

BSc in Computer Science

CM2005-01

Object – Oriented Programming End of term coursework

Hesron Muscat No – 170322450

March 2021

A screenshot of my OtoDeck application

OtoDecks_V2

JOHN KAH - CARINA (ENUI REMIX)
7 min : 5 sec

<< Play / Pause Stop >> ☐ Loop

CUE Save CUE Play LOAD Save to Playlist

0.5640103 1.0000000

0.3515508

DEMETER - UTOPIA (ORIGINAL MIX)
4 min : 27 sec

<< Play / Pause Stop >> ☐ Loop

CUE Save CUE Play LOAD Save to Playlist

0.2500000 1.0000000

0.3265509

Search for Music:

Track Title	Location	Length	
John Kah - Carina (Enui Remix)	C:\Users\hesro\Downloads\Joh...	7 min : 5 sec	Load Track
Can Believe (Nils Hoffmann Re...	C:\Users\hesro\Downloads\Ca...	5 min : 14 sec	Load Track
Nils Hoffmann - Balloons (Club ...	C:\Users\hesro\Downloads\Nils...	8 min : 16 sec	Load Track
bleep_10	C:\Users\hesro\Documents\B....	0 min : 18 sec	Load Track
Demeter - Utopia (Original Mix)	C:\Users\hesro\Downloads\De...	4 min : 27 sec	Load Track
Benny Benassi - Love Is Gonna...	C:\Users\hesro\Downloads\Ben...	6 min : 34 sec	Load Track
Jamie Woon - Lady Luck (Mad ...	C:\Users\hesro\Downloads\Ja...	7 min : 19 sec	Load Track
electro_smash	C:\Users\hesro\Documents\B....	2 min : 14 sec	Load Track

Requirements

- R1 – The application should contain all the basic functionality shown in class

R1A

The user has two options of how he can load a file into an audio player. The first one is by clicking on the 'LOAD' button on one of the players, or else he can choose to load a track from the playlist.

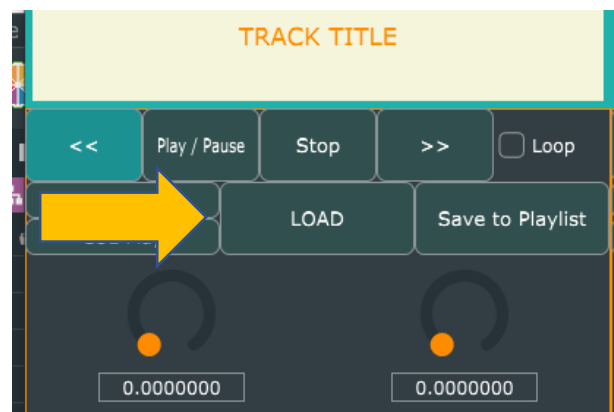



Figure 1: Load button on one of the players



Search for Music:			
Track Title	Location	Length	
John Kah - Carina (Enui Remix)	C:\Users\hesro\Downloads\Joh...	7 min : 5 sec	Load Track
Can Believe (Nils Hoffmann Re...	C:\Users\hesro\Downloads\Ca...	5 min : 14 sec	Load Track
Nils Hoffmann - Balloons (Club ...	C:\Users\hesro\Downloads\Nils...	8 min : 16 sec	Load Track

Figure 2: Load button on the playlist component

In the first case where the user chooses to load a file from the player, the code is found in the DeckGUI.cpp file. In the function called DeckGUI::buttonClicked(), one can find the code. When the button is clicked a file chooser object is created and a window pop up will appear to allow the user to choose the file. The file chosen is returned to the relevant functions of the player, waveformDisplay object, and the TrackTitle heading.

