component* existingComponentToUpdate)
{
    //colId 3 = load deckGUI1
    //colId 4 = load deckGUI2
    if(columnId == 3 || columnId == 4)
    {
        if(existingComponentToUpdate == nullptr)
        {
            //create textbutton pointer, textbutton is a child class of component, so able to store a text button onto a
            component pointer
            TextButton* btn = new TextButton{"Load"};
            //unique id -> rowNumber + columnId
            String id{std::to_string(rowNumber)+std::to_string(columnId)};
            btn->setComponentID(id);

```

```

//add itself as a button listener
btn->addListener(this);

existingComponentToUpdate = btn;
}
}

```

From there, anytime a ‘click’ input is received in the respective load buttons in playlistComponent, I retrieve the rowIndex and colIndex from the unique button id (concatenated rowID + colID). Depending on the row of the loadButton clicked in tableComponent within the library, it loads the track present in that particular row into the deckGUI. The column index of the tableComponent determines which deckGUI to load it into.

Playlist		Search		
Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete
Southern%20Jam%20Le...	0:08	Load	Load	Delete
Megaton%20Pitch%20B...	0:07	Load	Load	Delete
Counter melody%20Gui...	0:07	Load	Load	Delete
Adele%20-%20Rolling%...	3:53	Load	Load	Delete
aon_inspired.mp3	1:34	Load	Load	Delete
c_major_theme.mp3	1:46	Load	Load	Delete
bleep_2.mp3	0:50	Load	Load	Delete

buttonClicked function in ‘PlaylistComponent.h’:

```

//component override
void PlaylistComponent::buttonClicked(Button* button)
{
    //.....
    //load file into different deck depending on which col button is clicked
    if(colIndex == 3) //column index 3: deckGUI1
    {
        //reset playButton
    }
}

```

```

deckGUI1->playButton.setToggleState(false, dontSendNotification);
deckGUI1->player->loadURL(trackToPlay);
deckGUI1->waveformDisplay.loadURL(trackToPlay);
    deckGUI1->waveformDisplay.setPositionRelative(deckGUI1->player->getPositionRelative());
//changing track name displayed in deckGUI
deckGUI1->trackName.setText(trackToPlay.getFileName(), dontSendNotification);
}
if(collIndex == 4) //column index 4: deckGUI2
{
    //reset playButton
    deckGUI2->playButton.setToggleState(false, dontSendNotification);
    deckGUI2->player->loadURL(trackToPlay);
    deckGUI2->waveformDisplay.loadURL(trackToPlay);
    deckGUI2->waveformDisplay.setPositionRelative(deckGUI1->player->getPositionRelative());
//changing track name displayed in deckGUI
    deckGUI2->trackName.setText(trackToPlay.getFileName(), dontSendNotification);
}

```

-
- R1B: Can play two or more tracks

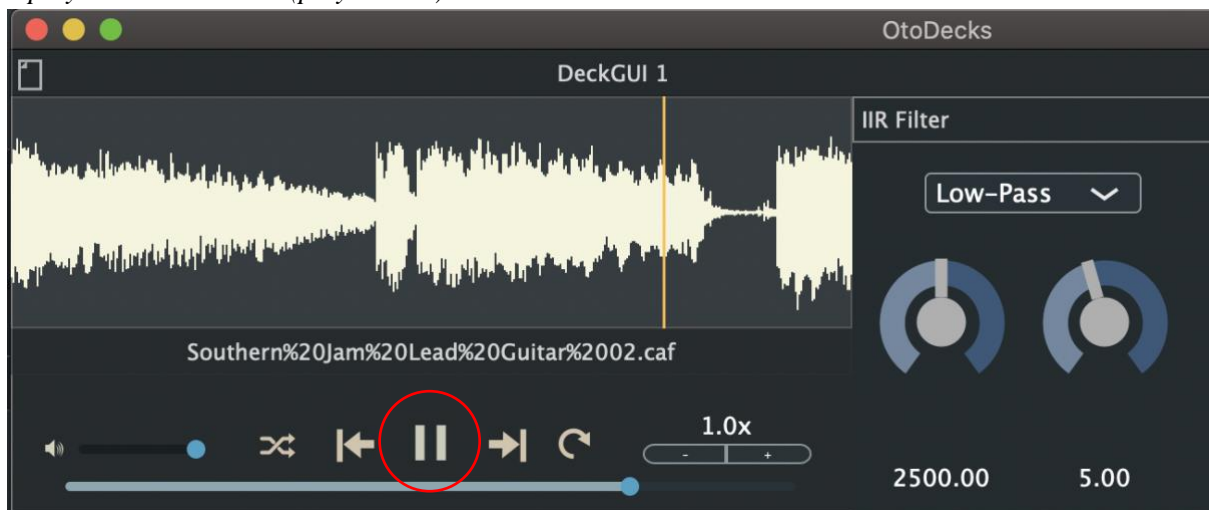
There are two deckGUIs available, hence up to two audio files can be played simultaneously. The controls and functions for playing the audio files are located in respective deckGUIs.

The playButton has two toggle states, when set to true, the playButton will start the audio from the player. When set to false, it will stop the audio playback.

** playButton set to false (pauses audio)*



**playButton set to true (plays audio)*



**(player is a declared DJAudioPlayer object)
'DeckGUI.cpp'*

```
void DeckGUI::buttonClicked(Button* button)
{
    //reset search to display trackEntries (NOT the search results vector)
    playlistComponent->trackFound = false;

    if (button == &playButton) //if button same as memory address of ____
    {
        //starts the audio player if button set to true, pauses if false
        if(button->getToggleState()) //true
            //if playButton toggle state = true (pause img)
```

