OTTO DECKS: A DJ APPLICATION WRITTEN IN JUCE

Purpose of the Application

Otto Decks is an audio application that was created using the JUCE framework: the underlying programming language of the framework is C++.

The main purpose of Otto Decks is to simulate the real behaviour of a DJ. This means that audio tracks can be loaded into players, their volumes adjusted and that a custom playlist can be created and saved for later use. In the same vein, the contents of the playlist can also be deleted.

Basic Functionality

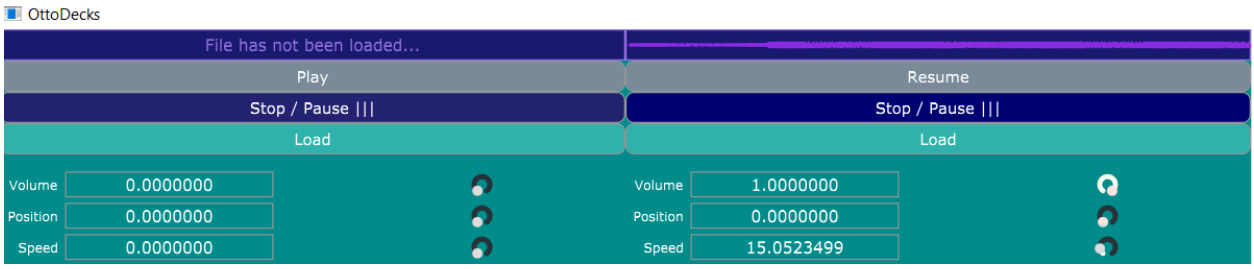
This section provides an overview of the basic functionality that was implemented during the lecture; it also includes additional features created for the final project.

Implemented Requirements Table

The table below provides an overview of Otto Deck's basic services.

Requirement	(Y/N)	Relevant Methods / Classes
R1A: can load audio files into audio players	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::loadURL(URL audioURL)
R1B: can play two or more tracks	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::playSong()
R1C: can mix the tracks by varying each of their volumes	Y	void ControlDeck::sliderValueChanged(Slider* slider) void DJAudioPlayer::setSpeed(double ratio)
R1D: can speed up and slow down the tracks	Y	void ControlDeck::sliderValueChanged(Slider* slider) void DJAudioPlayer::setGain(double gain)

High Level Description of the Features



The basic functionality of the Otto Decks application can be broken down into three buttons and sliders. Each of these buttons and sliders have specific tasks associated with them.

In particular, the three button in control of:

- ★ Playing or resuming the audio of a track.
- ★ Stopping or pausing a track's audio.
- ★ Uploading an audio file into one of the two players.

The audio-related functionality of the buttons is implemented in the DJAudioPlayer class, while the ControlDeck component is in charge of mapping each button to the appropriate function call that deals with the relevant audio effect. In other words, if the play button is clicked, the ControlDeck component will check that the button has been clicked and delegate the task of the audio being played to the DJAudioPlayer class.

In addition to the buttons, the sliders, positioned underneath them, are responsible for:

- ★ The volume of the audio.
- ★ The position of the audio.
- ★ The speed of the audio.

As in the case of the buttons, the audio-related parts are allocated to the DJAudioPlayer while the ControlDeck component ensures that – each time a slider's value is changed – the desired audio effect is communicated to the right function in the DJAudioPlayer class mentioned above.

Low Level Description of the Features

As stated above, the DJAudioPlayer class is responsible for the audio and playback processes of the tracks. For example, the loading of the tracks is dealt with in the following function below.

```
void DJAudioPlayer::loadURL(URL& audioURL)
{
        auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

        if (reader != nullptr)
        {
                std::unique_ptr<AudioFormatReaderSource> newSource(new
AudioFormatReaderSource(reader, true));
                transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
                readerSource.reset(newSource.release());
        }
}
```

In the same vein, functions such as playing or pausing the audio are carried out by the functions such as playSong() as shown in the example displayed below.

```
void DJAudioPlayer::playSong()
{
        transportSource.start();
}

void DJAudioPlayer::stopSong()
{
        transportSource.stop();
}
```

