Coursework Report for Endterm: Otodecks

R1: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop / start.

R1A: Component has custom graphics implemented in a paint function

DeckGUI.h
line 36: function declaration
DeckGUI.cpp
line 69: paint ( )
PlaylistComponent.h
line 55: function declaration
PlaylistComponent.cpp
line 111: paint ( )

Achieved by setting the customised color and styles for buttons, sliders, labels, and a text (search) box. I used setColour( ) function to change colors, setBounds( ) function to set location, setFont( ) and setText( ) functions to set text style, setSliderStyle( ) function to change the slider style, and setTextToShowWhenEmpty( ) function to show “Search...” text in the search box.
For example, the following code set the color of a play button to light pink:

```
// set color for buttons
playButton.setColour(juce::TextButton::buttonColourId, juce::Colour(255, 219, 255));
```

The following code set the text style and color for a volume label:

```
// set style for volume label
volLabel.setFont(juce::Font(15.0f, juce::Font::bold));
volLabel.setText("Volume", juce::dontSendNotification);
volLabel.setColour(juce::Label::textColourId, juce::Colours::lightyellow);
```

The following code set the slider style to the one with a vertival linear bar:

```
// set style for sliders
volSlider.setSliderStyle(juce::Slider::SliderStyle::LinearBarVertical);
```

R1B: Component enables the user to control the playback of a deck somehow

DeckGUI.h
line 44, 52: functions declaration
DeckGUI.cpp
line 126: buttonClicked( )
line 143: sliderValueChanged( )

In DeckGUI class, This is achieved by adding a play button, a stop button, a volume slider, a speed slider, a position slider, and a player declaring them as private instances in header file as well as implementing buttonClicked() function and sliderValueChanged() function in cpp file.

```
private:
    juce::TextButton playButton{ "PLAY" }
    juce::TextButton stopButton{ "STOP" }

    DJAudioPlayer* player;

    juce::Slider volSlider;
    juce::Slider speedSlider;
    juce::Slider posSlider;
```

In buttonClicked( ) function, if the button is a play button,
the player initializes the transportSource and calls the
start( ) function to start the music.

If the button is a stop button, the player calls stop( ) function to stop the music.

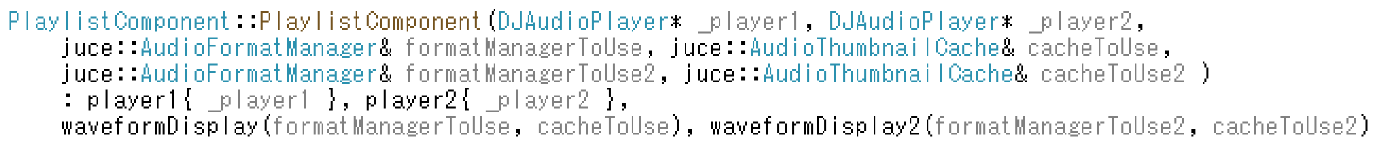
In sliderValueChanged( ) function, it checks the type of slider and depending on the slider type, the player instance calls setPositionRelative( ) function for a position slider, setSpeed( ) function for a speed slider, and setGain ( ) function for a volume slider.

PlaylistComponent.h
- line 31: constructor declaration
- line 64, 72: functions declaration

PlaylistComponent.cpp
- line 16: constructor initialization
- line 140: loadFromLibrary( )
- line 158: timerCallback( )

In PlaylistComponent class, this is achieved by adding waveforms declaring them as private instances in header file as well as integrating them into the PlaylistComponent class. This way, the user can control the playback by triggering a waveform while clicking on a LOAD1 / LOAD2 button.

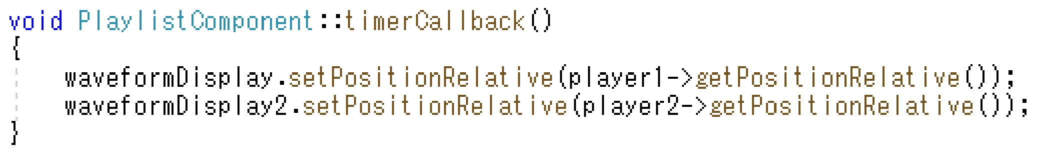
In constructor, it requires two sets of DJAudioPlayer, AudioFormatManager, and AudioThumbnailCache so two players and two waveforms can connect to the playback control. The following code is the constructor initializes players and waveforms with the parameters passed from MainComponent class.



```
PlaylistComponent::PlaylistComponent(DJAudioPlayer* _player1, DJAudioPlayer* _player2,
    juce::AudioFormatManager& formatManagerToUse, juce::AudioThumbnailCache& cacheToUse,
    juce::AudioFormatManager& formatManagerToUse2, juce::AudioThumbnailCache& cacheToUse2 )
    : player1{ _player1 }, player2{ _player2 },
    waveformDisplay(formatManagerToUse, cacheToUse), waveformDisplay2(formatManagerToUse2, cacheToUse2)
```

