OOP Final report

R1 – Basic program :

R1A: Loading Audio Files into Audio Players

· The DJ app allows users to load audio files into the decks directly or from a playlist.
· Users have the flexibility to choose tracks from their local storage or previously saved playlists within the app.

R1B: Playing Two or More Tracks

· The app supports playing two tracks simultaneously, each assigned to a separate deck. · Users can initiate playback of multiple tracks and manipulate them independently for mixing purposes.

R1C: Mixing Tracks by Varying Each of Their Volumes

· Users can mix tracks by adjusting the volume of each track individually.
· Controls for high, mid, and low frequencies are provided to fine-tune the audio mix.
· The app features crossfade functionality, enabling smooth transitions between tracks during mixing sessions.

R1D: Speed Control for Tracks

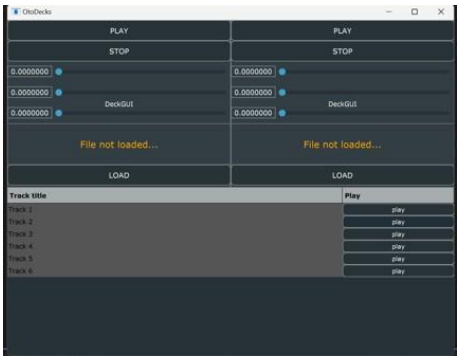
· The DJ app allows users to adjust the playback speed of tracks.
· Speed control functionality enables users to create dynamic mixes and match the tempo of different tracks seamlessly.

Additional Features:

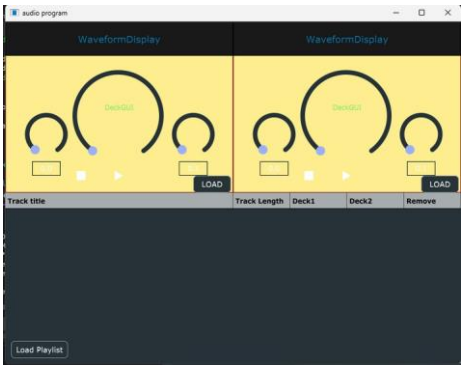
· Track Controls: The app offers a range of controls for track manipulation, including play, pause, rewind, forward, loop, and stop. These controls enhance the user experience by providing intuitive playback management.
· Volume, Position, Speed, High, Mid, Low Controls: Users have access to comprehensive controls for adjusting various aspects of the audio playback, including volume, speed, and frequency levels.
· Waveform Display: The app includes a waveform display feature, providing users with visual feedback on the audio waveform of loaded tracks. This feature aids in precise cueing and mixing.
· Playlist Management: Users can manage playlists within the app, allowing them to save and organize their favorite tracks for future use.
· Dual Decks: The DJ app incorporates two decks, enabling users to mix and play multiple tracks simultaneously.

