1. Abstract

This report presents an in-depth analysis of the development and enhancement of a DJ application built using the JUCE framework. The project achieves core functionalities—such as multi-track audio loading, simultaneous playback, individual volume control, and speed adjustments—while also incorporating advanced features inspired by professional DJ software. These include loop control, fast forward/rewind operations, dynamic display of track names and playback times, enhanced slider scrubbing, and a comprehensive playlist management system. The report discusses the implementation details, underlying logic, and design decisions made throughout the project.

2. Introduction

The primary objective of this project was to develop a robust DJ application that not only fulfills the baseline requirements of loading, playing, and mixing multiple audio tracks but also extends its functionality with a customized user interface and innovative features. From the outset, the design emphasized modularity and separation of concerns, ensuring that each component—such as the audio player, deck interface, playlist management system, and waveform display—addresses a specific aspect of the overall functionality. The application leverages object-oriented programming (OOP) principles and event-driven methodologies to create a dynamic and interactive user experience, with the JUCE framework providing the necessary multimedia and UI capabilities.

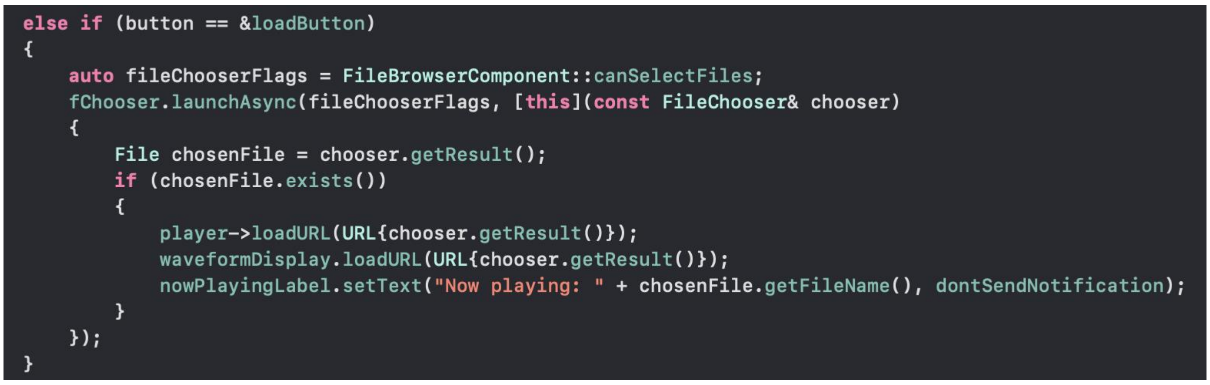
3. Basic Functionality

3.1 Loading Audio Files (R1A)

When a user clicks the "Load" button on the deck interface, a file chooser dialog is displayed, allowing the selection of an audio file. Once a file is selected, it is loaded into the corresponding audio player via the loadURL() method. This method creates an AudioFormatReaderSource to facilitate smooth playback while simultaneously updating the waveform display to reflect the loaded file. The logic behind the audio player is structured to manage playback seamlessly, and the visual update provides immediate feedback to the user.

From DeckGUI.cpp – Load Button Handler:

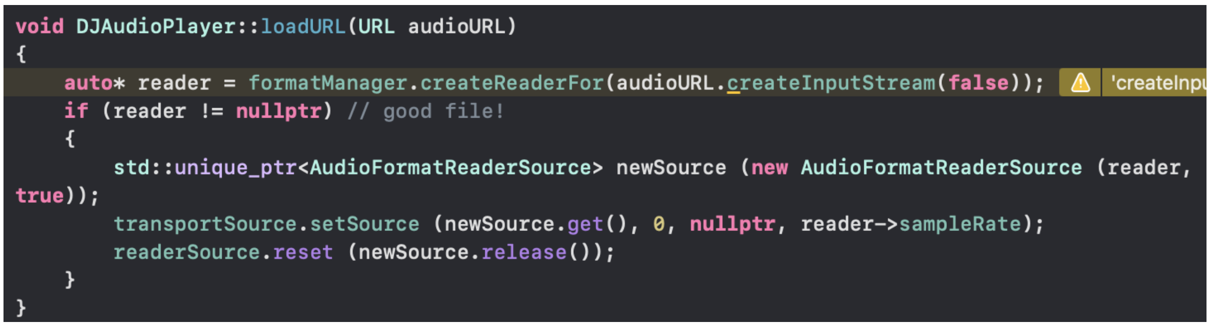
This snippet shows the file chooser launching and how the chosen file is loaded into both the audio player and waveform display.



```
else if (button == &loadButton)
{
    auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
    fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
    {
        File chosenFile = chooser.getResult();
        if (chosenFile.exists())
        {
            player->loadURL(URL{chooser.getResult()});
            waveformDisplay.loadURL(URL{chooser.getResult()});
            nowPlayingLabel.setText("Now playing: " + chosenFile.getFileName(), dontSendNotification);
        }
    });
}
```