loadFromLibrary( ) function loads a file for a player and for a waveform so it can trigger the timerCallback() function below for the waveform.

In timerCallback( ) function, the relative positions are set for waveforms so the play head can move correctly in a real time as music is played.



```
void PlaylistComponent::timerCallback()
{
    waveformDisplay.setPositionRelative(player1->getPositionRelative());
    waveformDisplay2.setPositionRelative(player2->getPositionRelative());
}
```

R2: Implementation of a music library component which allows the user to manage their music library

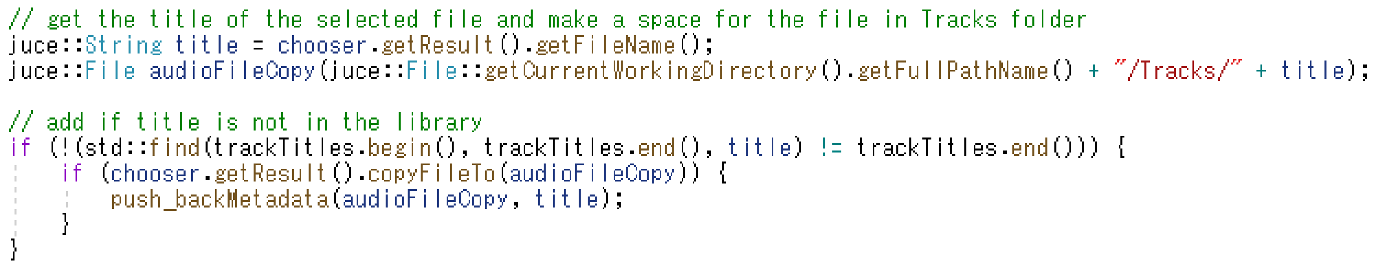
R2A: Component allows the user to add files to their library

PlaylistComponent.h
  line 81, 87: functions declaration
PlaylistComponent.cpp
  line 171: buttonClicked( )
  line 189: addFileToLibrary( )

Achieved by using browseForFileToOpen( ) function in FileChooser class to choose a file and by using functions from File class to manage the “Tracks” local directory and files.

In buttonClicked( ) function, call addFileToLibrary( ) function if the type of the clicked button is determined as the ADD button.

In addFileToLibrary ( ) function, firstly call resetAll( ) function and restoreLibrary( ) function to reset the vectors and library because the vector value for track information maybe changed by search operation. Then, use FileChooser class to select a file. If browserForFileToOpen( ) function returns TRUE, it checks whether a Tracks folder exists and create it if it doesn’t exist using File class. It then create a space for the file in Tracks folder, goes through the trackTitles vector, and copy the selected file into the space if the trackTitles doesn’t contain the title of the file. The following code does that process:



```
// get the title of the selected file and make a space for the file in Tracks folder
juce::String title = chooser.getResult().getFileName();
juce::File audioFileCopy(juce::File::getCurrentWorkingDirectory().getFullPathName() + "/Tracks/" + title);

// add if title is not in the library
if (!(std::find(trackTitles.begin(), trackTitles.end(), title) != trackTitles.end())) {
    if (chooser.getResult().copyFileTo(audioFileCopy)) {
        push_backMetadata(audioFileCopy, title);
    }
}
```

After the operation, call updateContent( ) function and repaint( ) function from TableListBox class to update the table component of the library.