R2 - User Interface Customization

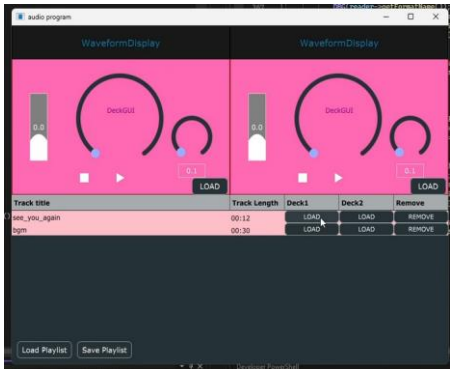
Template



Stage 1

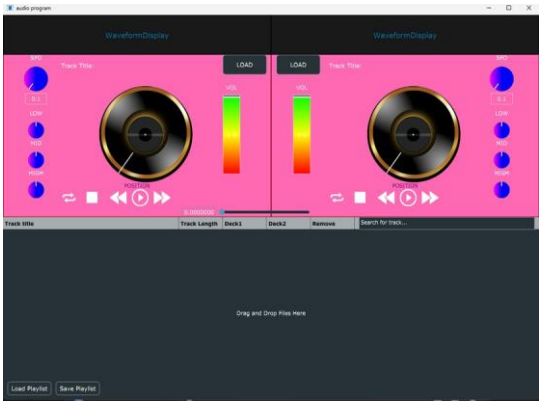


Stage 2

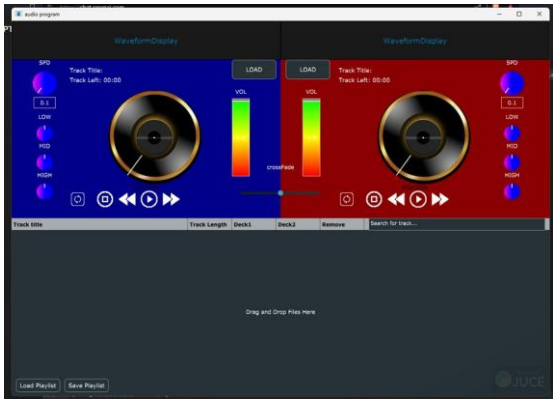


Stage 3

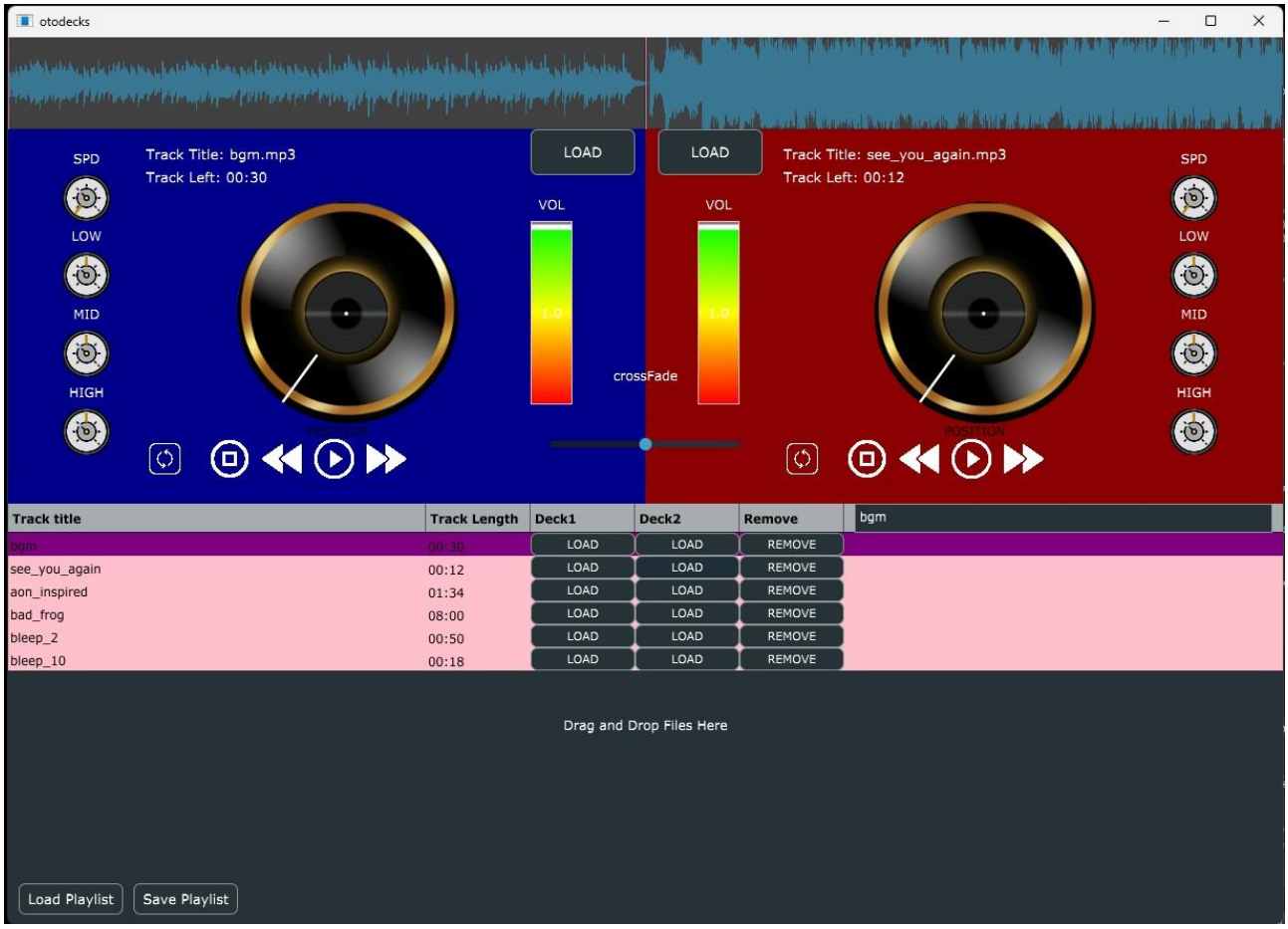
Stage 4



Stage 5



Stage 6 (Final product)



In customizing the user interface (UI) of the DJ app, I focused on enhancing its appearance and usability across six stages. Here's how I approached it:

First Stage:

· Initially, I revamped the layout by placing the waveform at the top of the deck and repositioning sliders and buttons to emulate a professional DJ player interface.
· I replaced standard sliders with different styles for a sleeker look and experimented with rotary sliders.
· Introduction of image buttons, such as play and stop buttons, for a more visually appealing interface.
· I explored various slider styles and incorporated a new feature allowing users to load playlists.
· Changed the background color to yellow to add vibrancy.

Second Stage:

· Implemented custom UI elements using the CustomLookAndFeel class, modifying the volume slider to a linear vertical bar with a rounded rectangle thumb.
· Integrated a 'save playlist' button to enable users to store songs for future use. · Altered the background color to pink for a softer visual aesthetic.

Third Stage:

· Applied alpha color effects to the waveform for a more dynamic visual experience.
· Enhanced the DJ player basics by adding icons to control buttons (play, pause, stop,

rewind, forward, loop).
· Introducing a new event listener and slider for crossfading between tracks.
· Included labels for volume and speed sliders to improve user understanding. · Added a search box in the playlist component for easier song navigation.
· Shifted the load button to the top for better accessibility.

Fourth Stage:

· Expanded UI functionality by introducing sliders for low, mid, and high frequencies, aligned vertically with the speed slider.
· Implemented gradient color effects to enhance the visual appeal of sliders. · Incorporated crossfade controls on top of the decks for intuitive mixing.
· Adjusted the volume slider's appearance, utilizing a gradient from green to red to indicate volume levels.
· Enhanced deck symmetry by mirroring sliders based on their position (0 and 1). · Added instructional labels to guide users in the playlist component.
· Introduced track tiles in the deck to display loaded song names.
· Refinement included relocating the search bar to the top beside the remove option in the playlist component.

Fifth Stage:

· Replaced numerical indicators with thematic colors (dark blue and dark red) for visual consistency.
· Redesigned icons for stop and loop functions for better clarity.
· Added a label to the crossfade slider for improved user guidance.