From DJAudioPlayer.cpp – Loading URL:

This snippet shows how the audio file is loaded and prepared for playback.



```
void DJAudioPlayer::loadURL(URL audioURL)
{
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));  ⚠ 'createInpu
    if (reader != nullptr) // good file!
    {
        std::unique_ptr<AudioFormatReaderSource> newSource (new AudioFormatReaderSource (reader,
true));
        transportSource.setSource (newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset (newSource.release());
    }
}
```

3.2 Playing Multiple Tracks (R1B)

To support multi-track playback, the application instantiates two independent audio player objects, each associated with its own deck interface. These players are connected to a mixer component that aggregates the audio outputs from both decks, enabling simultaneous playback. This design allows users to manipulate each track separately while ensuring that the final audio output is a cohesive mix.

From MainComponent.cpp – Deck and Mixer Setup:

This excerpt demonstrates adding two decks and connecting them to the mixer for simultaneous playback.

```
//==============================================================================
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
    player2.prepareToPlay(samplesPerBlockExpected, sampleRate);

    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);
}
```

3.3 Volume Mixing (R1C)

Volume control is implemented via individual sliders on each deck. Movement of a slider triggers the setGain() method on the corresponding audio player, allowing for precise control over each track's volume before they are mixed together. This granular control enables the DJ to balance levels effectively.

From DeckGUI.cpp – Volume Slider Configuration and Listener:

This snippet configures the volume slider and shows how its value is used to control gain.

```
// Configure sliders
volSlider.setRange(0, 100, 1);
volSlider.setValue(50);
```

```
void DeckGUI::sliderValueChanged (Slider* slider)
{
    if (slider == &volSlider)
    {
        player->setGain(volSlider.getValue() / 100.0);
    }
```

3.4 Playback Speed Control (R1D)

The speed control functionality is achieved by linking a speed slider in the deck interface to a method in the audio player that adjusts the playback speed. The method modifies the resampling ratio, which in turn changes the tempo of the audio playback in real time.

From DeckGUI.cpp – Speed Slider Configuration and Listener:

This shows how the speed slider is set up and linked to the audio player.

```
speedSlider.setRange(0.5, 2.0, 0.5);
speedSlider.setValue(1.0);
speedSlider.setNumDecimalPlacesToDisplay(2);
```

```
else if (slider == &speedSlider)
{
    player->setSpeed(speedSlider.getValue());
}
```

From DJAudioPlayer.cpp – Adjusting Playback Speed:

This snippet adjusts the resampling ratio based on the speed value.

```
void DJAudioPlayer::setSpeed(double ratio)
{
  if (ratio < 0 || ratio > 100.0)
    {
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 100" << std::endl;
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

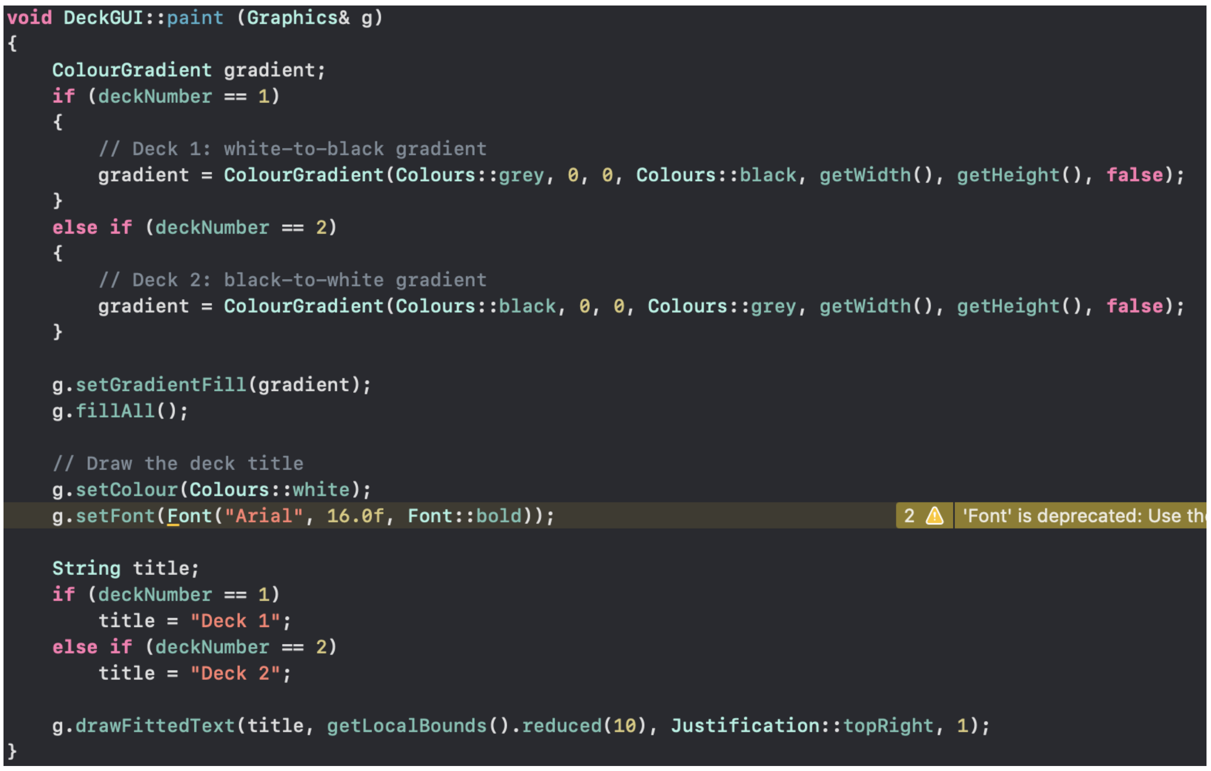
4. Customized User Interface

4.1 Redesigned Layout and Visual Enhancements (R2A)

The user interface has been significantly redesigned to elevate both aesthetics and functionality. Departing from the basic prototype provided in class, the new deck interface now features additional control elements such as Pause, Restart, Loop, Rewind (by 5 seconds), and Fast Forward (by 5 seconds) buttons. Each deck is distinguished by its unique gradient background, which not only enhances the visual appeal but also provides a visual cue for deck identification. Complementary labels, such as “Now Playing” and a dynamic time display, keep the user informed about the current track and its progress. The waveform display has also been enhanced to support customizable gradient and waveform colors.

From DeckGUI.cpp – Painting the Deck Interface:

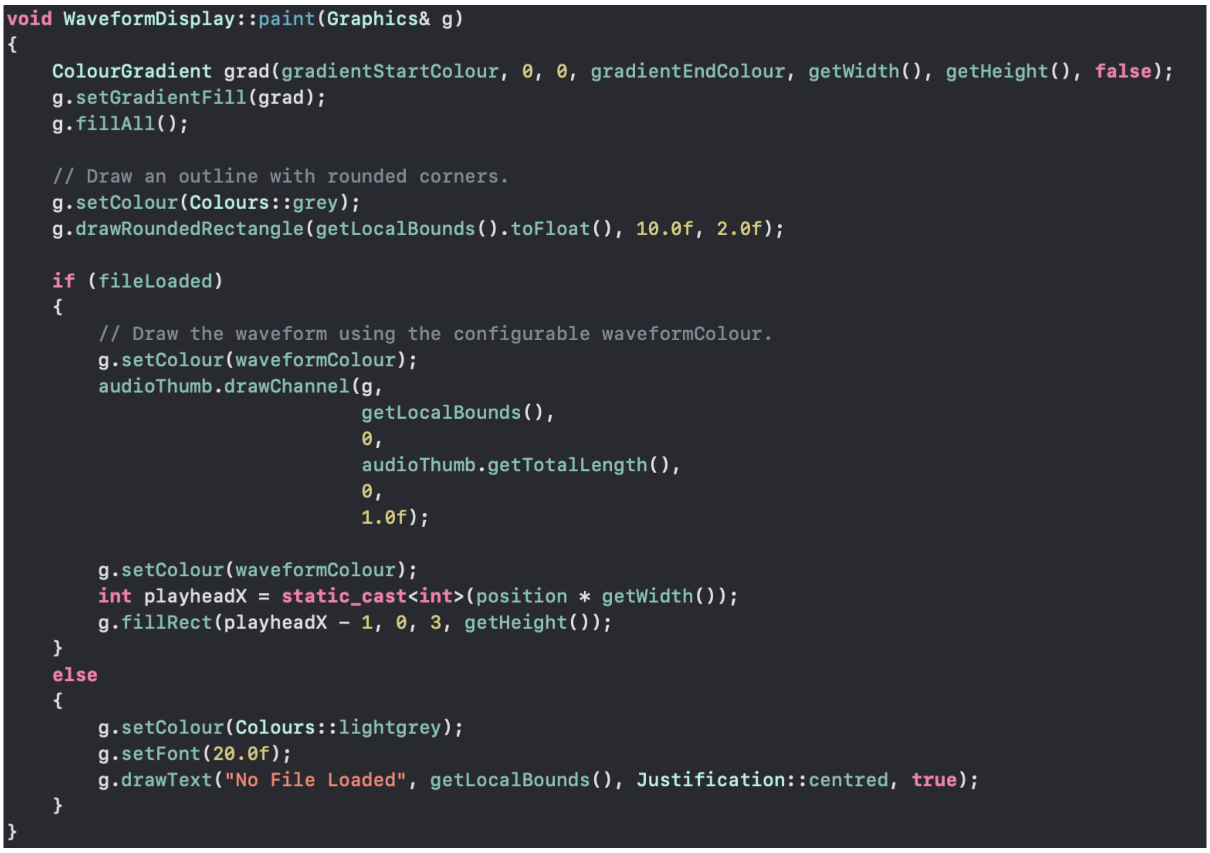
This snippet shows the use of gradient backgrounds and deck titles that enhance the UI.



