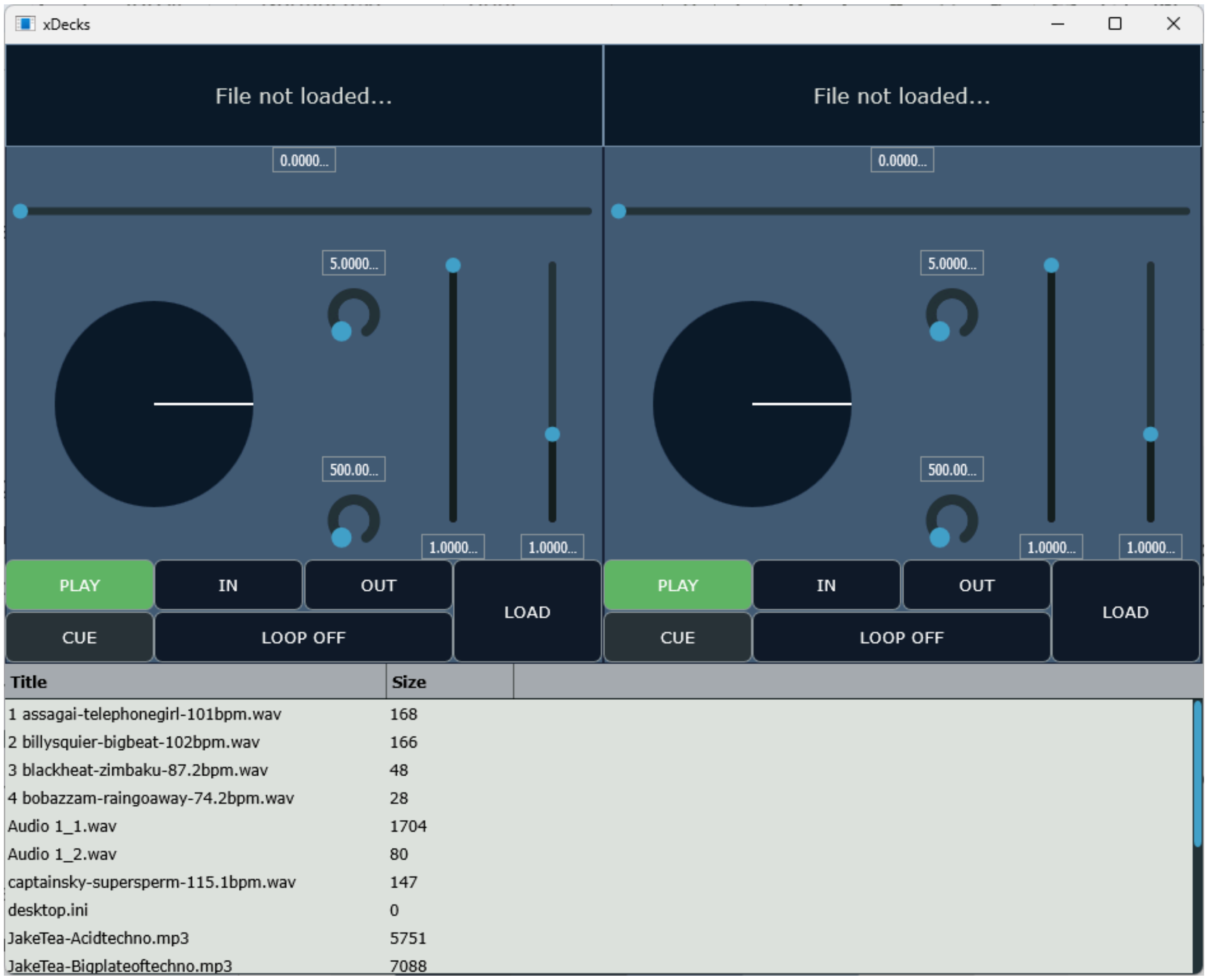
Student number: 210323041

Object Oriented Programming Final Project Report

xDecks



Title	Size
1 assagai-telephonegirl-101bpm.wav	168
2 billysquier-bigbeat-102bpm.wav	166
3 blackheat-zimbaku-87.2bpm.wav	48
4 bobazzam-raingoaway-74.2bpm.wav	28
Audio 1_1.wav	1704
Audio 1_2.wav	80
captainsky-supersperm-115.1bpm.wav	147
desktop.ini	0
JakeTea-Acidtechno.mp3	5751
JakeTea-Bigplateoftechno.mp3	7088