Final Stage:

· Completed the customization by incorporating matching images for sliders and adjusting pointer colors for coherence.
· Achieved a cohesive visual presentation with brown pointer colors for speed, low, mid and high sliders.
· These alterations culminated in the final product, presenting a visually appealing and user-friendly DJ app interface.

· Changed the project name on projucer from audio program to otodecks to match with coursework.

I customized the appearance of rotary and linear sliders by implementing a custom LookAndFeel class. Here's how I achieved this:

```cpp
void CustomLookAndFeel::drawLinearSlider(Graphics& g, int x, int y, int width, int height,
    float sliderPos, float minSliderPos, float maxSliderPos,
    const Slider::SliderStyle style, Slider& slider)
{
    if (style == Slider::SliderStyle::LinearBarVertical)
    {
            // Draw the track
            auto trackWidth = width;
            auto trackBounds = Rectangle<float>(x, y, trackWidth, height);

            // gradient colors for the track
            ColourGradient gradient(Colour(0xff00ff00), trackBounds.getTopLeft(),
                Colour(0xffff0000), trackBounds.getBottomLeft(), false);
            gradient.addColour(0.5, Colours::yellow); // Yellow in the middle

            // Fill the track with the gradient
            g.setGradientFill(gradient);
            g.fillRect(trackBounds);

            // Draw the thumb as a rounded rectangle
            auto thumbWidth = trackBounds.getWidth() * 1.5f;
            auto thumbHeight = 10;
            auto thumbBounds = Rectangle<float>(trackBounds.getX() + trackBounds.getWidth() * 0.5f - thumbWidth * 0.5f,
                sliderPos - thumbHeight * 0.5f, thumbWidth, thumbHeight);

            // Draw the thumb with a solid color
            g.setColour(Colours::grey);
            g.fillRoundedRectangle(thumbBounds, 4.0f);

            // Draw the outline of the thumb
            g.setColour(Colours::white);
            g.drawRoundedRectangle(thumbBounds, 10.0f, 5.0f);

            // Fill the remaining space above the track with grey
            auto remainingBounds = Rectangle<float>(x, y, trackWidth, thumbBounds.getY());
            g.setColour(Colours::grey);
            g.fillRect(remainingBounds);
    }
    else
    {
        // Fallback to the default LookAndFeel
        LookAndFeel_V4::drawLinearSlider(g, x, y, width, height, sliderPos, minSliderPos, maxSliderPos, style, slider);
    }
}
```

DrawLinearSlider function in customLookAndFeel.cpp

drawLinearSlider: This method is responsible for drawing a linear slider, specifically for the vertical orientation (`SliderStyle::LinearBarVertical`).It draws the track with a gradient fill, where the color changes from green at the top to red at the bottom with yellow in the middle.The thumb (slider handle) is drawn as a rounded rectangle filled with a solid grey color and outlined with white.The remaining space above the track is filled with grey.

```cpp
void CustomLookAndFeel::drawRotarySlider(Graphics& g, int x, int y, int width, int height,
    float sliderPos, float rotaryStartAngle, float rotaryEndAngle,
    Slider& slider)
{
    auto radius = jmin(width / 2, height / 2) - 4.0f;
    auto centreX = x + width * 0.5f;
    auto centreY = y + height * 0.5f;
    auto rx = centreX - radius;
    auto ry = centreY - radius;
    auto rw = radius * 2.0f;
    auto angle = rotaryStartAngle + sliderPos * (rotaryEndAngle - rotaryStartAngle);

    // Draw the background ellipse
    g.setColour(Colours::black);
    g.fillEllipse(rx, ry, rw, rw);

    if (slider.getName() == "posSlider")
    {
        // Draw the disc image for the "posSlider"
        auto originalImage = ImageCache::getFromMemory(BinaryData::disc_png, BinaryData::disc_pngSize);
        int newWidth = originalImage.getWidth();
        int newHeight = originalImage.getHeight();
        auto resizedImage = originalImage.rescaled(newWidth-100, newHeight-100, Graphics::highResamplingQuality);
        auto imageX = centreX - resizedImage.getWidth() / 2;
        auto imageY = centreY - resizedImage.getHeight() / 2;
        g.drawImageAt(resizedImage, imageX, imageY);
        g.setColour(Colours::white);
    }
    else {

        auto originalImage = ImageCache::getFromMemory(BinaryData::knob_png, BinaryData::knob_pngSize);
        int newWidth = originalImage.getWidth();
        int newHeight = originalImage.getHeight();
        auto resizedImage = originalImage.rescaled(newWidth - 80, newHeight - 80, Graphics::highResamplingQuality);
        auto imageX = centreX - resizedImage.getWidth() / 2;
        auto imageY = centreY - resizedImage.getHeight() / 2;
        g.drawImageAt(resizedImage, imageX, imageY);
        g.setColour(Colours::darkgoldenrod);

    }
    // Draw the pointer
    Path pointer;
    auto pointerLength = radius * 0.53f;
    auto pointerThickness = 2.5f;
    pointer.addRectangle(-pointerThickness * 0.5f, -radius, pointerThickness, pointerLength);
    pointer.applyTransform(AffineTransform::rotation(angle).translated(centreX, centreY));
    g.fillPath(pointer);
}
```

DrawRotarySlider function in customLookAndFeel.cpp

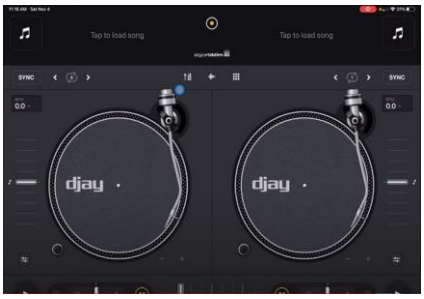
drawRotarySlider Method:Within the custom LookAndFeel class, I implemented the drawRotarySlider method. This method is responsible for rendering rotary sliders.I customized the appearance of rotary sliders by:Drawing a background ellipse with a black color fill.Loading a specific knob image using embedded binary data. For sliders named 'posSlider,' I used a disc image, while for others, I used a different knob image.Resizing the knob image to fit the slider's dimensions while maintaining its aspect ratio.Drawing a pointer on the knob to indicate the slider's position.

