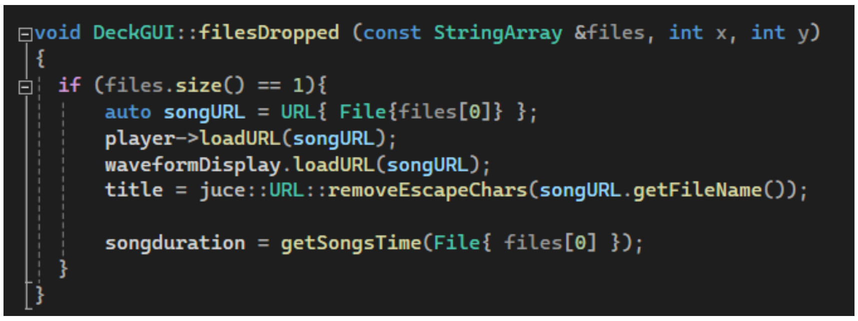
CM2005 Object Oriented Programming End-of-Term assignment

1. Description of how the basic program works (R1)

◦ **R1A: can load audio files into audio players**

Without considering the playlist component, the only way to directly load audio into the player is by dragging the audio file over the player.



```
void DeckGUI::filesDropped (const StringArray &files, int x, int y)
{
    if (files.size() == 1){
        auto songURL = URL{ File{files[0]} };
        player->loadURL(songURL);
        waveformDisplay.loadURL(songURL);
        title = juce::URL::removeEscapeChars(songURL.getFileName());

        songduration = getSongsTime(File{ files[0] });
    }
}
```

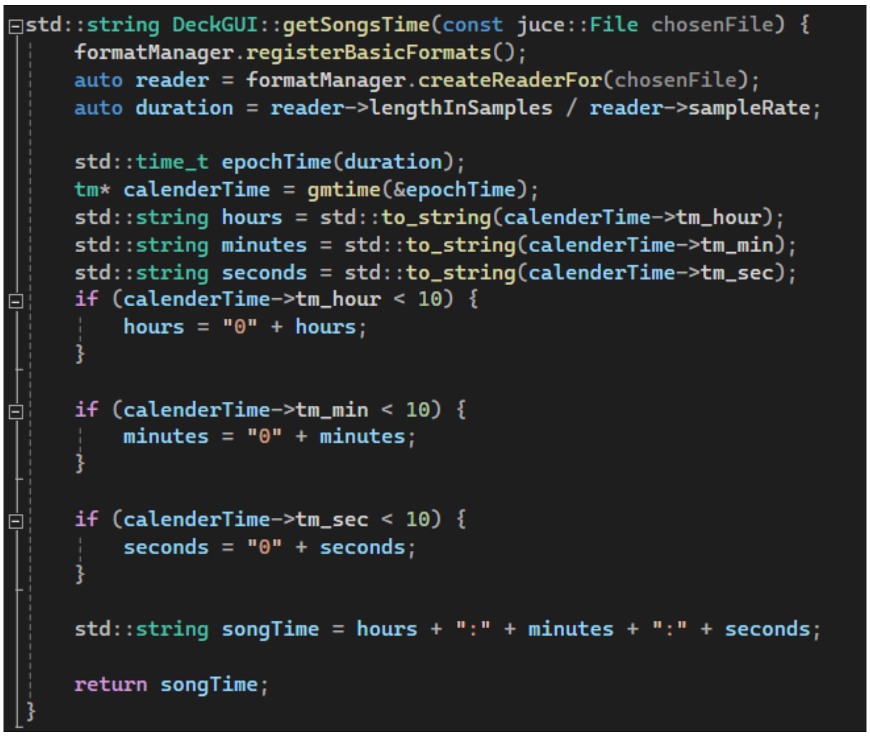
DeckGUI's filesDropped()

It works by utilising the filesDropped() function in the DeckGUI.cpp. After you release the audio file after dragging it over the player, the directory of the file will be stored as a string in the input array 'files'. Before loading the audio file, there will be an 'if' condition to ensure that the player only loads one song when the user drag-and-drops multiple audio files. To add on, the user could drag multiple audio files into the playlist below instead.