Introduction

For my Object-Oriented Programming final project, I had to create a DJing application that will allow users to mix two tracks together by changing the speed and volume of two separate tracks that can be loaded into a program.
In my early 20s, I played around with DJing and creating my own music, so when I heard that we were going to build a DJing application from scratch I was really excited, as it always amazed me, how this kind of software works. I followed the lectures and build the application along with Matthew, but eventually, I wanted my application to be unique. I named it **xDecks.**

How the program works

Even though the GUI of the program looks simple the underlying logic and everything that happens "under the hood" is quite impressive and complicated. The program consists of the following classes:

- DeckGui - manages the graphical interface. DeckGUI Object represents one deck in a DJ setup. That's where you control the application.
- DJAudioPlayer -
- MainComponent
- PlaylistComponent
- WaveformDisplay

There is also an **xDecksApplication** class that runs the whole program.

High-Level Architecture

At a high level, the application consists of the following key components:

1. **MainComponent**: this is where the whole program comes together. In MainComponenet other classes are called that are necessary for the application to run. The general look of the application is also set up here; the window is divided into a few separate parts.
2. **DeckGUI**: manages the graphical interface. DeckGUI Object represents one deck in a DJ setup. That's where the controls for playing, speed, volume, and filters are set up. This class also holds a place for a WaveFormDisplay. Each `DeckGUI` object is a self-contained unit with its own set of controls.
3. **WaveformDisplay**: Responsible for displaying the visual representation of the audio file. It also shows the part of the track that is currently played by drawing a vertical rectangle in a waveform. I added the visual representation of the part of the audio file that has already been played.
4. **DJAudioPlayer**: It handles all the audio processing. The instance of this class receives instructions such as play/pause command, speed, filters etc. from DeckGUI and applies them to the audio file.

5. **PlaylistComponent**: A component in the program that displays the tracks available to play. Next to each track is a play button. I created my playlist component to show the files in the Music folder. (hardcoded)

MainComponent

The `MainComponent` initializes two instances of `DJAudioPlayer: player2` and `player2` which are then passed on to the initialized instances of `DeckGUI: deck1` and `deck2`. The general layout of the app is also defined here. The virtual functions that are responsible for managing the audio `AudioFormatManager` and `MixerAudioSource` as well as the `AudioThumbnailCache` are also instantiated within this class.

DeckGUI

The `DeckGUI` is where all the controls and visual representations of the program come together. It interacts with the `DJAudioPlayer` to control the audio file using various buttons and sliders such as:

- `playButton`
- `cueButton`
- `volSlider`
- `speedSlider`
- `positionSlider`
- `waveformDisplay`
- `loadButton`
- `highKnob (custom)`
- `lowKnob (custom)`
- `inLoopButton (custom)`
- `outLoopButton (custom)`
- `loopButton (custom)`

Additional functionality

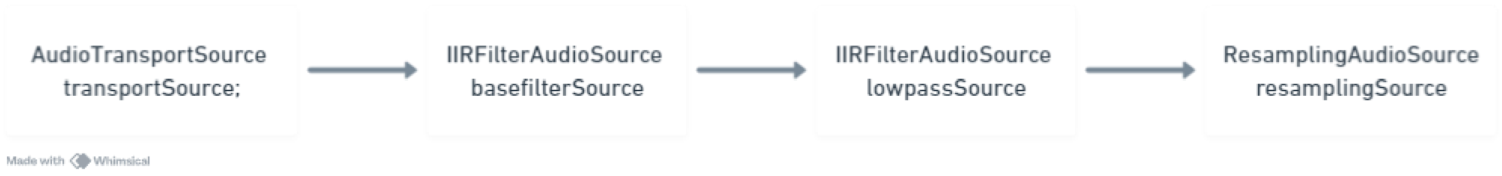
Apart from the basic functionality that was presented in the lectures, I've added 5 additional listeners (labelled "custom" above). `highKnob` and `lowKnob` are respectively high pass and low pass filters that cut the sounds above/below a certain frequency. The cut-off frequency is set using a knob. One of the functionalities very often used in digital DJing is looping this invaluable function allows DJs to extend the time of their favourite part of the audio file by repeatedly playing it. I implemented my looping function by adding the `inLoopButton,` `outLoopButton` and `loopButton` that respectively control when the loop starts, ends and whether you want to play it or not. Here the position of each component is defined in the `resized` function.

