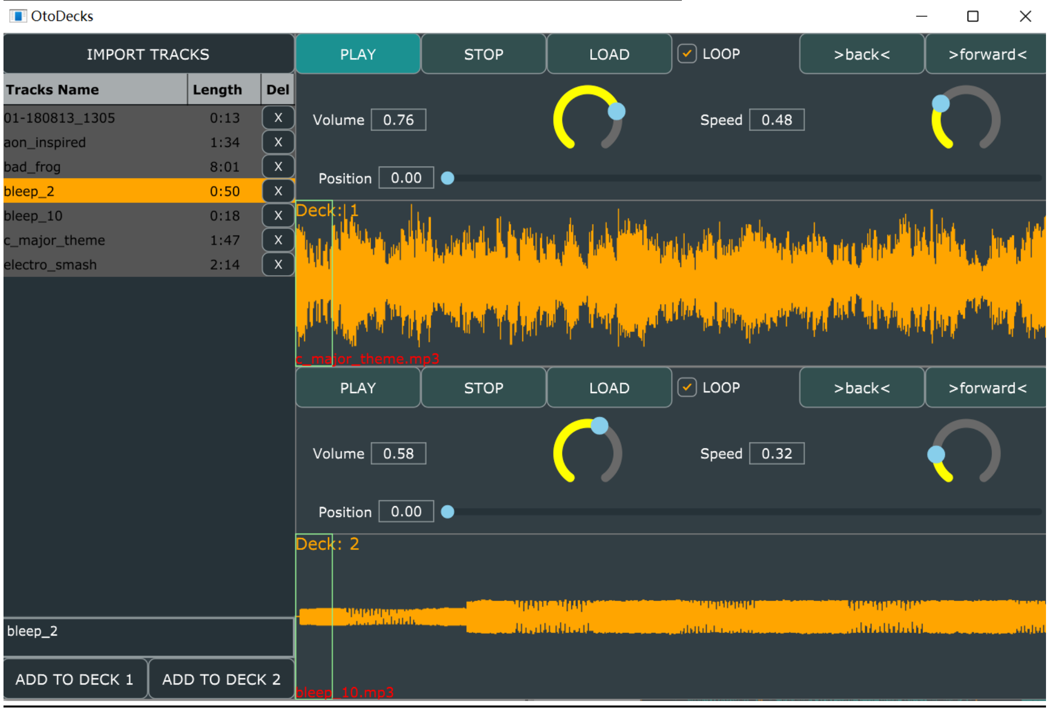
BSc Computer Science

CM2005 Object-Oriented Programming End-term Coursework Report

Candidate No. KJ2011

Mar 2022

A Screenshot of my OtoDecks DJ Application



Tracks Name	Length	Del
01-180813_1305	0:13	X
aon_inspired	1:34	X
bad_frog	8:01	X
bleep_2	0:50	X
bleep_10	0:18	X
c_major_theme	1:47	X
electro_smash	2:14	X

Requirements

Requirements	Achievement	Requirement	Achievement
R1A	YES	R3B	YES
R1B	YES	R3C	YES
R1C	YES	R3D	YES
R1D	YES	R3E	YES
R2A	YES	R4A	YES
R2B	YES	R4B	YES
R3A	YES	R4C	YES

Details of Requirements Achievement

- R1A: can load audio files into audio players

  The user has two options to load those music tracks into the audio player. The first is to click on the "LOAD" button on one of the players, the other is that he or she can choose to import the tracks into the playlist library first by clicking the "Import Tracks" button in the upper left corner, then select a track in the library and then press the "Add to Deck" button in the lower left corner to add the track to some of the chosen decks.

If a track has been loaded to a player, when user loaded another file in it, the previous one will be covered up automatically.

Figure 1: load button on the one of the players

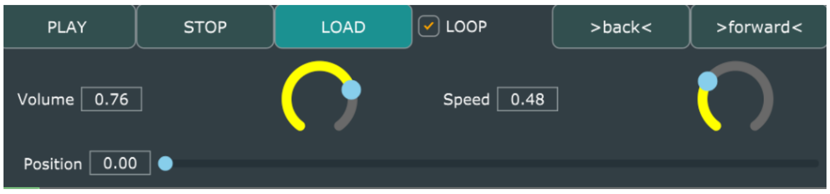
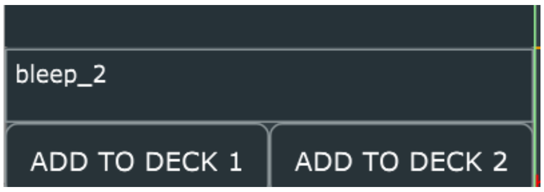
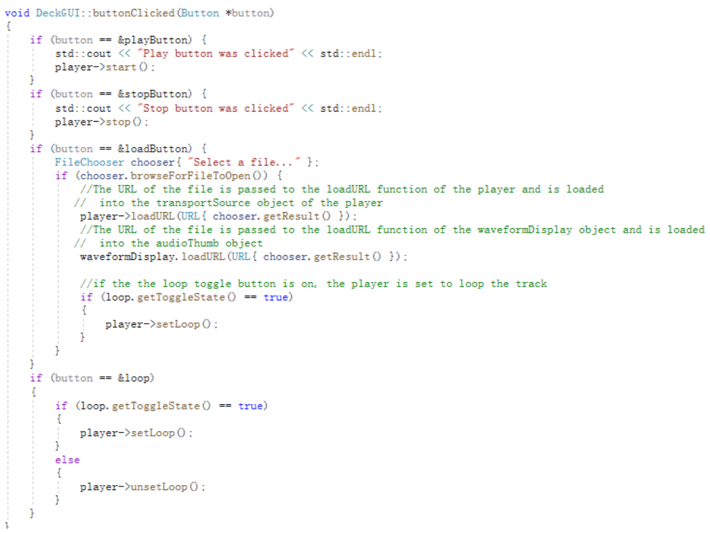


Figure 2: load button on the playlist component



For the first method, code can be found in the function called 'DeckGUI::buttonClicked()' in DeckGUI.cpp file. This feature implements that when the Load button is clicked, a file picker object is created and a window pops up that allows the user to select a trace file from the computer to load.