Initially, the audio directory will be converted into a Uniform Resource Locator (URL) from the string format. The URL is then used as input for the DJAudioPlayer::loadURL() and WaveformDisplay::loadURL() to load the audio into the player and display its wave respectively.
DJAudioPlayer::loadURL() essentially uses the Audio Transport Source class to take an audio file, process it, and allow it to be played, stopped, started, etc.
While WaveformDisplay::loadURL() uses the Audio Thumbnail class to quickly draw scaled views of the waveform shape of an audio file

Afterwards, the title of the song will be retrieved by removing the path from the audio directory. This is done by passing the audio directory as a string through the juce::URL::removeEscapeChars() function.

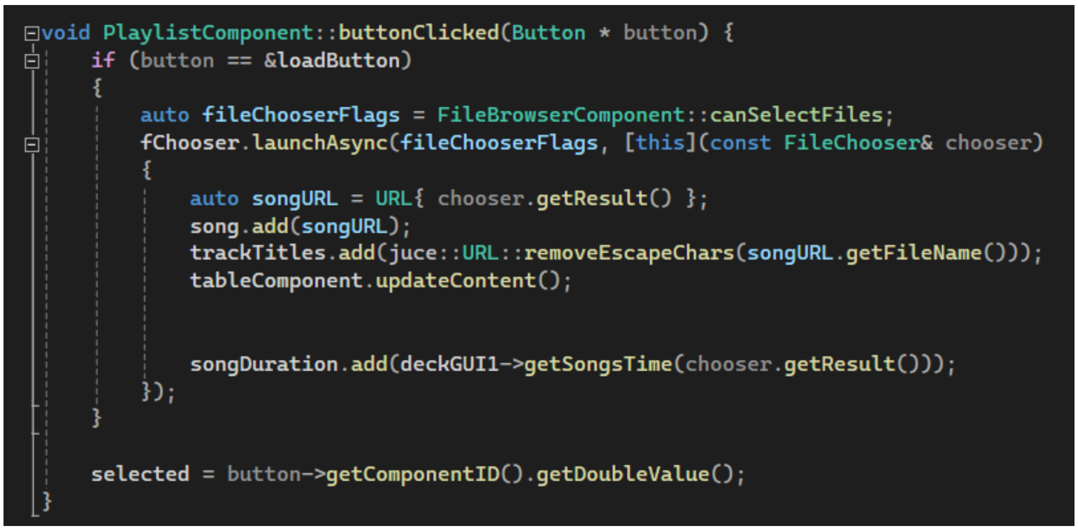
Lastly, the file's audio duration is retrieved by passing the audio directory through the function DeckGUI::getSongsTime(). DeckGUI::getSongsTime() is a function that I implemented in. It works by initially finding the length of an audio depending on its number of samples. Taking the result from dividing the total number of samples by the sample rate, I will convert the result into the epoch time format. The epoch time format is then converted to calendar time to get a more easily human-readable time format. By converting the hours, minutes, and seconds into strings, we can combine them into a single string. This output string shows the length of the loaded audio in the format of "hours:minutes: seconds".



```cpp
std::string DeckGUI::getSongsTime(const juce::File chosenFile) {
    formatManager.registerBasicFormats();
    auto reader = formatManager.createReaderFor(chosenFile);
    auto duration = reader->lengthInSamples / reader->sampleRate;

    std::time_t epochTime(duration);
    tm* calenderTime = gmtime(&epochTime);
    std::string hours = std::to_string(calenderTime->tm_hour);
    std::string minutes = std::to_string(calenderTime->tm_min);
    std::string seconds = std::to_string(calenderTime->tm_sec);
    if (calenderTime->tm_hour < 10) {
        hours = "0" + hours;
    }

    if (calenderTime->tm_min < 10) {
        minutes = "0" + minutes;
    }

    if (calenderTime->tm_sec < 10) {
        seconds = "0" + seconds;
    }

    std::string songTime = hours + ":" + minutes + ":" + seconds;

    return songTime;
}
```