The below image is the code found in the DeckGUI.cpp file, in the DeckGUI::buttonClicked() function.

```

//if the button clicked is the Load button, a file chooser gets created and the chosen file
//gets loaded into the player, and to the waveform display object
//Also, the track title and song length are sent to the title object
if (button == &loadFile)
{
    juce::FileChooser chooser{ "Please select a file..." };
    if (chooser.browseForFileToOpen())
    {
        fileLoaded = chooser.getResult();
        //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(juce::URL{ fileLoaded });
        //The URL of the file is passed to the loadURL function of the waveformDisplay object and is loaded
        // into the audioThumb object
        w_display->loadURL(juce::URL{ fileLoaded });
        //The filename and the song length of the chosen file is passed to the setTitle function found in the
        //TrackTitle class
        title->setTitle(fileLoaded.getFileNameWithoutExtension().toUpperCase(), player->getSongLength());

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

```

Figure 3: Code found in the DeckGUI.cpp file

The below image is a code snippet from the `AudioPlayer::loadURL()` function found in the `AudioPlayer.cpp` file. It takes a URL as a parameter and it is then passed to the `formatManager` to extract the formats that will be able to read as audio files.

If the reader is not a null pointer it will then load the data onto the `transportSource` for playing.

```

//Loading the file into the transportSource
void AudioPlayer::loadURL(juce::URL audioURL)
{
    juce::AudioFormatReader* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

    if (reader != nullptr)
    {
        std::unique_ptr<juce::AudioFormatReaderSource> newSource
            (new juce::AudioFormatReaderSource(reader, true));

        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset(newSource.release());
    }
    else
    {
        DBG("Something went wrong in the file");
    }
}

```

Figure 4: `loadURL()` function found in the `AudioPlayer.cpp` file

For the second option, where the user chooses to load a track from the playlist, when he presses the button, the component ID of the button is used to identify which track he chose so it can be loaded onto the player. The process of loading a track into the player is the same as above, using the `AudioPlayer::loadURL()` function. But before, there is some logic in deciding onto which player the track will be loaded. If player 1 (left) is currently playing and player 2(right) is not, the new track is loaded onto player 2. If both players are playing then the track is loaded onto player 1. If player 1 is not playing and player 2 is playing, the track is loaded onto player 1.

For each of the above condition, the new track is also passed to the `waveformDisplay` object, the title and song length are extracted and passed onto the relevant `TrackTitle` object.

The below image shows the code found in the `PlaylistComponent.cpp` file, in the `PlaylistComponent::buttonClicked()` function.

```
//This function is called whenever a button is clicked in the playlist component
//A button is passed as a parameter, whenever a button is clicked inside this component
void PlaylistComponent::buttonClicked(juce::Button* button)
{
    //The component id set in the refreshComponentForCell function is extracted from each button
    int id = std::stoi(button->getComponentID().toString());

    //If player 1 is playing and player 2 is not, the track is loaded into player 2
    //Also, the title is sent to the title2 pointer and same is done to the w_display2 pointer
    if (player1->transportSource.isPlaying() == true && player2->transportSource.isPlaying() == false)
    {
        player2->loadURL(juce::URL{ tracks[id] });
        titled2->setTitle(tracks[id].getFileNameWithoutExtension().toUpperCase(), player2->getSongLength());
        w_display2->loadURL(juce::URL{ tracks[id] });
    }

    //Vice versa, if player 1 is not playing and player 2 is playing, the track is loaded into player 1
    //As above, the track title and waveform display are sent to the according pointers
    if (player1->transportSource.isPlaying() == false && player2->transportSource.isPlaying() == true)
    {
        player1->loadURL(juce::URL{ tracks[id] });
        titled1->setTitle(tracks[id].getFileNameWithoutExtension().toUpperCase(), player1->getSongLength());
        w_display1->loadURL(juce::URL{ tracks[id] });
    }

    //If both players are playing, track is loaded into player 1
    if (player1->transportSource.isPlaying() == true && player2->transportSource.isPlaying() == true)
    {
        player1->loadURL(juce::URL{ tracks[id] });
        titled1->setTitle(tracks[id].getFileNameWithoutExtension().toUpperCase(), player1->getSongLength());
        w_display1->loadURL(juce::URL{ tracks[id] });
    }

    //If player 1 is not playing, track is loaded into player 1
    if (player1->transportSource.isPlaying() == false)
    {
        player1->loadURL(juce::URL{ tracks[id] });
        titled1->setTitle(tracks[id].getFileNameWithoutExtension().toUpperCase(), player1->getSongLength());
        w_display1->loadURL(juce::URL{ tracks[id] });
    }
}
```

Figure 5: `PlaylistComponent::buttonClicked()` implementation

The component ID is used to identify which location in the tracks vector was accessed to identify the track in the playlist.