```

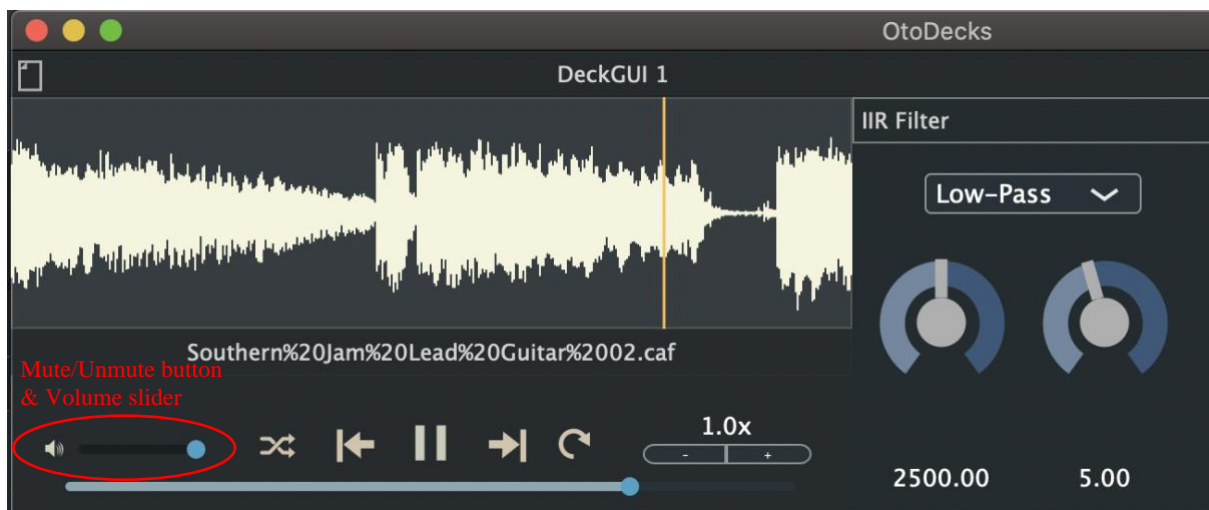
player->start();
else //false
    player->stop();
}

```

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

The volume of the audio files is controlled by the volume slider displayed on the bottom left portion of respective deckGUIs. It ranges from 0 – 100% of the volume, and there is an additional mute/unmute function beside the slider. The mute button has two states: when set to true, it will mute the audio, and when set to false, unmute the audio.

**Mute/Unmute button in deckGUI & volume slider*



The following code handles the logic for manually changing the slider value:
'DeckGUI.cpp'

```

void DeckGUI::sliderValueChanged (Slider *slider)
{
    if (slider == &volSlider)
    {
        //as long as volume > 0, volumeButton state = false (unmuted)
        if(slider->getValue()>slider->getMinimum())
    }
}

```

```

{
    volumeButton.setToggleState(false, dontSendNotification);
}

//adjusts volume of audio playback depending on slider value
player->setGain(slider->getValue());
}

```

And the following code handles the logic for clicking on the mute/unmute button:
'DeckGUI.cpp'

```

//mute/unmute button
if(button == &volumeButton)
{
    if(button == &volumeButton)
    {
        //if button is clicked (true) -> change button image to mute image and adjust gain value
        if(button->getToggleState())
        {
            player->setGain(0);
            volSlider.setValue(0);
        }

        //else if button state is (false), change button image to unmute image and adjust gain value
        else
        {
            player->setGain(1);
            volSlider.setValue(1);
        }
    }
}

```

-
- R1D: Can speed up and slow down the tracks

There is a slider that can increase/decrease the speed ratio of the tracks. Again, this slider is located in respective deckGUIs. Depending on the slider value set by the user, it will set the speed of the audio playback accordingly. There is a visual text indication on the ratio of the increments/decrements made just above the slider.



'DeckGUI.cpp'

```
void DeckGUI::sliderValueChanged (Slider *slider)
{
    if (slider == &speedSlider)
    {
        //set the speed of playback based on slider value
        player->setSpeed(slider->getValue());
    }
}
```

The slider style has been changed to IncDecButtons to assume a button appearance, and a text suffix “x” at the end of the slider text value (e.g ‘2.5x’) for the user to understand that this slider controls the playback speed ratio.

'DeckGUI.cpp'

```
//customise
speedSlider.setSliderStyle(Slider::SliderStyle::IncDecButtons);
//text of speedSlider
speedSlider.setColour(speedSlider.textBoxOutlineColourId, Colours::transparentWhite);
speedSlider.setTextValueSuffix("x");
```

R2: Custom Deck Control Implementation

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

I have two additional components (aside from deckGUI) that controls the deck playback:

1. **IIRFilterGUI component**
2. **RevFilterGUI component**

These components are contained in 'FilterGUI.cpp'. The components mainly consists of sliders that modifies the parameters/values necessary for the filters to be applied on the audio sources. The visuals of the sliders have a customised lookAndFeel settings that I have initialised in 'OtherLookAndFeel.cpp', which changes the slider colour and appearance.

I liked the default background colour, so I added some variation to the colours by applying a saturation or brightness modifier. The colours declared in paint have varied shades of navy blue, showcasing a 'dark' appearance theme.

There are additional functions within the deckGUIs that will be further explained in R3. All the buttons are represented by an image icon that is easily interpreted and customary to other audio software.

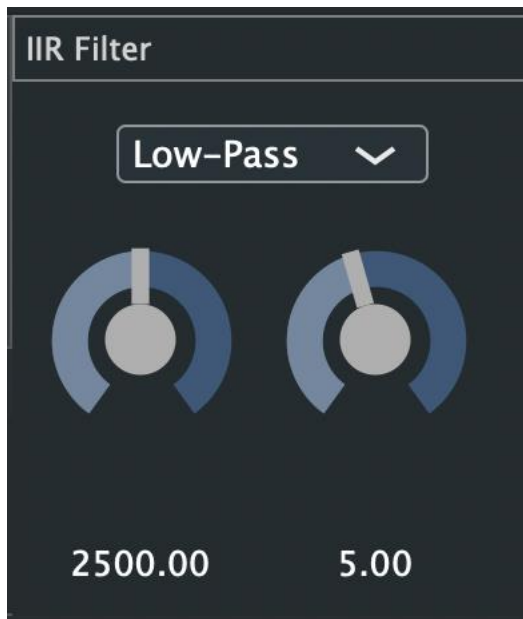
'Filter.h' + 'Filter.cpp' :

- a. *Contains logic for the actual filter.*
- b. *2 Filter classes are present here: IIR_Filter and RevFilter*

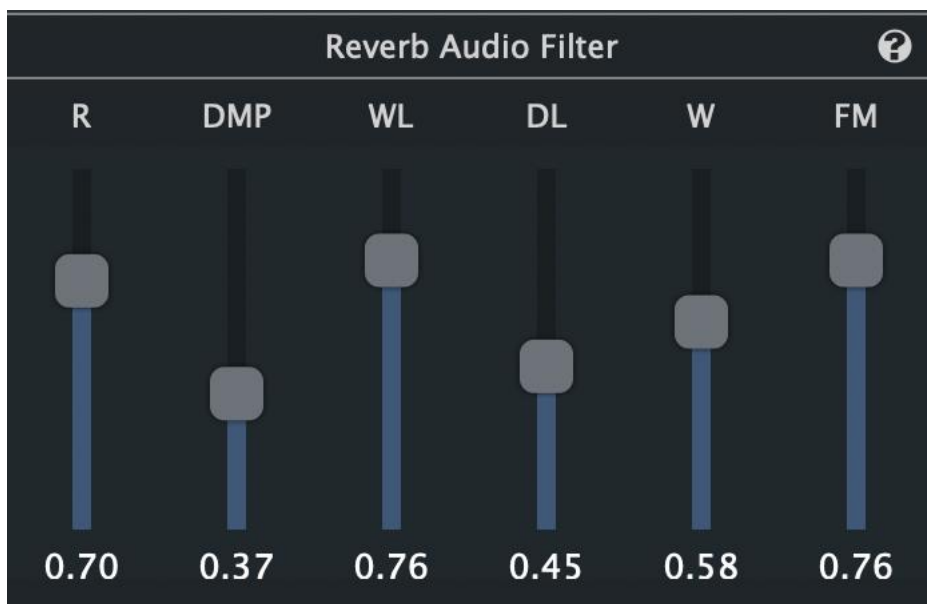
'FilterGUI.h' + 'FilterGUI.cpp' :

- a. *Contains component appearance for the filters in Filter class.*
- b. *2 FilterGUI classes are present here: IIRFilterGUI and RevFilterGUI*

IIRFilterGUI Component: 'FilterGUI.cpp'



RevFilterGUI Component: 'FilterGUI.cpp'



IIRFilterGUI's sliders uses a rotary slider style, and the appearance settings are initialised in the following code:

The code in 'OtherLookAndFeel.cpp' was referenced from:

<https://github.com/juce-framework/JUCE/blob/master/examples/GUI/LookAndFeelDemo.h>

<https://github.com/remberg/juceCustomSliderSample/blob/master/Source/myLookAndFeel.cpp>

Changing the colours and some settings taken from the referenced link mentioned in 'OtherLookAndFeel.cpp'