```
void DeckGUI::paint (Graphics& g)
{
    ColourGradient gradient;
    if (deckNumber == 1)
    {
        // Deck 1: white-to-black gradient
        gradient = ColourGradient(Colours::grey, 0, 0, Colours::black, getWidth(), getHeight(), false);
    }
    else if (deckNumber == 2)
    {
        // Deck 2: black-to-white gradient
        gradient = ColourGradient(Colours::black, 0, 0, Colours::grey, getWidth(), getHeight(), false);
    }

    g.setGradientFill(gradient);
    g.fillAll();

    // Draw the deck title
    g.setColour(Colours::white);
    g.setFont(Font("Arial", 16.0f, Font::bold));                              2 ⚠ 'Font' is deprecated: Use th

    String title;
    if (deckNumber == 1)
        title = "Deck 1";
    else if (deckNumber == 2)
        title = "Deck 2";

    g.drawFittedText(title, getLocalBounds().reduced(10), Justification::topRight, 1);
}
```

From WaveformDisplay.cpp – Drawing the Waveform:

This snippet shows how the waveform display is rendered with a gradient and playhead indicator.



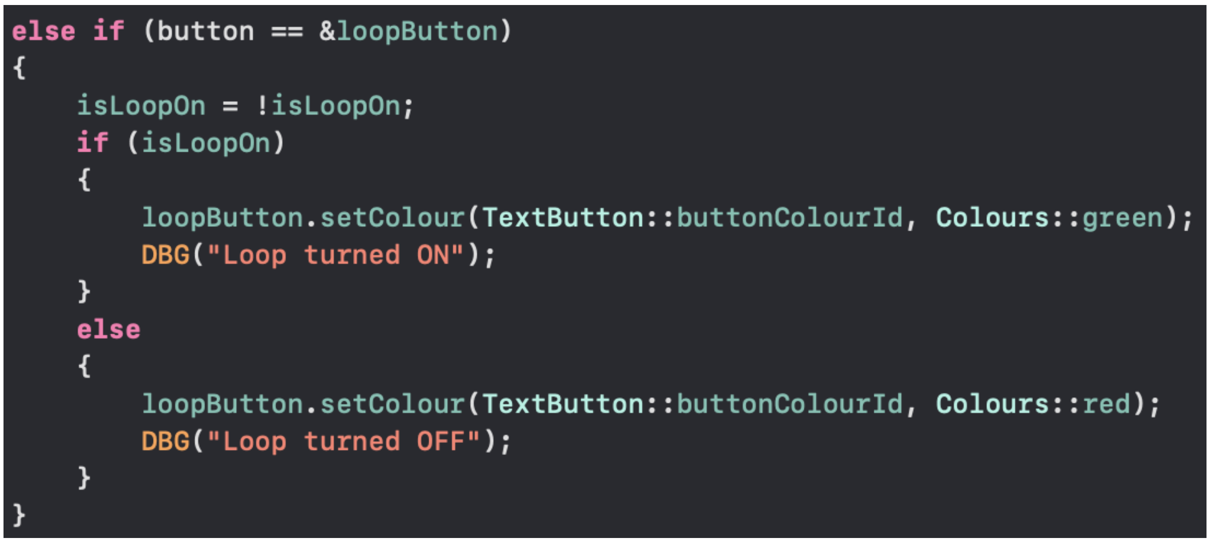
```cpp
void WaveformDisplay::paint(Graphics& g)
{
    ColourGradient grad(gradientStartColour, 0, 0, gradientEndColour, getWidth(), getHeight(), false);
    g.setGradientFill(grad);
    g.fillAll();