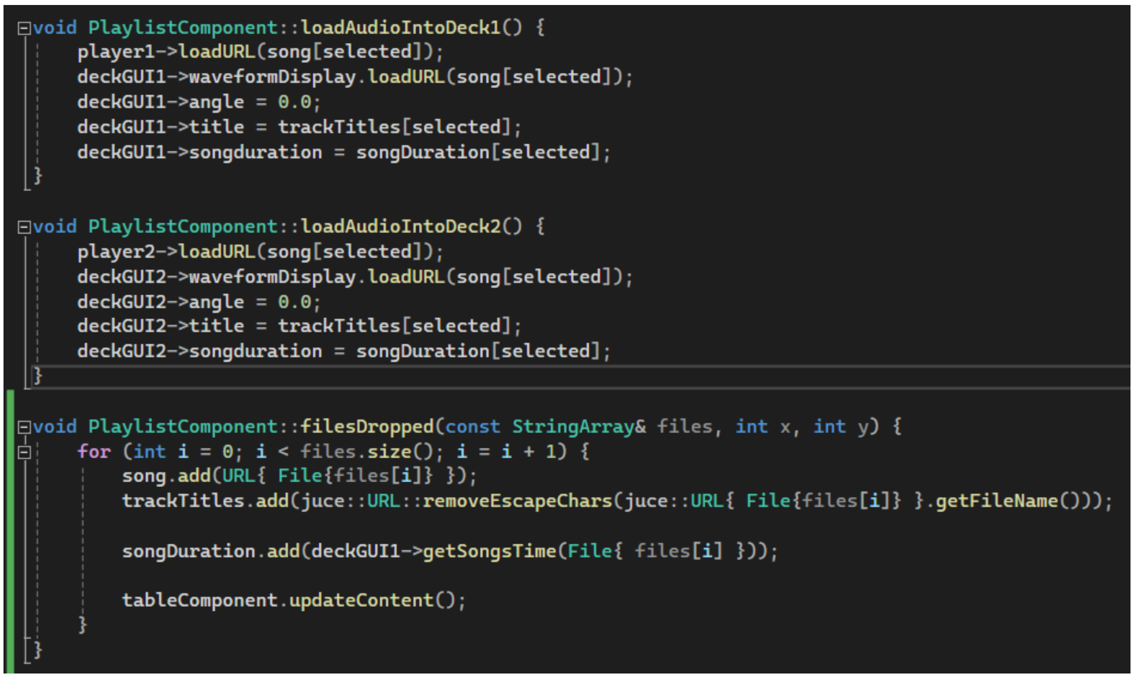
DeckGUI’s getSongsTime()

As for the playlist, the user could also load songs from there. The user has two methods to add songs to the players. The user could either add one song individually by pressing the “LOAD A SONG INTO THE PLAYLIST” button above the playlist or manually dragging multiple files onto the playlist (The code to drag in files is the same as the one in DeckGUI.cpp, except that it can add multiple files by iterating over the dragged files).



```cpp
void PlaylistComponent::buttonClicked(Button * button) {
    if (button == &loadButton)
    {
        auto fileChooserFlags = FileBrowserComponent::canSelectFiles;
        fChooser.launchAsync(fileChooserFlags, [this](const FileChooser& chooser)
        {
            auto songURL = URL{ chooser.getResult() };
            song.add(songURL);
            trackTitles.add(juce::URL::removeEscapeChars(songURL.getFileName()));
            tableComponent.updateContent();


            songDuration.add(deckGUI1->getSongsTime(chooser.getResult()));
        });
    }

    selected = button->getComponentID().getDoubleValue();
}
```

PlaylistComponent’s buttonClicked() to add a song into the playlist

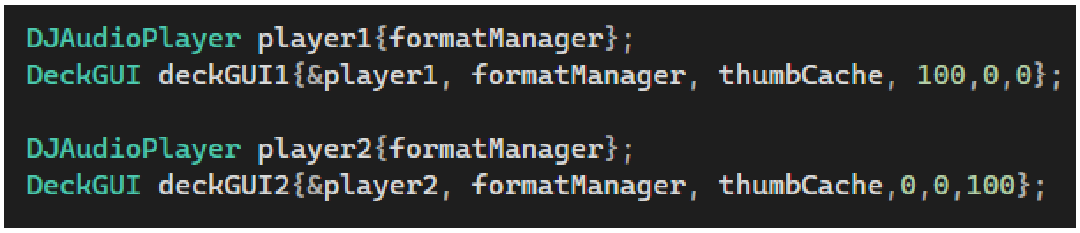


```
void PlaylistComponent::loadAudioIntoDeck1() {
    player1->loadURL(song[selected]);
    deckGUI1->waveformDisplay.loadURL(song[selected]);
    deckGUI1->angle = 0.0;
    deckGUI1->title = trackTitles[selected];
    deckGUI1->songduration = songDuration[selected];
}

void PlaylistComponent::loadAudioIntoDeck2() {
    player2->loadURL(song[selected]);
    deckGUI2->waveformDisplay.loadURL(song[selected]);
    deckGUI2->angle = 0.0;
    deckGUI2->title = trackTitles[selected];
    deckGUI2->songduration = songDuration[selected];
}

void PlaylistComponent::filesDropped(const StringArray& files, int x, int y) {
    for (int i = 0; i < files.size(); i = i + 1) {
        song.add(URL{ File{files[i]} });
        trackTitles.add(juce::URL::removeEscapeChars(juce::URL{ File{files[i]} }.getFileName()));

        songDuration.add(deckGUI1->getSongsTime(File{ files[i] }));

        tableComponent.updateContent();
    }
}
```