And the 'DJAudioPlayer::loadURL()' function passes the URL as an argument in the 'DJAudioPlayer.cpp' file to 'formatManager' to extract the format that will be read as an audio file. If the reader is not a null pointer, it will then load the file data to the 'transportSource' to play.



```
void DeckGUI::buttonClicked(Button *button)
{
    if (button == &playButton) {
        std::cout << "Play button was clicked" << std::endl;
        player->start();
    }
    if (button == &stopButton) {
        std::cout << "Stop button was clicked" << std::endl;
        player->stop();
    }
    if (button == &loadButton) {
        FileChooser chooser{ "Select a file..." };
        if (chooser.browseForFileToOpen()) {
            //The URL of the file is passed to the loadURL function of the player and is loaded
           //  into the transportSource object of the player
            player->loadURL(URL{ chooser.getResult() });
            //The URL of the file is passed to the loadURL function of the waveformDisplay object and is loaded
           //  into the audioThumb object
            waveformDisplay.loadURL(URL{ chooser.getResult() });

            //if the the loop toggle button is on, the player is set to loop the track
            if (loop.getToggleState() == true)
            {
                player->setLoop();
            }
        }
    }
    if (button == &loop)
    {
        if (loop.getToggleState() == true)
        {
            player->setLoop();
        }
        else
        {
            player->unsetLoop();
        }
    }
}
```

Figure 3:DeckGUI::buttonClicked() implementation.

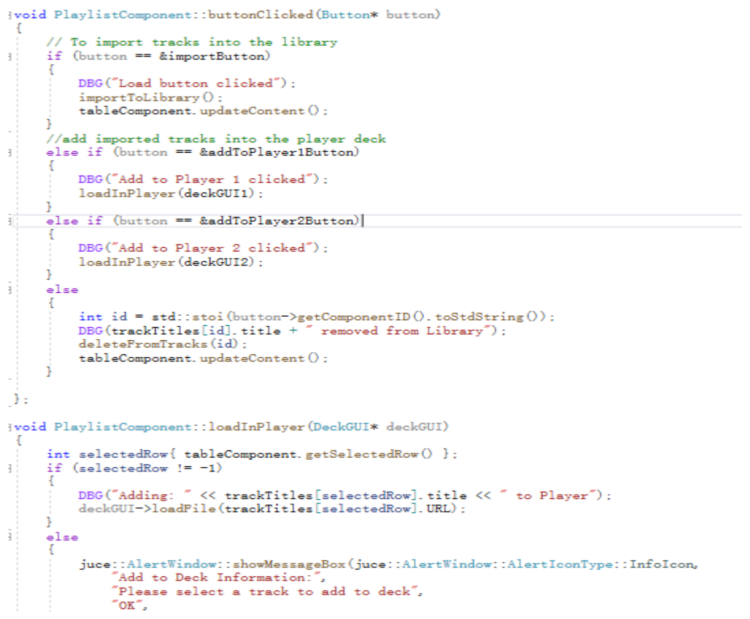
The second method is to first import the tracks into the playlist library and then load them into a certain card group by clicking the relevant button. The process code that loads the tracks into the player can be found in the

‘DJAudioPlayer.cpp’ file, locate the deck by calling the ‘loadInPlayer’ function and link the file to the ‘DeckGUI.cpp’ file.

In this way, the library will be able to link with the player’s deck in order to load the selected files into the player.

Besides, track is also passed to the ‘WaveformDisplay’ object as its title will be passed to relevant trackTitle object. When the object get the information of the loaded track, It will automatically draw its wave by the sound altitude and length. The latest loaded file will cover up the previous one in the player. And the loaded track’ s name will also be shown at the bottom of the waveform display deck.

Figure4: PlaylistComponent::buttonClicked() function implemented



```cpp
void PlaylistComponent::buttonClicked(Button* button)
{
    // To import tracks into the library
    if (button == &importButton)
    {
        DBG("Load button clicked");
        importToLibrary();
        tableComponent.updateContent();
    }
    //add imported tracks into the player deck
    else if (button == &addToPlayer1Button)
    {
        DBG("Add to Player 1 clicked");
        loadInPlayer(deckGUI1);
    }
    else if (button == &addToPlayer2Button)
    {
        DBG("Add to Player 2 clicked");
        loadInPlayer(deckGUI2);
    }
    else
    {
        int id = std::stoi(button->getComponentID().toStdString());
        DBG(trackTitles[id].title + " removed from Library");
        deleteFromTracks(id);
        tableComponent.updateContent();
    }

};

void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow{ tableComponent.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Adding: " << trackTitles[selectedRow].title << " to Player");
        deckGUI->loadFile(trackTitles[selectedRow].URL);
    }
    else
    {
        juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
            "Add to Deck Information:",
            "Please select a track to add to deck",
            "OK",
```