    // Draw an outline with rounded corners.
    g.setColour(Colours::grey);
    g.drawRoundedRectangle(getLocalBounds().toFloat(), 10.0f, 2.0f);

    if (fileLoaded)
    {
        // Draw the waveform using the configurable waveformColour.
        g.setColour(waveformColour);
        audioThumb.drawChannel(g,
                               getLocalBounds(),
                               0,
                               audioThumb.getTotalLength(),
                               0,
                               1.0f);

        g.setColour(waveformColour);
        int playheadX = static_cast<int>(position * getWidth());
        g.fillRect(playheadX - 1, 0, 3, getHeight());
    }
    else
    {
        g.setColour(Colours::lightgrey);
        g.setFont(20.0f);
        g.drawText("No File Loaded", getLocalBounds(), Justification::centred, true);
    }
}
```

4.2 Additional Event Listeners and Interactivity (R2B)

The enhanced interface employs a series of event listeners to handle user actions in real time. In addition to standard listeners for button clicks and slider adjustments, new listeners have been implemented to manage loop toggling, fast forward/rewind operations. These additions ensure that the application is not only visually appealing but also functionally robust and responsive.

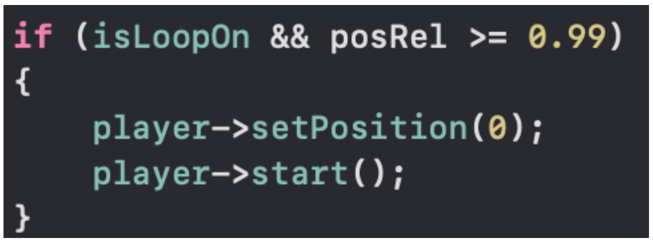
Loop Functionality in DeckGUI.cpp

Lambda Functions and Listeners:
I utilized lambda callbacks to handle user interactions such as button clicks for looping and navigation. When a loop button is toggled, a lambda function in the LoopController updates the loop state and changes the button’s color accordingly.



```
else if (button == &loopButton)
{
    isLoopOn = !isLoopOn;
    if (isLoopOn)
    {
        loopButton.setColour(TextButton::buttonColourId, Colours::green);
        DBG("Loop turned ON");
    }
    else
    {
        loopButton.setColour(TextButton::buttonColourId, Colours::red);
        DBG("Loop turned OFF");
    }
}
```

Timer Callbacks:
A timer continuously checks the playback position. If it detects that the playhead is nearing the track's end and looping is enabled, it triggers the loop restart functionality.

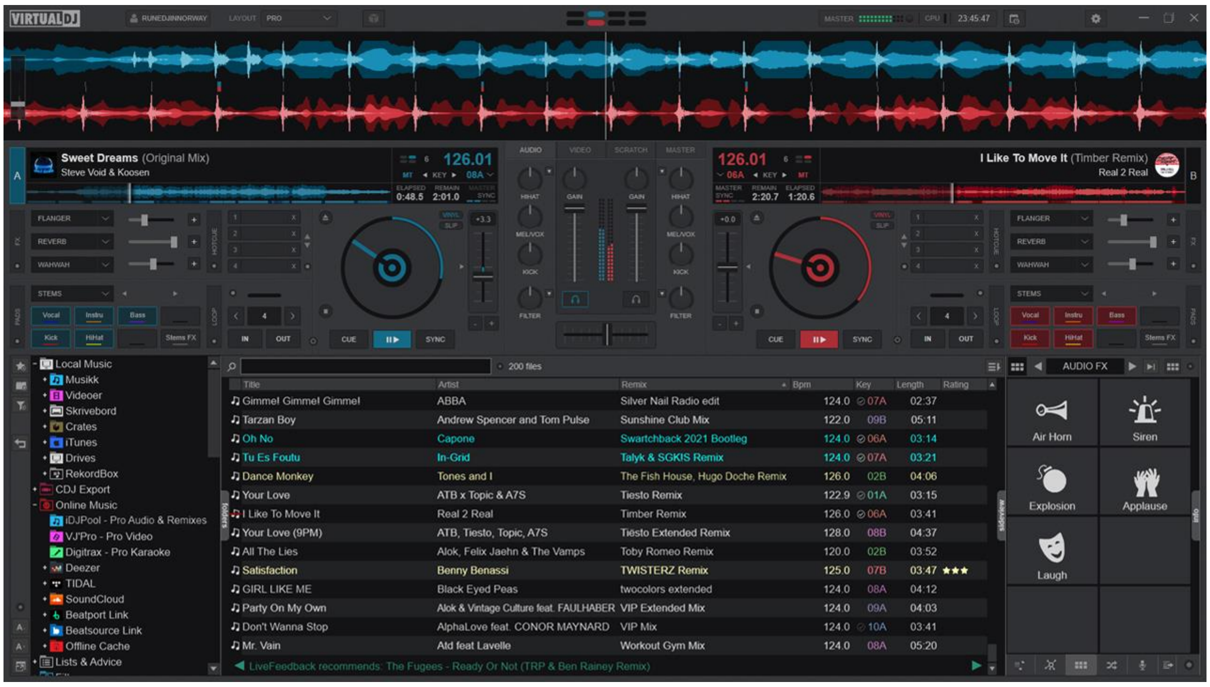


