# JUCE DJ App Manual

Table of contents

[JUCE DJ App Manual 1](#_Toc111399068)

[Introduction 2](#_Toc111399069)

[Basic functionality (R1) 2](#_Toc111399070)

[R1A: can load audio files into audio players 2](#_Toc111399071)

[Can play two or more tracks 3](#_Toc111399072)

[Can mix the tracks by varying each of their volumes 4](#_Toc111399073)

[Can speed up and slow down the tracks 4](#_Toc111399074)

[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. 4](#_Toc111399075)

[Component has custom graphics implemented in a paint function 4](#_Toc111399076)

[Component enables the user to control the playback of a deck somehow 5](#_Toc111399077)

[Implementation of a music library component which allows the user to manage their music library 5](#_Toc111399078)

[Component allows the user to add files to their library 5](#_Toc111399079)

[Component parses and displays meta data such as filename and song length 5](#_Toc111399080)

[Component allows the user to search for files 6](#_Toc111399081)

[Component allows the user to load files from the library into a deck 6](#_Toc111399082)

[The music library persists so that it is restored when the user exits then restarts the application 6](#_Toc111399083)

[Implementation of a complete custom GUI 6](#_Toc111399084)

[GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls 6](#_Toc111399085)

[Summary 9](#_Toc111399086)

Custom Knobs: <https://github.com/remberg/juceCustomSliderSample>

Colors: <https://www.ccoderun.ca/programming/doxygen/juce/namespacejuce_1_1Colours.html>

## Introduction

OtoDecks is a desktop DJ application written in C++ leveraging the JUCE Framework, which offers ready-made, customizable (GUI and audio related) components that are perfect for our use case.

The application allows to:

* Import tracks from the user’s computer to the application playlist
* Save the playlist so that it is reloaded as saved even if the app is closed and reopened
* Load a track to the left deck and control it autonomously. Load a track to the right deck and control it autonomously.
* Play two tracks simultaneously and control their speed, volume, track position, and reverb properties through intuitive GUI components such as knobs and sliders.

This application was developed as final project for the class of Object-Oriented Programming and thus the different application components are divided into a series of header and .cpp files, following the best practice of separation of concerns. Code documentation can be found for each function within the header files (following [these guidelines](https://developer.lsst.io/cpp/api-docs.html)).

In this report I will describe the structure of the application, highlighting how its features were implemented.

## Basic functionality (R1)

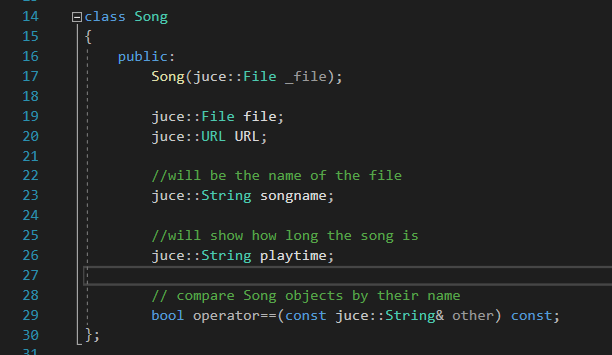
OtoDecks contains all the basic functionality shown in class. These include:

* loading audio files into the audio player decks (R1A)
* playing 2 tracks at the same time (R1B)
* mixing the tracks by varying the volume of each track (R1C)
* speeding up or down the tracks (R1D) Can load audio files into audio players

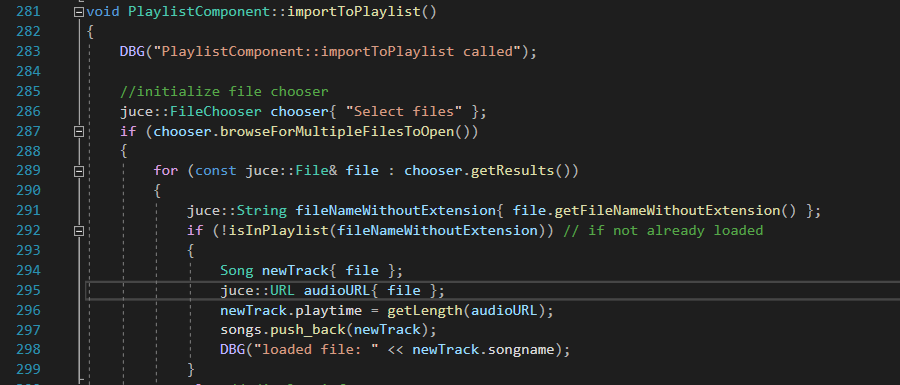
### R1A: can load audio files into audio players