R2B: Component parses and displays meta data such as filename and song length

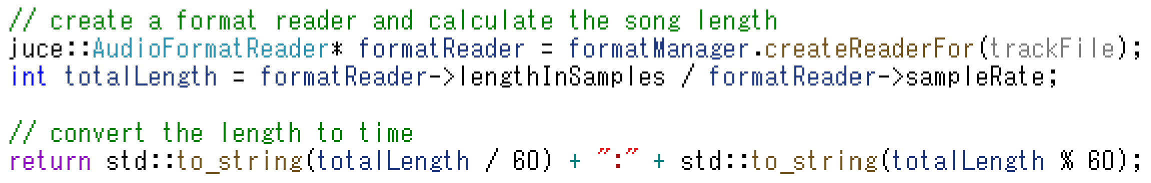
PlaylistComponent.h
  line 101, 114, 123: functions declaration
PlaylistComponent.cpp
  line 235: paintCell( )
  line 260: push_backMetadata( )
  line 274: getTrackLength( )

Achieved by accessing trackTitles vector and trackLengths vector drawing the text extracted from the vectors.

In paintCell( ) function, if column ID is equal to 0, access trackTitles vector to display the track (file) titles. Else if column ID is equal to 3, access trackLengths vector to display the track (file) lengths.

In push_backMetadata( ) function, update all the related vectors, trackFiles, trackTitles, and trackLengths. This is called whenever there is any update in other functions. It receives a file and its title as parameters and they are parsed in advance in other function. For instance, in the above screenshot of code from addFileToLibrary( ) function in R2A, the function parses the title and file data from the selected file.

In getTrackLength( ) function, create an AudioFormatManager instance to create a format reader for the track file. To calculate the track length, access lengthInSamples variable in the format reader and devide it by sampleRate to get the total length in seconds. Convert the total length to time in string and return. The following code describes the calculation and the conversion processes:



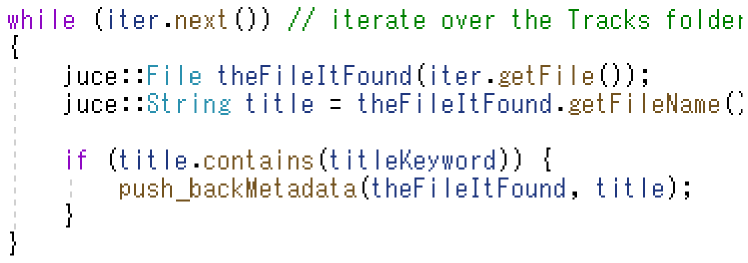
```
// create a format reader and calculate the song length
juce::AudioFormatReader* formatReader = formatManager.createReaderFor(trackFile);
int totalLength = formatReader->lengthInSamples / formatReader->sampleRate;

// convert the length to time
return std::to_string(totalLength / 60) + ":" + std::to_string(totalLength % 60);
```

R2C: Component allows the user to search for files

PlaylistComponent.h
   line 132: function declaration
PlaylistComponent.cpp
   line 294: textEditorReturnKeyPressed( )

Achieved by implementating textEditorReturnKeyPressed( ) function which is the pure virtual function of TextEditor class to display the matched files in the library when return key is pressed after user inputs the keyword.

In textEditorReturnKeyPressed( ) function, firstly empty all the titles and files in vectors with resetAll( ) helper function to reconstruct the new list from the search result. Get the title keyword and Tracks folder path. Then go through the Tracks folder and push back the data only if the track title contains the keyword. The list of push backed data is then displayed by updating the table component at the end. The code on the right describes the searching process:



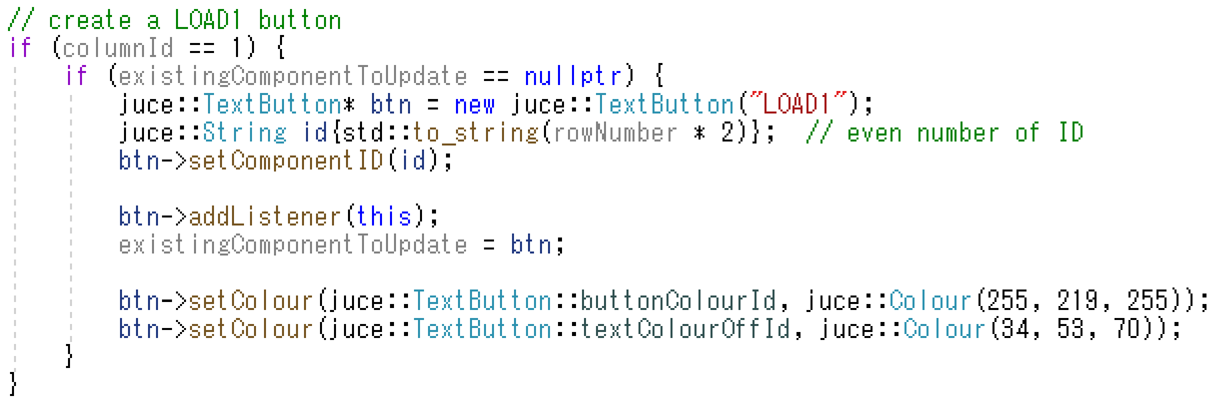
```
while (iter.next()) // iterate over the Tracks folde
{
    juce::File theFileItFound(iter.getFile());
    juce::String title = theFileItFound.getFileName(

    if (title.contains(titleKeyword)) {
        push_backMetadata(theFileItFound, title);
    }
}
```