The audio-related functions above are used by the ControlDeck component, which matches each audio-related function to its corresponding button as shown in the following code snippet. The code

snippet below is part of the buttonClicked method which is an override of Button::Listener. It uses a pointer to the DJAudioPlayer class to refer to methods from it.

```
if (button == &playButton)
    {
        player->setPosition(0);
        player->playSong();
        paused = false;
    }

    if (button == &stopButton)
    {
        player->stopSong();
        paused = true;
    }
```

Another important part of the audio-related tasks is to distinguish the various states of the play button. For example – when the audio has been paused – the text of the play button changes to 'Resume'; a code-related purpose on how to accomplish it is shown in the example below.

```
void ControlDeck::displayPlayButtonText(bool pauseButtonStatus)
{
    std::string playButtonText = "";
    switch (pauseButtonStatus)
    {

    case true:
        playButtonText = "Resume";
        break;

    case false:
        playButtonText = "Play";
        break;
    }

    playButton.setButtonText(playButtonText);
}
```

The displayPlayButtonText function, which is called after the if statements in the buttonClicked method have been executed, takes a bool parameter as an argument that determines which text will be displayed for the play button in the wake of other buttons having been clicked.

Custom Deck Control

This section deals with the custom control decks that were created for the final project, providing an overview of their purpose.

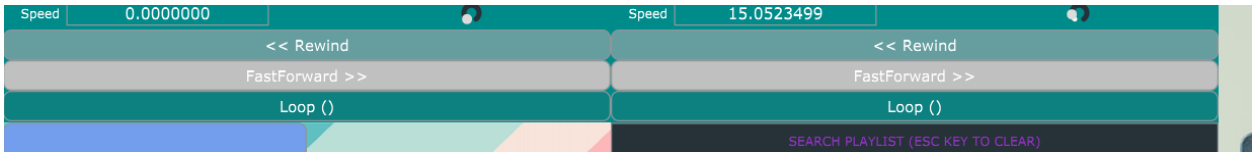
Implemented Requirements Table

The table below gives an overview of the custom control decks that are used to control the audio's playback of the tracks.

Requirement	(Y/N)	Relevant Methods / Classes

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 stop/ start.	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::fastForwardSong() void DJAudioPlayer::rewindSong() void DJAudioPlayer::startLoop()
R2A: Component has custom graphics implemented in a paint function.	Y	void MainDJAudioManager::paint (juce::Graphics& graphics) void MainDJAudioManager::resized() void ControlDeck::paint(juce::Graphics& graphics) void ControlDeck::repaintButtons() void ControlDeck::repaintSliders()
R2B: Component enables the user to control the playback of a deck somehow.	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::fastForwardSong() void DJAudioPlayer::rewindSong() void DJAudioPlayer::startLoop()

High Level Description of the Features



In addition to the basic controls, three new buttons have been added. Building on the original buttons, they contribute to the Otto Decks application as listed below:

- ★ The rewinding of a track.
- ★ The fastforwarding of a track.
- ★ The looping of a track.

The audio-related components live in the DJAudioPlayer class while the ControlDeck deals with assigning each button to the appropriate function in the DJAudioPlayer class.

Additionally, the layout of the ControlDeck component's layout has also been changed, with the sliders having been made rotary. Moreover, the sliders and buttons have the ability to change their colours depending on their state of activity.

Low Level Description of the Features

Similarly to the implementation of the basic audio features, the more advanced audio control features are part of the DJAudioPlayer class. For example, the rewind method is implemented as shown in the table below.

void DJAudioPlayer::rewindSong()

```
{
	int currentPosition = transportSource.getCurrentPosition();

	if (currentPosition > 0.5)
	{
		setPosition(currentPosition - 5.0);
	}

	else
	{
		setPosition(0.0);
	}
}
```

As follows, this method is called by the ControlDeck's buttonClicked function as shown in the code snippet below.

```
if (button == &rewindButton)
   {
      player->rewindSong();
      paused = false;
   }
```

The ControlDeck component also includes a way to repaint the button and sliders depending on their state: the function is called repaintButtont() which is called in the paint function. Its implementation details are shown below.

```
void ControlDeck::repaintButtons()
{

   // set the colour for the play button if mouse over OR it has stopped playing
   if (playButton.isOver() || playButton.isMouseOver())
   {
      playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkblue);
   }

   // set the colour of the play button when the mouse is not hovering over it
   else
   {
      playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::lightslategrey);
   }

   // set the colour of the stop button if mouse is over OR it has stopped playing
   if (stopButton.isOver() || stopButton.isMouseOver())
   {
      stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::aquamarine);
   }
```

A variation of the repaintButton method is reused for the MusicControlDeck, so that the effect of the buttons changing colour is consistent throughout the entire application.