DJAudioPlayer

Here is where the magic “happens”.Instances of this class are responsible for controlling the digital audio signal and managing the low-level architecture and backend of the application. It is responsible for loading the file into the program; `loadURL`, adjusting the speed of the playback by increasing/decreasing the sampling ratio; `setSpeed,` as well as other controls such as play, pause, gain etc.

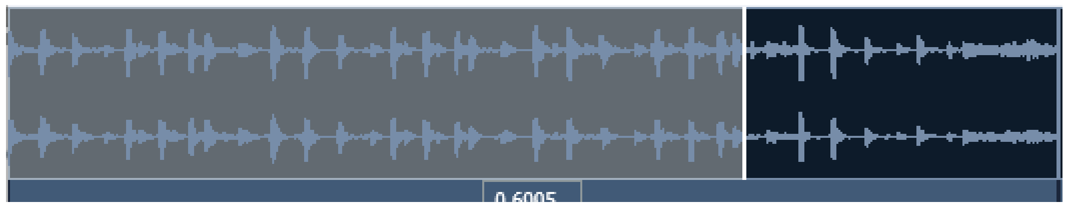
Additional functionality

In DeckGUI I added low and high pass filter knobs. Their value is then passed on to the functions `setLowPass` and `setHighPass` in the `DjAudioPlayer`. In order for this to work as planned I had to instantiate the `IIRFilterAudioSource` for each filter. The signal is then passed linearly like in the diagram below.[1]


AudioTransportSource transportSource;
IIRFilterAudioSource basefilterSource
IIRFilterAudioSource lowpassSource
ResamplingAudioSource resamplingSource
Made with Whimsical

WaveformDisplay

The instance of this class receives the `AudioFormatManager` and `AudioThumbnailCache` in order to generate a visual representation of the audio file. This object is called from the instance of the `DeckGUI` to be drawn for each deck. It is fairly simple class in comparison to the `DeckGUI` or `DJAudioPlayer`. As an additional feature, I’ve added the visual representation of what has already been played.



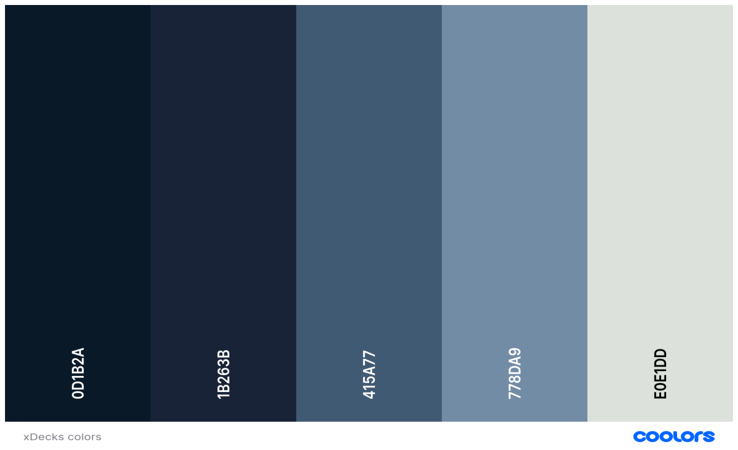
[1] Code based on the: mg79 *et al.* (2022) *Bass treble mid equaliser*, *JUCE*. Available at: https://forum.juce.com/t/bass-treble-mid-equaliser/52245 (Accessed: 23 August 2024).