The main way to do it is to use the JUCE URL class which combined with the use of the Input Output Stream ( **#include <fstream>** ) and can load URLs to browse files on the PC. The JUCE URL class also brings other useful functions with itself.

Loading audio files has been additionally enhanced. A new class called Song has been created, respectively in **Song.cpp** and **Song.h**, which have the following properties and member functions: constructor (used when new song is added), playtime, songname, file, URL and a Boolean operator to compare song names.



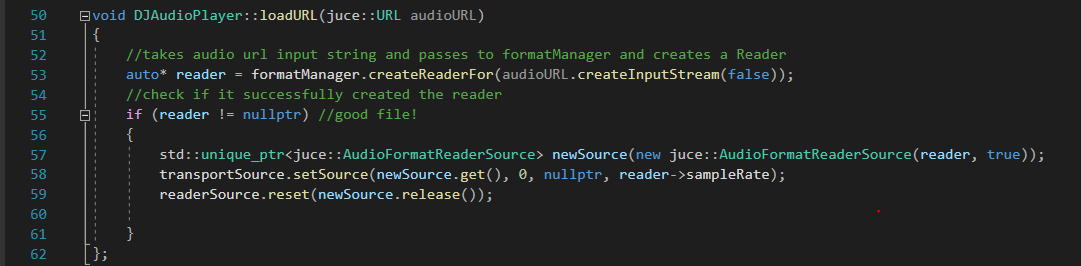
When a new song is imported into the app, the **importToPlaylist()** function creates a new Song object and adds it to the playlist by constructing it with a new URL. This happens in **PlaylistComponent.cpp** circa ln281:



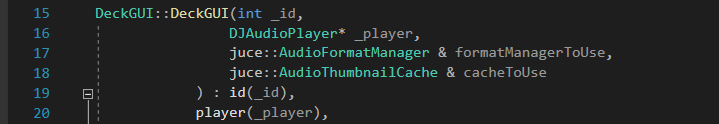
The song can be added and played in the audio players constructed in **DJAudioPlayer.cpp** and **.h**.

### Can play two or more tracks

The DJAudioPlayer.cpp and .h files are responsible for loading in the passed Song object and setting the **readerSource** and **transportSource** to the newly added **juce::AudioFormatReaderSource** unique pointer. This is circa ln50 in **DJAudioPlayer.cpp**:



Since the app constructs two separate DeckGUI’s, each DeckGUI has an autonomous DJAudioPlayer and, therefore, allows two tracks to be played at the same time.



Creating a third DeckGUI component to mix a third DJAudioPlayer would become redundant as commercial DJ decks usually have two decks for mixing.

### Can mix the tracks by varying each of their volumes

A volume slider has been added circa ln72 in **DeckGUI.cpp**. A function called **sliderValueChanged** (circa ln193 in **DeckGUI.cpp**) would then run whenever the slider value is changed by the user. The function takes in a Slider object as a parameter and check if the slider changed is the one that controls the volume. If it is, the function would get the value of the slider and pass it in to the constructed DJAudioPlayer object inside that DeckGUI.

The passed value from the Slider to the DJAudioPlayer must be between 0 and 1. The DJAudioPlayer’s member function **setGain()** controls the output volume by taking in a **double** as a parameter, checks it for error values below 0 and above 1, and then set’s the gain to the **transportSource** to the value. This is circa ln70 in **DJAudioPlayer.cpp**.