The other functions to load audio files into the decks and playlist

◦ **R1B: can play two or more tracks**

By initialising DeckGUI twice as deckGUI1 and deckGUI2 in the MainComponent.h, it will allow two instances of the player to exist. Allowing the user to load and play at most two audios at the same time.

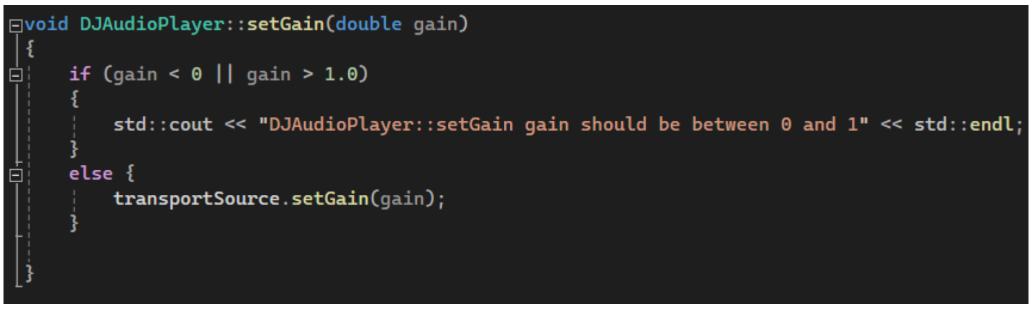


```
DJAudioPlayer player1{formatManager};
DeckGUI deckGUI1{&player1, formatManager, thumbCache, 100,0,0};

DJAudioPlayer player2{formatManager};
DeckGUI deckGUI2{&player2, formatManager, thumbCache,0,0,100};
```

Defined player1, player2, deckGUI1, and deckGUI2 in MainComponent.h

◦ **R1C: can mix the tracks by varying each of their volumes**

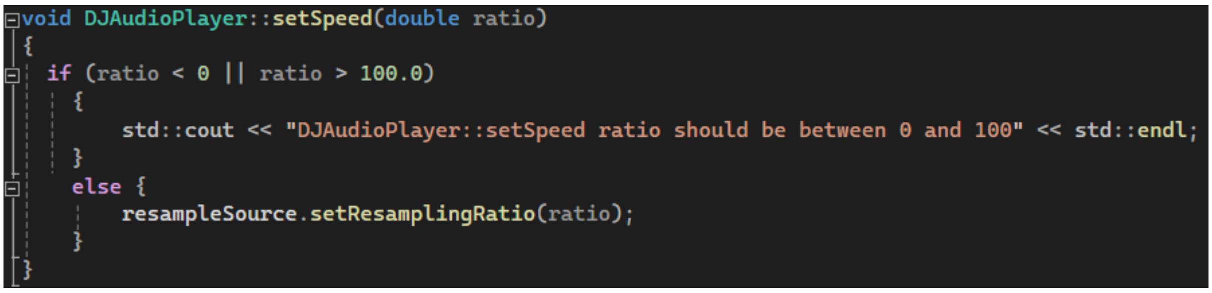
Varying the volume of each track is the purpose of the volume rotary slider. Every time the user interacts with the volume slider (with the in-file name of volSlider), the value is returned and used as input for the function DJAudioPlayer::setGain(). This function uses the Audio Transport Source class, where it will use the input value (which ranges from 0.0 to 2.0) and multiply it by the outgoing samples. This will increase or decrease the decibel output value of the track, allowing the track to be played at 0% ranges to 100% of the audio's default volume.



```
void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl;
    }
    else {
        transportSource.setGain(gain);
    }

}
```

DJAudioPlayer’s setGain()