PlaylistComponent

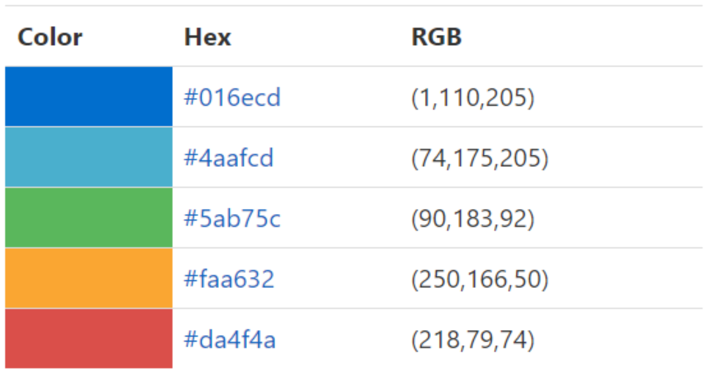
This class shows a table of the tracks available to play for the DJ. I left it untouched except for the functionality that it shows all the files in my music folder. The path to the folder is hard-coded and the function is AI-generated. I removed the play button and the artist columns. I left the component as a peek into what is available in the Music folder.

Customised interface and new features

Most DJing application I came across were dark and mostly in gray colors. I've decided to give my application a bit of modern and minimalistic look. I decided to choose the following colour palette for the different components:



And the following palette for the buttons and controls[2]:



Color	Hex	RGB
🟦	#016ecd	(1,110,205)
🟦	#4aafcd	(74,175,205)
🟩	#5ab75c	(90,183,92)
🟧	#faa632	(250,166,50)
🟥	#da4f4a	(218,79,74)

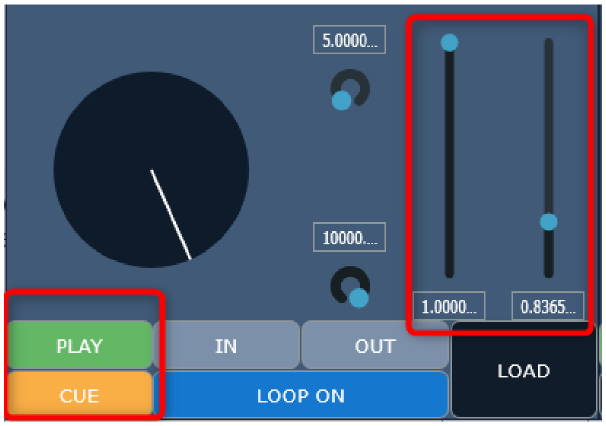
I took my inspiration not only from other software but also from hardware and tried to imitate the CDJ, DJ controllers and record players.

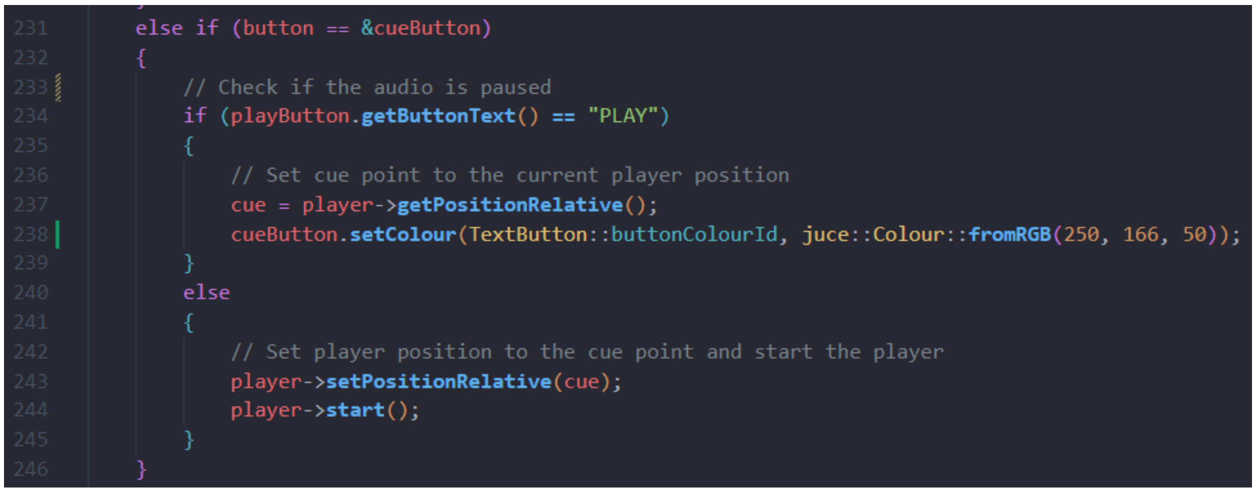
1. **Play / pause and cue buttons:** Namely, when it comes to the play/pause button and cue button on the bottom left on the CDJ-2000NXS2, as well as the speed slider on the right. The

[2] https://www.color-hex.com/color-palette/2714

same goes for Technics SL-1200 MK2. One noticeable feature is that there is no separate button for play and pause. It's one button that, depending on the state, has both of those functions. That's how I also implemented my play/pause button.