```
/* customise rotary slider appearance -> used in FilterGUI */  
void OtherLookAndFeelV1::drawRotarySlider(Graphics& g, int x, int y, int width, int height, float sliderPos,
```

```

        float rotaryStartAngle, float rotaryEndAngle, Slider& slider)
{
    const float radius = jmin(width / 2, height / 2) * 0.85f;
    const float centreX = x + width * 0.5f;
    const float centreY = y + height * 0.5f;
    const float rx = centreX - radius;
    const float ry = centreY - radius;
    const float rw = radius * 2.0f;
    const float angle = rotaryStartAngle
        + sliderPos
        * (rotaryEndAngle - rotaryStartAngle);

    g.setColour(Colour(0xff39587a));
    Path filledArc;
    filledArc.addPieSegment(rx, ry, rw + 1, rw + 1, rotaryStartAngle, rotaryEndAngle, 0.6);

    g.fillPath(filledArc);

    g.setColour(Colour(0xff39587a).brighter());
    Path filledArc1;
    filledArc1.addPieSegment(rx, ry, rw + 1, rw + 1, rotaryStartAngle, angle, 0.6);

    g.fillPath(filledArc1);

    Path p;
    const float pointerLength = radius * 0.63f;
    const float pointerThickness = radius * 0.2f;
    p.addRectangle(-pointerThickness * 0.5f, -radius - 1, pointerThickness, pointerLength);
    p.applyTransform(AffineTransform::rotation(angle).translated(centreX, centreY));
    g.setColour(Colours::whitesmoke.darker());
    g.fillPath(p);

    const float dotradius = radius * (float)0.4;
    const float dotradius2 = rw * (float)0.4;
    g.setColour(Colours::whitesmoke.darker());
    g.fillEllipse(centreX - (dotradius),
        centreY - (dotradius),
        dotradius2, dotradius2);
}

```

The sliders in RevFilter are initialised in a similar manner. However, the slider styles here are linear, and therefore, in a separate function. The slider thumbs for RevFilterGUI have been changed to a rounded rectangle instead of using an ellipse:

'OtherLookAndFeel.cpp'

```
/* customise slider thumb appearance for FilterGUI */
void OtherLookAndFeelV1::drawRoundThumb (Graphics& g, float x, float y, float diameter, Colour colour, float
outlineThickness)
{
    auto halfThickness = outlineThickness * 0.5f;

    Path p;
    p.addRoundedRectangle(x + halfThickness, y + halfThickness, diameter - outlineThickness, diameter -
outlineThickness, 5.0f);

    DropShadow (Colours::black, 1, {}).drawForPath (g, p);

    g.setColour (Colour(0xff39587a).withMultipliedSaturation(0.2));
    g.fillPath (p);

    // g.setColour (colour.darker());
    g.strokePath (p, PathStrokeType (outlineThickness));
}
```

An OtherLookAndFeelV1 object will be created in the *'FilterGUI.h'* file as shown here:

```
/* customise slider appearance */
OtherLookAndFeelV1 v1;
```

The sliders, labels and other misc objects will then be initialised in the constructors of *'IIRFilterGUI'* and *'RevFilterGUI'* classes with the helper function in *'Init.cpp'*.

RevFilterGUI class in 'FilterGUI.cpp'

```
RevFilterGUI::RevFilterGUI(RevFilter* _revFilter) : revfilter(_revFilter)
{
    /* SLIDERS */
    Init::sliderOptions(this, &roomSizeSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0,
1.0, 0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);
```

```

    Init::sliderOptions(this, &dampingSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0,
1.0, 0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);

    Init::sliderOptions(this, &wetLevelSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0,
1.0, 0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);

    Init::sliderOptions(this, &dryLevelSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0,
1.0, 0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);

    Init::sliderOptions(this, &widthSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0, 1.0,
0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);

    Init::sliderOptions(this, &freezeModeSlider, this, Slider::LinearVertical, Slider::TextBoxBelow, false, 50, 10, 0.0,
1.0, 0.01, &v1, roomSizeSlider.textBoxOutlineColourId, Colours::transparentWhite);

/* LABEL */

    Init::labelOptions(this, &label, "Reverb Audio Filter", dontSendNotification, Justification::centred, 14.0f,
label.textColourId, Colours::lightgrey);

    Init::labelOptions(this, &roomSize, "R", dontSendNotification, Justification::centred, 14.0f, label.textColourId,
Colours::lightgrey);

    Init::labelOptions(this, &damping, "DMP", dontSendNotification, Justification::centred, 14.0f, label.textColourId,
Colours::lightgrey);

    Init::labelOptions(this, &wetLevel, "WL", dontSendNotification, Justification::centred, 14.0f, label.textColourId,
Colours::lightgrey);

    Init::labelOptions(this, &dryLevel, "DL", dontSendNotification, Justification::centred, 14.0f, label.textColourId,
Colours::lightgrey);