Image Handling:I retrieved the knob images from embedded binary data using the ImageCache::getFromMemory function.To ensure the images fit the sliders properly, I resized them while maintaining their aspect ratio.

By implementing these customizations in the LookAndFeel class, I achieved a visually enhanced and cohesive appearance for sliders in my application. Additionally, the images were edited by me on Canva to ensure they complemented the overall aesthetic seamlessly.

R3 – Cross-Application Feature Integration: Research, Analysis, and Implementation Process

DJ
Apps from
youtube



After reviewing a YouTube video for common features found in DJ applications, I decided to integrate those features into my own app. The main components I focused on were **crossfade, basic controls (play, pause, stop, rewind, forward, loop), mid, low, and high frequency controls, track title, and track left functionalities.**

Here's how I engineered them into my application:

Firstly, start with **crossfade**,I implemented this feature in the main component to draw on top of the DeckGUI. I created a slider and label for crossfade, along with a function named sliderValueChanged in the MainComponent.h. Then, in MainComponent.cpp, I used addAndMakeVisible to display the slider and label in the main function, added an event listener for the crossfader slider, and set bounds for them in the resized function. The callback function "void sliderValueChanged" handles all the work for the crossfade slider.

```
private:
    //==============================================
    // Your private member variables go here...
    Slider crossfadeSlider;
    Label crossFade;
```

Slider & Label in Maincomponent.h

```
24
25          addAndMakeVisible(deckGUI1);
26          addAndMakeVisible(deckGUI2);
27          addAndMakeVisible(playlistComponent);
28
29          addAndMakeVisible(crossfadeSlider);
30
31          crossfadeSlider.addListener(this);
32          crossfadeSlider.setRange(0.0, 1.0);
33          crossfadeSlider.setTextBoxStyle(juce::Slider::NoTextBox, true, 0, 0);
34          crossfadeSlider.setValue(0.5);
35
36          addAndMakeVisible(crossFade);
37          crossFade.setText("crossFade", juce::dontSendNotification);
38          crossFade.setJustificationType(Justification::centredTop);
39          crossFade.attachToComponent(&crossfadeSlider, false);
```

crossfadeSlider and crossFade in MainComponent()

```
82
83  void MainComponent::resized()
84  {
85      deckGUI1.setBounds(0, 0, getWidth() / 2, getHeight() / 1.9);
86      deckGUI2.setBounds(getWidth() / 2, 0, getWidth() / 2, getHeight() / 1.9);
87
88      playlistComponent.setBounds(0, getHeight()/1.9, getWidth(), getHeight()/2.1);
89      crossfadeSlider.setBounds(getWidth()/2 - 100, getHeight()/2.5, 200, 100);
90  }
```

crossfadeSlider setbounds in MainComponent::resized

```cpp
void MainComponent::sliderValueChanged(Slider* slider) {
    if (slider == &crossfadeSlider) {
        // Get the current volume values from both DeckGUI instances
        float currentVol1 = deckGUI1.volSlider.getValue();
        float currentVol2 = deckGUI2.volSlider.getValue();

        // Get the crossfade value from the slider
        float crossfadeValue = slider->getValue();

        // Calculate the adjusted volume levels based on the crossfade value
        float adjustedVol1 = currentVol1 * (1 - crossfadeValue);
        float adjustedVol2 = currentVol2 * crossfadeValue;

        // Apply the adjusted volume levels to the players
        player1.setGain(adjustedVol1);
        player2.setGain(adjustedVol2);
    }
}
```

Void sliderValueChanged in MainComponent.cpp

In the function, we first check if the slider that triggered the change is the crossfade slider by comparing their memory addresses. Then, we retrieve the current volume levels from both DeckGUI instances using their respective volume sliders. These volume levels represent the current volume slider of the audio players associated with each deck. Next, we get the current value of the crossfade slider, which indicates the position of the crossfade slider. This value ranges from 0 to 1, representing the proportion of volume levels between the two decks. Using this crossfade value, we calculate the adjusted volume levels for both decks. The volume level of the first deck is adjusted by reducing its current volume level by a factor of (1 - crossfadeValue), while the volume level of the second deck is adjusted by multiplying its current volume level by the crossfade value. Finally, we apply these calculated adjusted volume levels to the respective audio players associated with each deck, effectively controlling the volume balance between the two decks based on the position of the crossfade slider.