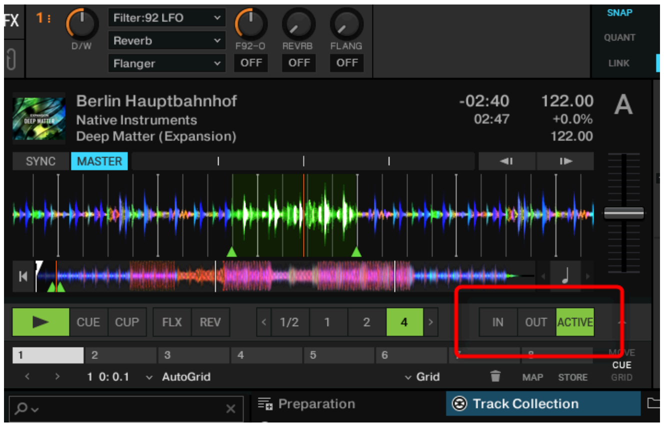
The CUE button, when pressed while audio is paused sets the cue point. To do this it makes use of the `getPositionRelative` function. Then when the CUE button is pressed while the audio is playing it moves the play head to the previously set cue point using `setPositionRelative`. If no cue point was set it moves to the start of the audio file.

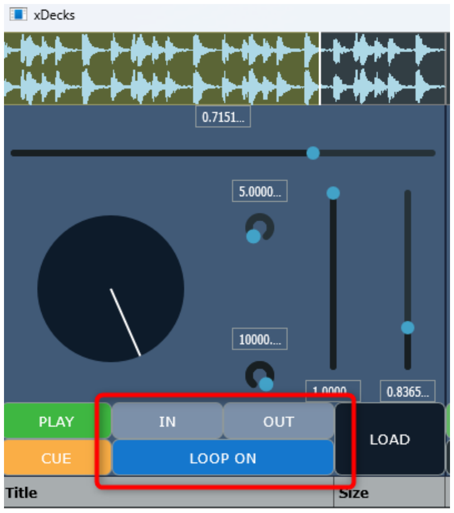




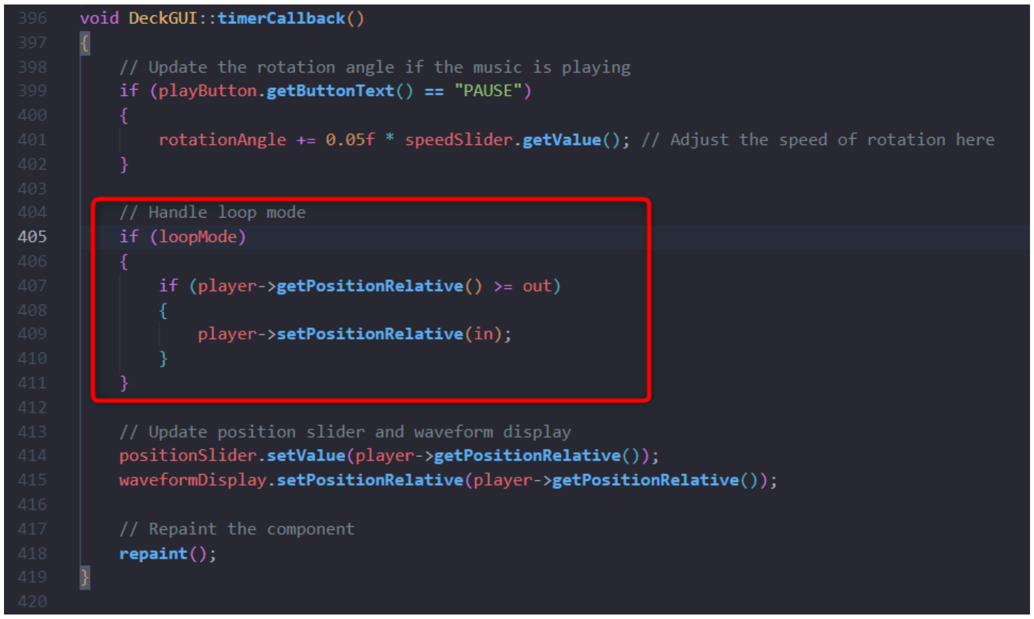
```
        else if (button == &cueButton)
        {
            // Check if the audio is paused
            if (playButton.getButtonText() == "PLAY")
            {
                // Set cue point to the current player position
                cue = player->getPositionRelative();
                cueButton.setColour(TextButton::buttonColourId, juce::Colour::fromRGB(250, 166, 50));
            }
            else
            {
                // Set player position to the cue point and start the player
                player->setPositionRelative(cue);
                player->start();
            }
        }
```