```
if (isLoopOn && posRel >= 0.99)
{
    player->setPosition(0);
    player->start();
}
```

5. New Feature Implementation (R3)

5.1 Analysis of Virtual DJ

For this feature, I analyzed **VirtualDJ**, a well-known DJ application that offers advanced looping, fast forward/rewind, and dynamic track display features. The screenshot below illustrates VirtualDJ’s interface, highlighting its loop control and navigation functionalities:



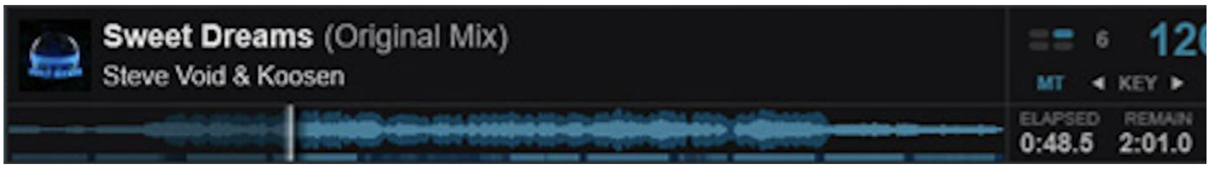
200 files

Title	Artist	Remix	Bpm	Key	Length	Rating
Gimmel Gimmel Gimmel	ABBA	Silver Nail Radio edit	124.0	07A	02:37	
Tarzan Boy	Andrew Spencer and Tom Pulse	Sunshine Club Mix	122.0	09B	05:11	
Oh No	Capone	Swartchback 2021 Bootleg	124.0	06A	03:14	
Tu Es Foutu	In-Grid	Talyk & SGKIS Remix	124.0	07A	03:21	
Dance Monkey	Tones and I	The Fish House, Hugo Doche Remix	126.0	02B	04:06	
Your Love	ATB x Topic & A7S	Tiesto Remix	122.9	01A	03:15	
I Like To Move It	Real 2 Real	Timber Remix	126.0	06A	03:41	
Your Love (9PM)	ATB, Tiesto, Topic, A7S	Tiësto Extended Remix	128.0	08B	04:37	
All The Lies	Alok, Felix Jaehn & The Vamps	Toby Romeo Remix	120.0	02B	03:52	
Satisfaction	Benny Benassi	TWISTERZ Remix	125.0	07B	03:47	★★★
GIRL LIKE ME	Black Eyed Peas	twocolors extended	124.0	08A	04:12	
Party On My Own	Alok & Vintage Culture feat. FAULHABER	VIP Extended Mix	124.0	09A	04:03	
Don't Wanna Stop	AlphaLove feat. CONOR MAYNARD	VIP Mix	124.0	10A	03:41	
Mr. Vain	Atd feat Lavelle	Workout Gym Mix	124.0	08A	05:20	

LiveFeedback recommends: The Fugees - Ready Or Not (TRP & Ben Rainey Remix)

Analysis:

Dynamic Track Information:
Real-time updates of track names and playback times are displayed prominently. This immediate feedback keeps DJs informed about the current position in the track, enhancing performance accuracy.



Sweet Dreams (Original Mix)
Steve Void & Koosen

ELAPSED 0:48.5
REMAIN 2:01.0

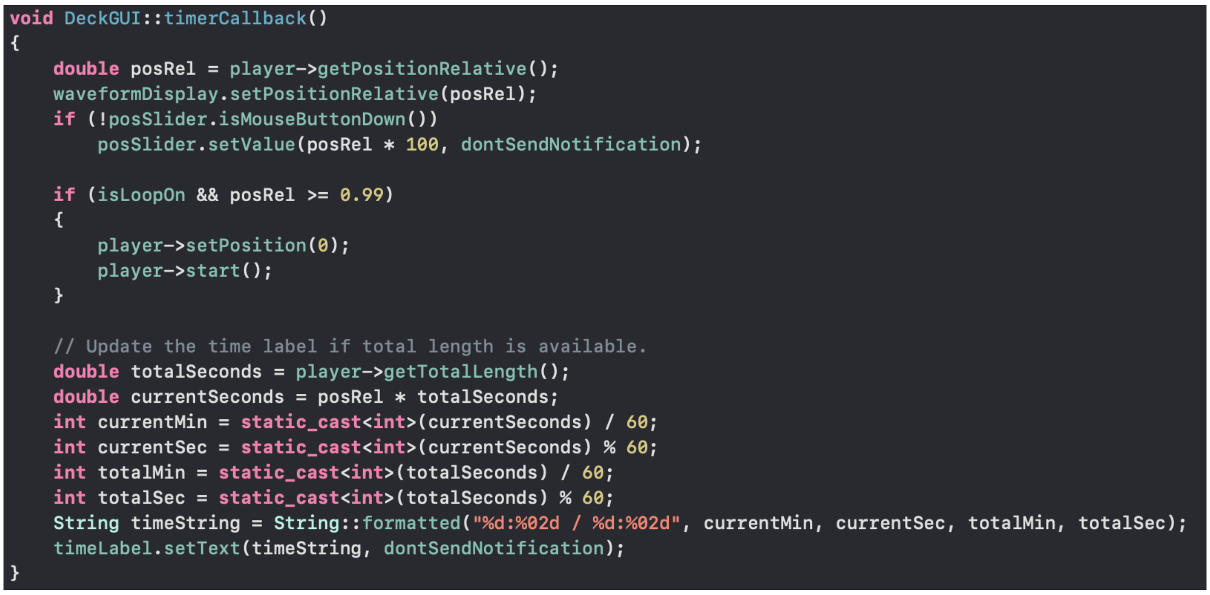
Implementation:

Inspired by VirtualDJ's approach, I engineered the new feature using object-oriented design principles and event-driven programming. Here's how I structured the implementation:

From DeckGUI.cpp - Timer Callbacks:
DisplayManager oversees updating the dynamic track information on the interface, such as the "Now Playing" label and the current versus total track time display.

OOP concept:
This class uses composition by integrating with other components (like LoopController

and NavigationController) to fetch and display up-to-date track information based on real-time events.