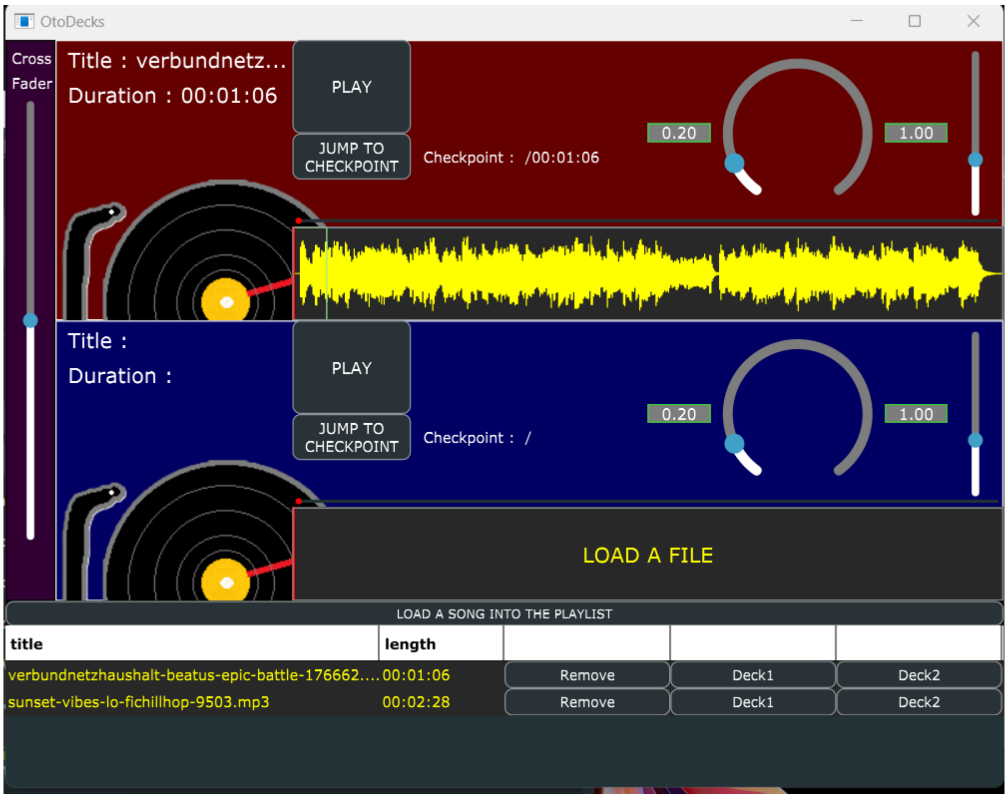
◦ **R1D: can speed up and slow down the tracks**
Changing the speed play rate of a track is the purpose of the speed linear slider. Every time the user interacts with the speed slider (with the in-file name of speedSlider), the value is returned and used as input for the function DJAudioPlayer::setSpeed(). This function uses the Resampling Audio Source class, where it will use the input value (which ranges from 0.0 to 3.0) to change the resampling ratio accordingly. This will allow the track to be played at 0% ranges to 300% of the audio’s original play speed.



```
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);
    }
}
```

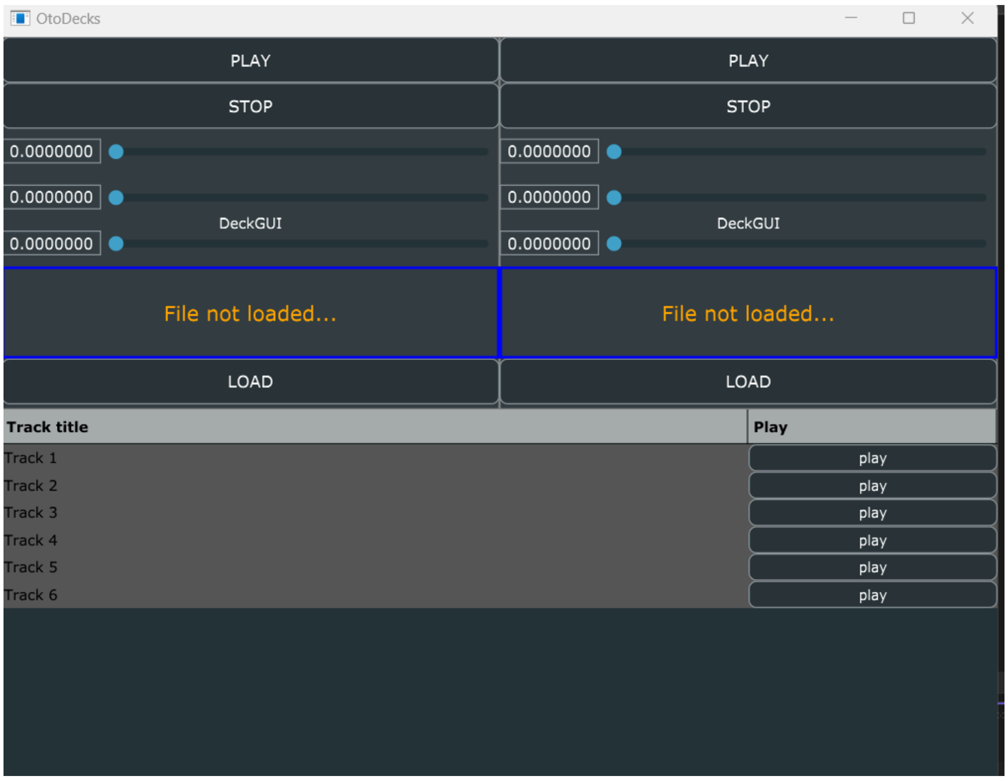
DJAudioPlayer’s setSpeed()