2. **Loop:** I also looked at several DJing applications. I found VirtualDJ and Traktor Pro 3 the most interesting. I have the Traktor Pro 3 installed on my computer and one functionality that I often use is the Loop. Traktor has 3 buttons to manage the loop: IN, OUT and ACTIVE. This functionality is very handy for DJing and I decided to implement it also in my application.

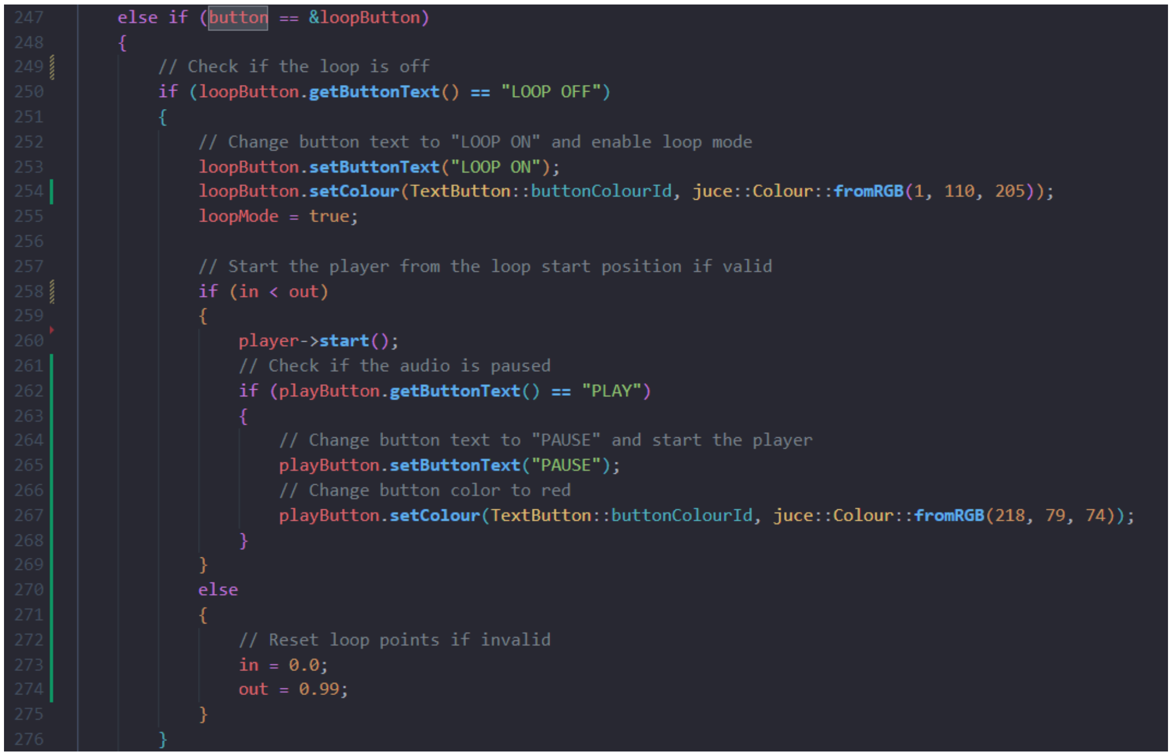




The logic for the loop function works as follows: the IN and OUT are initially set to, respectively, the start and end of the audio file.
<u>Happy flow:</u> At any given time the user sets a new IN point, then a new OUT point, and then they can turn the loop on, by pressing the LOOP ON/OFF switch. The sample of the audio is then played in a loop. When no points are set by the user, the whole audio file will be looped.
<u>Unhappy flow:</u> The user sets the IN and OUT points and then decides to get out of the loop and set a new IN point that is greater then previous OUT point. In this case, the program automatically sets the new OUT point to be at the end of the audio file. Analogous to this situation, when the OUT point is smaller than the IN point the program sets IN to be at the start of the audio.
IN and OUT buttons, similarly to the CUE button rely on `getPositionRelative` function.
The functionality is implemented in the loop switch button logic. It switches the `loopMode` to `true` and checks whether IN is smaller than OUT. When the `loopMode` is switched to `true` the `timerCallBack` function handles the changing the position of the play head to be between IN and OUT bounds.



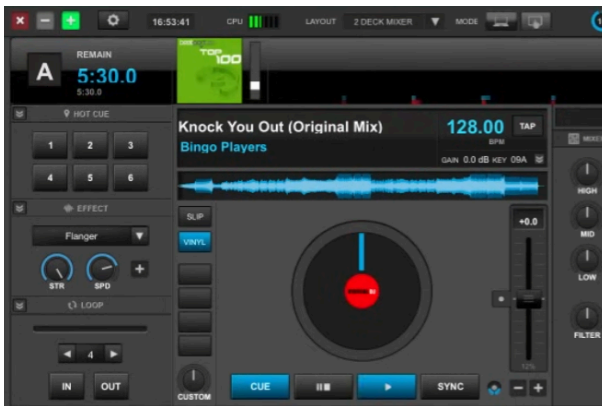
```cpp
void DeckGUI::timerCallback()
{
    // Update the rotation angle if the music is playing
    if (playButton.getButtonText() == "PAUSE")
    {
        rotationAngle += 0.05f * speedSlider.getValue(); // Adjust the speed of rotation here
    }

    // Handle loop mode
    if (loopMode)
    {
        if (player->getPositionRelative() >= out)
        {
            player->setPositionRelative(in);
        }
    }

    // Update position slider and waveform display
    positionSlider.setValue(player->getPositionRelative());
    waveformDisplay.setPositionRelative(player->getPositionRelative());

    // Repaint the component
    repaint();
}
```