R1B

By using the MixerAudioSource object, we were able to play two tracks at the same time. Since both players are independent from each other, the MainComponent only has access to both and the MixerAudioSource object, called mixerSource, is found in the MainComponent.h header file. The prepareToPlay() function of mixerSource is then called inside the prepareToPlay() function of the MainComponent. Player 1 and player 2 were added as sources so mixerSource will be able to play both players at the same time. Also, we needed to call the getNextAudioBlock() function of the mixerSource object inside the getNextAudioBlock() function of the MainComponent. Likewise, the releaseResources() function was called inside the releaseResources() function of the MainComponent. If one would want to add more players, one just needs to add the new player as an input source to the mixerSource.

The below images show the code relating to the above.

```
//The mixerAudioSource object needed to be able to play 2 tracks at the same time
juce::MixerAudioSource mixerSource;
```

Figure 6: Object declaration inside the MainComponent.h header file

```
mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
mixerSource.addInputSource(&player1, false);
mixerSource.addInputSource(&player2, false);
```

Figure 7: Code snippet found in the MainComponent::prepareToPlay() function inside the MainComponent.cpp file

```
//Getting the next audio block from the buffer to play
void MainComponent::getNextAudioBlock (const juce::AudioSourceChannelInfo& bufferToFill)
{
    mixerSource.getNextAudioBlock(bufferToFill);
}
```

Figure 8: Code snippet found in the MainComponent::getNextAudioBlock() function inside the MainComponent.cpp file

```

//Releasing any resource used by player1, player 2 and MixerSource
void MainComponent::releaseResources()
{
    player1.releaseResources();
    player2.releaseResources();
    mixerSource.releaseResources();
}

```

Figure9: Code snippet found in the `MainComponent::releaseResources()` function inside the `MainComponent.cpp` file

R1C

By using a slider inside the DeckGUI component one is able to adjust and vary the volume of each player. By implementing the `DeckGUI::sliderValueChanged()` function inside the `DeckGUI.cpp` file, the DeckGUI component listens for changes and whenever the slider of the volume is changed, the gain of the player is updated by the new value.

Whenever the slider is moved, the `AudioPlayer::setGain()` function of the respective player is called. In this way the volume of each player can be varied.

```

//Implement a slider listener
void DeckGUI::sliderValueChanged(juce::Slider* slider)
{
    //Whenever the gain slider value changes, the player's gain is changed according to the
    //slider's value
    if (slider == &gain)
    {
        player->setGain(slider->getValue());
    }
}

```

Figure 10: The implementation of the `DeckGUI::sliderValueChanged()` function inside the `DeckGUI.cpp` file

```

//Setting the gain of the transportSource using the function from the AudioTransportSource class
void AudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
    }
    else
    {
        transportSource.setGain(gain);
    }
}

```

Figure 11: The implementation of the `AudioPlayer::setGain()` function inside the `AudioPlayer.cpp` file

When the user plays the track for the first time after loading it onto the player, the volume is set to a default value of 0.25. The user can then change the volume as he wants.

R1D

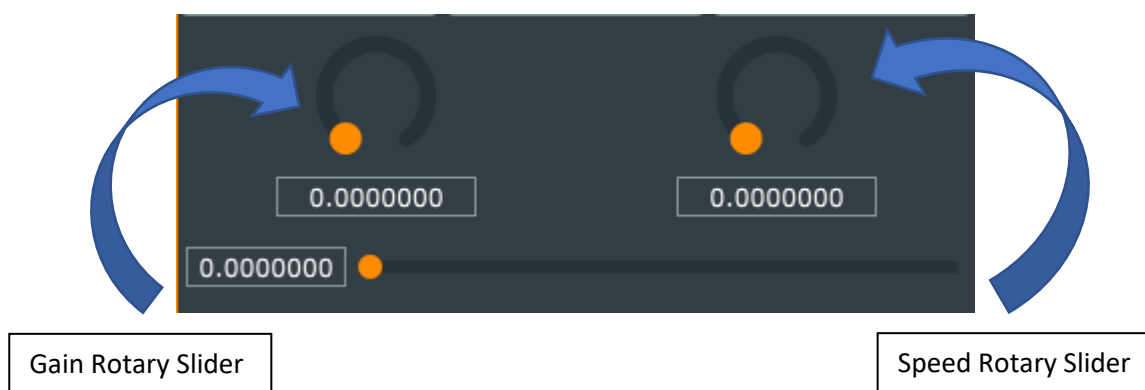
Just like the gain, the speed at which the track is played is controlled by a slider in the DeckGUI component. Whenever the slider is moved or value is changed, the `AudioPlayer::setSpeed()` function gets called. The new value of the slider is passed as a parameter. Inside this function, we are using a `ResamplingAudioSource` object to change the resampling ratio which results in changing the speed at which the track is played.

```
//Whenever the speed slider value changes, the player's speed is changed according to the
//slider's value
if (slider == &speed)
{
    player->setSpeed(slider->getValue());
}
```