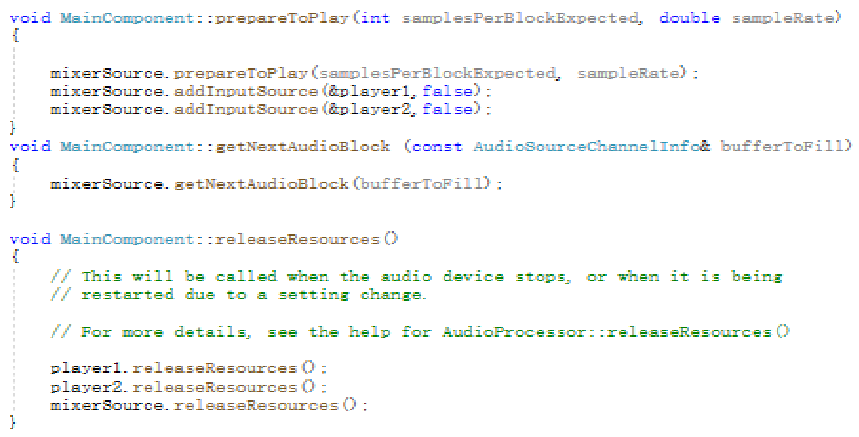
- <u>R1B: Can play two or more tracks</u>

  This function was achieved by using the ‘MixerAudioSource’ object, which called ‘mixerSource’ , makes the user able to play two tracks at the same time but play independently in each player. It can be found in the MainComponent.h file. Here is a screenshot of pieces of its code.

  Next, in order to play those tracks, we need to call the ‘prepareToPlay()’ function of ‘mixerSource’ inside the ‘MainComponent’ .

  Then we need to do the same operation on ‘getNextAudioBlock()’ function and ‘releaseResources()’ function so that the latest add file can cover up the previous one and play. If one would want to add more players, one just needs to add the new player as an input source to the ‘mixerSource’ .

Figure 5: function called in the MainComponent.cpp file



```
void MainComponent::prepareToPlay(int samplesPerBlockExpected, double sampleRate)
{

    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
}
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    mixerSource.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.

    // For more details, see the help for AudioProcessor::releaseResources()

    player1.releaseResources();
    player2.releaseResources();
    mixerSource.releaseResources();
}
```

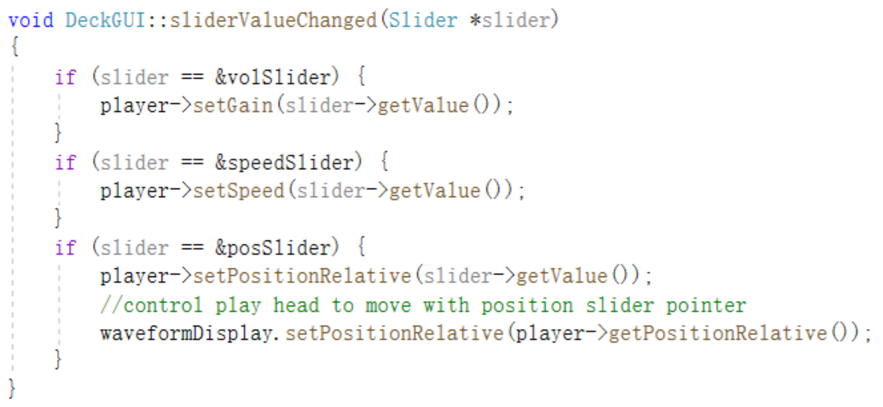
- <u>R1C: can mix the tracks by varying each of their volumes</u>

Each deck is independent from the other and contains a volume control slider. By implementing the ‘DeckGUI::sliderValueChanged()’ function inside the DeckGUI.cpp file, the ‘DeckGUI’ component listens for changes whenever the value of volume slider is changed.

Whenever the slider is moved, the ‘DJAudioPlayer::setGain()’ function of the respective player is called as value will be updated to the deck.

In this way, the volume of each player can change. Users can change the volume slider value by dragging their pointer or by entering a number in the preceding text field. Because the maximum and minimum values are already set from 0 to 1, If the use input value smaller or greater than the bound, the input value will not exceed or fall below it.

Figure 6: The DeckGUI::sliderValueChanged() function implemented inside the DeckGUI.cpp file



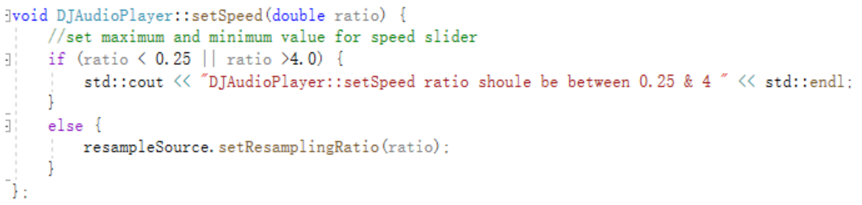
```
void DeckGUI::sliderValueChanged(Slider *slider)
{
    if (slider == &volSlider) {
        player->setGain(slider->getValue());
    }
    if (slider == &speedSlider) {
        player->setSpeed(slider->getValue());
    }
    if (slider == &posSlider) {
        player->setPositionRelative(slider->getValue());
        //control play head to move with position slider pointer
        waveformDisplay.setPositionRelative(player->getPositionRelative());
    }
}
```

- <u>R1D: can speed up and slow down the tracks</u>

  Just like the volume controller, the speed control slider is also set in ‘DeckGUI’ component. Whenever the slider value is changed, the ‘DJAudioPlayer::setSpeed()’ function gets called and we use a ‘ResamplingAudioSource’ object to change the resampling ratio to control the playing speed of tracks.

