# CM2005 Object Oriented Programming

# Endterm assignment

## 1 Introduction

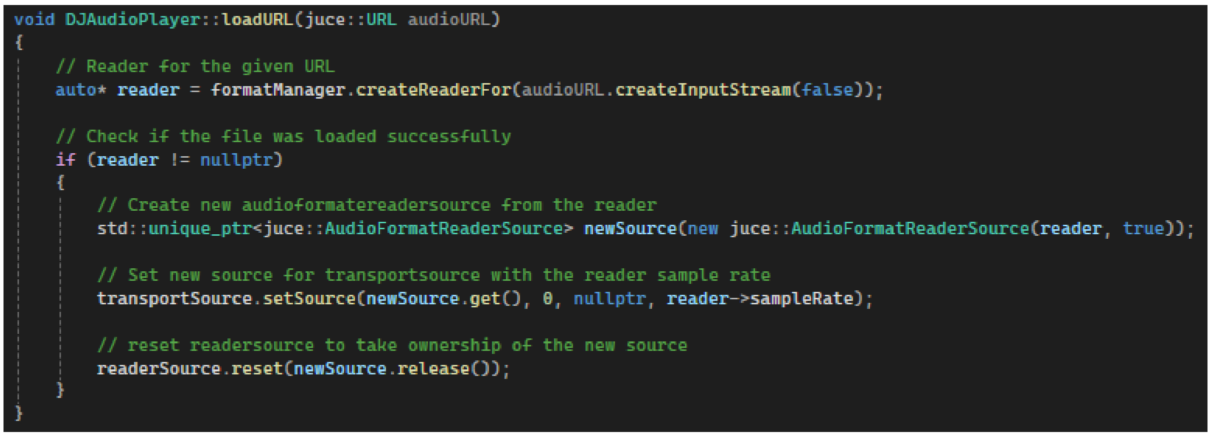
In this report, I will describe the basic functions of the Otodecks DJ application and how it was improved by implementing a new feature that was designed after an actual DJ application and changing the user interface. Initial version of the application let users load and play two audio tracks, control playback speed, and adjust volume. I changed the user interface to improve this by making it visually appealing. In addition, I introduced drum pads, which inspired from a real DJ program and let users mix music using different beats. This report will give an overview of the development of each component.

## 2 Basic Program

As shown in class, I was told to put the basic features of a DJ application into practice. The audio players could play multiple tracks, mix tracks by varying their volume levels, load audio files, and control the speed of each track.

### 2.1 Load Audio Files into Audio Players

There are three ways to load audio files into the system using the application. The DJAudioPlayer component is in charge of controlling the playback of audio. Along with controlling play, pause, and speed, it loads files. By initializing the audio source, DJAudioPlayer's loadURL function makes sure the file is correctly read and ready for playback.

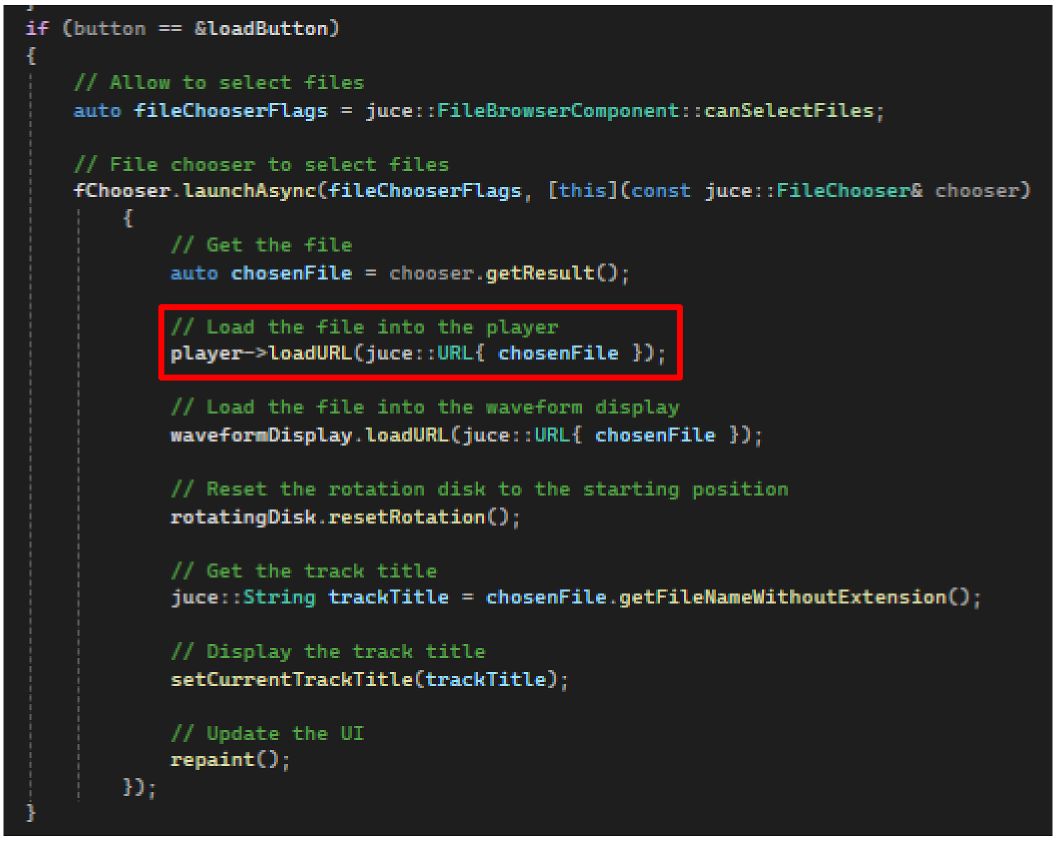


```cpp
void DJAudioPlayer::loadURL(juce::URL audioURL)
{
    // Reader for the given URL
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

    // Check if the file was loaded successfully
    if (reader != nullptr)
    {
        // Create new audioformatereadersource from the reader
        std::unique_ptr<juce::AudioFormatReaderSource> newSource(new juce::AudioFormatReaderSource(reader, true));

        // Set new source for transportsource with the reader sample rate
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);

        // reset readersource to take ownership of the new source
        readerSource.reset(newSource.release());
    }
}
```

*Img 1: DJAudioPlayer::loadURL code*

**Method 1:**

In the first method, tracks are loaded via a button click using the DeckGUI component, which offers a user interface. A file chooser dialog box enables the user to choose an audio file when the Load button is clicked. DeckGUI sends a valid file to DJAudioPlayer for playback if it is selected.

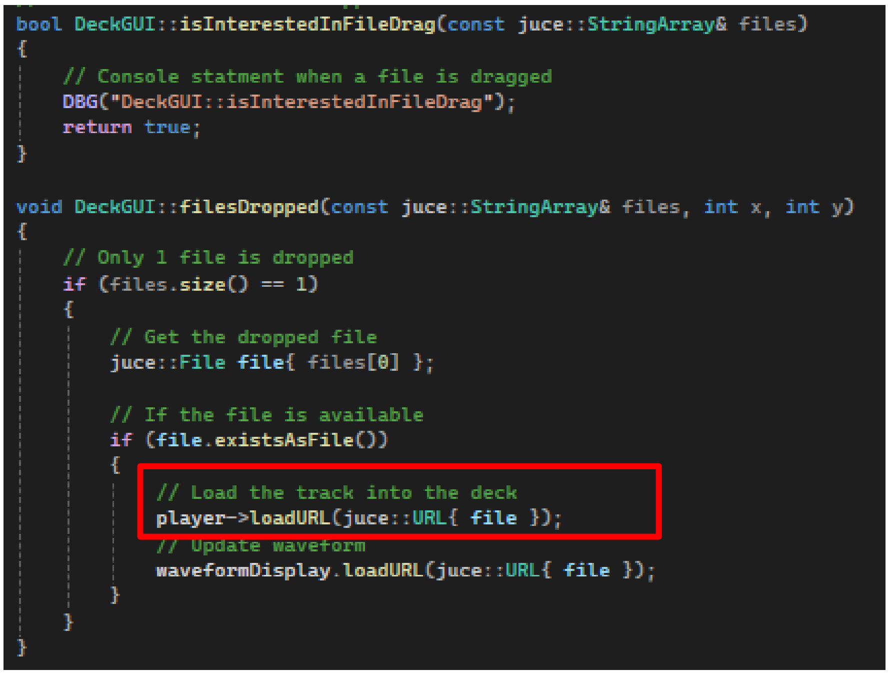


```cpp
if (button == &loadButton)
{
    // Allow to select files
    auto fileChooserFlags = juce::FileBrowserComponent::canSelectFiles;

    // File chooser to select files
    fChooser.launchAsync(fileChooserFlags, [this](const juce::FileChooser& chooser)
        {
            // Get the file
            auto chosenFile = chooser.getResult();

            // Load the file into the player
            player->loadURL(juce::URL{ chosenFile });

            // Load the file into the waveform display
            waveformDisplay.loadURL(juce::URL{ chosenFile });

            // Reset the rotation disk to the starting position
            rotatingDisk.resetRotation();

            // Get the track title
            juce::String trackTitle = chosenFile.getFileNameWithoutExtension();

            // Display the track title
            setCurrentTrackTitle(trackTitle);

            // Update the UI
            repaint();
        });
}
```

*Img 2: DeckGUI::buttonclicked code*

**Method 2:**

With the second method, users can drag and drop an audio file into DeckGUI straight from the file explorer. As with the load button method, DeckGUI records the file path when a file is dropped onto the deck and sends it to DJAudioPlayer. This method provides a more user-friendly way to load tracks without bringing up a file dialog.