Figure 12: The implementation of the `DeckGUI::sliderValueChanged()` function inside the `DeckGUI.cpp` file

```
//Setting the speed using the function from the ResamplingAudioSource class
void AudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0 || ratio > 100.0)
    {
        DBG("DJAudioPlayer::setSpeed ratio should be between 0 and 100");
    }
    else
    {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

Figure 13: The implementation of the `AudioPlayer::setSpeed()` function inside the `AudioPlayer.cpp` file



- R2 – 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 start/stop.

R2A

As one may see from the screenshot of my application, I chose a colour scheme of light sea green, dark orange, and beige as background for the Track title heading. All the buttons found on the top part of the Deck below the heading are painted in a dark slate grey colour but whenever the mouse is over the button, the respective button will change colour to dark cyan. Also, whenever a track is being played and the rewind button or the fast forward buttons are pressed they change colour to dark orange. This is to indicate that they are active since the button need to be held down for the operation to start and keep going. If the buttons are released the operation related to them is stopped.

The same principle is applied to the CUE Play button. All the logic is implemented in a dedicated function inside the DeckGUI.cpp file called DeckGUI::buttonsRepainting(). Then, this function is called inside the paint function. I chose to separate the code related to the painting of the buttons from that related to the sliders for better organization of code.

The below are code snippets from the buttonsRepainting function.

```
//Function takes care of the painting of the buttons
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);
    }
}
```

Figure 14: Implementation of the DeckGUI::buttonsRepainting() function inside the DeckGUI.cpp file

```
//The below 3 statements sets the colour of the buttons darkorange whenever the buttons toggles are on
cue_play.setColour(juce::TextButton::buttonOnColourId, juce::Colours::darkorange);
rewind.setColour(juce::TextButton::buttonOnColourId, juce::Colours::darkorange);
fforward.setColour(juce::TextButton::buttonOnColourId, juce::Colours::darkorange);

//This sets the colour of the tick of the loop togglebutton
loop.getLookAndFeel().setColour(juce::ToggleButton::tickColourId, juce::Colours::orange);
```

Figure 15: Part of the code inside the `DeckGUI::buttonsRepainting()` function inside the `DeckGUI.cpp` file

The same is applied to the sliders. The gain and speed sliders are set to be a rotary style slider and the relevant painting code is organized inside a dedicated function which is called inside the `DeckGUI::paint()` function. The function is called `DeckGUI::sliderRepainting()` and can be found inside the `DeckGUI.cpp` file.

```
//The function that takes care of the colours to be used by the sliders
void DeckGUI::slidersRepainting()
{
    gain.setColour(juce::Slider::thumbColourId, juce::Colours::darkorange);
    speed.setColour(juce::Slider::thumbColourId, juce::Colours::darkorange);
    position.setColour(juce::Slider::thumbColourId, juce::Colours::darkorange);