  Users can change the speed slider value by dragging their pointer or by entering a number in the preceding text field. Because the maximum and minimum values are already set from 0.25 to 4. If the use input value smaller or greater than the bound, the value will not exceed or fall below it.

Figure 7: The DJAudioPlayer::setSpeed() function implemented inside the DJAudioPlayer.cpp file



```
void DJAudioPlayer::setSpeed(double ratio) {
    //set maximum and minimum value for speed slider
    if (ratio < 0.25 || ratio >4.0) {
        std::cout << "DJAudioPlayer::setSpeed ratio shoule be between 0.25 & 4 " << std::endl;
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
};
```

- <u>R2A: Component has custom graphics implemented in a paint function</u>

  Specific interface can be shown from the screenshot of my application on the first page.

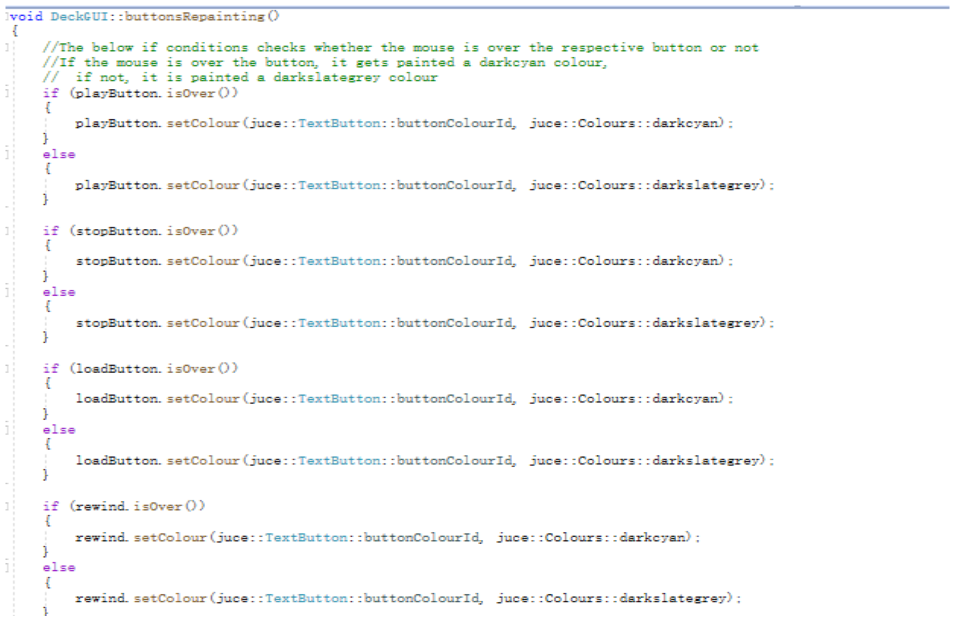
  Although the main background color is gray, each component has its own different decision of colors sets. For example, whenever the mouse hovers over a button, the corresponding button changes color to alert the user that it can be chosen.

  In addition, when the user continues to press the forward or back buttons, it also changes its color to remind the user that you are in control. If you release the buttons, the operations associated with them stop and the color of the buttons changes back to their original color.

  These control code you can find in the 'DeckGUI::buttonRepainting() ' function in 'DeckGUI.cpp' file, here is it's screenshot.

  In addition, the slider has been adapted into a rotary one, which saves more space in the interface. Its sliding trajectory and pointer color changes to be brighter, increasing the contrast of the color and making it easier to be seen

  Figure 8: Piece code of screenshot for the 'DeckGUI::buttonsRepainting()' function implemented inside the DeckGUI.cpp file



```
void DeckGUI::buttonsRepainting()
{
    //The below if conditions checks whether the mouse is over the respective button or not
    //If the mouse is over the button, it gets painted a darkcyan colour,
    //  if not, it is painted a darkslategrey colour
    if (playButton.isOver())
    {
        playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkcyan);
    }
    else
    {
        playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkslategrey);
    }

    if (stopButton.isOver())
    {
        stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkcyan);
    }
    else
    {
        stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkslategrey);
    }

    if (loadButton.isOver())
    {
        loadButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkcyan);
    }
    else
    {
        loadButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkslategrey);
    }

    if (rewind.isOver())
    {
        rewind.setColour(juce::TextButton::buttonColourId, juce::Colours::darkcyan);
    }
    else
    {
        rewind.setColour(juce::TextButton::buttonColourId, juce::Colours::darkslategrey);
    }
```

- <u>R2B: Component enables the user to control the playback of a deck somehow</u>

  There are two ways to achieve this functionality. Instead, do this by adjusting the value of the position slider or by clicking the forward/back buttons