    Init::labelOptions(this, &width, "W", dontSendNotification, Justification::centred, 14.0f, label.textColourId,
Colours::lightgrey);

    Init::labelOptions(this, &freezeMode, "FM", dontSendNotification, Justification::centred, 14.0f,
label.textColourId, Colours::lightgrey);

//labels attach to slider
roomSize.attachToComponent(&roomSizeSlider, false);
damping.attachToComponent(&dampingSlider, false);
wetLevel.attachToComponent(&wetLevelSlider, false);
dryLevel.attachToComponent(&dryLevelSlider, false);
width.attachToComponent(&widthSlider, false);
freezeMode.attachToComponent(&freezeModeSlider, false);

/* IMAGEBUTTON */
auto infolImage = ImageCache::getFromMemory(BinaryData::info_png, BinaryData::info_pngSize);
infoButton.setImages(true, true, true, infolImage, 1, Colours::lightgrey, Image(nullptr), 1,
Colours::transparentWhite, Image(nullptr), 1, Colours::transparentBlack);
addAndMakeVisible(infoButton);

```

```
infoButton.setToolTip("R: Room-size, DMP: Damping, WL: Wet-level, DL: Dry-level, W: Width, FM: Freeze-
mode");
}
```

IIRFilterGUI class in 'FilterGUI.cpp'

```
IIRFilterGUI::IIRFilterGUI(IIR_Filter *_filter) : filter(_filter)
{
    /* SLIDERS */
    Init::sliderOptions(this, &freqSlider, this, Slider::RotaryHorizontalDrag, juce::Slider::TextBoxBelow, false, 100,
15, 1, 5000, 0.01, &v1, freqSlider.textBoxOutlineColourId, Colours::transparentWhite);
    Init::sliderOptions(this, &qSlider, this, Slider::RotaryHorizontalDrag, juce::Slider::TextBoxBelow, false, 100, 15,
1, 10, 0.01, &v1, qSlider.textBoxOutlineColourId, Colours::transparentWhite);

    //value
    freqSlider.setValue(2500);
    qSlider.setValue(5);

    /* COMBO BOX */
    addAndMakeVisible(&filterMenu);
    filterMenu.addItem("Low-Pass", 1);
    filterMenu.addItem("Band-Pass", 2);
    filterMenu.addItem("High-Pass", 3);
    //id
    filterMenu.setSelectedId(1);
    //listener
    filterMenu.addListener(this);

    /* LABEL */
    Init::labelOptions(this, &label, "IIR Filter", dontSendNotification, Justification::centredLeft, 14.0f,
label.textColourId, Colours::lightgrey);
}
```

Slider helper function in 'Init.cpp'

```
/* helper fn to initialise slider settings */
```

```

void Init::sliderOptions(Component *component, Slider *slider, Slider::Listener *listener, Slider::SliderStyle style,
Slider::TextEntryBoxPosition textPos, bool readOnly, int textBoxW, int textBoxH, double rangeStart, double
rangeEnd, double increment, LookAndFeel *otherLookAndFeel, Slider::ColourIds colourid, Colour colour)
{
    /* addAndMakeVisible
    addListener
    setSliderStyle
    setTextBoxStyle
    setColour
    setRange
    setLookAndFeel
    setColour */
    component->addAndMakeVisible(slider);
    slider->addListener(listener);
    slider->setSliderStyle(style);
    slider->setTextBoxStyle(textPos, readOnly, textBoxW, textBoxH);
    slider->setColour(slider->textBoxOutlineColourId, Colours::transparentWhite);
    slider->setRange(rangeStart, rangeEnd, increment);
    slider->setLookAndFeel(otherLookAndFeel);
    slider->setColour(colourid, colour);
}

```

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

The sliders in both FilterGUI components effectively modifies the parameters/values necessary for the filters to be applied on the audio sources. Both FilterGUI classes take in a reference object to the actual respective filters, where the logic for applying the filter is stored. As mentioned, in the Filter files, there are **2 classes**:

1. IIR_Filter

This filter wraps an IIRFilterAudioSource around the ResamplingAudioSource in the DJAudioPlayer class. The IIRFilter takes in two coefficients: freq and q. These values are then passed into the IIR::setCoefficients function, before applying it to the ResamplingAudioSource.

The IIRFilterGUI class takes in a reference of IIR_Filter declared in DJAudioPlayer:

'DJAudioPlayer.h'

```

AudioTransportSource transportSource;

//the constructor for ResamplingAudioSource receives a pointer to an AudioSource, which is transportSource
in this case

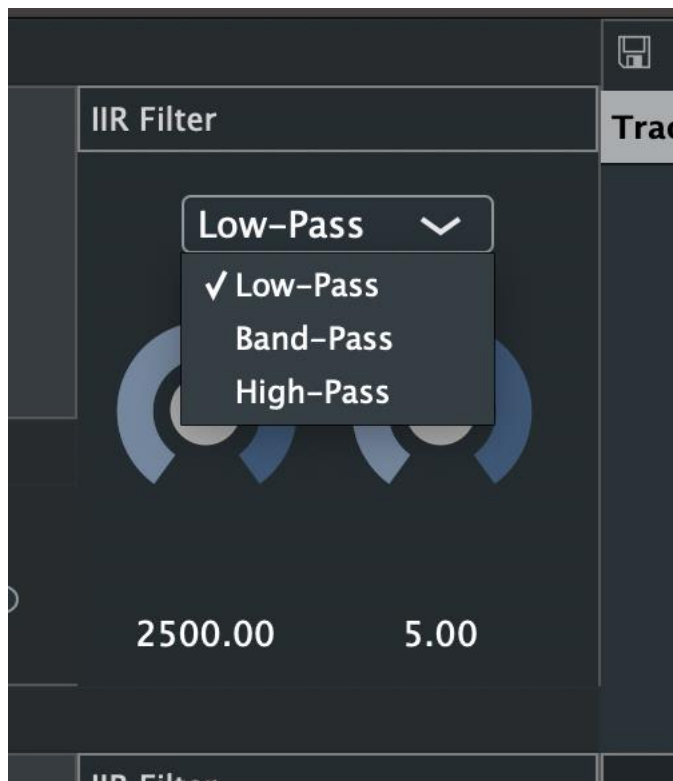
ResamplingAudioSource resampleSource{&transportSource, false, 2};