Music Library Manager

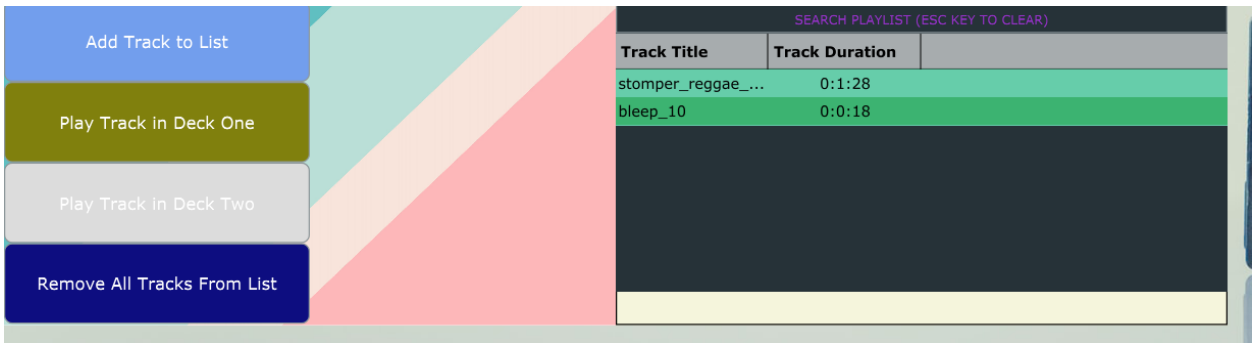
This section is concerned with the MusicLibraryManager that is based on the PlayListComponent programmed in class.

Implemented Requirements Table

The table below offers a glimpse into the Music Library Manager component that is tasked with the organisation of the playlist.

Requirement	(Y/N)	Relevant Methods / Classes
R3A: Component allows the user to add files to their library.	Y	void MusicControlDeck::buttonClicked(Button* button) void MusicControlDeck::populateTracksFile() MusicControlDeck::checkIfTrackAlreadyLoaded(TrackFile& trackFile) juce::Array<juce::File> MusicControlDeck::loadInTracks() void MusicControlDeck::removeAllContentsFromFile() void PlayListManager::addTrackToTracksVector(TrackFile& trackFile) void PlayListManager::deleteTracksFromTable()
R3B: Component parses and displays meta data such as filename and song length.	Y	juce::String TrackFile::getFileName() void TrackFile::setFileLength(juce::String length) uce::String AudioMetaData::getAudioTrackLength(juce::URL& audioURL) double DJAudioPlayer::getFileLengthSeconds() juce::String AudioMetaData::getFormattedAudioString(double& seconds)
R3C: Component allows the user to search for files.	Y	void PlayListManager::searchThePlaylist(juce::String inputtedText)
R3D: Component allows the user to load files from the library into a deck.	Y	void PlayListManager::addTrackToPlayerOne() void PlayListManager::addTrackToPlayerTwo()
R3E: The music library persists so that it is restored when the user exits then restarts the application.	Y	void PlayListManager::loadTracksFile() void PlayListManager::saveTracksFile()

High Level Description of the Features



The implementation of the playlist is one of the most complex in the entire Otto Decks application; it contains two components and classes as depicted below.

- ★ The MusicControlDeck component whose main objective is the creation of the tracks.txt file; it is also in charge of assigning each button to a specific role.
- ★ The PlayListManager component which is tasked with handling the contents of the playlist table displays.
- ★ The TrackFile class which deals with tasks such as the retrieval of track file specific properties such as the name or file length. This information is used by the PlayListManger to populate the playlist.
- ★ The AudioMetaData class which supervises the retrieval of audio-related mediadata such as the length of the file.