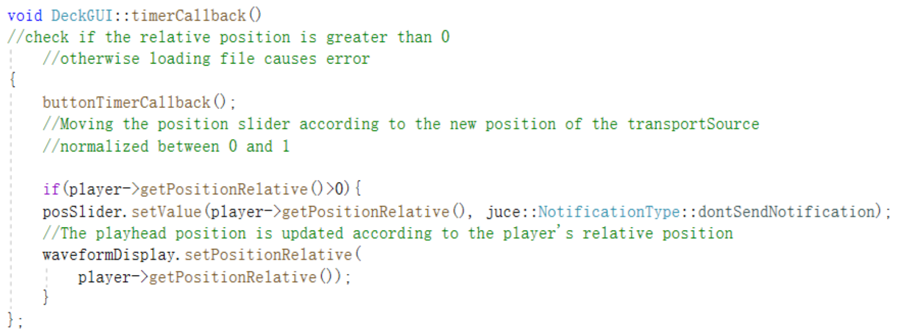
  For the first method, it's like the volume and speed slider controllers that call functions in the waveformDisplay.cpp file. When the slider value changes, it updates to the feature and changes the music playback progress position relatively. Both position slider pointer and play-head in waveform display deck will change together, and the value of track length in the front of the slider will also change correspondingly.

  Another way is through button control. When the user continues to press the forward/back buttons, the play-head moves forward or backward at a constant rate. Here, I use the Timer class to check the button status so that it can go forward/backward in a constant rate. If you release the button, it changes back to the default color and stops the operation. Then the music will then continue to play at its original speed, starting from where it is now released from operation.

  Furthermore, ticking the 'loop' checkbox will enable the loop function, which allows the track to replay from the very beginning when it has finished. Without ticking this option, the music will stop playing when it reaches the end.

  Regardless of the method, the pointer slider on the play-head and position will always change its value and position with the progress of the play.

  Figure 9: function to check button status

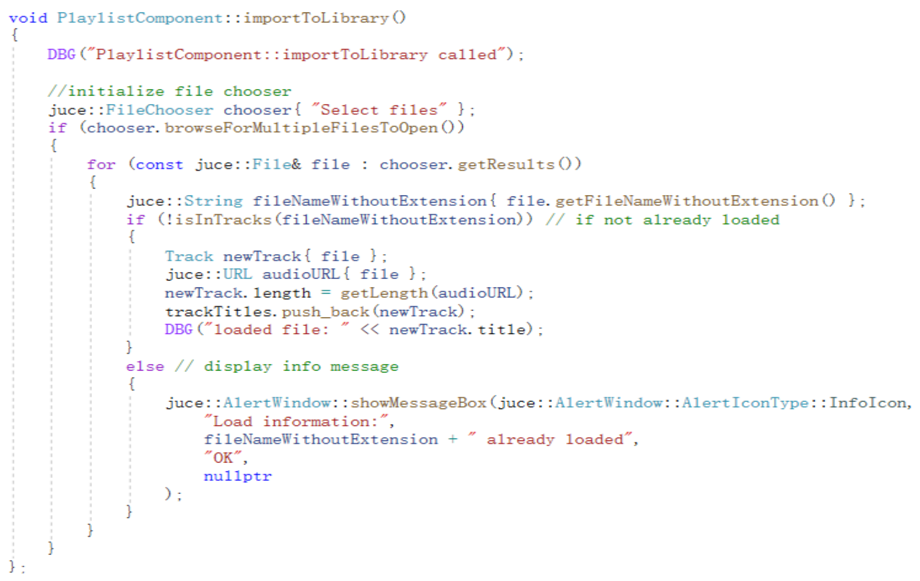
```
void DeckGUI::timerCallback()
//check if the relative position is greater than 0
    //otherwise loading file causes error
{
    buttonTimerCallback();
    //Moving the position slider according to the new position of the transportSource
    //normalized between 0 and 1

    if(player->getPositionRelative()>0){
    posSlider.setValue(player->getPositionRelative(), juce::NotificationType::dontSendNotification);
    //The playhead position is updated according to the player's relative position
    waveformDisplay.setPositionRelative(
        player->getPositionRelative());
    }
};
```

- <u>R3A: Component allows the user to add files to their library</u>

  This is implemented by using text buttons in the PlaylistComponent.cpp file. Whenever the 'Import Tracks button' is clicked, just like the 'Load' Button function, a window pops up that lets the user find the tracks and import them into the playlist library.

  This is done by calling the 'PlaylistComponent::importToLibrary()' function, which is located in the 'PlaylistComponent.cpp' file. If the file has already been added, it will not be added again, and a message will pop up to remind the user.



```
void PlaylistComponent::importToLibrary()
{
    DBG("PlaylistComponent::importToLibrary called");

    //initialize file chooser
    juce::FileChooser chooser{ "Select files" };
    if (chooser.browseForMultipleFilesToOpen())
    {
        for (const juce::File& file : chooser.getResults())
        {
            juce::String fileNameWithoutExtension{ file.getFileNameWithoutExtension() };
            if (!isInTracks(fileNameWithoutExtension)) // if not already loaded
            {
                Track newTrack{ file };
                juce::URL audioURL{ file };
                newTrack.length = getLength(audioURL);
                trackTitles.push_back(newTrack);
                DBG("loaded file: " << newTrack.title);
            }
            else // display info message
            {
                juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
                    "Load information:",
                    fileNameWithoutExtension + " already loaded",
                    "OK",
                    nullptr
                );
            }
        }
    }
};
```