R2D: Component allows the user to load files from the library into a deck

PlaylistComponent.h
   line 81, 64, 146: functions declaration
PlaylistComponent.cpp
   line 171: buttonClicked( )
   line 140: loadFromLibrary( )
   line 335: refreshComponentForCell( )

Achieved by creating load buttons in refreshComponentForCell( ) function as user adds music in a library and by calling buttonClicked( ) function when buttons are clicked.

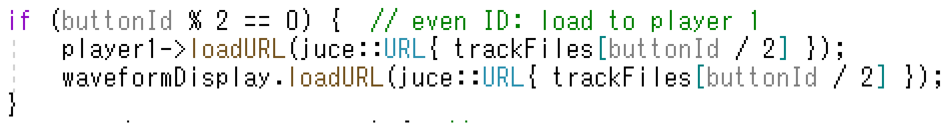
In refreshComponentForCell( ) function, if the passed columnId is equal to 1 it creates a LOAD1 button, assigns an even number of ID to the button, add a listener, set the button color to light pink, and update the existing component. If the passed columnId is equal to 2 it processes almost same for the LOAD1 button except it creates a LOAD2 button and assigns an odd number of ID. The following code is the process of creating a LOAD1 button for example:



```
// create a LOAD1 button
if (columnId == 1) {
    if (existingComponentToUpdate == nullptr) {
        juce::TextButton* btn = new juce::TextButton("LOAD1");
        juce::String id{std::to_string(rowNumber * 2)};  // even number of ID
        btn->setComponentID(id);

        btn->addListener(this);
        existingComponentToUpdate = btn;

        btn->setColour(juce::TextButton::buttonColourId, juce::Colour(255, 219, 255));
        btn->setColour(juce::TextButton::textColourOffId, juce::Colour(34, 53, 70));
    }
}
```

In buttonClicked( ) function, if the button is not an ADD button, it retrieves the button ID and calls loadFromLibrary( ) function passing the button ID.

In loadFromLibrary( ) function, it checks whether the parameter of button ID is even or odd, load a file to player 1 and waveform 1 if it's even ID, and load to player 2 and waveform 2 if it's odd ID. For example, if button ID is 4 it accesses trackFiles[2] and load the file into player 1 and waveform 1 in the following code:



```
if (buttonId % 2 == 0) {  // even ID: load to player 1
    player1->loadURL(juce::URL{ trackFiles[buttonId / 2] });
    waveformDisplay.loadURL(juce::URL{ trackFiles[buttonId / 2] });
}
```

R2E: The music library is restored when the user exits then restarts the application

PlaylistComponent.h
    line 158: function declaration
PlaylistComponent.cpp
    line 378: restoreLibrary( )

Achieved by saving files the user added in Tracks folder locally and calling restoreLibrary( ) function in the constructor so when user restarts the application the PlaylistComponent class always does this restoring process first.

In restoreLibrary( ) function, access the Tracks folder, iterate over the files saved under the folder, and push back every file data into the subjected vectors using push_backMetadata( ) function. Update the table component to display the push backed track data in the library. The following code describes the whole function:

```cpp
void PlaylistComponent::restoreLibrary()
{
    juce::String sMyFolderPath(juce::File::getCurrentWorkingDirectory().getFullPathName() + "/Tracks");
    juce::File myFolder(sMyFolderPath);
    if (myFolder.isDirectory()) // Tracks folder exists
    {
        juce::DirectoryIterator iter(juce::File(sMyFolderPath), true);

        while (iter.next()) // iterate over the Tracks folder
        {
            juce::File theFileItFound(iter.getFile());
            juce::String title = theFileItFound.getFileName();
            push_backMetadata(theFileItFound, title);
        }
        // update the library
        tableComponent.updateContent();
        tableComponent.repaint();
    }
}
```