### Can speed up and slow down the tracks

Similar to the slider for volume control, a slider is created to control the speed of the playback, circa ln58 in **DeckGUI.cpp**. The same function sliderValueChanged would then get the value of that slider and pass it to the **setSpeed()** member function of the DJAudioPlayer object of that specific DeckGUI. The **setSpeed()** function, circa ln86 in **DJAudioPlayer.cpp** takes in a double as a parameter and after checking if that value is within the acceptable constraints, it passes it to the **.setResamplingRatio()** method of the **resampleSource**. This is how the song would speed up or slow down in its playback speed.

## 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.

The customised deck control is achieved in several ways. Firstly, a “Forward step” and “Backward step” buttons are added to the DeckGUI. The second additional control mechanisms are the AxisModificator plots. Last but not least, above the WaveformDisplay component, a PlaybackBar is added.

### Component has custom graphics implemented in a paint function

The PlaybackBar redraws a Slider object with a new LookAndFeel style by also using the player’s **getPositionRelative()** to check where the track is at. Data is passed to the PlaybackBar to update the Slider happens in the **timerCallback()** function in **DeckGUI.cpp** circa ln227. This makes it not appear as a slider but as an interactive progress bar.

### Component enables the user to control the playback of a deck somehow

The “Forward step” and “Backward step” buttons, when pressed, increment/decrement the DJAudioPlayer object’s **getPositionRelative()** value which in turn makes the song play a bit seconds later in or earlier in the playback. This happens circa ln171 in **DeckGUI.cpp**.

The AxisModificator plots allow the user to drag inside and set reverb values that change the output of the song playing. This does not change the playtime of the track but affects the music output. This is created as a separate Class in **AxisModificator.cpp** and **.h** files and are instantiated in DeckGUI.

The PlaybackBar is created in **PlaybackBar.cpp** and **.h** files. The PlaybackBar allows the user to drag it or click inside it to change where in the track the player should continue playing from. This is seen in **sliderValueChanged()** in **DeckGUI.cpp** circa ln203.

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

A music library component is added to the app. The component is called PlaylistComponent and is created in PlaylistComponent.cpp and PlaylistComponent.h.

### Component allows the user to add files to their library

The component can import audio files from the user’s PC. This is carried out in the **importToPlaylist()** function circa ln281 in **PlaylistComponent.cpp**.It instantiates a juce::FileChooser which then uses its **browseForMultipleFilesToOpen()** method to check if more than one files are chosen. It then runs a loop to iterate over each file and if the file is not already in the playlist, saves the file as a Song object (created in **Song.cpp** and **Song.h**) and adds it to the ***songs* std::vector<Song>** which constitutes the actual playlist. If the selected song is already in the playlist, the user will see an alert message.

### Component parses and displays meta data such as filename and song length

The **importToPlaylist()** runs a loop to iterate over each file and saves the file name without the extension of the file. It also checks how long the track is by using the **getLength()** function circa ln296 in **PlaylistComponent.cpp**. The **getLength()** is defined circa ln337 which in turn uses a calculative function called **secondsToMinutes()** defined circa ln350 which converts seconds into minutes. The PlaylistComponent instantiates a DJAudioPlayer object called **metadataParser** which the Playlist uses to call its function **.getLengthInSeconds** which in turn uses the same method **.getLengthInSeconds** but from the **AudioTransportSource** which is in DJAudioPlayer. This is how the length is retrieved.

Once all songs’ playtime and names are saved, the **paintCell()** function circa ln129 in PlaylistComponent.cpp iterates over them and fills the cells of the playlist table with this metadata.

### Component allows the user to search for files

A search box is created circa ln32 in PlaylistComponent.cpp.Searching happens in the **searchPlaylist()** function circa ln369 which first of all checks if the query string is empty. If it is not empty, run the **whereInPlaylist()** function by passing it the query as a parameter. Circa ln388, **whereInPlaylist()** iterates over the vector of Song objects called **songs**, if any part of a Song’s name contains the user’s query, the searched song is found and the function returns an integer. Back in **searchPlaylist(),** this integer is passed onto the playlist’s **selecrRow()** function which the uses the integer to count the rows and mark the n-th row as selected. This indicates to the user his searched song.