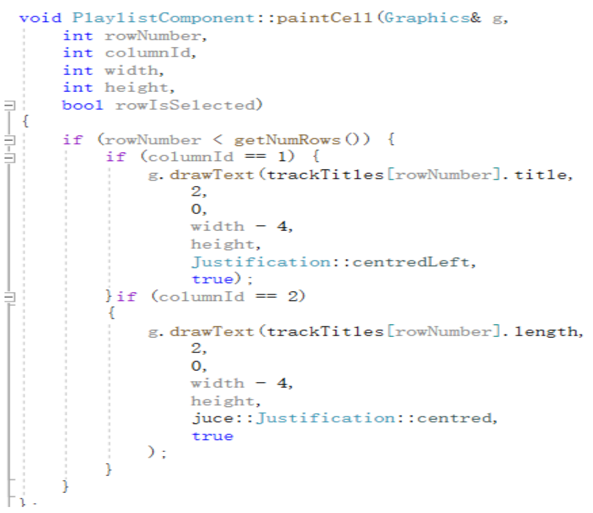
Figure 10: The implementation of PlaylistComponent::importToLibrary().

- **<u>R3B: Component parses and displays meta data such as filename and song length</u>**

For this point, I used a temporary 'DJAudioPlayer' object called 'PlayerParsingforMetaData' so that it could operate on the file to extract information.

For each track found in the Track vector, after import, the file name and its length appear in a separate column in the playlist. This can be done by code in the 'PlaylistComponent::paintCell()' function in the PlaylistComponet.cpp file. By adding an ID to each imported track, we can extract its position and load it into the pointer player.

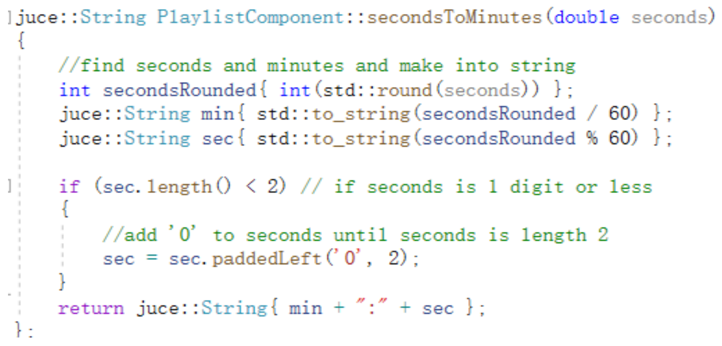
In addition, I implemented a custom function that can be calculated in minutes and seconds and returned as a string, showing the length of the track in a more readable way.



```
void PlaylistComponent::paintCell(Graphics& g,
    int rowNumber,
    int columnId,
    int width,
    int height,
    bool rowIsSelected)
{
    if (rowNumber < getNumRows()) {
        if (columnId == 1) {
            g.drawText(trackTitles[rowNumber].title,
                2,
                0,
                width - 4,
                height,
                Justification::centredLeft,
                true);
        }if (columnId == 2)
        {
            g.drawText(trackTitles[rowNumber].length,
                2,
                0,
                width - 4,
                height,
                juce::Justification::centred,
                true
            );
        }
    }
};
```

Figure 11: PlaylistComponent::paintCell() function implemented in PlaylistComponent.cpp file

Figure 12: calculate function for track length.



```cpp
juce::String PlaylistComponent::secondsToMinutes(double seconds)
{
    //find seconds and minutes and make into string
    int secondsRounded{ int(std::round(seconds)) };
    juce::String min{ std::to_string(secondsRounded / 60) };
    juce::String sec{ std::to_string(secondsRounded % 60) };

    if (sec.length() < 2) // if seconds is 1 digit or less
    {
        //add '0' to seconds until seconds is length 2
        sec = sec.paddedLeft('0', 2);
    }
    return juce::String{ min + ":" + sec };
};
```

- <u>R3C: Component allows the user to search for files</u>

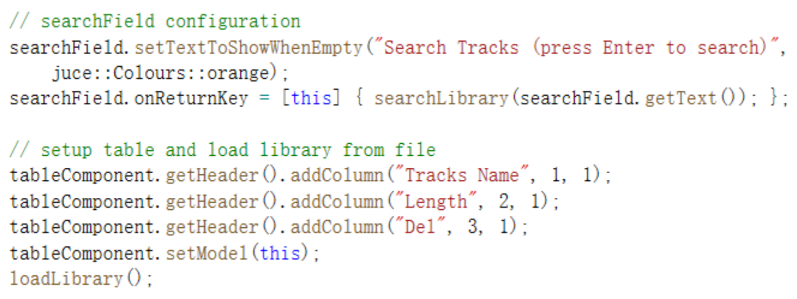
  I used the 'JUCE Label' class to achieve this point. I created 2 labels, one is called 'tableComponent' which is for contain and locate the imported track file separately in the playlist library and extract its information. Another one is called 'searchField' which allows text input and compare the input keyword with imported file name.

  Whenever the user enters a keyword and presses the Enter button on the keyboard, the function 'PlaylistComponent::searchLibrary()' is called, the implementation of which can be found in the 'PlaylistComponent.cpp' file.

  Search field will then compare the input keywords to find an exact match, and if it returns true, the relative column will turn orange as it will be selected.

  Otherwise, if there are no matching keywords, nothing will happen and the user may want to remove those keywords in the search field.