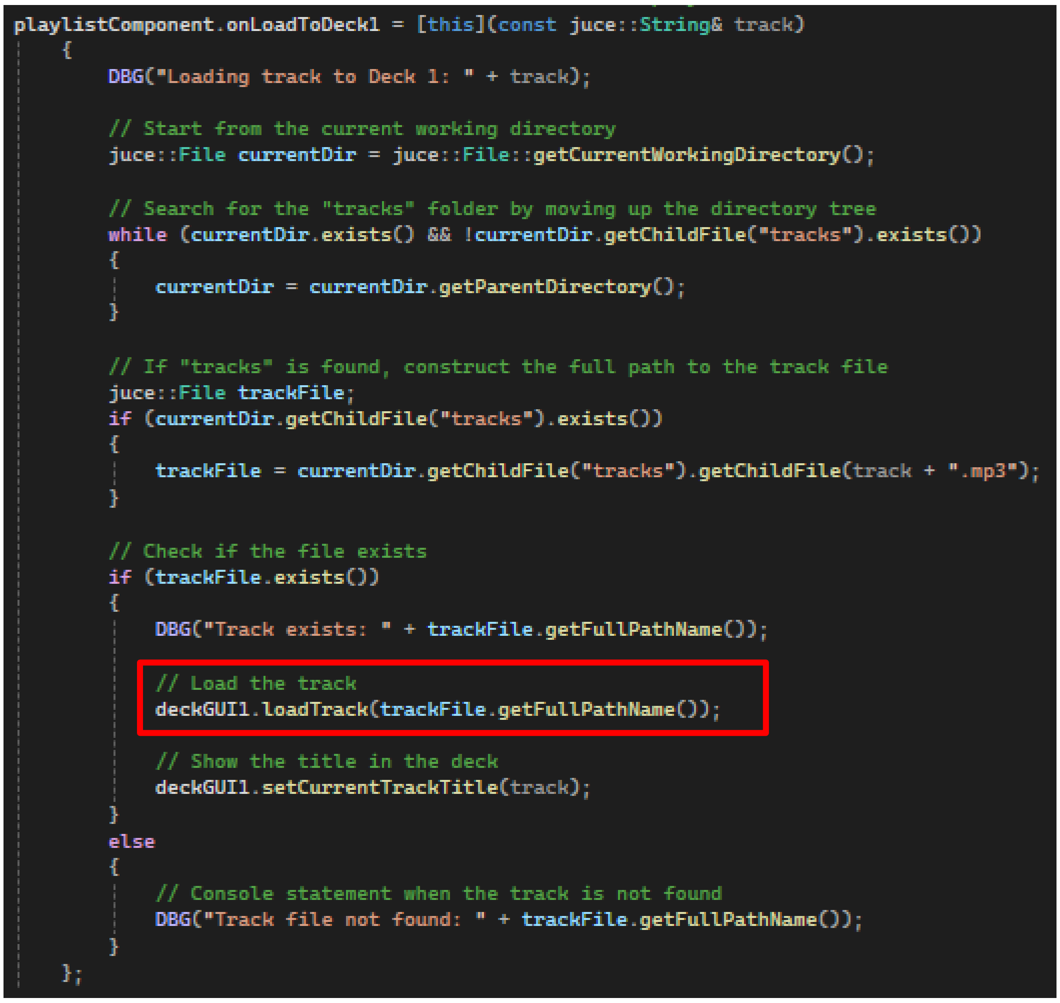


```
bool DeckGUI::isInterestedInFileDrag(const juce::StringArray& files)
{
    // Console statment when a file is dragged
    DBG("DeckGUI::isInterestedInFileDrag");
    return true;
}

void DeckGUI::filesDropped(const juce::StringArray& files, int x, int y)
{
    // Only 1 file is dropped
    if (files.size() == 1)
    {
        // Get the dropped file
        juce::File file{ files[0] };

        // If the file is available
        if (file.existsAsFile())
        {
            // Load the track into the deck
            player->loadURL(juce::URL{ file });
            // Update waveform
            waveformDisplay.loadURL(juce::URL{ file });
        }
    }
}
```

*Img 3: DeckGUI::isInterestedInFileDrag and DeckGUI::filesDropped code*

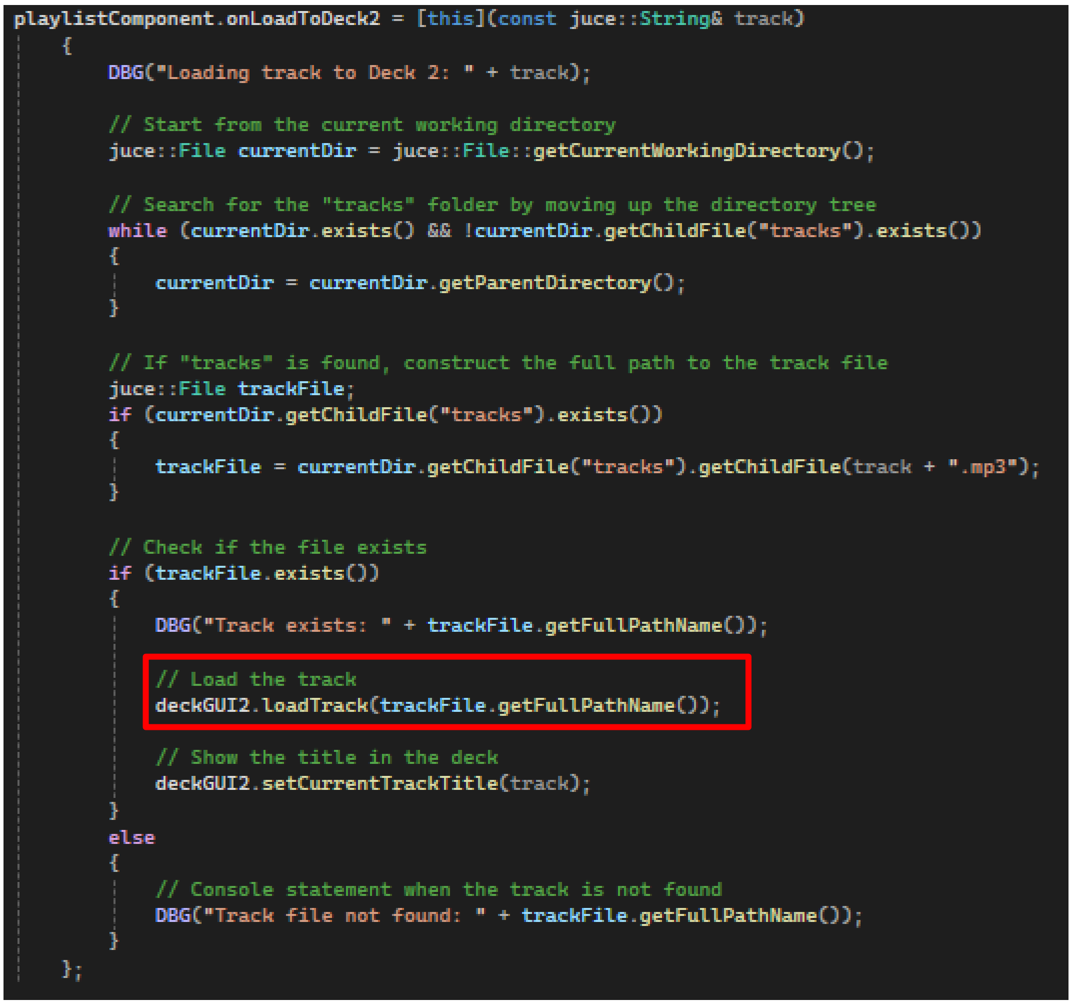
**Method 3:**

Using the third method, user can choose a track from the Playlist Table Component and click a specific Load button to load it into Deck 1 or Deck 2. The file path of the chosen track is retrieved and passed to DeckGUI by the MainComponent's onLoadToDeck1 or onLoadToDeck2 callback. The tracks are also saved in a folder called tracks. If the folder does not exist, the program would create the folder. To guarantee smooth track selection from an existing list of available tracks, DeckGUI then makes a call to DJAudioPlayer.



```
playlistComponent.onLoadToDeck1 = [this](const juce::String& track)
    {
        DBG("Loading track to Deck 1: " + track);

        // Start from the current working directory
        juce::File currentDir = juce::File::getCurrentWorkingDirectory();

        // Search for the "tracks" folder by moving up the directory tree
        while (currentDir.exists() && !currentDir.getChildFile("tracks").exists())
        {
            currentDir = currentDir.getParentDirectory();
        }

        // If "tracks" is found, construct the full path to the track file
        juce::File trackFile;
        if (currentDir.getChildFile("tracks").exists())
        {
            trackFile = currentDir.getChildFile("tracks").getChildFile(track + ".mp3");
        }

        // Check if the file exists
        if (trackFile.exists())
        {
            DBG("Track exists: " + trackFile.getFullPathName());

            // Load the track
            deckGUI1.loadTrack(trackFile.getFullPathName());

            // Show the title in the deck
            deckGUI1.setCurrentTrackTitle(track);
        }
        else
        {
            // Console statement when the track is not found
            DBG("Track file not found: " + trackFile.getFullPathName());
        }
    };
```