    gain.setColour(juce::Slider::rotarySliderFillColourId, juce::Colours::lightseagreen);
    speed.setColour(juce::Slider::rotarySliderFillColourId, juce::Colours::lightseagreen);
    position.setColour(juce::Slider::trackColourId, juce::Colours::lightseagreen);
}
```

Figure 16: The implementation of the `DeckGUI::sliderRepainting()` function inside the `DeckGUI.cpp` file

R2B

Apart from the start and stop functionality, the user can move backwards and forwards along the track by using the rewind and fast forward buttons. The user needs to hold these buttons down and the play head will move forwards or backwards at a constant rate. For the buttons to be active while the user holds down the buttons, I used the `Timer` class to check if the button is still down or not at a certain interval. If the button is released, it will change back to the default colour and the operation stops. He can also choose to move to any position by using the position slider.

The user can also, tick the loop checkbox and this will enable the loop function. This means that whenever a track has finished, the player goes back to the start position and starts playing again.

All the above code can be found in the DeckGUI::buttonClicked() function in the DeckGUI.cpp file. Also, since the Timer class was used, I needed to implement the timerCallback() function.

I dedicated a separate function for the rewind, fast forward and CUE play buttons that is then called inside the DeckGUI::timerCallback() function. This is to separate button callback code from that of the position slider and the waveformDisplay timer callback code.

```
//A function that gets called instead the timerCallback function
void DeckGUI::buttonTimerCallback()
{
    //If the rewind button is down and cue is not being used,
    //Also, if the player is playing, the state of the rewind button is set to on
    //and the player rewind function gets called
    if (rewind.isDown() && cue == false)
    {
        if (player->transportSource.isPlaying())
        {
            rewind.setToggleState(true, false);
            player->rewind();
        }
    }
    //else the toggle state is set to false
    else
    {
        rewind.setToggleState(false, false);
    }
}
```

Figure 17: Part of the code in the DeckGUI::buttonTimerCallback() function in the DeckGUI.cpp file

```
//Implementing this function since we inherit from the timer class
void DeckGUI::timerCallback()
{
    //The function gets called
    buttonTimerCallback();

    //Moving the position slider according to the new position of the transportSource
    //normalized between 0 and 1
    position.setValue(player->getPositionRelative(),juce::NotificationType::dontSendNotification);
    //The playhead position is updated according to the player's relative position
    w_display->setPositionRelative(player->getPositionRelative());
}
```

Figure 18: DeckGUI::timerCallback() function in the DeckGUI.cpp file

- R3 – Implementation of a music library component which allows the user to manage their music library

R3A

This problem was solved by using a Text button in the DeckGUI component. Whenever the Save to Playlist button is pressed, the loaded track will be added to the vector of files that holds all the tracks saved in the playlist.

This is done by calling the PlaylistComponent::addAudioFile() function found in the PlaylistComponent.cpp file. Before the file is added, a check is made to see if the file already exists in the playlist, if it does exist, it will not be added again.

The below 2 images show the code found in the DeckGUI::buttonClicked() function where the call for the PlaylistComponent::addAudioFile() is made, and the code in the PlaylistComponent.cpp file where the implementation of the PlaylistComponent::addAudioFile() is found.

```
//Whenever the SaveToPlaylist button is pressed, the loaded track gets added to the playlist
// by the addAudioFile function passing the file
if (button == &SaveToPlaylist)
{
    trackList->addAudioFile(&fileLoaded);
}
```

Figure 19: Code that is found in the DeckGUI::buttonClicked() function relating to the Save to Playlist button

```
//A function that enables adding of file to the tracklist
void PlaylistComponent::addAudioFile(juce::File* audioFile)
{
    //bool variable used as a signal whether the file already exists or not
    bool alreadyExists{ false };

    for (juce::File& file : tracks)
    {
        //if the filename of the file in the tracks vector is equal to the filename of the audiofile passed
        // the bool variable is set to true and loop exits
        if (file.getFileName().equalsIgnoreCase(audioFile->getFileName()))
        {
            alreadyExists = true;
            break;
        }
    }

    //If alreadyExists is still false and the file's passed directory string is empty
    //the file is pushed back the vector
    //The playlist's content is updated and the paint function is recalled
    if (alreadyExists == false && audioFile->getFullPathName() != "")
    {
        tracks.push_back(*audioFile);
        playlist.updateContent();
        playlist.repaint();
    }
}
```

Figure 20: Implementation of the PlaylistComponent::addAudioFile() function in the PlaylistComponent.cpp file

R3B

To solve this problem, I used a temporary `AudioPlayer` object so that the `PlaylistComponent` could operate on the file to extract information regarding each track in the playlist. For each track found in the 'tracks' vector, the file name, location on disk and song length are displayed in a separate column in the `PlaylistComponent`. These are displayed by implementing the code in the `PlaylistComponent::paintCell()` function found in the `PlaylistComponet.cpp` file.

For the title of the track and location on disk, a function that is a member of the `JUCE::File` class was used to extract such data. On the other hand, to display the song length, I implemented a custom function found in the `AudioPlayer.cpp` file where the song length is calculated and returned as a string in the form of minutes and seconds. This is done because the function that operates on the `AudioTransportSource` only returns the song length in seconds.