While the PlayListManager is the main force behind the organisation of the way the playlist is depicted on screen, the MusicControlDeck has four buttons that deliver the following duties:

- ★ Adding a track to the playlist table.
- ★ Playing a selected track in the playlist in either of the available audio players.
- ★ Removing all tracks from the playlist.

The format used to store the playlist is a txt file; its contents are stored in a vector utilised by the PlayListManager class in order to display the tracks in a table. The PlayListManager is also responsible for loading and saving the tracks.txt file.

Low Level Description of the Features

The creation of the txt file and / or writing of a single track into is achieved as shown in the function below.

```
void MusicControlDeck::populateTracksFile()
{
   std::ofstream fileList;

   // append the track list into the existing playlist
   fileList.open("tracks.txt", std::fstream::app);

   juce::Array<juce::File> files = loadInTracks();

   for (File& file : files)
   {
      TrackFile trackFile{file};

      if (!checkIfTrackAlreadyLoaded(trackFile))
      {
         filesAlreadyLoaded.push_back(trackFile);
         fileList << trackFile.getTrackFileProperties().getFullPathName() << "\n";
         playList->addTrackToTracksVector(trackFile);
      }
   }

   fileList.close();
}
```

The method checkIfTrackAlreadyLoaded contains a check for whether a certain track has already been added to the tracks.txt file. The way this is done is displayed in the table below.

```
bool MusicControlDeck::checkIfTrackAlreadyLoaded(TrackFile& trackFile)
{
    bool trackAlreadyLoaded = false;

    for (TrackFile& existingTrackFile : filesAlreadyLoaded)
    {
        if (existingTrackFile.getFileName() == trackFile.getFileName())
        {
            trackAlreadyLoaded = true;
        }
    }

    return trackAlreadyLoaded;
}
```

The PlayListManager component also contains functions such as loading the track into a tracks vector as shown below.

```
void PlayListManager::addTrackToTracksVector(TrackFile& trackFile)
{
    File file = trackFile.getTrackFileProperties();
    juce::URL audioURL{ file };
    trackFile.setFileLength(audioMetaData->getAudioTrackLength(audioURL));
    trackList.push_back(trackFile);
}
```

Custom GUI

This section deals with the custom GUI that was created for the final project.

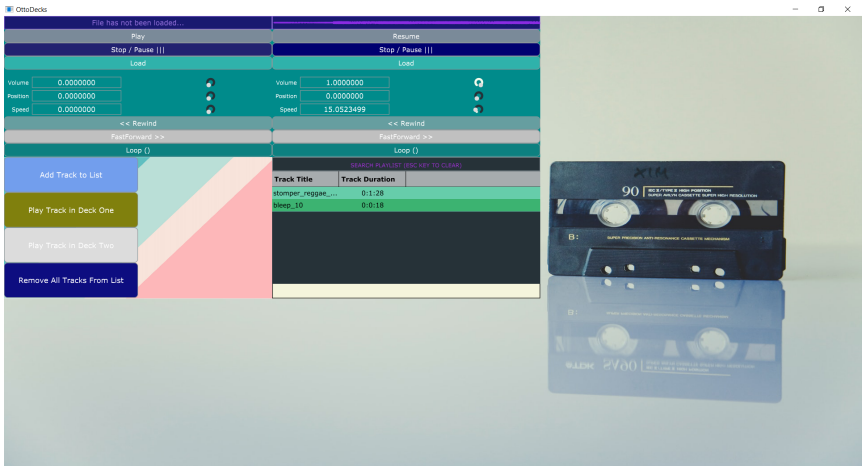
Implemented Requirements Table

The table below showcases the way the custom GUI has been rendered different from the original layout of the Otto Decks application.