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

After selectin a song from the playlist either by mouse click or by search, the user can add it to either DeckGUI by using a button like “Add to left” or “Add to right”.This will call the **loadInPlayer()** function circa ln 259 in **PlaylistComponent.cpp**. It takes a DeckGUI object as a parameter, searches the playlist’s rows for which is the currently selected song and calls the DeckGUI’s **loadFile()** function by passing the Song’s URL. If no song is selected, the user will see a prompt window.

### The music library persists so that it is restored when the user exits then restarts the application

A function **savePlaylist()** is defined circa ln219 in PlaylistComponent.cpp which saves a .csv file on the user’s PC. Then for each Song object in the songs vector, it will save the path to the file and the length of the song in the **.csv**.

When the user closes the JUCE app and the PlaylistComponent is destroyed, in **~PlaylistComponent()** circa ln62 in PlaylistComponent.cpp, it calls the **savePlaylist()** function.

When the user start up the app again and PlaylistComponent is instantiated, **loadPlaylist()** is called circa ln45. **loadPlaylist()** is defined circa ln.234 in **PlaylistComponent.cpp**, it opens the .csv created from **savePlaylist()**, reads each line and loads each File by passing in the file path. This way all Song objects are again pushed into the **songs** vector and the playlist is available to the user again.

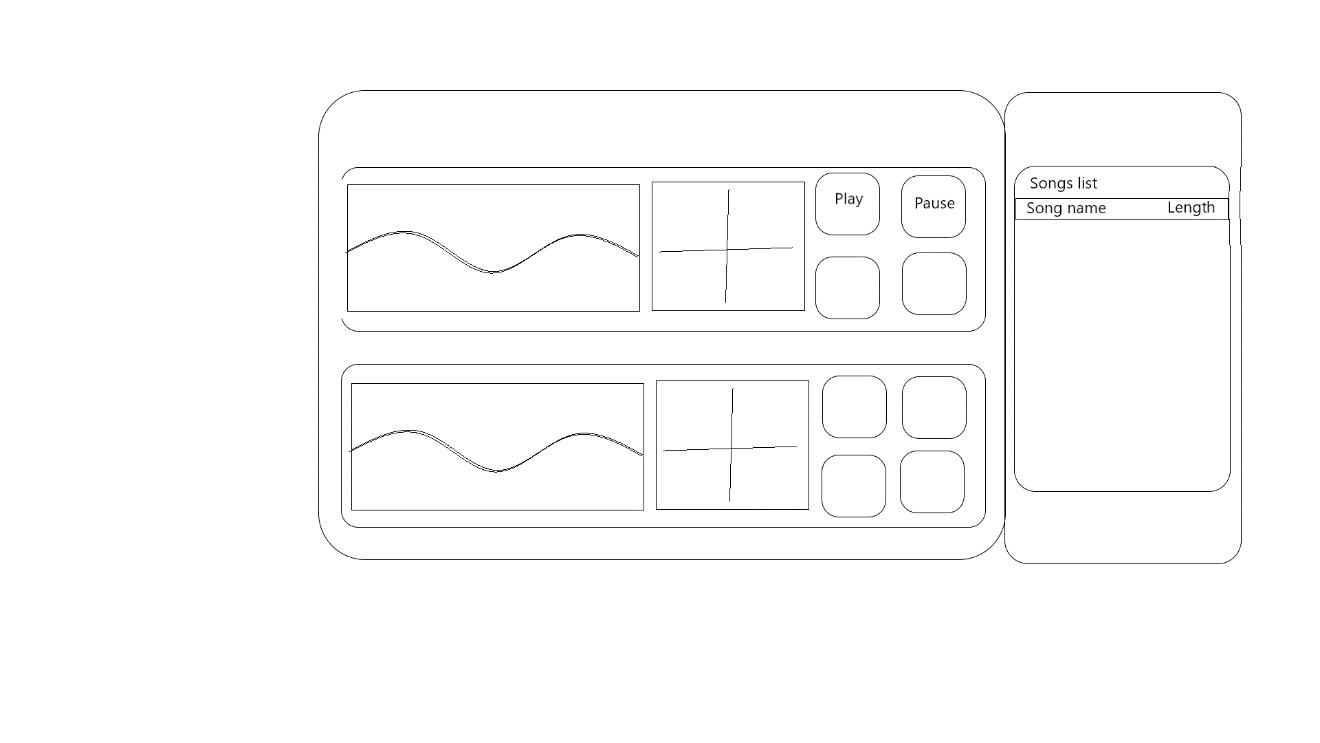
## Implementation of a complete custom GUI