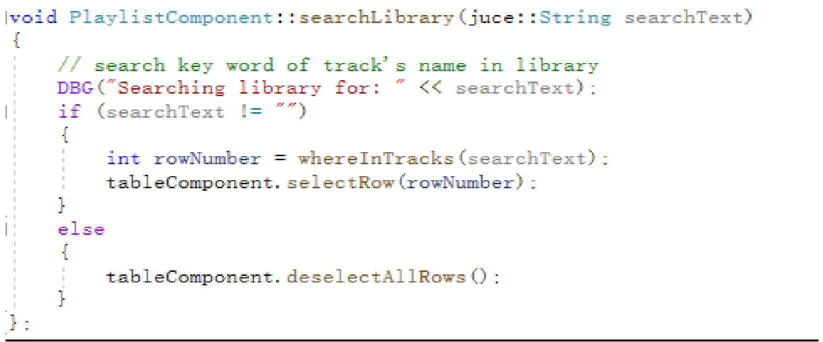
Figure 13: searchField configuration in the PlaylistComponent.cpp file



```cpp
// searchField configuration
searchField.setTextToShowWhenEmpty("Search Tracks (press Enter to search)",
    juce::Colours::orange);
searchField.onReturnKey = [this] { searchLibrary(searchField.getText()); };

// setup table and load library from file
tableComponent.getHeader().addColumn("Tracks Name", 1, 1);
tableComponent.getHeader().addColumn("Length", 2, 1);
tableComponent.getHeader().addColumn("Del", 3, 1);
tableComponent.setModel(this);
loadLibrary();
```

Figure 14: implementation of PlaylistComponent::searchLibrary() function in PlaylistComponent.cpp file



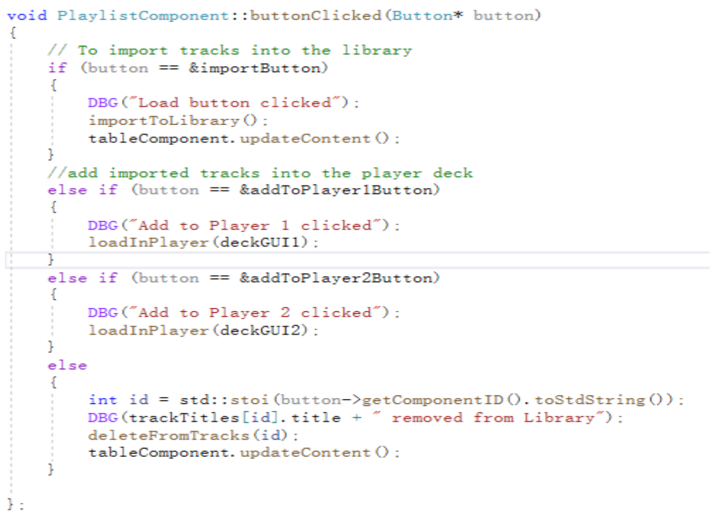
```
void PlaylistComponent::searchLibrary(juce::String searchText)
{
    // search key word of track's name in library
    DBG("Searching library for: " << searchText);
    if (searchText != "")
    {
        int rowNumber = whereInTracks(searchText);
        tableComponent.selectRow(rowNumber);
    }
    else
    {
        tableComponent.deselectAllRows();
    }
};
```

- **<u>R3D: Component allows the user to load files from the library into a deck</u>**

  This is very similar to the function of the load button in the 'deckGUI.cpp' file. However, the difference is that after importing the tracks, we need to choose which one should be loaded into the player.

  So, first we create a 'TableListBox' object to put in all added tracks separately and put these box in order with a ID from top to the bottom in the library, then we call the 'PlaylistComponent::loadInPlayer()' function in the PlaylistComponent.cpp file.

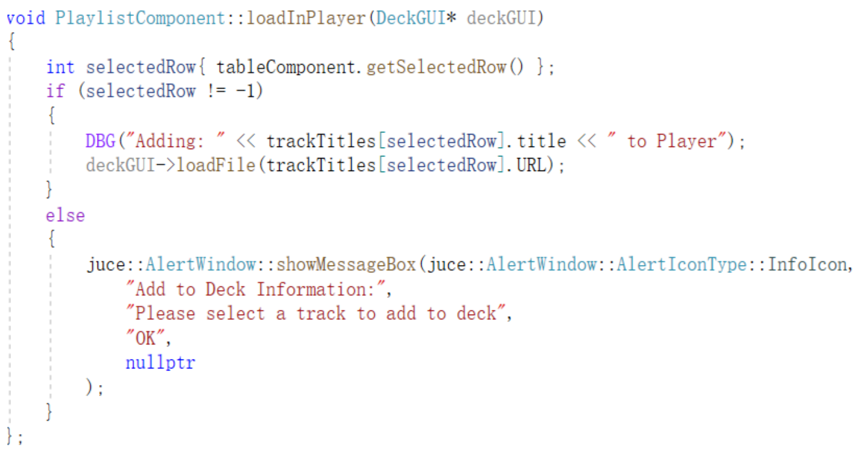
  With the "getSelectedRow" method, we can select and position each track position in the playlist. When you click the Relative button, the selected track is added to the deck accordingly.



```
void PlaylistComponent::buttonClicked(Button* button)
{
    // To import tracks into the library
    if (button == &importButton)
    {
        DBG("Load button clicked");
        importToLibrary();
        tableComponent.updateContent();
    }
    //add imported tracks into the player deck
    else if (button == &addToPlayer1Button)
    {
        DBG("Add to Player 1 clicked");
        loadInPlayer(deckGUI1);
    }
    else if (button == &addToPlayer2Button)
    {
        DBG("Add to Player 2 clicked");
        loadInPlayer(deckGUI2);
    }
    else
    {
        int id = std::stoi(button->getComponentID().toStdString());
        DBG(trackTitles[id].title + " removed from Library");
        deleteFromTracks(id);
        tableComponent.updateContent();
    }
};
```

Figure 15: function of buttonClicked() to add tracks into deck

Figure 16: implementation of PlaylistComponent::loadInPlayer() function.