2. Description of how you went about customising the user interface (R2)



title	length			
verbundnetzhaushalt-beatus-epic-battle-176662....	00:01:06	Remove	Deck1	Deck2
sunset-vibes-lo-fichillhop-9503.mp3	00:02:28	Remove	Deck1	Deck2

My DJ application layout



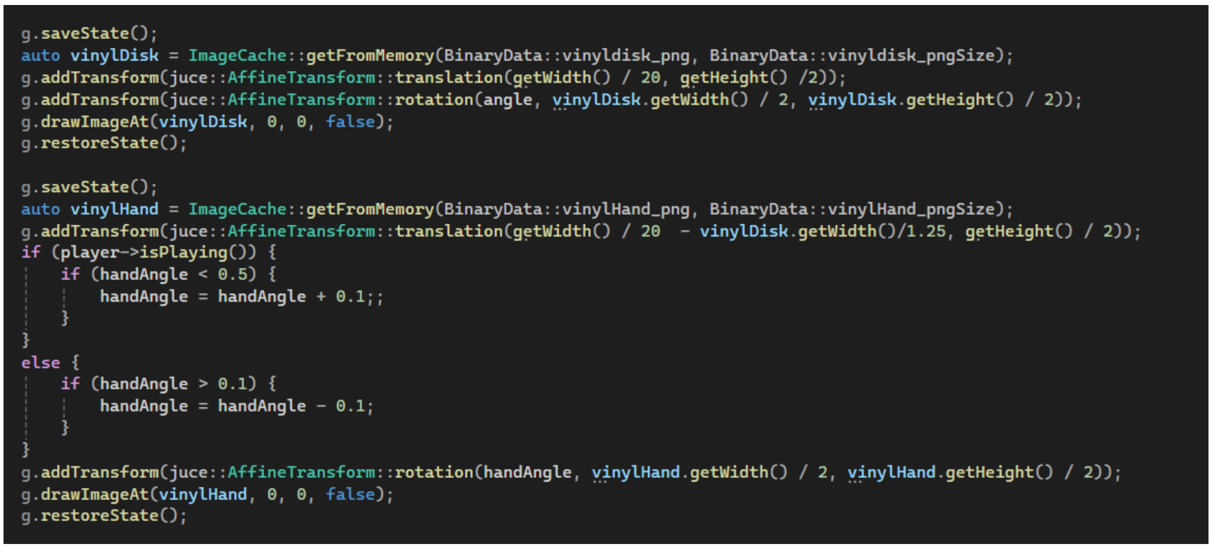
Track title	Play
Track 1	play
Track 2	play
Track 3	play
Track 4	play
Track 5	play
Track 6	play

The default DJ application layout

◦ R2A: GUI layout is significantly different from the basic DeckGUI shown in class