```
void DeckGUI::timerCallback()
{
    double posRel = player->getPositionRelative();
    waveformDisplay.setPositionRelative(posRel);
    if (!posSlider.isMouseButtonDown())
        posSlider.setValue(posRel * 100, dontSendNotification);

    if (isLoopOn && posRel >= 0.99)
    {
        player->setPosition(0);
        player->start();
    }

    // Update the time label if total length is available.
    double totalSeconds = player->getTotalLength();
    double currentSeconds = posRel * totalSeconds;
    int currentMin = static_cast<int>(currentSeconds) / 60;
    int currentSec = static_cast<int>(currentSeconds) % 60;
    int totalMin = static_cast<int>(totalSeconds) / 60;
    int totalSec = static_cast<int>(totalSeconds) % 60;
    String timeString = String::formatted("%d:%02d / %d:%02d", currentMin, currentSec, totalMin, totalSec);
    timeLabel.setText(timeString, dontSendNotification);
}
```

5.1.1 Implementation of Playlist Component to make it fully functional (NOT NEW)

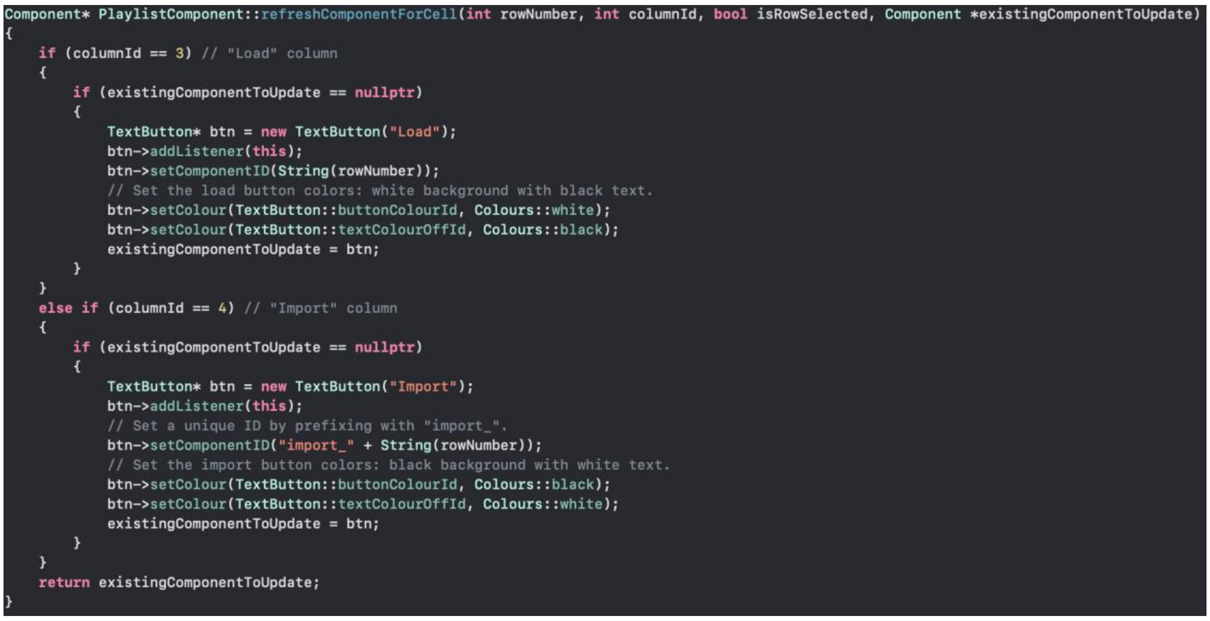
The project also introduces a comprehensive playlist management system. This component employs a table-based layout to display track titles, durations, and interactive buttons for both loading and importing tracks. Initially, the code provided for class offered the visual elements of the buttons, but they lacked functional integration. Recognizing that a comprehensive playlist is essential for a professional DJ—after all, a DJ is defined by the ability to seamlessly mix multiple tracks—this functionality was prioritized and fully implemented in the final design.

When the “Import” button is clicked, a file chooser dialog appears, allowing the selection of a new audio file. Upon selection, the application extracts the audio file's metadata to determine its duration and updates the track title accordingly. Furthermore, clicking the “Load” button opens a dialog that allows the user to select which deck should load the track through a custom deck selection component that utilizes lambda callbacks.

This modular design not only encapsulates the logic for track management but also mimics the functionality found in industry-standard DJ software. The use of inheritance (extending table models) and event-driven programming ensures that the component is both scalable and maintainable.

From PlaylistComponent.cpp – Refreshing Cells with Interactive Buttons:

This code demonstrates how "Load" and "Import" buttons are created in the playlist table.