Secondly,moving on to the **basic features**, such as play, pause, stop, rewind, forward, and loop, I extended the given template to include extra functionalities for pause, rewind, forward, and loop in the DJAudioPlayer.h and cpp files. These features allow users to control playback and manipulate audio tracks effectively. Images for buttons are loaded via Projucer, making them accessible through ImageCache::getFromMemory, and associated with variables.

```cpp
auto playImage = ImageCache::getFromMemory(BinaryData::play_png, BinaryData::play_pngSize);
auto pauseImage = ImageCache::getFromMemory(BinaryData::pause_png, BinaryData::pause_pngSize);
auto stopImage = ImageCache::getFromMemory(BinaryData::stop_png, BinaryData::stop_pngSize);
auto loopImage = ImageCache::getFromMemory(BinaryData::loop_png, BinaryData::loop_pngSize);
auto rewindImage = ImageCache::getFromMemory(BinaryData::previous_png, BinaryData::previous_pngSize);
auto forwardImage = ImageCache::getFromMemory(BinaryData::next_png, BinaryData::next_pngSize);


playButton.setImages(true, true, true, playImage, 1.0f, Colours::white, Image(nullptr), 1.0f, Colours::transparentWhite, pauseImage, 1.0f, Colours::transparentWhite);
stopButton.setImages(true, true, true, stopImage, 1, Colours::white, Image(nullptr), 1, Colours::transparentWhite, stopImage, 1, Colours::transparentWhite);
loopButton.setImages(true, true, true, loopImage, 1, Colours::white, Image(nullptr), 1, Colours::transparentWhite, loopImage, 1, Colours::transparentWhite);
rewindButton.setImages(true, true, true, rewindImage, 1.0f, Colours::white, Image(nullptr), 1.0f, Colours::transparentWhite, rewindImage, 1.0f, Colours::transparentWhite);
forwardButton.setImages(true, true, true, forwardImage, 1.0f, Colours::white, Image(nullptr), 1.0f, Colours::transparentWhite, forwardImage, 1.0f, Colours::transparentWhite);
```

Accessing Image with variables in DeckGUI::DeckGUI()

When the user clicks the play button, it transitions to a pause state, visually represented by an image swap. This functionality is managed through the playButton object, where images for various button states are set. For the normal state, a play image is displayed with full opacity and a white background. When hovered over, there's no image change, but the button background becomes transparent white. However, when the button is clicked, indicating a pause action, the play image is replaced with a pause image, both with full opacity and a transparent white background. This visual cue provides users with clear feedback on the playback state, enhancing the overall user experience of the application.

Similarly, **rewind and forward** functionalities are implemented, allowing users to navigate through tracks by adjusting the playback position. Rewinding moves the track back by 5 seconds, while

forwarding moves it forward by the same duration. These functionalities ensure seamless track navigation within the app.

```
/** rewind track by 5 seconds */
void DJAudioPlayer::rewind()
{
    //check if position is more than 0 after rewind 5secs

    if ((transportSource.getCurrentPosition() - 5.0) > 0)
    {
        transportSource.setPosition(transportSource.getCurrentPosition() - 5.0);
    }
    else  //else set position to beginning of the track
    {
        transportSource.setPosition(0);
    }
}
```

rewind function in DJAudioPlayer.cpp

rewind(): This method rewinds the track by 5 seconds. It first checks if the current position of the track minus 5 seconds is greater than 0, indicating that there is enough time to rewind without going before the start of the track. If so, it sets the position of the transport source to the current position minus 5 seconds. If the position would be less than 0 after rewinding, it sets the position to the beginning of the track (position 0).

```
/** forward track by 5 seconds */
void DJAudioPlayer::forward()
{

    if ((transportSource.getCurrentPosition() + 5.0) < transportSource.getLengthInSeconds())
    {
        transportSource.setPosition(transportSource.getCurrentPosition() + 5.0);
    }
    else
    {
        transportSource.setPosition(transportSource.getLengthInSeconds());
    }
}
```

forward function in DJAudioPlayer.cpp

forward(): This method moves the track forward by 5 seconds. It checks if adding 5 seconds to the current position of the track would exceed the total length of the track (in seconds). If not, it sets the position of the transport source to the current position plus 5 seconds. If the position would exceed the length of the track after forwarding, it sets the position to the end of the track (its total length).

```
/** toggles the loop button */
void DJAudioPlayer::toggleLoop()
{
    if (loop) {
        loop = false;
    }
    else {
        loop = true;
    }
}

/** check if loop is on */
bool DJAudioPlayer::isLooping()
{
    return loop;
}
```

toggleLoop in DJAudioPlayer.cpp

```
}

if (button == &loopButton)
{
    repaint();
    player->toggleLoop();

    //set the loop button colours according to the loop state
    if (player->isLooping())
    {
        std::cout << "looping" << std::endl;
        loopButton.setToggleState(true, juce::dontSendNotification);
    }
    else
    {
        loopButton.setToggleState(false, juce::dontSendNotification);
    }
}
```

loopButton in DeckGui.cpp

Loop, allowing users to repeat songs. This feature is controlled by a boolean property, enabling users to toggle loop mode on and off. By default, loop is enabled (set to true) since only one song can play at a time in a deck. Users have the option to disable loop mode if they prefer. The status of loop mode is indicated by an image: white indicates loop mode is off, while purple signifies loop mode is on.

All of these functions are connected with DeckGUI, where we implement our controls. Therefore, within the DeckGUI class, the buttonClicked function serves as the event listener. It ensures that the interface updates accordingly whenever changes are made to these controls.

Mid, low and high. These are similar functions as well.