*Img 4: MainComponent onLoadToDeck1 code*

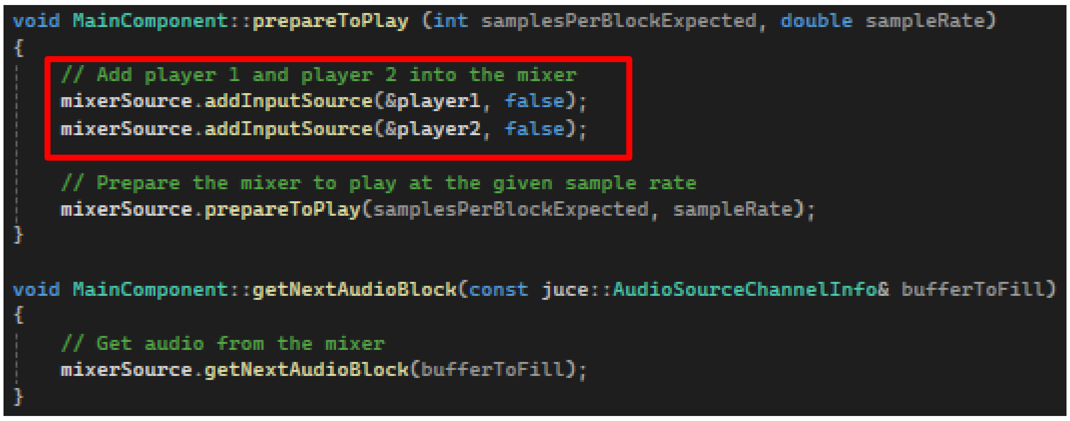


```
playlistComponent.onLoadToDeck2 = [this](const juce::String& track)
    {
        DBG("Loading track to Deck 2: " + track);

        // Start from the current working directory
        juce::File currentDir = juce::File::getCurrentWorkingDirectory();

        // Search for the "tracks" folder by moving up the directory tree
        while (currentDir.exists() && !currentDir.getChildFile("tracks").exists())
        {
            currentDir = currentDir.getParentDirectory();
        }

        // If "tracks" is found, construct the full path to the track file
        juce::File trackFile;
        if (currentDir.getChildFile("tracks").exists())
        {
            trackFile = currentDir.getChildFile("tracks").getChildFile(track + ".mp3");
        }

        // Check if the file exists
        if (trackFile.exists())
        {
            DBG("Track exists: " + trackFile.getFullPathName());

            // Load the track
            deckGUI2.loadTrack(trackFile.getFullPathName());

            // Show the title in the deck
            deckGUI2.setCurrentTrackTitle(track);
        }
        else
        {
            // Console statement when the track is not found
            DBG("Track file not found: " + trackFile.getFullPathName());
        }
    };
```

*Img 5: MainComponent onLoadToDeck2 code*

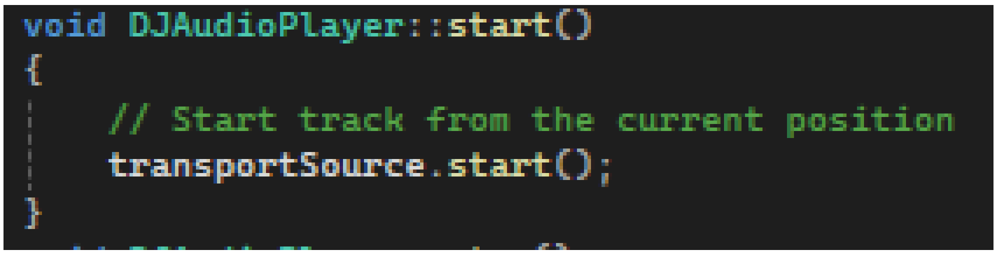
## 2.2 Play Two Tracks

Using two separate decks, the application enables users to play multiple tracks at once. To play two tracks simultaneously, each DeckGUI instance runs its own instance of DJAudioPlayer. The juce::MixerAudioSource, which combines audio from both decks for simultaneous playback, is responsible of providing this functionality. A button press in each DeckGUI initiates the play functionality, which calls DJAudioPlayer::start() to start playing the track that is currently loaded.



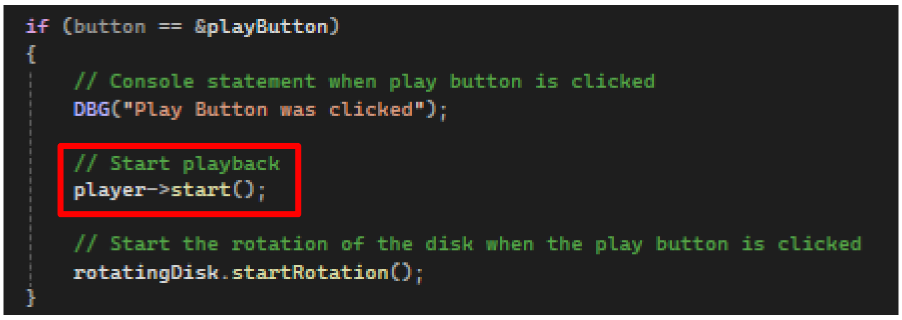
```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    // Add player 1 and player 2 into the mixer
    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);

    // Prepare the mixer to play at the given sample rate
    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);
}

void MainComponent::getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)
{
    // Get audio from the mixer
    mixerSource.getNextAudioBlock(bufferToFill);
}
```

*Img 6: MainComponenet::prepareToPlay and MainComponent::getNextAudioBlock code*



```
void DJAudioPlayer::start()
{
    // Start track from the current position
    transportSource.start();
}
```

*Img 7: DJAudioPlayer::start() code*

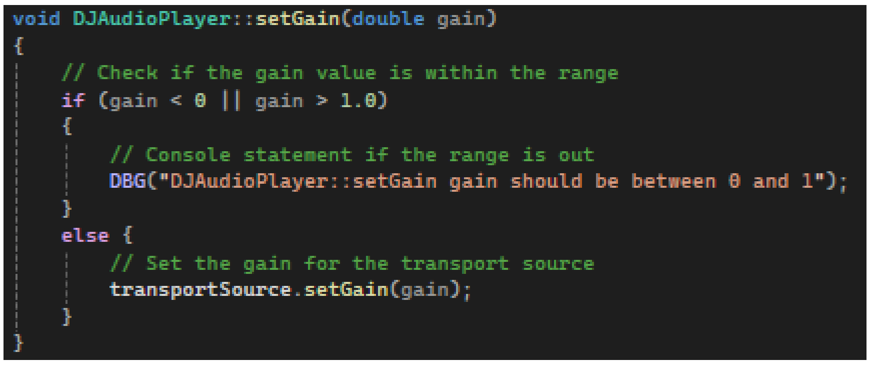


```
if (button == &playButton)
{
    // Console statement when play button is clicked
    DBG("Play Button was clicked");

    // Start playback
    player->start();

    // Start the rotation of the disk when the play button is clicked
    rotatingDisk.startRotation();
}
```

*Img 8: DeckGUI::buttonclicked code*

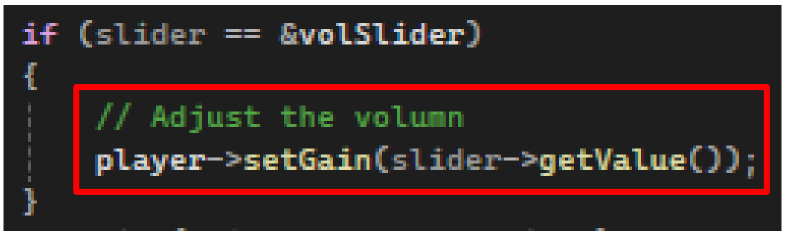
## 2.3 Mix Tracks by Varying their Volumes

By changing the volume of each deck separately, the application enables track mixing to improve the user experience and give greater control over the audio output. This is done by adjusting the gain of the related DJAudioPlayer using a slider control in the DeckGUI. Users can blend tracks together for a smooth listening experience by adjusting the volume sliders.



```cpp
void DJAudioPlayer::setGain(double gain)
{
    // Check if the gain value is within the range
    if (gain < 0 || gain > 1.0)
    {
        // Console statement if the range is out
        DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
    }
    else {
        // Set the gain for the transport source
        transportSource.setGain(gain);
    }
}
```

*Img 9: DJAudioPlayer::setGain code*

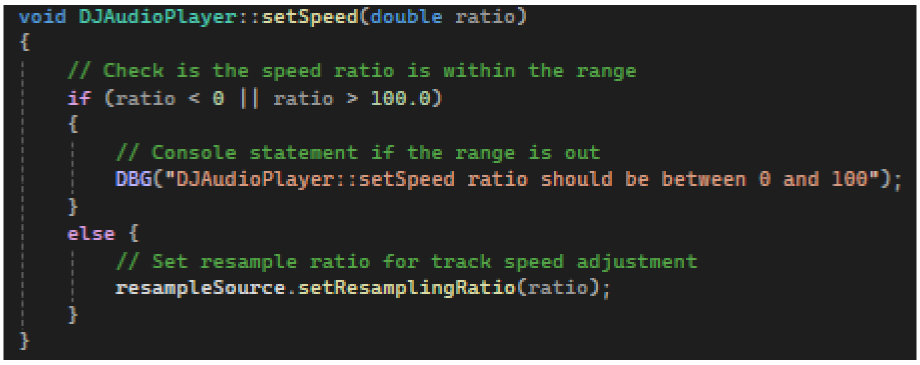


```cpp
if (slider == &volSlider)
{
    // Adjust the volumn
    player->setGain(slider->getValue());
}
```