```cpp
Component* PlaylistComponent::refreshComponentForCell(int rowNumber, int columnId, bool isRowSelected, Component *existingComponentToUpdate)
{
    if (columnId == 3) // "Load" column
    {
        if (existingComponentToUpdate == nullptr)
        {
            TextButton* btn = new TextButton("Load");
            btn->addListener(this);
            btn->setComponentID(String(rowNumber));
            // Set the load button colors: white background with black text.
            btn->setColour(TextButton::buttonColourId, Colours::white);
            btn->setColour(TextButton::textColourOffId, Colours::black);
            existingComponentToUpdate = btn;
        }
    }
    else if (columnId == 4) // "Import" column
    {
        if (existingComponentToUpdate == nullptr)
        {
            TextButton* btn = new TextButton("Import");
            btn->addListener(this);
            // Set a unique ID by prefixing with "import_".
            btn->setComponentID("import_" + String(rowNumber));
            // Set the import button colors: black background with white text.
            btn->setColour(TextButton::buttonColourId, Colours::black);
            btn->setColour(TextButton::textColourOffId, Colours::white);
            existingComponentToUpdate = btn;
        }
    }
    return existingComponentToUpdate;
}
```

6. Methods, Logic, and Design Rationale

The design and implementation of the DJ application are grounded in solid object-oriented design principles. Each class within the project, whether it is the audio player, deck interface, playlist component, or waveform display, is responsible for a specific part of the functionality, promoting modularity and ease of maintenance. The separation of concerns inherent in this design ensures that modifications in one module have minimal impact on others.

Event-driven programming is central to the application's responsiveness. The use of event listeners captures user interactions—such as button clicks, slider adjustments, and scroll events—allowing the application to react dynamically to user input. Lambda functions are utilized for callback management, particularly in the playlist component, thereby streamlining the code and reducing complexity.

From a technical standpoint, the application leverages JUCE's robust audio processing classes, including AudioTransportSource, ResamplingAudioSource, and MixerAudioSource, to manage audio playback, speed control, and mixing. These classes ensure high performance and reliability, which are critical for live DJ applications. The combination of these technical choices reflects a commitment to building a professional-grade application that is both flexible and scalable.

7. Conclusion

The DJ application developed in this project not only meets the fundamental requirements of loading, playing, and mixing audio tracks but also significantly enhances the overall user experience through a highly customized interface and additional advanced features. The integration of a robust playlist management system, loop control, fast forward/rewind functionality, dynamic track information display, and refined slider scrubbing exemplifies the application's commitment to innovation and usability. Through careful adherence to object-oriented design and event-driven programming principles, and by leveraging the powerful capabilities of the JUCE framework, this project has produced a comprehensive and professional tool suitable for live DJ performances.

8. Appendices

Appendix A: Full Code Listings and References

- **DeckGUI.cpp & DeckGUI.h:** These files contain the detailed implementation of the deck interface, including new controls for Pause, Restart, Loop, Fast Forward, and Rewind, as well as the event listeners for these actions.
- **DJAudioPlayer.cpp & DJAudioPlayer.h:** These files detail the audio playback logic, including methods for loading audio files, adjusting playback speed, and mixing audio streams.
- **MainComponent.cpp & MainComponent.h:** These files demonstrate the integration of multiple decks and the configuration of the mixer to ensure simultaneous playback.
- **PlaylistComponent.cpp & PlaylistComponent.h:** The playlist management system is implemented in these files, showcasing track import functionality, deck selection dialogs, and interactive table-based track listings.
- **WaveformDisplay.cpp & WaveformDisplay.h:** These files implement the custom waveform rendering logic, including methods for setting gradient and waveform colors.