/* IIR filter on mixerSource */
IIR_Filter filter{&resampleSource};

```

For the IIRFilter, there is an additional selection box for the user to choose between 3 types of coefficients: lowpass, bandpass and highpass. Depending on the user's selection, the type of filter-pass (lowpass/bandpass/highpass) for the coefficients to be passed into will be applied to the resampleSource.

Visual of ComboBox:



IIRFilterGUI class in 'FilterGUI.cpp':

Setting the filterpass based on selection of combo box

```

/* sets the correct filter based on the user's selection */
void IIRFilterGUI::comboBoxChanged(ComboBox *comboBox)
{
    if(comboBox == &filterMenu)
    {

```

```

int state = comboBox->getSelectedId();
switch(state)
{
    //setLowPass IIRCoefficients
    case lowpass:
    {
        filter->setLowPass();
        break;
    }
    //setBandPass IIRCoefficients
    case bandpass:
    {
        filter->setBandPass();
        break;
    }
    //setHighPass IIRCoefficients
    case highpass:
    {
        filter->setHighPass();
        break;
    }
}
}
}

```

The coefficients (freq and q) are modified using values derived from the 2 sliders that comes along with the component IIRFilterGUI. The coefficients and types of pass filter to use will then be passed into the ResamplingAudioSource, which will then alter the playback.

The slider values will be passed into variables freq and q. These set values are updated whenever the user moves the slider, and they are passed into the coefficients of the IIRFilter.

IIRFilterGUI class in 'FilterGUI.cpp':

```

/* get freq/q value from respective sliders */
void IIRFilterGUI::sliderValueChanged (Slider* slider)
{
    if(slider == &freqSlider)
    {
        //set the freq variable for IIR coefficients based on freqSlider value
        filter->freq = slider->getValue();
    }
}

```

```

    //check for comboBox selection
    comboBoxChanged(&filterMenu);
}
if(slider == &qSlider)
{
    //set the q variable for IIR coefficients based on qSlider value
    filter->q = slider->getValue();
    //check for comboBox selection
    comboBoxChanged(&filterMenu);
}
}

```

**Take note that as this filter sits in the DJAudioPlayer class, it only modifies the corresponding audio playback. (one per deck)*

The logic of applying the filter to the audio source will be contained in 'Filter.h'. This is where the coefficients will be set to the filter pass and applied to the audio source.

Create the IIRFilterAudioSource variable in 'Filter.h'

```

/* filterSource to apply on &resampleSource (reference to resampleSource in deckGUI) */
IIRFilterAudioSource filterSource;

```

IIR_Filter class in 'Filter.cpp':

```

void IIR_Filter::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{
    globalSampleRate = sampleRate;
    filterSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void IIR_Filter::releaseResources()
{
    filterSource.releaseResources();
}

void IIR_Filter::getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
{
    filterSource.getNextAudioBlock(bufferToFill);
}

```

```

}

/* sets coefficients of lowpass IIR filter to be applied to audio source */
void IIR_Filter::setLowPass()
{
    IIRCoefficients lowPassFilter = IIRCoefficients::makeLowPass(globalSampleRate, freq, q);
    filterSource.setCoefficients(lowPassFilter);
}

/* sets coefficients of bandpass IIR filter to be applied to audio source */
void IIR_Filter::setBandPass()
{
    IIRCoefficients bandPassFilter = IIRCoefficients::makeBandPass(globalSampleRate, freq, q);
    filterSource.setCoefficients(bandPassFilter);
}

/* sets coefficients of highpass IIR filter to be applied to audio source */
void IIR_Filter::setHighPass()
{
    IIRCoefficients highPassFilter = IIRCoefficients::makeHighPass(globalSampleRate, freq, q);
    filterSource.setCoefficients(highPassFilter);
}

```

And then when the IIR_Filter object is created, it then takes in the ResamplingAudioSource and uses that as its basis. The audio playback will then be based on the IIR_Filter audio source.

Creating the IIR_Filter object in 'DJAudioPlayer.h'

```

/* IIR filter on mixerSource */
IIR_Filter filter{&resampleSource};

```

Playing the audio source from filter in 'DJAudioPlayer.cpp':

```

void DJAudioPlayer::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    // transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    // resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    filter.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void DJAudioPlayer::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)

```

```

{
//  resampleSource.getNextAudioBlock(bufferToFill);
    filter.getNextAudioBlock(bufferToFill);
}

void DJAudioPlayer::releaseResources()
{
//  to complete the life cycle of transportSource
    transportSource.releaseResources();
    resampleSource.releaseResources();
    filter.releaseResources();
}

```

2. RevFilter

RevFilter wraps a ReverbAudioSource around the MixerAudioSource in MainComponent. The RevFilterGUI class then takes in a reference of RevFilter declared in MainComponent.

'MainComponent.h'

```

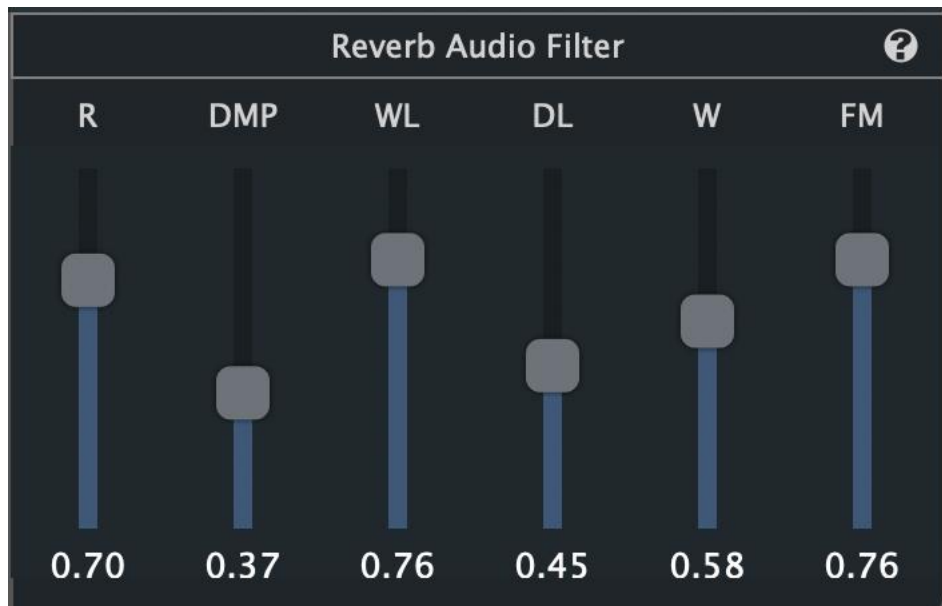
/* mixer source for two audioSources from two deckGUIs */
    MixerAudioSource mixerSource;
/* REV filter GUI */
    RevFilterGUI revFilterGUI{&revFilter};
/* reverb audio filter */
/* takes in reference to mixerSource to wrap filter around it */
    RevFilter revFilter{&mixerSource};

```

The ReverbFilter consists of 6 different parameters:

- a. Room size
- b. Damping
- c. Wet level
- d. Dry level
- e. Width
- f. Freeeze mode

Therefore, the need for 6 sliders to set the values for all of the parameters.



The slider values obtained from the *RevFilterGUI* component are passed into the parameters. These parameter values are updated whenever the user moves the slider.

Depending on slider values from *RevFilterGUI*, the *ReverbFilter* alters the *MixerAudioSource* in various ways.

RevFilterGUI class in 'FilterGUI.cpp'