*Img 10: DeckGUI::sliderValueChanged code*

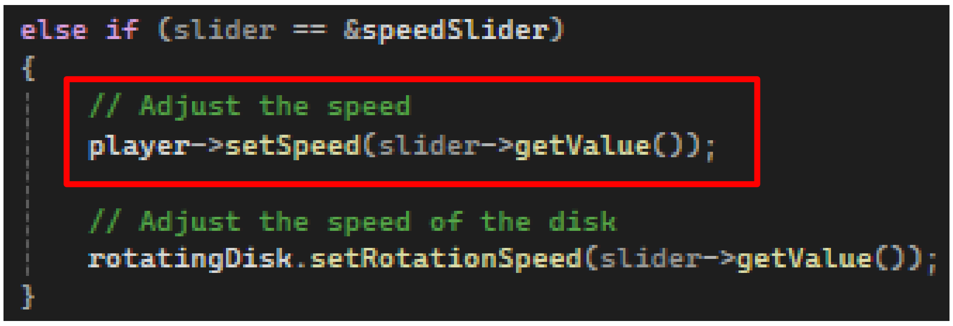
## 2.4 Speed Up and Slow Down Tracks

Using a speed control slider, users can adjust the DJAudioPlayer's playback rate by speeding up or slowing down tracks. In a live mix, this is helpful for beatmatching or changing the tempo of tracks.



```cpp
void DJAudioPlayer::setSpeed(double ratio)
{
    // Check is the speed ratio is within the range
    if (ratio < 0 || ratio > 100.0)
    {
        // Console statement if the range is out
        DBG("DJAudioPlayer::setSpeed ratio should be between 0 and 100");
    }
    else {
        // Set resample ratio for track speed adjustment
        resampleSource.setResamplingRatio(ratio);
    }
}
```

*Img 11: DJAudioPlayer::setSpeed code*



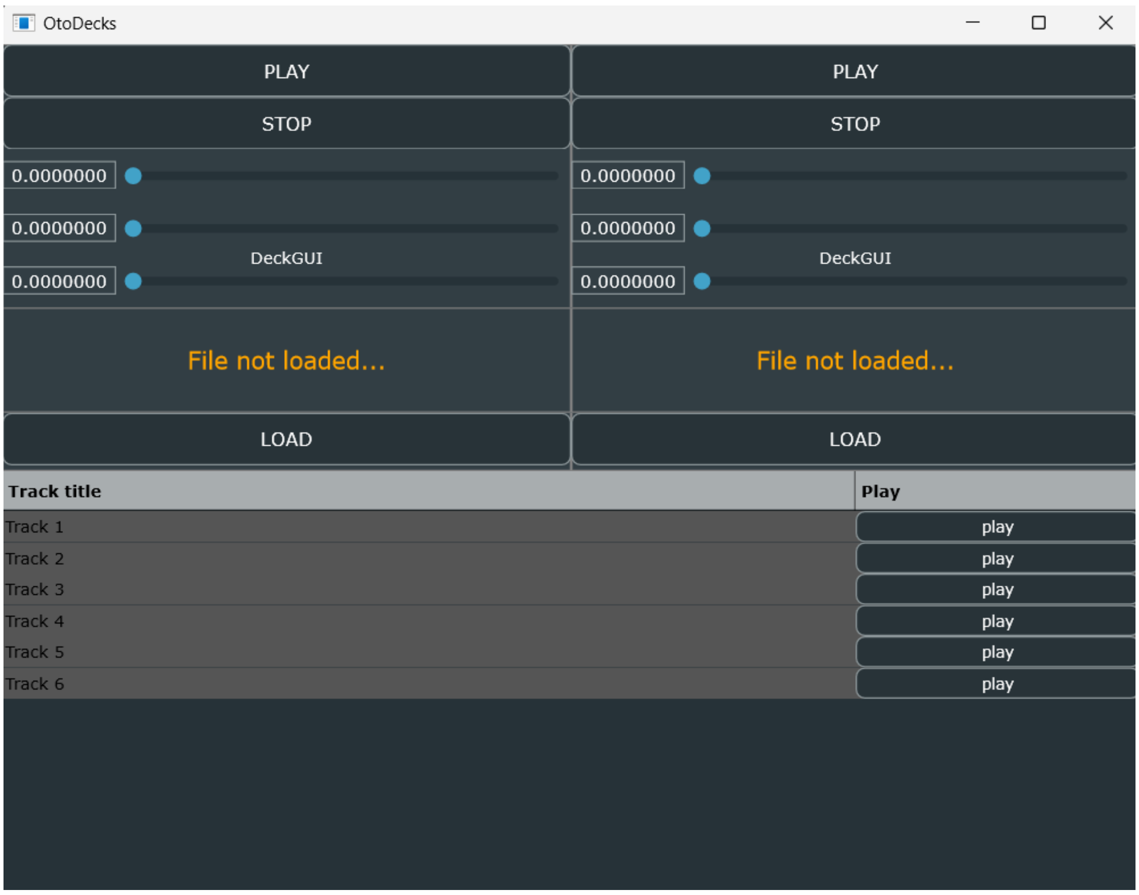
```cpp
else if (slider == &speedSlider)
{
    // Adjust the speed
    player->setSpeed(slider->getValue());

    // Adjust the speed of the disk
    rotatingDisk.setRotationSpeed(slider->getValue());
}
```

*Img 10: DeckGUI::sliderValueChanged code*

# 3 Custom User Interface

## 3.1 Layout is different from basic DeckGUI

Compared to the simple DeckGUI used in class, the GUI layout of my project has undergone an extensive change. For comparison, I have included the user interface of the class layout below.



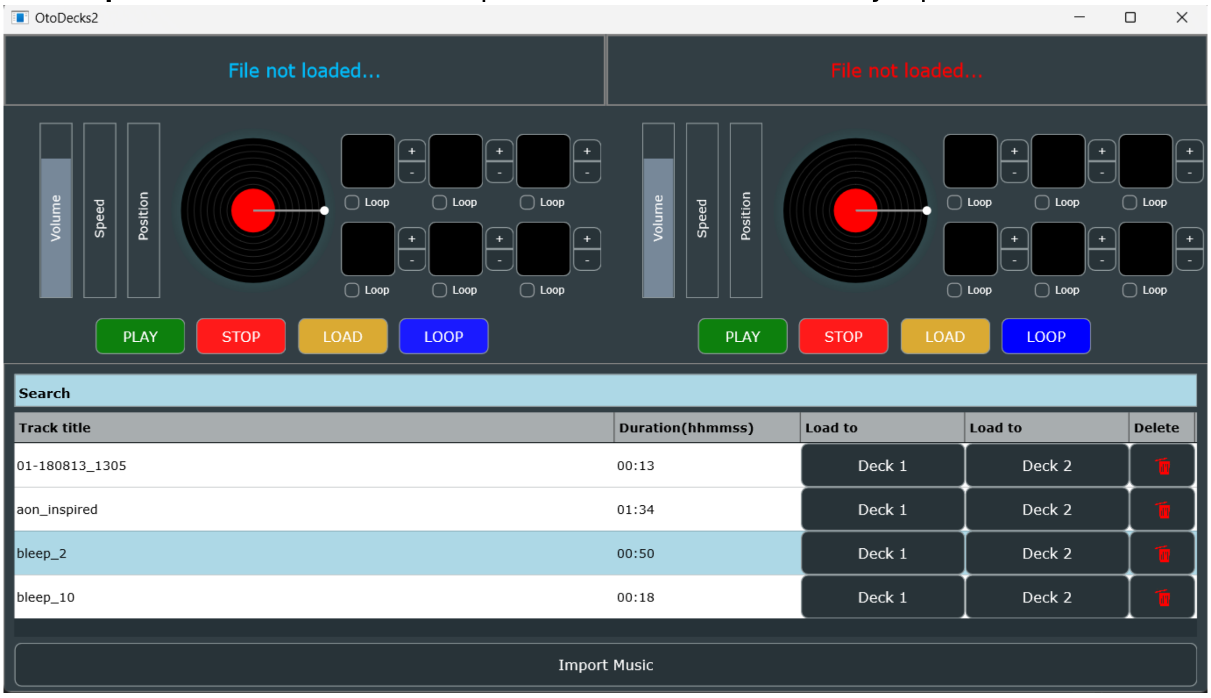
| Track title | Play |
|---|---|
| Track 1 | play |
| Track 2 | play |
| Track 3 | play |
| Track 4 | play |
| Track 5 | play |
| Track 6 | play |

*Img 11: Basic DeckGUI (Shown In Class)*

In my customized version,

1. **Rotating Disk**: To improve the DJ-style experience and make it more engaging and aesthetically pleasing, I have added a rotating disk visualization in my modified version.
2. **Drum Pads**: I have included drum pads with loop functionality so that users can dynamically create and modify loops. Additionally, each drum pad has its own volume slider.
3. **Sliders**: For improved control, the position, speed, and volume sliders have been modified.
4. **Search Box**: To facilitate track selection, a search box has been included.
5. **Playlist Tracks**: This feature gives users more control over how they manage their music collection by allowing them to import new tracks into the playlist.