```
//A function that returns the total time of the track in minutes and seconds as a String
juce::String AudioPlayer::getSongLength()
{
    double length_minutes{ transportSource.getLengthInSeconds() / 60 };

    //Variables needed to seperate the integer and fractional value of the calculated minutes
    //from the return value in seconds of the function getLengthInSeconds
    double fractional_part, integer_part;
    fractional_part = modf(length_minutes, &integer_part);

    //two variables that hold the calculated minutes and seconds
    std::string minutes{ std::to_string((int)integer_part) };
    std::string seconds{ std::to_string((int)round(fractional_part * 60)) };

    juce::String final_string{ minutes + " min : " + seconds + " sec" };

    return final_string;
}
```

Figure 21: The `AudioPlayer::getSongLength()` function found in the `AudioPlayer.cpp` file

```

void PlaylistComponent::paintCell(juce::Graphics& g,
    int rowNumber,
    int columnId,
    int width,
    int height,
    bool rowIsSelected)
{
    //The next file in the tracks vector is loaded into the tmp player so that the songlength
    // is extracted
    tmp_player->loadURL(juce::URL(tracks[rowNumber]));

    //At column 1 the file name / track title is displayed
    if (columnId == 1)
    {
        g.drawText(tracks[rowNumber].getFileNameWithoutExtension(),
            2,
            0,
            width - 4,
            height,
            juce::Justification::horizontallyCentred,
            true);
    }

    //At column 2 the location of the file is displayed
    if (columnId == 2)
    {
        g.drawText(tracks[rowNumber].getFullPathName(),
            2,
            0,
            width - 4,
            height,
            juce::Justification::horizontallyCentred,
            true);
    }