```
/* get params value from respective sliders */
void RevFilterGUI::sliderValueChanged (Slider* slider)
{
    if(slider == &roomSizeSlider)
        revfilter->params.roomSize = slider->getValue();

    if(slider == &dampingSlider)
        revfilter->params.damping = slider->getValue();

    if(slider == &wetLevelSlider)
        revfilter->params.wetLevel = slider->getValue();

    if(slider == &dryLevelSlider)
        revfilter->params.dryLevel = slider->getValue();

    if(slider == &widthSlider)
        revfilter->params.width = slider->getValue();

    if(slider == &freezeModeSlider)
```

```

    revfilter->params.freezeMode = slider->getValue();
}

```

**Because the MixerAudioSource contains inputs from both audio players, this filter directly alters playbacks from both players at once.*

The logic for setting these parameters will be in the callback audio function getNextAudioBlock(), thus the slider values will be constantly passed into the filter.

Create ReverbAudioSource variable in 'Filter.h'

```

/* reverb audio source to be applied to audio source */
/* apply to &mixerSource in MainComponent */
ReverbAudioSource reverbSource;

```

Preparing and playing reverbSource (based on &mixerSource)

```

void RevFilter::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{
    reverbSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void RevFilter::releaseResources()
{
    reverbSource.releaseResources();
}

void RevFilter::getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
{
    //set params during this callback
    reverbSource.setParameters(params);
    reverbSource.getNextAudioBlock(bufferToFill);
}

```

And then when the RevFilter object is created, it then takes in the MixerAudioSource and uses that as its basis. The MixerAudioSource takes in 2 inputs from both DJAudioPlayer objects. The audio playback will then be based on the RevFilter audio source.

Creating revFilter object in 'MainComponent.h'

```
/* reverb audio filter */  
/* takes in reference to mixerSource to wrap filter around it */  
RevFilter revFilter{&mixerSource};
```

Playing the audio source from revFilter in 'MainComponent.cpp'

```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)  
{  
    // player1.prepareToPlay(samplesPerBlockExpected, sampleRate);  
    // player2.prepareToPlay(samplesPerBlockExpected, sampleRate);  
  
    mixerSource.addInputSource(&player1, false);  
    mixerSource.addInputSource(&player2, false);  
  
    revFilter.prepareToPlay(samplesPerBlockExpected, sampleRate);  
}  
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)  
{  
    revFilter.getNextAudioBlock(bufferToFill);  
}  
  
void MainComponent::releaseResources()  
{  
    // This will be called when the audio device stops, or when it is being  
    // restarted due to a setting change.  
  
    player1.releaseResources();  
    player2.releaseResources();  
    mixerSource.releaseResources(); //not necessary  
    revFilter.releaseResources();  
}
```


R3: Music Library Implementation

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

In my audio app, the music library also serves as a playlist, which I have named the component after (playlistComponent).

I have a separate TrackEntry class that stores data of the audio file. It takes in the URL and duration of the audio file. The file name will then be retrieved from the URL in the constructor.

Class to store audio file data 'TrackEntry.h'

```
class TrackEntry
{
public:
    /* constructor takes in URL and juce::string duration */
    TrackEntry(URL _audioURL, juce::String _duration);

    /* returns TrackEntry in juce::String for **PRINTING** */
    juce::String toString();
    /* returns TrackEntry in juce::String for **CSV file** */
    juce::String toCSV();

// private:
    juce::String title;
    juce::String artist;
    juce::String duration;
    URL audioURL;
};
```

'TrackEntry.cpp'

```
TrackEntry::TrackEntry( URL _audioURL, juce::String _duration)
: audioURL(_audioURL)
{
    //get file name from URL
    title = audioURL.getFileName();

    //convert duration to 2sf and replace '.' with ':'
    //e.g 3.149865 -> 3:14
```

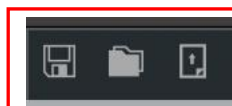
```
duration = juce::String(_duration).substring(0, 4).replaceCharacter('.', ':');
}
```

A local vector of <TrackEntry> named trackEntries will be declared in PlaylistComponent to store all the audio files' data once they are loaded into the playlist. The tableComponent will be updated to display any new files inserted or when any files are deleted through the deleteButton.

'PlaylistComponent.h'

```
/* <TrackEntry> vector to store loaded songs and save songs */
std::vector<TrackEntry> trackEntries;
```

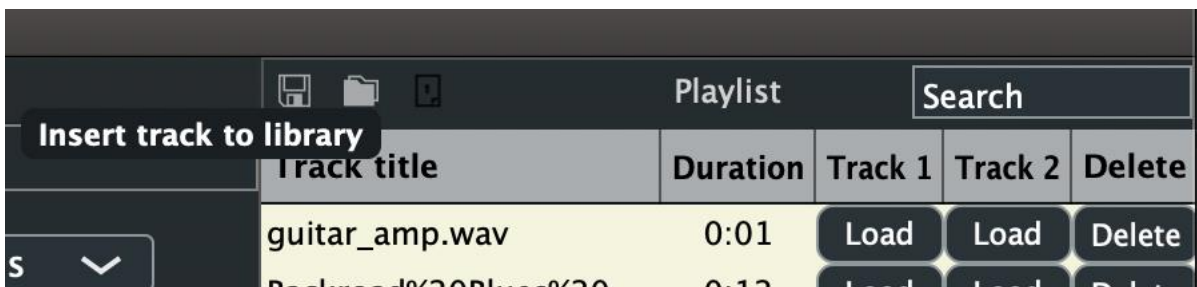
In the playlistComponent, there are 3 menu functions: Save library, load library, insert file to library (shown below in the corresponding order: left to right)



Playlist		Search		
Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete
Southern%20Jam%20Le...	0:08	Load	Load	Delete
Megaton%20Pitch%20B...	0:07	Load	Load	Delete

The insertButton in playlistComponent controls the function to insert a file into the library. By receiving a 'click' input on the insert button, it prompts the File Chooser window, similar to how the load file function works in deckGUI. From there, upon a valid file selection made by the user, a TrackEntry object will be pushed back to the trackEntries vector. The tableComponent will then be refreshed to display the newly inserted file.

**mouse hover tool tips explaining button function*



Playlist		Search		
Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete

buttonClicked function in 'PlaylistComponent.cpp'