6. **Playlist Buttons**: The table component also has buttons for Deck 1 and Deck 2 to load smoothly onto the corresponding decks and delete buttons to remove tracks.
7. **Waveform**: The waveform display's new colour scheme enhances both its appearance and visibility.
8. **Loop**: Each track now has a loop button that lets users easily repeat tracks.



| Track title | Duration(hhmmss) | Load to | Load to | Delete |
|---|---|---|---|---|
| 01-180813_1305 | 00:13 | Deck 1 | Deck 2 | |
| aon_inspired | 01:34 | Deck 1 | Deck 2 | |
| bleep_2 | 00:50 | Deck 1 | Deck 2 | |
| bleep_10 | 00:18 | Deck 1 | Deck 2 | |

*Img 12: Customise Version of DeckGUI*

Compared to the original version, these improvements make the application more feature-rich and easy to use.

## 3.2 New Event Listeners

I added a number of new event listeners to my project that were not in the original DeckGUI codebase that was used in class. Among the significant additions are:

1. **Loop for track**: With dedicated event listeners, users can turn on looping for particular tracks.

2. **Rotating Disk**: Track playback is used by an event listener to dynamically update the rotating disk's rotation in response to user interactions.

3. **Drum Pad**: Users can activate loop functionality and trigger sound samples by tapping drum pads with event listeners.

4. **Track Import and Search**: Tracks can be filtered by user input in the search box and new tracks can be added to the playlist by event listeners.

5. **Playlist Buttons**: Now, when you click the Deck 1 or Deck 2 button, an event listener loads the chosen track onto the appropriate deck. When a delete button with an event listener is clicked in each track row, the track is deleted from the playlist.

6. **Loop and Volume Controls for drum pads**: Users can dynamically change the volume levels and turn on looping for particular drum pad tracks with dedicated event listeners.

Compared to the original version, these additions greatly increase the application's functionality and make it more interactive. Users now have more control over playback, looping, and track management than just playing and pausing tracks.

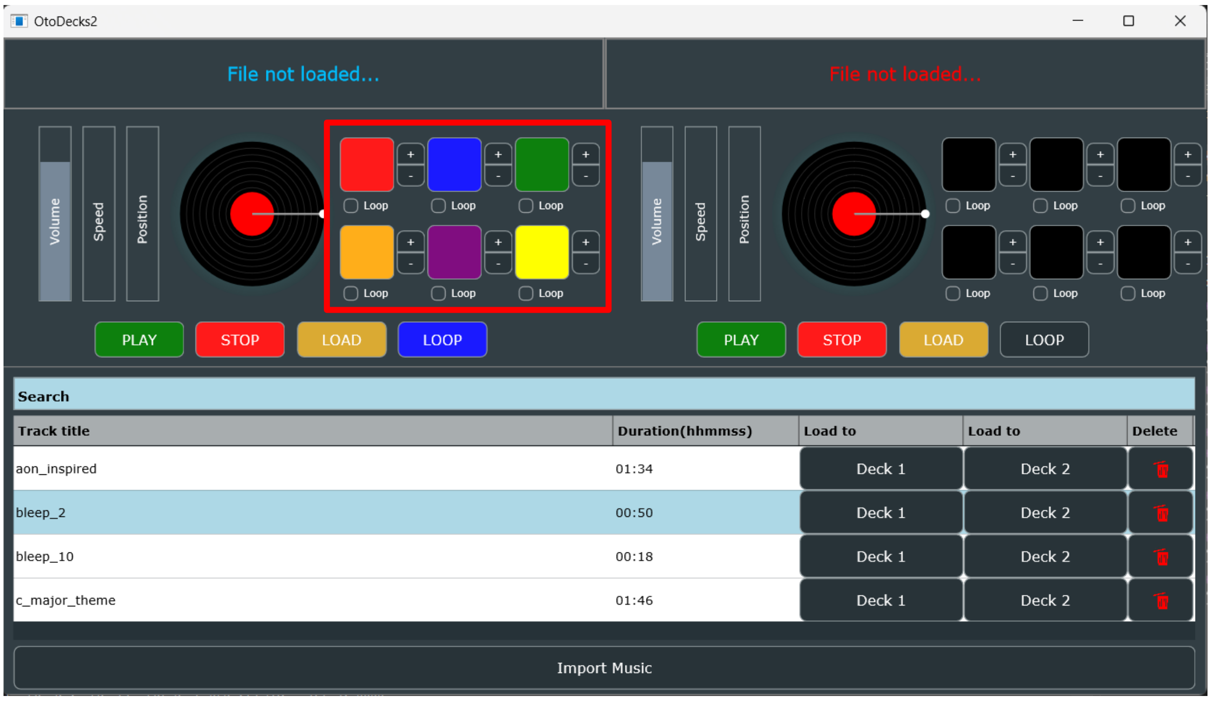
# 4 Drum Pads: Research and Implementation

## 4.1 Research and Feature Selection

I have studied the well-known DJ platform Serato DJ Pro. Track synchronization, beat matching, looping, effects processing, and MIDI controller support are just a few features this program offers. The drum pad sampler was one unique feature. With the help of this feature, DJs can add unique beats and effects to live performances by instantly triggering drum sounds, loops, and samples.

*Img 13: Serato DJ Pro physical dj deck*

This is why the drum pad feature was chosen to be used in this project. Designing a system that would allow users to activate a drum pad and trigger six distinct sounds, change volume levels, enable looping, and get visual feedback was the aim. Live remixing and beat layering are made possible by this feature, which greatly expands creative possibilities.



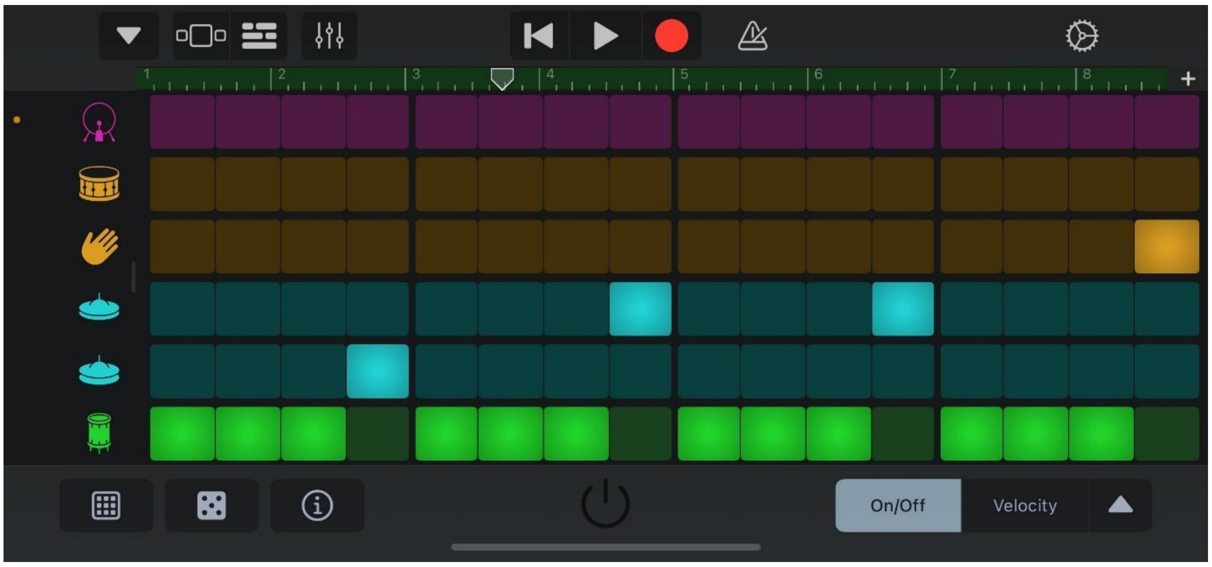
| Track title | Duration(hhmmss) | Load to | Load to | Delete |
|---|---|---|---|---|
| aon_inspired | 01:34 | Deck 1 | Deck 2 | 🗑 |
| bleep_2 | 00:50 | Deck 1 | Deck 2 | 🗑 |
| bleep_10 | 00:18 | Deck 1 | Deck 2 | 🗑 |
| c_major_theme | 01:46 | Deck 1 | Deck 2 | 🗑 |