Next, mid, low, and high frequency controls are implemented using filter controls. I created filter controls for each band to adjust the gain of specific frequency ranges in the audio signal. These controls provide users with the ability to fine-tune the tonal balance or equalization of the audio playback.

```
void DJAudioPlayer::setLBFilter(double gain) {
    audioLowFilter.setCoefficients(juce::IIRCoefficients::makeLowShelf(thisSampleRate, 500, 1.0 / juce::MathConstants<double>::sqrt2, gain));
};

void DJAudioPlayer::setMBFilter(double gain) {
    audioMidFilter.setCoefficients(juce::IIRCoefficients::makePeakFilter(thisSampleRate, 3250, 1.0 / juce::MathConstants<double>::sqrt2, gain));
};

void DJAudioPlayer::setHBFilter(double gain) {
    audioHighFilter.setCoefficients(juce::IIRCoefficients::makeHighShelf(thisSampleRate, 5000, 1.0 / juce::MathConstants<double>::sqrt2, gain));
};
```

Filters in DJAudioPlayer.cpp

setLBFilter(double gain): This method sets the low-frequency filter. It uses the IIRCoefficients::makeLowShelf function to create a low-shelf filter with the specified sample rate,
cutoff frequency (500 Hz), Q factor (1.0 / sqrt(2)), and gain. The gain parameter controls the amplification or attenuation of low-frequency audio signals.

setMBFilter(double gain): This method sets the mid-frequency filter. It uses the IIRCoefficients::makePeakFilter function to create a peak filter with the specified sample rate,
center frequency (3250 Hz), Q factor (1.0 / sqrt(2)), and gain. The gain parameter adjusts the gain of the mid-frequency audio signals.

setHBFilter(double gain): This method sets the high-frequency filter. It utilizes the IIRCoefficients::makeHighShelf function to create a high-shelf filter with the specified sample rate,
cutoff frequency (5000 Hz), Q factor (1.0 / sqrt(2)), and gain. The gain parameter controls the amplification or attenuation of high-frequency audio signals.

These methods allow adjusting the gain of specific frequency ranges (low, mid, and high) in the audio signal, providing control over the tonal balance or EQ (equalization) of the audio playback.

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

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

    if (slider == &posSlider)
    {
        player->setPositionRelative(slider->getValue());
    }
    if (slider == &lowSlider) {
        DBG("MainComponent::sliderValueChanged: low slider " << slider->getValue());
        player->setLBFilter(slider->getValue());
    }

    if (slider == &midSlider) {
        DBG("MainComponent::sliderValueChanged: Mid slider " << slider->getValue());
        player->setMBFilter(slider->getValue());
    }

    if (slider == &highSlider) {
        DBG("MainComponent::sliderValueChanged: High slider " << slider->getValue());
        player->setHBFilter(slider->getValue());
    }
}
```

Band Slider in DeckGui.cpp

In deckGUI, I create slider for each band to control them by calling the function from DJAudioPlayer.lowSlider, midSlider, highSlider: These sliders control the low, mid, and high audio filters, respectively. When their values change, they call the setLBFilter, setMBFilter, and setHBFilter methods of the DJAudioPlayer instance to adjust the gain of the corresponding frequency bands.

Additionally, **track title and track left** functionalities are implemented. In the DJAudioPlayer class, a boolean variable called trackLoaded is used to determine if track information should be displayed. The trackLeft function calculates the remaining time of the audio track being played, providing users with valuable information about the track's duration.

```cpp
void DeckGUI::paint(Graphics& g)
{


    g.fillAll(themeColor); //background color


    g.setColour(Colours::black);
    g.setFont(14.0f);

    g.drawText("POSITION", getWidth() / 2 - 30, getHeight() -80, 80, 20,
        Justification::centred, true);

    //if file is loaded, calls the DJAudioPlayer to get the title and length left of the track that is currently playing
    if (player->trackLoaded)
    {
        g.setColour(Colours::white);
        g.setFont(16.0f);
        //display track title
        g.drawText("Track Title: " + player->trackPlayingTitle, getWidth() / 5 + 10, getHeight() / 5 + 15, 220, 15, Justification::left, true);
        //display track audio left
        g.drawText("Track Left: " + player->trackLeft(), getWidth() / 5 + 10, getHeight() / 5 + 35, 220, 20, Justification::left, true);
    }

}
```

drawing TrackTile and Track Left in DeckGui.cpp

```cpp
juce::String DJAudioPlayer::trackLeft()
{
    int trackLengthLeft = transportSource.getLengthInSeconds() - transportSource.getCurrentPosition();

    // Calculate minutes and seconds
    int mins = trackLengthLeft / 60;
    int secs = trackLengthLeft % 60;

    // Format the time string
    std::stringstream timeString;
    timeString << std::setfill('0') << std::setw(2) << mins << ":" << std::setw(2) << secs;

    // Return the formatted time string
    return timeString.str();
}
```

TrackLeft function in DJAudioPlayer.cpp

The **trackLeft function** within the DJAudioPlayer class is responsible for determining the remaining time of the currently playing audio track. It carries out this task through several steps: Firstly, it calculates the time remaining by subtracting the current playback position from the total length of the track. Then, it converts this time into minutes and seconds, utilizing division and modulus operations to obtain the respective values. Subsequently, it formats these values into a string with two digits each, ensuring uniformity using std::setfill('0') and std::setw(2). Finally, it returns this formatted time string, providing an accurate representation of the remaining time left in the track.

The **PlaylistComponent** is the core feature for managing playlists in the app. It lets users load tracks into a table, showing their titles, lengths, and options to load or remove them. Clicking "Load" adds a track to a deck for playback, while "Remove" deletes it from the playlist. Users can also load saved playlists or save the current one. Additionally, there's a search box for finding specific songs quickly. In short, it's a straightforward tool for organizing and playing music tracks.

Button Click Handling (buttonClicked method):

The buttonClicked method is responsible for handling button click events within the PlaylistComponent. Here's a more detailed explanation of how it works:

```cpp
void PlaylistComponent::buttonClicked(Button* button)
{


    if (button == &loadPlaylistButton)
    {
        if (getNumRows() >= 5)
        {
            // Hide the middle label
            middleLabel.setVisible(false);
        }

        auto fileChooserFlags =
            FileBrowserComponent::canSelectMultipleItems;
        fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
            {

                auto selectedFiles = chooser.getResults();
                for (const auto& file : selectedFiles)
                {
                    trackTitles.push_back(file.getFileNameWithoutExtension());
                    trackURL.push_back(URL{ File{file.getFullPathName()} });
                    std::cout << "URL: " << file.getFullPathName() << std::endl;
                    trackLength.push_back(getTrackLength(file));
                }
                tableComponent.updateContent();

            });
        tableComponent.updateContent();

    }
    else if (button == &saveePlaylistButton)
```

loadPlaylistButton in PlaylistComponent.cpp

· Load Playlist Button: When the "Load Playlist" button is clicked, it opens a file chooser dialog allowing the user to select multiple audio files to add to the playlist. Once selected, the track titles, URLs, and lengths of the selected tracks are added to the respective vectors (trackTitles, trackURL, trackLength). The table component is then updated to reflect the changes in the playlist.

```
void PlaylistComponent::buttonClicked(Button* button)
{


    if (button == &loadPlaylistButton) { ... }
    else if (button == &savePlaylistButton)
    {

        savePlaylist();
    }
    else
```

SavePlaylistButton in PlaylistComponent.cpp

· **Save Playlist Button**: When the "Save Playlist" button is clicked, it opens a file for writing and saves the current playlist to a CSV file. Each line in the CSV file contains the track title, URL, and length separated by commas. If the file cannot be opened or an error occurs during writing, an error message is displayed to the console.

```
162
163 void PlaylistComponent::buttonClicked(Button* button)
164 {
165
166
167     if (button == &loadPlaylistButton) { ... }
194     else if (button == &savePlaylistButton) { ... }
199     else
200     {
201         //get first index of button id which is row
202         int row = std::stoi(button->getComponentID().toStdString().substr(0, 1));
203         //get second index of button id which is column
204         int col = std::stoi(button->getComponentID().toStdString().substr(1, 2));
205
206         std::cout << "Clicked button in row: " << row << ", column: " << col << std::endl;
207
208         // Now you can safely use row and col indices
209         if (col == 3)
210         {
211             // Your existing implementation for loading to deck1
212             //load the url into deck1 player and waveform, and sets the waveform position
213             std::cout << "URL for deckGUI1: " << URL(trackURL[row]).toString(false) << std::endl;
214
215             deckGUI1->player->loadURL(URL{ trackURL[row] });
216             deckGUI1->waveformDisplay.loadURL(URL{ trackURL[row] });
217             deckGUI1->waveformDisplay.setPositionRelative(deckGUI1->player->getPositionRelative());
218
219             //calls paint so that the track title and length is displayed immediately as the track is loaded in
220             deckGUI1->repaint();
221         }
222         if (col == 4)
223         {
224             // Your existing implementation for loading to deck1
225             //load the url into deck1 player and waveform, and sets the waveform position
226             deckGUI2->player->loadURL(URL{ trackURL[row] });
227             deckGUI2->waveformDisplay.loadURL(URL{ trackURL[row] });
228             deckGUI2->waveformDisplay.setPositionRelative(deckGUI2->player->getPositionRelative());
229
230             //calls paint so that the track title and length is displayed immediately as the track is loaded in
231             deckGUI2->repaint();
232         }
233         if (col == 5) { ... }
241     }
242 }
243
```

Load to Deck Buttons (Column 3 and 4) in PlaylistComponent.cpp

· **Load to Deck Buttons (Column 3 and 4)**: These buttons are located in each row of the playlist table and are used to load tracks to Deck 1 or Deck 2 (associated with `DeckGUI` instances). When clicked, they load the corresponding track to the selected deck, update the waveform display, and repaint the DeckGUI to display track information.

```cpp
void PlaylistComponent::buttonClicked(Button* button)
{


    if (button == &loadPlaylistButton) { ... }
    else if (button == &savePlaylistButton) { ... }
    else
    {
        //get first index of button id which is row
        int row = std::stoi(button->getComponentID().toStdString().substr(0, 1));
        //get second index of button id which is column
        int col = std::stoi(button->getComponentID().toStdString().substr(1, 2));

        std::cout << "Clicked button in row: " << row << ", column: " << col << std::endl;

        // Now you can safely use row and col indices
        if (col == 3) { ... }
        if (col == 4) { ... }
        if (col == 5)
        {
            //remove the title, url, length from the vectors
            trackTitles.erase(trackTitles.begin() + row);
            trackURL.erase(trackURL.begin() + row);
            trackLength.erase(trackLength.begin() + row);
            tableComponent.updateContent();
        }
    }
}
```

remove Button in PlaylistComponent.cpp

· **Remove Button (Column 5)**: This button is also located in each row of the playlist table and is used to remove tracks from the playlist. When clicked, it removes the track title, URL, and length from the respective vectors, and updates the table component to reflect the changes in the playlist.

Playlist Loading and Saving (loadPlaylist and savePlaylist methods):

```cpp
void PlaylistComponent::loadPlaylist()
{
    // Open playlist file for reading
    juce::File playlist(filepath);
    if (playlist.existsAsFile())
    {
        juce::FileInputStream input(playlist);
        input.setPosition(0);

        // Read file line by line
        while (!input.isExhausted())
        {
            // Read line
            std::string line = input.readNextLine().toStdString();
            std::cout << line << std::endl; // Print the line to console

            // Use stringstream to parse the line
            std::stringstream ss(line);

            // Temporary variables for metadata of track
            std::string _trackTitles;
            std::string _trackURL;
            std::string _trackLength;