```
//component override
void PlaylistComponent::buttonClicked(Button* button)
```

```

{
    else if(button == &insertButton)
    {
        //add to <TrackEntry> upon valid file selection
        FileChooser chooser{"Select a file..."};
        if (chooser.browseForFileToOpen())
        {
            URL audioURL = URL{chooser.getResult()};
            /* adding to <trackEntries> vector and update tableComponent with new push_back */
            addToTrackEntries(audioURL,
deckGUI1->player->getTrackDurationWithoutLoadingIntoDeck(audioURL));
        }
    }
}

```

addToTrackEntries() in 'PlaylistComponent.cpp' as seen above

```

/* adding to <trackEntries> vector and update tableComponent with new push_back */
void PlaylistComponent::addToTrackEntries(URL audioURL, juce::String duration)
{
    trackEntries.push_back(TrackEntry{audioURL, duration});
    tableComponent.updateContent();
}

```

-
- R3B: Component parses and displays meta data such as filename and song length
 - R3C: Component allows the user to search for files

As mentioned above in R3A, I have initialised a vector<TrackEntry> named trackEntries to store TrackEntry objects. These TrackEntry objects contain variables for the URL, file name and track length. Everytime a file is loaded into playlistComponent, a TrackEntry object will be pushed back into the vector containing all the necessary arguments.

The library extracts data from the trackEntries vector and displays them in the tableComponent. The main details of the files displayed are the filename and the track duration.

Playlist		Search		
Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete
Southern%20Jam%20Le...	0:08	Load	Load	Delete

A search bar is present at the top right corner of the playlistComponent that allows the user to enter a text input. Upon hitting the 'return' key, the component retrieves the text input value from the user and searches for any suitable substring matches within the vector trackEntries through the *searchForTrack()* function.

If there is a successful match, the matched TrackEntry objects will be pushed back into a new local vector<TrackEntry> trackEntriesFound, and set the class var bool trackFound to true. Depending on the bool value trackFound, the tableComponent will display results from different vectors.

Vector declaration in 'PlaylistComponent.h'

```
/* <TrackEntry> vector to store loaded songs and save songs */
std::vector<TrackEntry> trackEntries;
/* local <TrackEntry> for storing search results */
std::vector<TrackEntry> trackEntriesFound;
```

However, if there are no successful searches made by the user, a message box will popup indicating that there are no valid files found.

Searching for track in vector trackEntries, 'PlaylistComponent.cpp'

```
void PlaylistComponent::searchForTrack(juce::String searchInput)
{
    //reset trackFound
    trackFound = false;
    for(int i=0; i<trackEntries.size(); i++)
    {
        //check whether any trackEntries contains the string input by the user
        if(trackEntries[i].title.containsIgnoreCase(searchInput))
        {
            trackFound = true;
            //push back to a different class vector trackEntriesFound
        }
    }
}
```

```

        trackEntriesFound.push_back(trackEntries[i]);
    }
}

//if not found, pop up message box telling user
if(!trackFound)
    AlertWindow::showMessageBox(AlertWindow::WarningIcon, "Invalid search", "No tracks were found");
}

```

In the scenario where [bool trackFound = false], and that either no search input has been made, the default vector to be displayed by tableComponent will be trackEntries. However, in the scenario where [bool trackFound = true], meaning that a search input has been entered by the user, the vector displayed by the tableComponent will be set to trackEntriesFound.

The paint function takes reference from either trackEntries or trackEntriesFound. The rowNum parameter in paint takes reference from the getNumRows() function as well:

*Which vector to paint (*rowNumber from getNumRows())*

```

//TableListBoxModel override
void PlaylistComponent::paintCell(Graphics& g, int rowNum, int columnId, int width, int height, bool
rowsIsSelected)
{
    //condition for which <TrackEntry> to paint depending on whether user has made a search
    if(trackFound)
    {
        if(columnId == 1) //title
            g.drawText(trackEntriesFound[rowNum].title, 2, 0, width - 4, height, Justification::centredLeft, true);
        if(columnId == 2) //duration
            g.drawText(trackEntriesFound[rowNum].duration, 2, 0, width - 4, height,
Justification::horizontallyCentred, true);
    }
    else
    {
        if(columnId == 1) //title
            g.drawText(trackEntries[rowNum].title, 2, 0, width - 4, height, Justification::centredLeft, true);
        if(columnId == 2) //duration
            g.drawText(trackEntries[rowNum].duration, 2, 0, width - 4, height, Justification::horizontallyCentred,
true);
    }
}
}

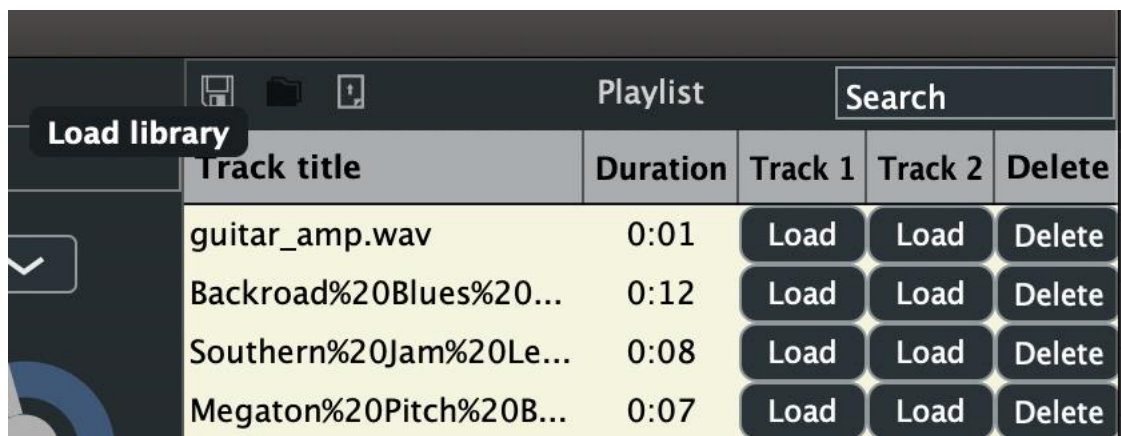
```

Which vector *numOfRows* to use for painting:

```
//TableListBoxModel override
int PlaylistComponent::getNumRows()
{
    //condition for which <TrackEntry> to display depending on whether user has made a search
    if(trackFound)
    {
        return int(trackEntriesFound.size());
    }
    else
    {
        return int(trackEntries.size());
    }
}
```

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

The user is able to load a file from the tableComponent into the deckGUI by clicking on the loadButton in tableComponent (which is in playlistComponent).



The screenshot shows a music application window. At the top, there are icons for file operations and a 'Search' input field. Below the search field is a table with the following columns: 'Track title', 'Duration', 'Track 1', 'Track 2', and 'Delete'. The table contains four rows of data. A 'Load library' button is visible on the left side of the table.

Track title	Duration	Track 1	Track 2	Delete
guitar_amp.wav	0:01	Load	Load	Delete
Backroad%20Blues%20...	0:12	Load	Load	Delete
Southern%20Jam%20Le...	0:08	Load	Load	Delete
Megaton%20Pitch%20B...	0:07	Load	Load	Delete

In both conditions of bool *trackFound* (true or false), the displayed rows of respective vectors have load buttons in them, and accommodates loading of the audio track into either *deckGUI1* or *deckGUI2*. In the button initialisation of *tableComponent*, a unique *buttonID* has been set to each button by

concatenating the row number and column index. This way, I am able to retrieve what button is clicked based on the row number and column index.

Creating the buttons in tableComponent's rows in 'PlaylistComponent.cpp'

```
//override
Component* PlaylistComponent::refreshComponentForCell(int rowNum, int colId, bool isRowSelected,
Component* existingComponentToUpdate)
{
    //colId 3 = load deckGUI1
    //colId 4 = load deckGUI2
    if(colId == 3 || colId == 4)
    {
        if(existingComponentToUpdate == nullptr)
        {
            //create textbutton pointer, textbutton is a child class of component, so able to store a text button onto a
            //component pointer
            TextButton* btn = new TextButton("Load");
            //unique id -> rowNum + colId
            String id{std::to_string(rowNum)+std::to_string(colId)};
            btn->setComponentID(id);
            //add itself as a button listener
            btn->addListener(this);
            existingComponentToUpdate = btn;
        }
    }
    //colId 5 = delete from vector
    if(colId == 5)
    {
        if(existingComponentToUpdate == nullptr)
        {
            //create deletebutton pointer, deletebutton is a child class of component, so able to store a text button onto
            //a component pointer
            TextButton* btn = new TextButton("Delete");
            //unique id -> rowNum + colId
            String id{std::to_string(rowNum)+std::to_string(colId)};
            btn->setComponentID(id);
            //add itself as a button listener
            btn->addListener(this);
        }
    }
}
```

```

        existingComponentToUpdate = btn;
    }
}
return existingComponentToUpdate;
}

```

If the bool trackFound is set to true, meaning that an input has been entered by the user and there is a valid search result, when the loadButton of that search result has been clicked, the URL of the file will be taken from trackEntriesFound vector. Similarly, if the bool trackFound is false, it will simply take the URL from the original trackEntries vector.

Hence, depending on whether a search input has been done, the tableComponent displays different data sets (TrackEntry vectors). But for both data sets, the function to load the songs displayed in tableComponent will be present. The logic for this can be found in *PlaylistComponent::buttonClicked()*, listening for any 'click' input made on the loadButton.

Logic to load track into the player in 'PlaylistComponent.cpp'

```

//component override
void PlaylistComponent::buttonClicked(Button* button)
{
    //...
    //for buttons inside of tableComponent rows
    else
    {
        //rowIndex to determine what track to retrieve from trackEntries
        int rowIndex = std::stoi(button->getComponentID().toString().substr(0,1));
        //colIndex to determine what deck to load track
        int colIndex = std::stoi(button->getComponentID().toString().substr(1,2));

        //local var to store track to play depending on tableComponent's rowIndex
        URL trackToPlay;

        //assign value to trackCounter for deckGUI to follow (when clicking on fwdButton)
        trackCounter = rowIndex;

        //if user search successful, tableComponent will display different details
        if(trackFound)
        {
            //get URL from trackEntriesFound
            trackToPlay = trackEntriesFound[rowIndex].audioURL;
        }
        else
    }
}

```



```

{
    //get URL from trackEntries
    trackToPlay = trackEntries[rowIndex].audioURL;
}

//load file into different deck depending on which col button is clicked
if(collIndex == 3) //deckGUI1
{
    //reset playButton
    deckGUI1->playButton.setToggleState(false, dontSendNotification);
    deckGUI1->player->loadURL(trackToPlay);
    deckGUI1->waveformDisplay.loadURL(trackToPlay);
    deckGUI1->waveformDisplay.setPositionRelative(deckGUI1->player->getPositionRelative());
    //changing track name displayed in deckGUI
    deckGUI1->trackName.setText(trackToPlay.getFileName(), dontSendNotification);
}
if(collIndex == 4) //deckGUI2
{
    //reset playButton
    deckGUI2->playButton.setToggleState(false, dontSendNotification);
    deckGUI2->player->loadURL(trackToPlay);
    deckGUI2->waveformDisplay.loadURL(trackToPlay);
    deckGUI2->waveformDisplay.setPositionRelative(deckGUI1->player->getPositionRelative());
    //changing track name displayed in deckGUI
    deckGUI2->trackName.setText(trackToPlay.getFileName(), dontSendNotification);
}
//delete from trackEntries vector AND update tableComponent
if(collIndex == 5)
{
    trackEntries.erase(trackEntries.begin()+rowIndex);
    tableComponent.updateContent();
}
}
}

```

**There is additional delete function that deletes the track from the trackEntries vector and updates the tableComponent to display the updated trackEntries vector.*

After loading the track into the deckGUI, the rest of the player functions can be accessed in deckGUI.

References

- JUCE, Class Index List, 2020, <https://docs.juce.com/master/index.html>
- High Pass IIR example, JUCE forum, May 2017, <https://forum.juce.com/t/high-pass-iir-example/22632>,
- SimpleIcon, 2018, <http://simpleicon.com>
- Remberg, juceCustomSliderSample, 10 Jan 2018, <https://github.com/remberg/juceCustomSliderSample>
- JUCE forum, [solved] setLooping(), 1 Nov 2018, <https://forum.juce.com/t/solved-setlooping/30586>
- WidgetsDemo JUCE examples, 2017, https://codesearch.isocpp.org/acted19/main/j/juce/juce_5.4.1+really5.4.1~repack-3/examples/GUI/WidgetsDemo.h