A complete custom GUI is created for the app. Components’ layout is changed, new components added, button and slider styles are changed, main colour scheme is changed.

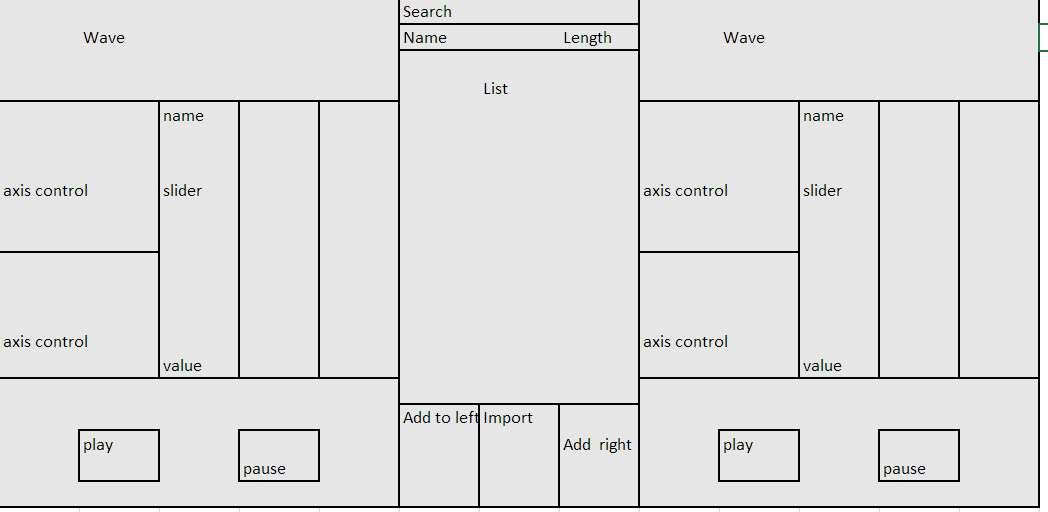
### GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls

These are some initial sketches and design drafts of how to layout the complete app.

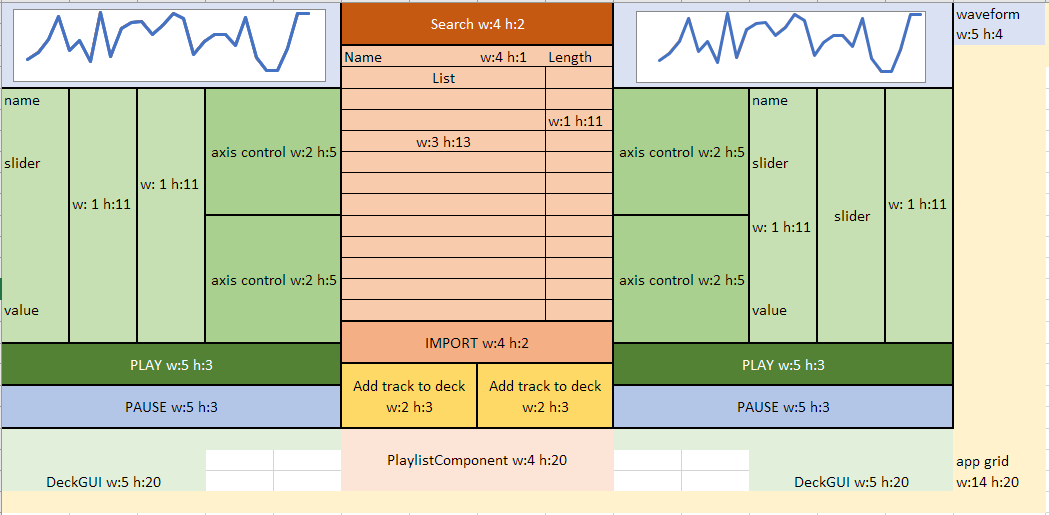
The initial idea was to have one AxisModificator component for each DeckGUI. However, this idea lates changed to having 2 AxisModificators as it turned out JUCE offered several reverb variables to use which allowed this. This first design also did not leave any space for sliders so it needed reworking.