*Img 14: Cutomized DJ deck*

## 4.2 Engineering the Feature: Classes and OOP Constructs

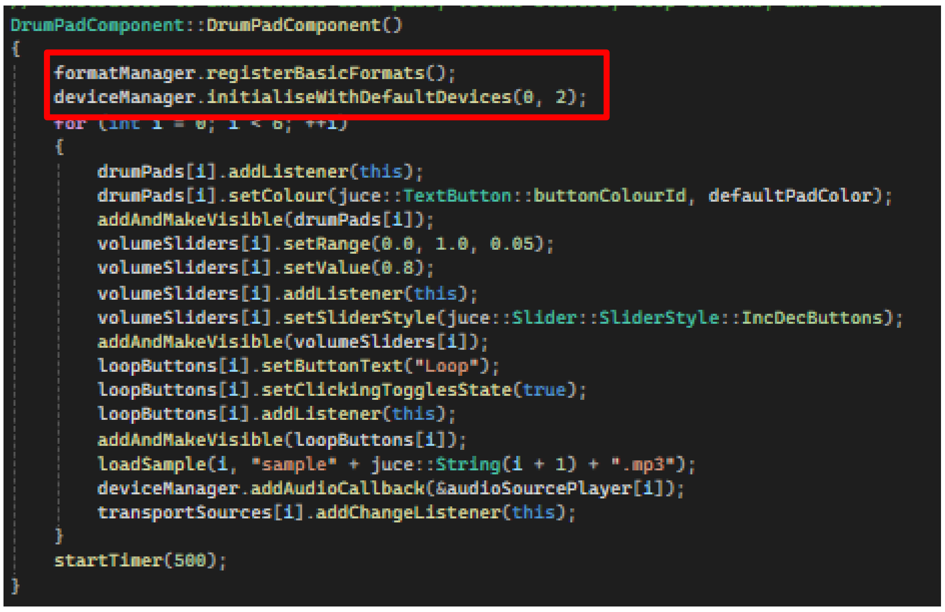
The DrumPadComponent class was used to construct the drum pad feature. Six drum pads, each associated with a distinct sound, are covered in this class. A track is played when a user clicks on a pad. Additionally, the pads feature loop buttons for repeating the track and volume sliders for controlling the volume.

I created the drum sounds in GarageBand and saved them as a mp3 file. All the files are stored in a folder called samples.



*Img 15: Sample track from garageband*

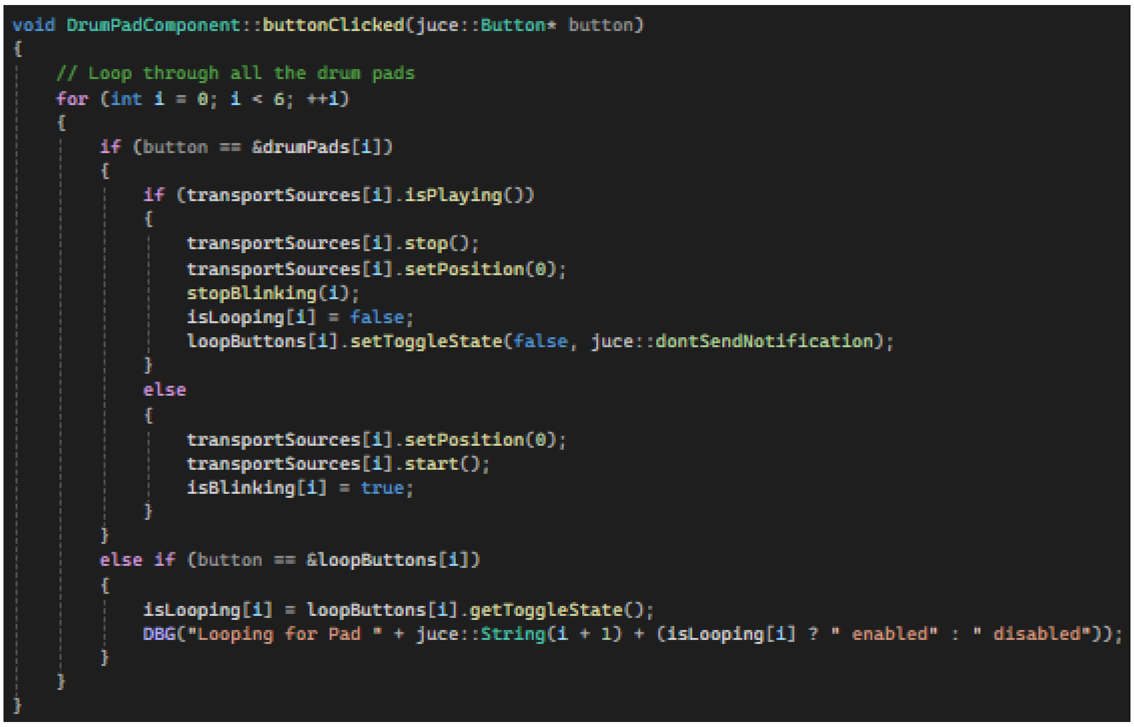
When the application launches, the tracks are loaded. FormatManager, which aids in reading audio files, and deviceManager, which controls sound playback, are used to set up the audio system.



```
DrumPadComponent::DrumPadComponent()
{
    formatManager.registerBasicFormats();
    deviceManager.initialiseWithDefaultDevices(0, 2);
    for (int i = 0; i < 6; ++i)
    {
        drumPads[i].addListener(this);
        drumPads[i].setColour(juce::TextButton::buttonColourId, defaultPadColor);
        addAndMakeVisible(drumPads[i]);
        volumeSliders[i].setRange(0.0, 1.0, 0.05);
        volumeSliders[i].setValue(0.8);
        volumeSliders[i].addListener(this);
        volumeSliders[i].setSliderStyle(juce::Slider::SliderStyle::IncDecButtons);
        addAndMakeVisible(volumeSliders[i]);
        loopButtons[i].setButtonText("Loop");
        loopButtons[i].setClickingTogglesState(true);
        loopButtons[i].addListener(this);
        addAndMakeVisible(loopButtons[i]);
        loadSample(i, "sample" + juce::String(i + 1) + ".mp3");
        deviceManager.addAudioCallback(&audioSourcePlayer[i]);
        transportSources[i].addChangeListener(this);
    }
    startTimer(500);
}
```

*Img 16: DrumPadComponent::DrumPadComponent code*

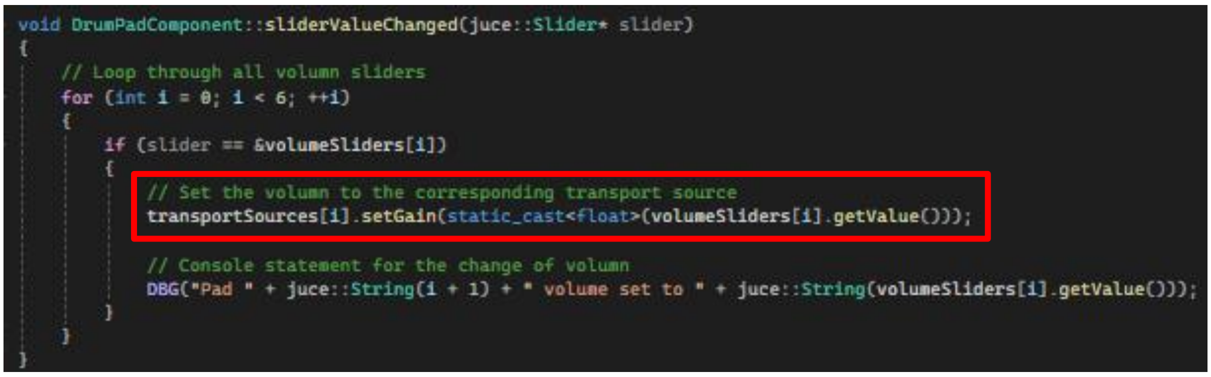
The application determines whether the sample is already playing when a user presses a drum pad. If so, the playback position is reset, and the sound is stopped. When the track reaches the end, it restarts if the loop button is pressed.



```cpp
void DrumPadComponent::buttonClicked(juce::Button* button)
{
    // Loop through all the drum pads
    for (int i = 0; i < 6; ++i)
    {
        if (button == &drumPads[i])
        {
            if (transportSources[i].isPlaying())
            {
                transportSources[i].stop();
                transportSources[i].setPosition(0);
                stopBlinking(i);
                isLooping[i] = false;
                loopButtons[i].setToggleState(false, juce::dontSendNotification);
            }
            else
            {
                transportSources[i].setPosition(0);
                transportSources[i].start();
                isBlinking[i] = true;
            }
        }
        else if (button == &loopButtons[i])
        {
            isLooping[i] = loopButtons[i].getToggleState();
            DBG("Looping for Pad " + juce::String(i + 1) + (isLooping[i] ? " enabled" : " disabled"));
        }
    }
}
```