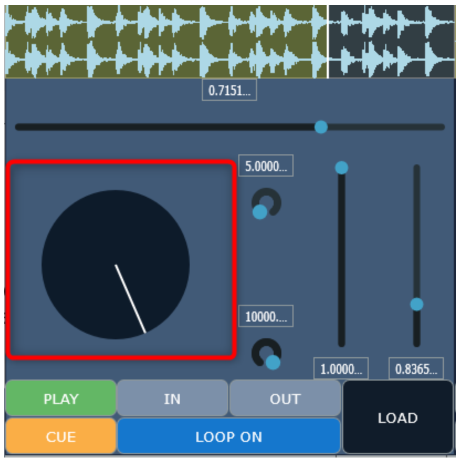


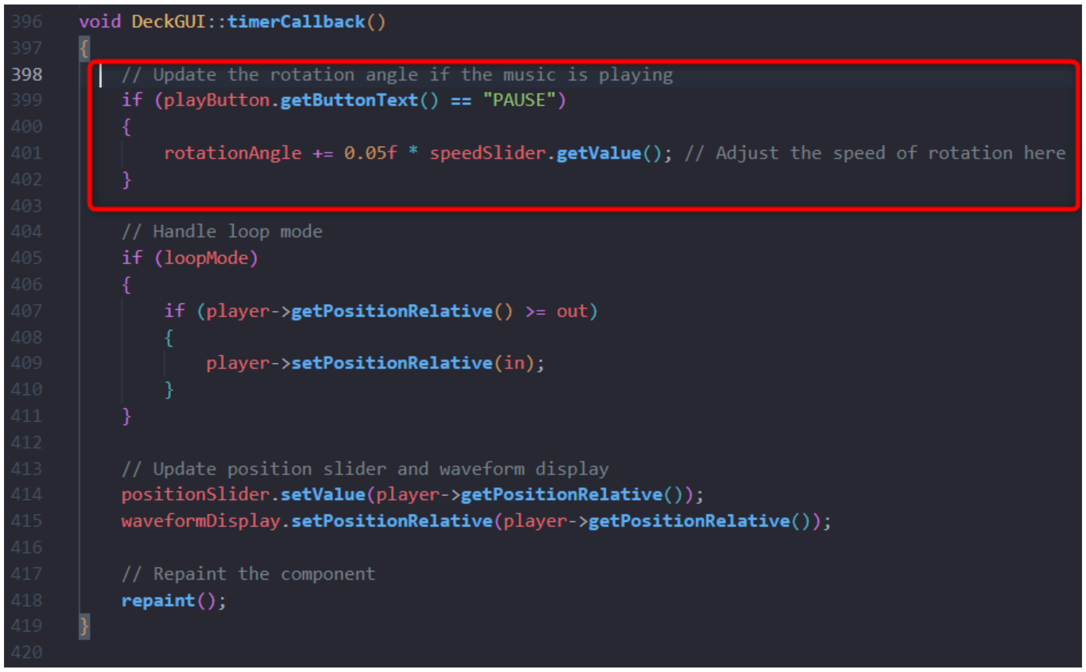
```cpp
    else if (button == &loopButton)
    {
        // Check if the loop is off
        if (loopButton.getButtonText() == "LOOP OFF")
        {
            // Change button text to "LOOP ON" and enable loop mode
            loopButton.setButtonText("LOOP ON");
            loopButton.setColour(TextButton::buttonColourId, juce::Colour::fromRGB(1, 110, 205));
            loopMode = true;

            // Start the player from the loop start position if valid
            if (in < out)
            {
                player->start();
                // Check if the audio is paused
                if (playButton.getButtonText() == "PLAY")
                {
                    // Change button text to "PAUSE" and start the player
                    playButton.setButtonText("PAUSE");
                    // Change button color to red
                    playButton.setColour(TextButton::buttonColourId, juce::Colour::fromRGB(218, 79, 74));
                }
            }
            else
            {
                // Reset loop points if invalid
                in = 0.0;
                out = 0.99;
            }
        }
```

3. **Spinning wheel:** Virtual DJ has this spinning wheel in the middle. I don't see any purpose for it apart from being aesthetically pleasing. I really liked this idea and wanted to add it to my project.