Then the second draft accommodated sliders and two AxisModificators per DeckGUI.

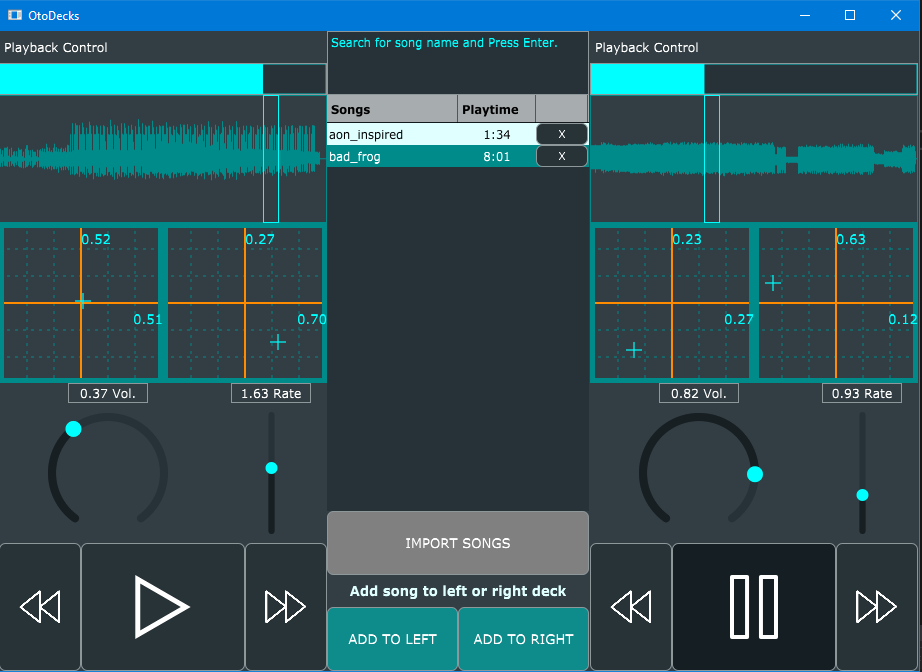


The third draft attempted to split out the width and height “cells” each component would take up. Colour-coding and writing the X and Y values for size of each component would make planning easier.



After having this third draft, it took some user testing and participant feedback to receive some ideas for changes.

The final design implemented “forward” and “backward” button in each DeckGUI, and also a Playback progress bar which users can use to control where the song plays. The AxisModificators are put horizontally together and under them would be two sliders, one of which made rotary. This provided a variety of visual components and made the UI interesting and engaging. The participants were then happy with the final results.



The main colour scheme is made to follow around nuances of cyan.

One of the sliders is made into a Rotary style circa ln.81, and the PlaybackBar mentioned earlier is a more advanced way of styling a Slider.

The Buttons are changed from **TextButton**s into **DrawableButton**s. Drawables are created to load the required .svg images for the button icons. Drawables and DrawableButtons are created circa ln.37 in **DeckGUI.cpp**. The Play button is now checking the state of the player and if it is playing, the Play button turns into a Pause button. Checking whether the button should start or pause the song is done in **buttonClicked()** circa ln.158 in particular. A Boolean called **isOn** is used for this purpose.

## Summary

The JUCE app is an amazing project to work since it allows fun testing and debugging during development. The JUCE framework is a unique way of learning Object Oriented Programming since it offers up GUI and Audio processing while also requiring a good understanding of SOC, Inheritance, overloading and polymorphism.