*Img 17: DrumPadComponent::buttonClicked code*

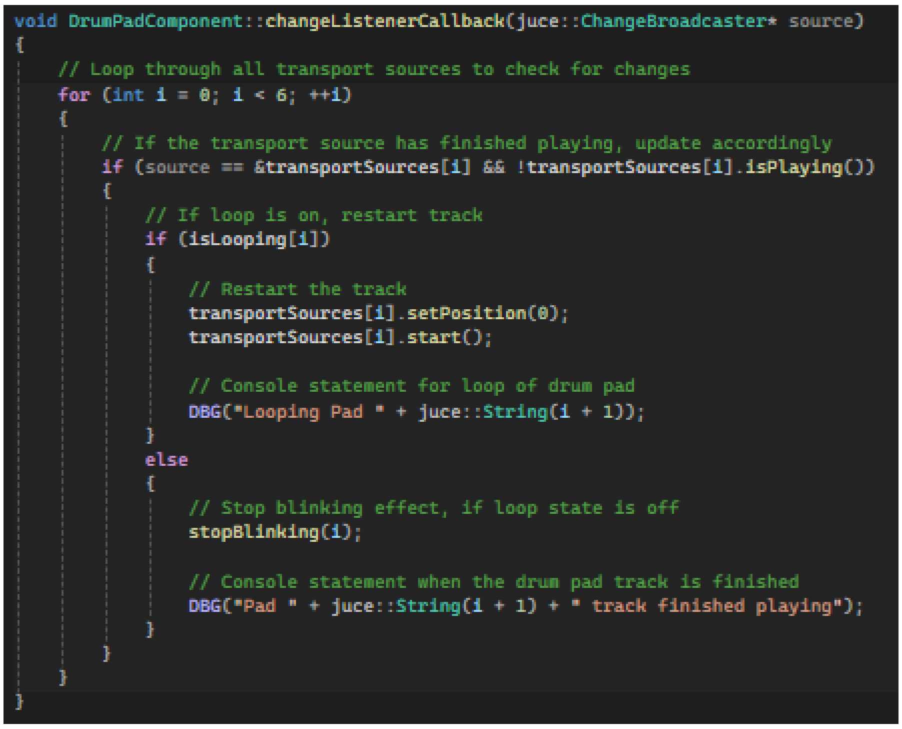
Sliders that dynamically modify each sample's gain are used to implement volume control. To ensure that users can mix drum sounds smoothly, the setGain() function adjusts the playback volume based on the value on the slider.



```cpp
void DrumPadComponent::sliderValueChanged(juce::Slider* slider)
{
    // Loop through all volumn sliders
    for (int i = 0; i < 6; ++i)
    {
        if (slider == &volumeSliders[i])
        {
            // Set the volumn to the corresponding transport source
            transportSources[i].setGain(static_cast<float>(volumeSliders[i].getValue()));

            // Console statement for the change of volumn
            DBG("Pad " + juce::String(i + 1) + " volume set to " + juce::String(volumeSliders[i].getValue()));
        }
    }
}
```

*Img 18: DrumPadComponent::sliderValueChanged code*

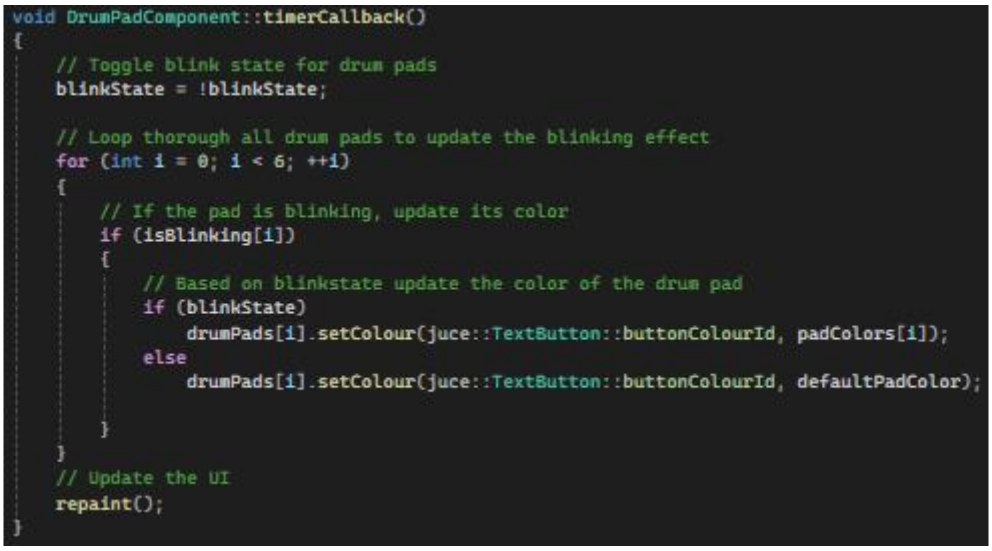
The application monitors playback state changes to manage looping. In the event that a sample ends, and looping is activated, the track starts over. In the event that playback is interrupted, the pad stops to blink.



```cpp
void DrumPadComponent::changeListenerCallback(juce::ChangeBroadcaster* source)
{
    // Loop through all transport sources to check for changes
    for (int i = 0; i < 6; ++i)
    {
        // If the transport source has finished playing, update accordingly
        if (source == &transportSources[i] && !transportSources[i].isPlaying())
        {
            // If loop is on, restart track
            if (isLooping[i])
            {
                // Restart the track
                transportSources[i].setPosition(0);
                transportSources[i].start();

                // Console statement for loop of drum pad
                DBG("Looping Pad " + juce::String(i + 1));
            }
            else
            {
                // Stop blinking effect, if loop state is off
                stopBlinking(i);

                // Console statement when the drum pad track is finished
                DBG("Pad " + juce::String(i + 1) + " track finished playing");
            }
        }
    }
}
```

*Img 19: DrumPadComponent::changeListenerCallback code*

During playing, the drum pads use a blinking effect to give visual feedback. An animation that indicates an active drum pad is produced by a timer that switches the pad color every 500 milliseconds.



```cpp
void DrumPadComponent::timerCallback()
{
    // Toggle blink state for drum pads
    blinkState = !blinkState;

    // Loop thorough all drum pads to update the blinking effect
    for (int i = 0; i < 6; ++i)
    {
        // If the pad is blinking, update its color
        if (isBlinking[i])
        {
            // Based on blinkstate update the color of the drum pad
            if (blinkState)
                drumPads[i].setColour(juce::TextButton::buttonColourId, padColors[i]);
            else
                drumPads[i].setColour(juce::TextButton::buttonColourId, defaultPadColor);

        }
    }
    // Update the UI
    repaint();
}
```

*Img 20: DrumPadComponent::timerCallback code*

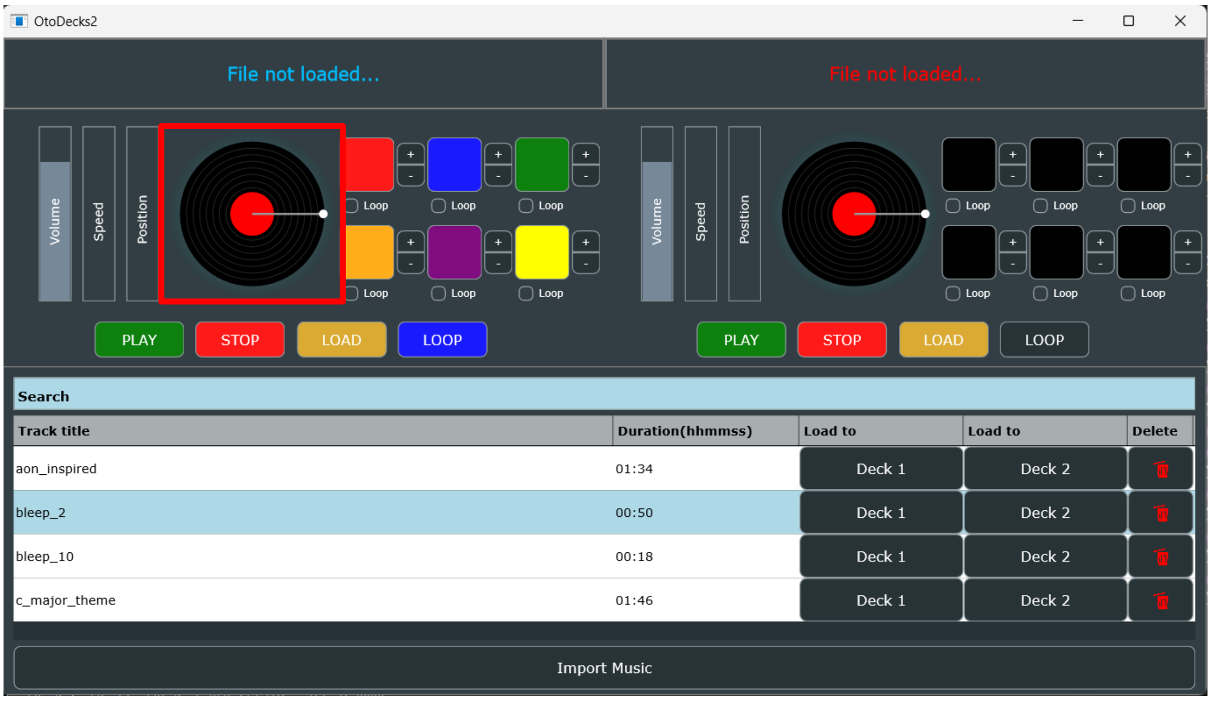
Through the use of object-oriented principles, this feature was designed to be reusable and modular. All functionality is contained in the DrumPadComponent class, which keeps the codebase manageable and structured. While JUCE's event-driven model enables smooth user interaction, AudioTransportSource ensures effective real-time playback control.

## 5 Feature Expansions

Other than the drum pads, several new features were added to improve the user experience and make the application more interesting. These features, which were created to enhance mixing capabilities, give users more control, and make the interface more interactive, were also influenced by actual DJ software.

### 5.1 Rotating disk

A rotating disk animation was added to the application to improve its visual appeal. This feature provides a dynamic visual representation of the track that is currently playing by simulating a real DJ turntable. Users get a more engaging experience because the disk's rotation speed is in sync with the track's playback speed.