The spinning wheel is drawn in the `paint` function. But the rotation is applied using the `timerCallBack` function. This allows for the circle to spin only when the music is playing.



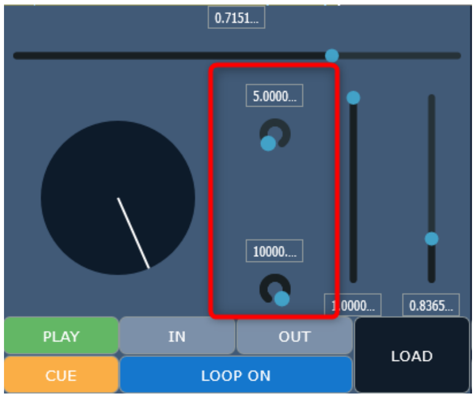




```
void DeckGUI::timerCallback()
{
    // Update the rotation angle if the music is playing
    if (playButton.getButtonText() == "PAUSE")
    {
        rotationAngle += 0.05f * speedSlider.getValue(); // Adjust the speed of rotation here
    }

    // Handle loop mode
    if (loopMode)
    {
        if (player->getPositionRelative() >= out)
        {
            player->setPositionRelative(in);
        }
    }

    // Update position slider and waveform display
    positionSlider.setValue(player->getPositionRelative());
    waveformDisplay.setPositionRelative(player->getPositionRelative());

    // Repaint the component
    repaint();
}
```

4. **High and low pass filters:** In modern DJing, filters play a crucial role. They allow for more interesting transitions between the two tracks. I find this functionality to add this professional sound and vibe to the track. As mentioned in the DJAudioPlayer section, I found the code for this in the JUCE forum.

In the DeckGUI they are just simple rotary sliders, but for the implementation, the two functions needed to be added in the DJAudioPlayer class as well as two instances of the `IIRFilterAudioSource` needed to be added in the header file.


0.7151...
5.0000...
10000....
1.0000...
0.8365...
PLAY
IN
OUT
LOAD
CUE
LOOP ON