    //At column 3 the song length in minutes and seconds is displayed
    if (columnId == 3)
    {
        g.drawText(tmp_player->getSongLength(),
            2,
            0,
            width - 4,
            height,
            juce::Justification::horizontallyCentred,
            true);
    }
}

```

Figure 22: The code found in the `PlaylistComponent::paintCell()` function in the `PlaylistComponent.cpp` file

R3C

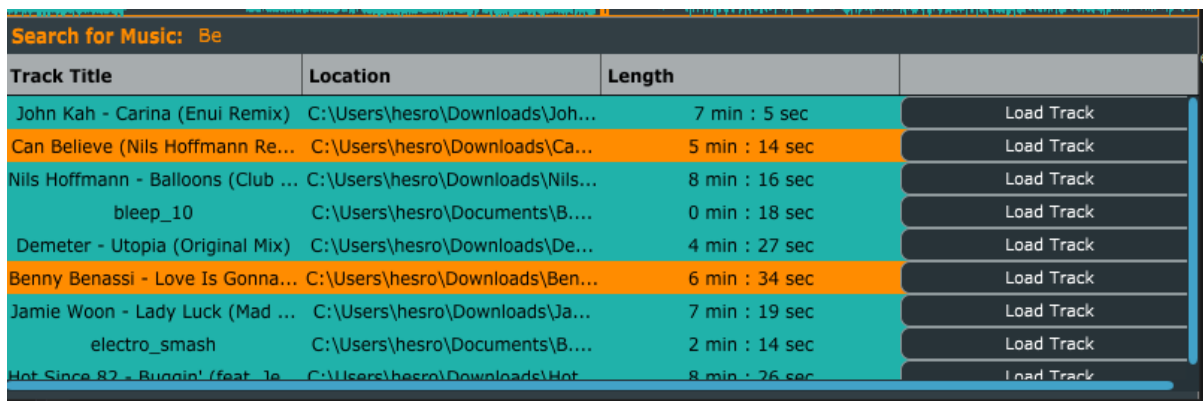
To allow the user to search for tracks in the playlist, I used the JUCE Label class. I created 2 labels, one that has only text “Search for music”, which I called ‘searchField’ and another label which allows text input. I called the latter, ‘inputText’.

Then, I attached these two labels together so they will always stay together. This means that one may only set the bounds for one label.

Whenever some text is inputted in the ‘inputText’ label and the return button is pressed a function is called. This function is the PlaylistComponent::searchTrack() function. Its implementation can be found in the PlaylistComponent.cpp file.

This function tries to find an exact match between the filename and the inputted text. Also, it returns a positive result whenever the inputted text is also a substring of the filename of each track inside the track list.

Whenever there is a positive result, the relevant row is set to be selected, the contents of the playlist are updated and the paint function is called once again. The rows are set to be painted a dark orange colour whenever they are set as selected. This will indicate to the user that the row highlighted is a returned result.



The screenshot shows a window titled "Search for Music: Be" with a table of search results. The table has four columns: Track Title, Location, Length, and an action button. The results are as follows:

Track Title	Location	Length	
John Kah - Carina (Enui Remix)	C:\Users\hesro\Downloads\Joh...	7 min : 5 sec	Load Track
Can Believe (Nils Hoffmann Re...	C:\Users\hesro\Downloads\Ca...	5 min : 14 sec	Load Track
Nils Hoffmann - Balloons (Club ...	C:\Users\hesro\Downloads\Nils...	8 min : 16 sec	Load Track
bleep_10	C:\Users\hesro\Documents\B...	0 min : 18 sec	Load Track
Demeter - Utopia (Original Mix)	C:\Users\hesro\Downloads\De...	4 min : 27 sec	Load Track
Benny Benassi - Love Is Gonna...	C:\Users\hesro\Downloads\Ben...	6 min : 34 sec	Load Track
Jamie Woon - Lady Luck (Mad ...	C:\Users\hesro\Downloads\Ja...	7 min : 19 sec	Load Track
electro_smash	C:\Users\hesro\Documents\B...	2 min : 14 sec	Load Track
Hot Since 82 - Buggin' (feat. Je...	C:\Users\hesro\Downloads\Hot	8 min : 26 sec	Load Track

Figure 23: An example of a search in the PlaylistComponent

```

//A function that lets the user search for a track
void PlaylistComponent::searchTrack(juce::String text)
{
    //At the beginning of each call of this function all the rows in the playlist are deselected
    playlist.deselectAllRows();

    //Looping through each file in the tracks vector,
    //the file name is compared with the text passed as an exact match or
    //if it contains the text as a substring
    for (int i = 0; i < tracks.size(); ++i)
    {
        if (tracks[i].getFileNameWithoutExtension().equalsIgnoreCase(text)
            || tracks[i].getFileNameWithoutExtension().containsIgnoreCase(text))
        {
            //if one of the above conditions return true, the respective row is set as selected
            //content is updated and repaint function is called
            playlist.selectRow(i, false, false);
            playlist.updateContent();
            playlist.repaint();
        }
    }
}

```

Figure 24: The `PlaylistComponent::searchTrack()` function found in the `PlaylistComponent.cpp` file

```

//2 Label objects that are used to let the user search for music
juce::Label searchField;
juce::Label inputText;

```

Figure 25: The two labels that are used for searching found in the private section of the `PlaylistComponent.h` header file.

```

void PlaylistComponent::paint (juce::Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId)); // clear the background

    g.setColour (juce::Colours::grey);
    g.drawRect (getLocalBounds(), 1); // draw an outline around the component

    //Setting the inputText label to be editable so the user can enter text to search music
    inputText.setEditable(true);
    //Setting the colour of the text
    inputText.setColour(juce::Label::textColourId, juce::Colours::darkorange);
    //When user presses the return button the searchTrack function is called passing the text entered by the user
    inputText.onTextChanged = [this] {searchTrack(inputText.getText()); };