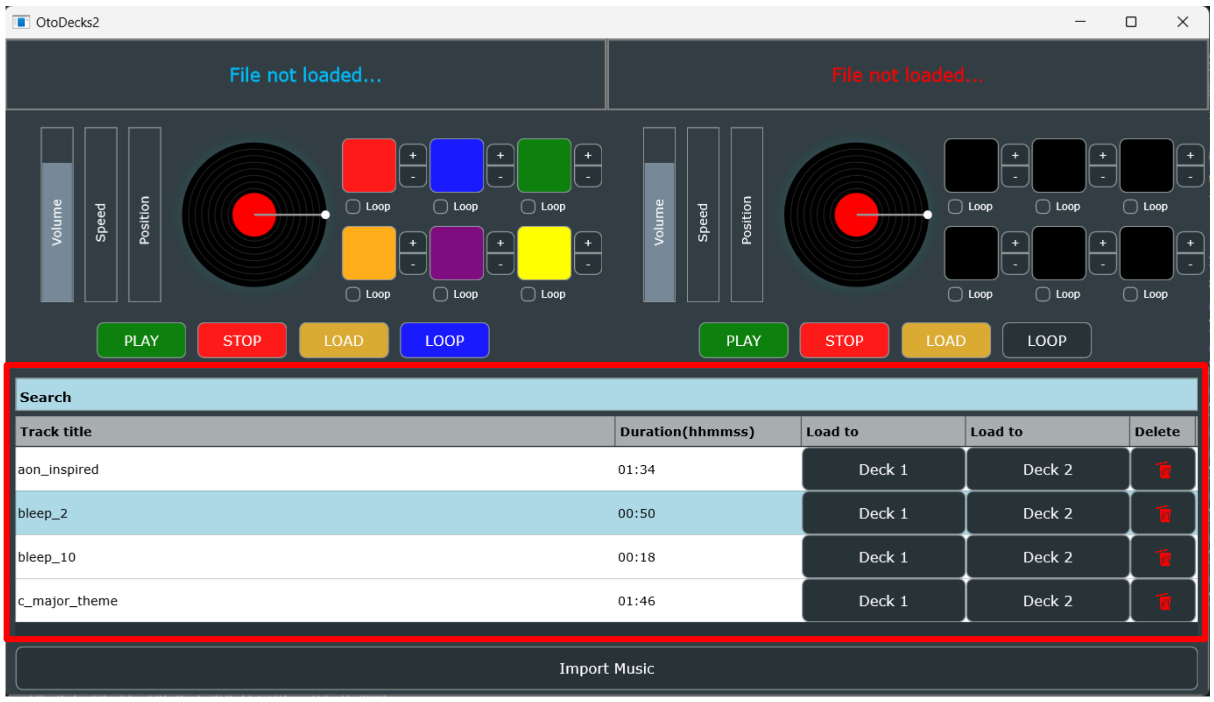


| Track title | Duration(hhmmss) | Load to | Load to | Delete |
|---|---|---|---|---|
| aon_inspired | 01:34 | Deck 1 | Deck 2 | 🗑 |
| bleep_2 | 00:50 | Deck 1 | Deck 2 | 🗑 |
| bleep_10 | 00:18 | Deck 1 | Deck 2 | 🗑 |
| c_major_theme | 01:46 | Deck 1 | Deck 2 | 🗑 |

*Img 21: Customize DJ Deck highlighting the rotating disk*

### 5.2 Playlist Table: Search Box + Button

The playlist section now has a search box to help with track selection. By entering keywords or partial names, users can quickly locate tracks thanks to this feature. This feature improves efficiency and usability, particularly for users managing a large song library. The user now has the access to the track duration as well. The table also contains 3 buttons for loading tracks into deck 1 and deck 2 and the delete button with the trash icon. The icon was made by me using Canva. The icon is stored in the sample folder.



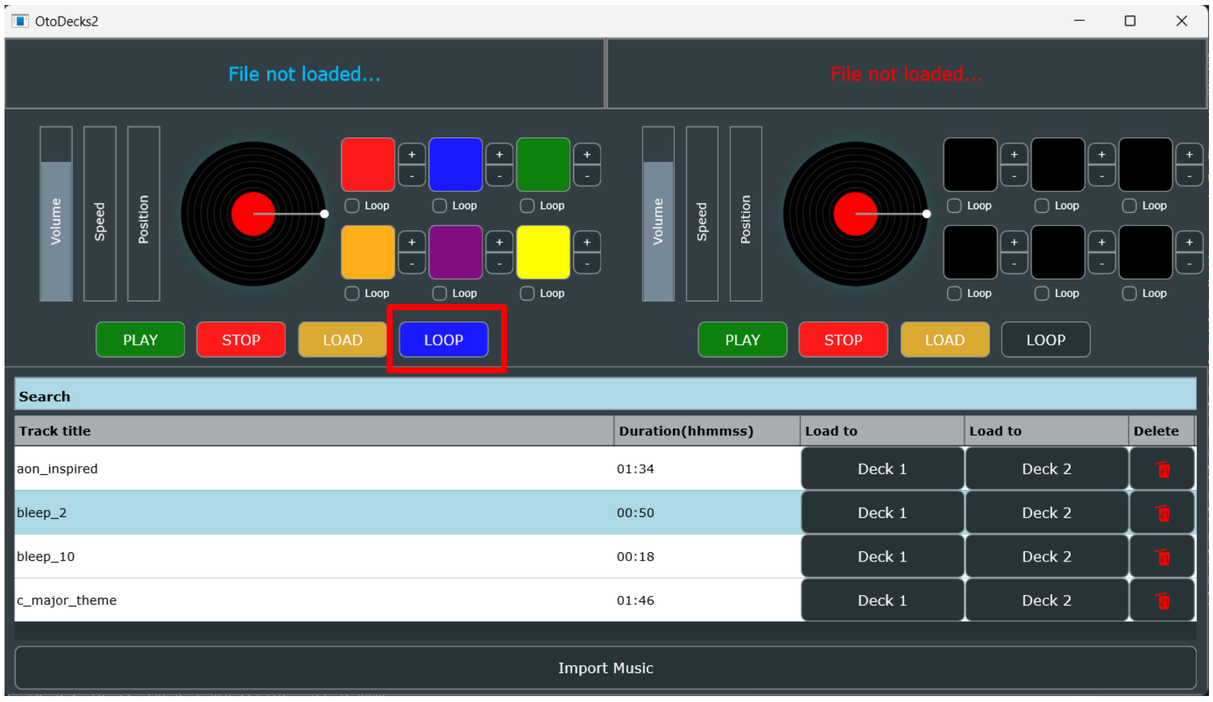
Search

| Track title | Duration(hhmmss) | Load to | Load to | Delete |
|---|---|---|---|---|
| aon_inspired | 01:34 | Deck 1 | Deck 2 | 🗑 |
| bleep_2 | 00:50 | Deck 1 | Deck 2 | 🗑 |
| bleep_10 | 00:18 | Deck 1 | Deck 2 | 🗑 |
| c_major_theme | 01:46 | Deck 1 | Deck 2 | 🗑 |

*Img 22: Customize DJ Deck highlighting the search box*

## 5.3 Loop Track

To let users repeat a particular track, a looping feature was added. When remixing or extending a specific beat, this feature is crucial. Dedicated buttons allow users to set loop points, and the application makes sure the track plays continuously over and over. JUCE's audio buffer management and exact timing controls were used to implement the looping functionality.



Search

| Track title | Duration(hhmmss) | Load to | Load to | Delete |
|---|---|---|---|---|
| aon_inspired | 01:34 | Deck 1 | Deck 2 | 🗑 |
| bleep_2 | 00:50 | Deck 1 | Deck 2 | 🗑 |
| bleep_10 | 00:18 | Deck 1 | Deck 2 | 🗑 |
| c_major_theme | 01:46 | Deck 1 | Deck 2 | 🗑 |

*Img 23: Customize DJ Deck highlighting the loop button*

## 6 Conclusion

In summary, the Otodecks DJ application was developed by adding new features to improve usability and creativity while also implementing essential DJ functionalities. Features like drum pads, a rotating disk, a search box, and looping controls were added to enhance the user interface. All things considered, the project gave practical experience in creating an entertaining DJ application and showcased JUCE's audio development capabilities.

## 7 Overall Experience

I learned a lot from this project and was able to get better at audio programming and JUCE. The development process became interesting and satisfying as a result of overcoming obstacles like adding new features and improving the user interface. Future projects will benefit greatly from the practical experience with sound processing and user interface design.

## 8 References

Lamity, K. (2022). *OtoDecks*. Available at: https://github.com/kevlamity/OtoDecks [Accessed 20 Feb 2025]

Ezema, E. (2022). *TheDJisOP*. Available at: https://github.com/Ezema/TheDJisOP [Accessed 20 Feb 2025]

Karnik, M., Oakley, I. & Nisi, V. (sept 2013). *Performing Online and Offline: How DJs Use Social Networks*. Available at: https://www.researchgate.net/publication/256971853_Performing_Online_and_Offline_How_DJs_Use_Social_Networks [Accessed 21 Feb 2025]

Apple. *GarageBand*. Available at: https://www.apple.com/mac/garageband/ [Accessed 27 Feb 2025]

Serato. *Hercules DJControl Inpulse T7*. Available at: https://serato.com/dj/hardware/hercules-djcontrol-inpulse-t7 [Accessed 28 Feb 2025]