            // Save track titles (first part of the line) into vector
            getline(ss, _trackTitles, ',');
            trackTitles.push_back(_trackTitles);

            // Save track URL (second part of the line) into vector
            getline(ss, _trackURL, ',');
            trackURL.push_back(URL(_trackURL));

            // Save track length (last part of the line) into vector
            getline(ss, _trackLength, ',');
            trackLength.push_back(_trackLength);
        }
    }
    else
    {
        // Handle the case where the playlist file doesn't exist
        std::cout << "Playlist file not found!" << std::endl;
    }
}
```

loadPlaylist function in PlaylistComponent.cpp

· **Load Playlist**: The loadPlaylist method reads the playlist file line by line, parsing each line to extract the track title, URL, and length. It then adds this information to the corresponding vectors (trackTitles, trackURL, trackLength). If the playlist file does not exist, an error message is displayed to the console.

```
342  /** save the playlist */
343  void PlaylistComponent::savePlaylist()
344  {
345      // Open playlist for writing
346      std::fstream playlist;
347
348      // Get the file path as a String
349      juce::String filePathString = filepath.getFullPathName();
350
351      // Convert the String to std::string
352      std::string filePathStdString = filePathString.toStdString();
353
354      // Open the file with the converted file path
355      playlist.open(filePathStdString, std::fstream::out);
356
357      // Check if the file is successfully opened
358      if (playlist.is_open())
359      {
360          // Save the strings in the vector to the csv file
361          for (int i = 0; i < trackURL.size(); ++i)
362          {
363              playlist << trackTitles[i] << "," << trackURL[i].toString(false) << "," << trackLength[i] << "\n";
364          }
365
366          // Close the file
367          playlist.close();
368      }
369      else
370      {
371          // Handle error opening the file
372          std::cerr << "Error: Unable to open file for writing: " << filePathStdString << std::endl;
373      }
374  }
```

SavePlaylist function in PlaylistComponent.cpp

· **Save Playlist**: The savePlaylist method opens a file for writing and saves the current playlist to a CSV file. It iterates over the vectors containing track information, writing each track's title, URL, and length to a separate line in the file. If an error occurs during file writing, an error message is displayed to the console.

Search Functionality (searchText and returnSearch methods):

```
375  int PlaylistComponent::searchText(const juce::String& inputText)
376  {
377      auto iter = std::find_if(trackTitles.begin(), trackTitles.end(),
378          [&inputText](const juce::String& tracks) {
379              return tracks.contains(inputText);
380          });
381
382      return (iter != trackTitles.end()) ? std::distance(trackTitles.begin(), iter) : 0;
383  }
384
385
386  void PlaylistComponent::returnSearch(juce::String inputText)
387  {
388      if (inputText != "")
389      {
390          int rowNumber = searchText(inputText);
391          tableComponent.selectRow(rowNumber);
392      }
393      else
394      {
395          tableComponent.deselectAllRows();
396      }
397  }
398
```

SearchText and returnSearch function in PlaylistComponent.cpp

· **Search Text**: The searchText method searches for a specific text within the track titles stored in the trackTitles vector. It uses the std::find_if algorithm to search for tracks

containing the input text. If a match is found, it returns the index of the track in the `trackTitles` vector.

· Return Search: The returnSearch method is triggered whenever the text in the search box (searchBox) changes. It calls the searchText method to search for the input text within the track titles. If a match is found, it selects the corresponding row in the playlist table using the selectRow method. If the search box is empty, it deselects all rows in the table.

Another method for loading files exists, akin to the one found in DeckGUI, where we utilize functions like isInterestedInFileDrag and filesDropped to facilitate file dropping for playlist loading. However, the distinction lies in how these files are processed: upon dropping, they are directly pushed into the respective vector to update the PlaylistComponent, thereby streamlining the file loading process.

```
243
244     /** implement FileDragAndDropTarget */
245     bool PlaylistComponent::isInterestedInFileDrag(const StringArray& files)
246     {
247         std::cout << "DeckGUI::isInterestedInFileDrag" << std::endl;
248         return true;
249     }
250
251     /** implement FileDragAndDropTarget */
252     void PlaylistComponent::filesDropped(const StringArray& files, int x, int y)
253     {
254         std::cout << "DeckGUI::filesDropped" << std::endl;
255         //load all files into playlist
256         if (files.size() >= 1)
257         {
258             for (int i = 0; i < files.size(); ++i)
259             {
260                 //push the track title into vector
261                 trackTitles.push_back(File{ files[i] }.getFileNameWithoutExtension());
262                 //push the track URL into vector
263                 trackURL.push_back(URL{ File{files[i]} }.toString(false));
264                 //push the track length into vector
265                 trackLength.push_back(getTrackLength(File{ files[i] }));
266
267                 std::cout << "URL for deckGUI1: " << URL(trackURL[i]).toString(true) << std::endl;
268             }
269             //update the table
270             tableComponent.updateContent();
271         }
272     }
```

isInterestedInFileDrag and filesDropped in PlaylistComponent.cpp

In conclusion, taking cues from the YouTube DJ app, I've upgraded my own application with various improvements. By adopting and tweaking features found in the YouTube DJ app, I've made my application more user-friendly and in line with modern expectations. Learning from existing platforms, I've polished my app over time to provide a smoother and more intuitive experience for users.

Reference :

· Icons - https://www.flaticon.com/search?type=icon&search-group=all&word=volume+knob&license=&color=&shape=&current_section=&author_id=&pack_id=&family_id=&style_id=&type=

· Image - https://www.canva.com/

· Youtube - https://www.youtube.com/watch?v=moNxobdTV3k