Requirement	(Y/N)	Relevant Methods / Classes
R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls.	Y	void MainDJAudioManager::paint (juce::Graphics& graphics) void MainDJAudioManager::resized() void MusicControlDeck::paint(juce::Graphics& graphics) void MusicControlDeck::resized() void MusicControlDeck::repaintButtons()
R4B: GUI layout includes the custom Component from R2.	Y	void ControlDeck::paint(juce::Graphics& graphics) void ControlDeck::resized() void ControlDeck::repaintButtons()

		void ControlDeck::repaintSliders() void MusicControlDeck::paint(juce::Graphics& graphics) void MusicControlDeck::resized() void MusicControlDeck::repaintButtons()
R4C: GUI layout includes the music library component from R3.	Y	void PlayListManager::paint(juce::Graphics& graphics) void PlayListManager::resized()

High Level Description of the Features



The Otto Deck layout consists of the following building blocks, which each contribute to the whole look of the application:

- ★ The two control decks, positioned next to each other.
- ★ The MusicControlDeck and the playlist table, which are underneath the two decks.
- ★ A custom background that displays an audio cassette.
- ★ A custom background incorporated within the MusicControlDeck that shows three triangles of differing colours.

While the opportunity to overwrite the LookAndFeel class was an option, it proved to be too much of a hassle to be worthwhile. Therefore, the approach above was chosen in order to create a layout that was easy on the eyes, but also more manageable in terms of the coding effort. The design went through several changes, taking feedback from my partner and friends into account in terms of the application not being too jarring or bright.

Low Level Description of the Features

The implementation of the background images was achieved by creating a cache of the relevant image in the MainDJAudioManager.cpp constructor and then referring to it in the paint method. The table below showcases how this was accomplished.

```
// add background image
    // credit @ https://unsplash.com/@markusspiske
```

```
    backgroundImage = ImageCache::getFromMemory(BinaryData::background_png,
BinaryData::background_pngSize);

void MainDJAudioManager::paint (juce::Graphics& graphics)
{
    graphics.drawImage(backgroundImage, getLocalBounds().toFloat());
}
```

In terms of customising each specific component, the paint and resized functions were utilised as shown in the example for the MusicControlDeck below.

```
void MusicControlDeck::paint(juce::Graphics& graphics)
{
    graphics.fillAll(juce::Colours::whitesmoke);
    graphics.drawImage(backgroundImage, getLocalBounds().toFloat());

    //A call to the buttons and sliders painting functions
    repaintButtons();
}

void MusicControlDeck::resized()
{
    double rowH = getHeight() / 4.0;

    // play, stop and load button positions
    loadTrack.setBounds(0, 0, getWidth() / 2.0, rowH * 1.0);
    loadToDeckOne.setBounds(0, rowH * 1.0, getWidth() / 2.0, rowH * 1.0);
    loadToDeckTwo.setBounds(0, rowH * 2.0, getWidth() / 2.0, rowH * 1.0);
    removeTrack.setBounds(0, rowH * 3.0, getWidth() / 2.0, rowH * 1.0);
}
```

The resized method was of particular importance for the ControlDeck because of the positioning of the label buttons that had to be aligned in the same area as the rotary sliders. Therefore, as can be seen below, the x coordinate of location had to be moved by 50 centimetres in order to create space for the label buttons.

```
void ControlDeck::resized()
{
    double rowH = getHeight() / 10.5;

    // Waveform Display
    waveformDisplay.setBounds(0, 0, getWidth(), rowH * 1);

    // Play, Stop and Load button positions
    playButton.setBounds(0, rowH * 1.0, getWidth(), rowH * 1.0);
    stopButton.setBounds(0, rowH * 2.0, getWidth(), rowH * 1.0);
    loadButton.setBounds(0, rowH * 3.0, getWidth(), rowH * 1.0);

    // Slider Positions
    volumeSlider.setBounds(50, rowH * 4.2, getWidth(), rowH * 1.5);
    positionSlider.setBounds(50, rowH * 5.2, getWidth(), rowH * 1.5);
    speedSlider.setBounds(50, rowH * 6.2, getWidth(), rowH * 1.5);

    // Rewind, Fastforward and Loop buttons
    rewindButton.setBounds(0, rowH * 7.5, getWidth(), rowH * 1.0);
    fastForwardButton.setBounds(0, rowH * 8.5, getWidth(), rowH * 1.0);
    loopButton.setBounds(0, rowH * 9.5, getWidth(), rowH * 1.0);
}
```