    //Below are the settings for the searchField label
    searchField.setText("Search for Music: ", juce::NotificationType::dontSendNotification);
    searchField.setFont(juce::Font(16.0f, juce::Font::bold));
    //Attaching the searchField label to go with the inputText
    searchField.attachToComponent(&inputText, true);
    searchField.setColour(juce::Label::textColourId, juce::Colours::darkorange);
    searchField.setJustificationType(juce::Justification::right);
}

```

Figure 26: The code found in the `PlaylistComponent::paint` function found in the `PlaylistComponent.cpp` file

R3E

To solve the problem, I used a text file where upon running of the program, the program will check at the path passed upon declaration if it exists, and if not, it will be created. If it already exists, it means that possibly there are paths of songs stored on disk that were added to the playlist before. If so, it will extract each path from this text file, a file for each

path and will get stored in the tracks vector. This process is done by the PlaylistComponent::loadDirectories() function call in the constructor of the PlaylistComponent.

When the program is closed, a function called PlaylistComponent::savePlaylistToFile() is called in the class destructor. This function will take the path of each file saved in the tracks vector and write it to the same text file that was used for reading for the next time the program is opened.

```
//A function that reads the playlist and saves the URLs to file
void PlaylistComponent::savePlaylistToFile()
{
    //A file output stream used to write to file
    juce::FileOutputStream stream{ playlistFile };

    //If the file output stream opened ok, the stream position is set at the beginning of the file
    //For each file in tracks vector the directory string is written to file added a carriage return
    //for ease of reading when the file is read back
    if (stream.openedOk() == true)
    {
        stream.setPosition(0);

        for (auto &file : tracks)
        {
            juce::String directory{ file.getFullPathName()+"\n"};
            DBG(directory);
            stream.writeText(directory, false, false, nullptr);
        }

        //stream is flushed to make sure that everything in the stream is written to file
        stream.flush();
    }
}
```

Figure 27: Implementation of the PlaylistComponent::savePlaylistToFile() function in the PlaylistComponent.cpp file

```
//A function to load the URLs of the files found in the playlist textfiles
void PlaylistComponent::loadDirectories()
{
    //An input stream is opened indicating the file that is used to read from
    juce::FileInputStream stream{ playlistFile };

    //if stream opened ok, a string is created for each line and a new file
    //is created from the line read and added to the tracks vector
    if (stream.openedOk() == true)
    {
        while (!stream.isExhausted())
        {
            juce::String str = stream.readLine();
            tracks.push_back( juce::File{ str } );
        }
    }
    else
    {
        DBG("INPUT STREAM failed to open");
    }
}
```

Figure 28: Implementation of the PlaylistComponent::loadDirectories() function in the PlaylistComponent.cpp file

- R4 – Implementation of a complete custom GUI

R4A/B/C

As one can see from the screenshot found on the first page of this document, the layout of the program is that of the Track Title on top of each deck where the track title, together with the song length is displayed whenever a track is loaded either from choosing a file from disk or from the playlist component.

Just below the track title, one finds a group of buttons which allow the user to control the playback of the track. These are painted a certain colour but whenever the mouse is over any button, this button will change colour. In addition to this, the rewind (<<), fast forward(>>) and the CUE Play button, when they are in an ON state, these will change colour as well to indicate that they are on, since they need to be held down for the operation to occur. If the buttons are released they switch back to an OFF state.

Below the button, one finds 2 rotary sliders, one for the gain and another for the speed, and a horizontal slider for the position of the play head. The thumbs of each slider are painted a dark orange colour and the filled part of the slider is painted with a light sea green colour.

Below the slider, the waveformDisplay object is painted. The waveform is painted in a light sea green colour with a dark orange play head to match the colour scheme of the application. All the above makes up a deck and there are two of these in my program.

Below them, one finds the Search function together with the playlist component. The background colour of each row is light sea green and when a row is clicked on or selected, the background changes to dark orange.

Comments

I am happy with what I managed to achieve during this coursework and feel that I have learned a lot of new things as this was my first GUI application. I think that the functionality of the program where the user can save a cue position, play from this position and moving along the track using the buttons or the slider were my strong points and feel that I implemented this functionality in an efficient way.

Some improvements to this application might be more functionality to the playlist. A delete track button could be also added as an example. Also, a custom look to the slider would give a much more custom look to the application. I would be happy to further enhance this application, use it myself and also share it with other who might want to experiment with it.