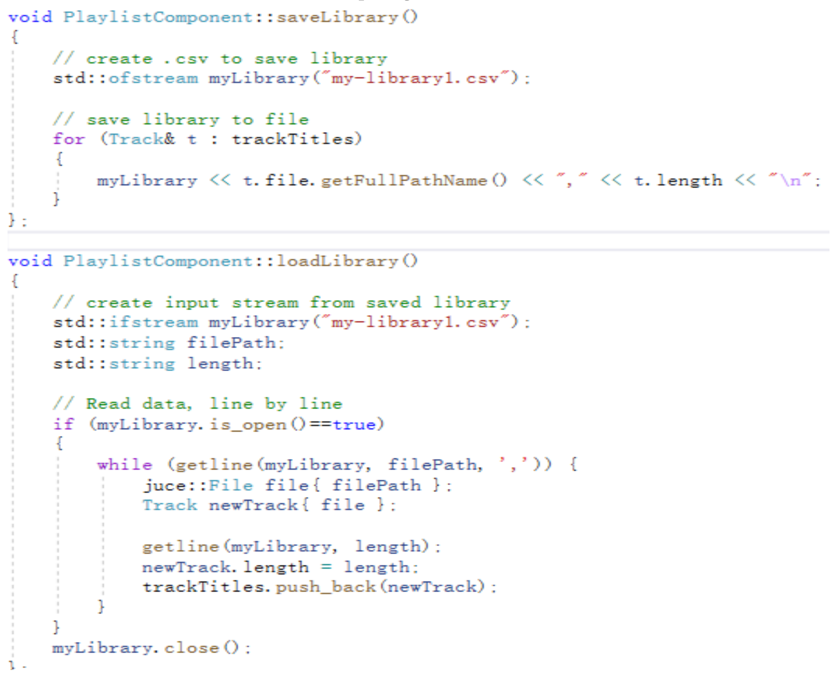


```
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    int selectedRow{ tableComponent.getSelectedRow() };
    if (selectedRow != -1)
    {
        DBG("Adding: " << trackTitles[selectedRow].title << " to Player");
        deckGUI->loadFile(trackTitles[selectedRow].URL);
    }
    else
    {
        juce::AlertWindow::showMessageBox(juce::AlertWindow::AlertIconType::InfoIcon,
            "Add to Deck Information:",
            "Please select a track to add to deck",
            "OK",
            nullptr
        );
    }
};
```

- <u>R3E: The music library persists so that it is restored when the user exits then restarts the application</u>

  To solve this problem, I used a csv file, while the program was running, and created an "ifstream" named "myLibrary", the program will check for the existence of the path passed when declaring, and if it does not exist, it will be created. If it already exists, it means that the songs that were previously added to the playlist are stored on disk and nothing will happen.

  All these are done by the 'PlaylistComponent::loadLibrary()' function call in the constructor of the 'PlaylistComponent'. When the program is closed, a function called 'PlaylistComponent::saveLibrary()' is called in the class destructor. This function will search the full path name and length of the file when next time the program runs.



```
void PlaylistComponent::saveLibrary()
{
    // create .csv to save library
    std::ofstream myLibrary("my-library1.csv");

    // save library to file
    for (Track& t : trackTitles)
    {
        myLibrary << t.file.getFullPathName() << "," << t.length << "\n";
    }
};

void PlaylistComponent::loadLibrary()
{
    // create input stream from saved library
    std::ifstream myLibrary("my-library1.csv");
    std::string filePath;
    std::string length;

    // Read data, line by line
    if (myLibrary.is_open()==true)
    {
        while (getline(myLibrary, filePath, ',')) {
            juce::File file{ filePath };
            Track newTrack{ file };

            getline(myLibrary, length);
            newTrack.length = length;
            trackTitles.push_back(newTrack);
        }
    }
    myLibrary.close();
};
```

Figure 17: The library relative functions implemented in the PlaylistComponent.cpp file

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

R4B: GUI layout includes the custom Component from R2

R4C: GUI layout includes the music library component from R3

From the screenshot of the application interface on the first page of this document, you can see that the layout is different because it has been rearranged into the left and right main section modes, with more buttons and interface elements added to control.

For the ‘DeckGUI’ at right, forward button, backward button and loop tick-box were added, and the style of volume and speed slider was changed into rotary with a brighter color pointer.

The titles of these played tracks are also added to the lower-left corner of the waveform to show the loaded tracks. Many decks are placed in the upper left corner so that the user knows which player they are controlling.

For the playlist library on the left, the control buttons, search bar, and track list appear as a whole set. Its interface is designed to meet the needs of aesthetic applications.

Improvements

I think some improvements to the program may be added in the future. For example, the file location of a track in your computer may appear in a playlist.

There are also more features like damping or wetting that can be added to the deck so that users can manipulate these music files more creatively and make this DJ app more fun. I would love to further enhance this app in the future. Thank you.