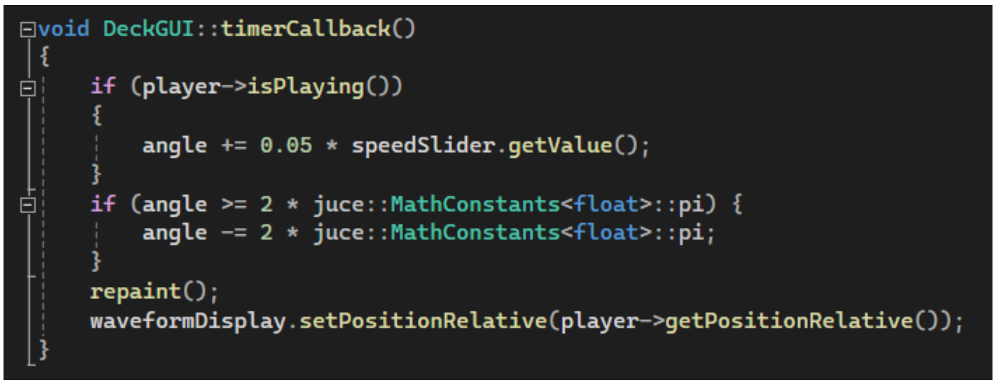
What makes a person want to use a DJ application is not just the functionality it provides, it's also how the application looks visually.

To make the DJ application look less empty, I've decided to implement the spinning vinyl disk and the cue arm. Both the vinyl disk and the cue arm are drawn by me and saved as PNG. The cue arm will rotate onto the vinyl disk whenever the track is being played, and rotate out of whenever the track stops. They can rotate because of the 'addTransform(juce::AffineTransform::rotation(angle,x,y));', where it will rotate the graphic 'g'. Additionally, by utilising the 'saveState()' and 'restoreState()', I can make the vinyl disk and the cue arm rotate independently. Lastly, just for a nice touch-up, the vinyl disk's rotation speed does increase depending on the playing speed of the song (speedSlider.getValue();).



```cpp
g.saveState();
auto vinylDisk = ImageCache::getFromMemory(BinaryData::vinyldisk_png, BinaryData::vinyldisk_pngSize);
g.addTransform(juce::AffineTransform::translation(getWidth() / 20, getHeight() /2));
g.addTransform(juce::AffineTransform::rotation(angle, vinylDisk.getWidth() / 2, vinylDisk.getHeight() / 2));
g.drawImageAt(vinylDisk, 0, 0, false);
g.restoreState();

g.saveState();
auto vinylHand = ImageCache::getFromMemory(BinaryData::vinylHand_png, BinaryData::vinylHand_pngSize);
g.addTransform(juce::AffineTransform::translation(getWidth() / 20  - vinylDisk.getWidth()/1.25, getHeight() / 2));
if (player->isPlaying()) {
    if (handAngle < 0.5) {
        handAngle = handAngle + 0.1;;
    }
}
else {
    if (handAngle > 0.1) {
        handAngle = handAngle - 0.1;
    }
}
g.addTransform(juce::AffineTransform::rotation(handAngle, vinylHand.getWidth() / 2, vinylHand.getHeight() / 2));
g.drawImageAt(vinylHand, 0, 0, false);
g.restoreState();
```

A section of DeckGUI's paint()

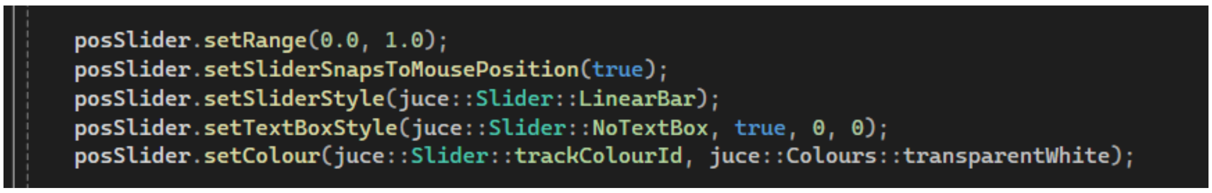


```cpp
void DeckGUI::timerCallback()
{
    if (player->isPlaying())
    {
        angle += 0.05 * speedSlider.getValue();
    }
    if (angle >= 2 * juce::MathConstants<float>::pi) {
        angle -= 2 * juce::MathConstants<float>::pi;
    }
    repaint();
    waveformDisplay.setPositionRelative(player->getPositionRelative());
}
```

DeckGUI's timerCallback()

I find the position slider (posSlider) is out of place because its length is not the same as the song's waveform, so we have to approximate where we want to skip. Additionally, the numerical value in the text box beside it is useless. These attributes make it uncomfortable to use the posSlider.

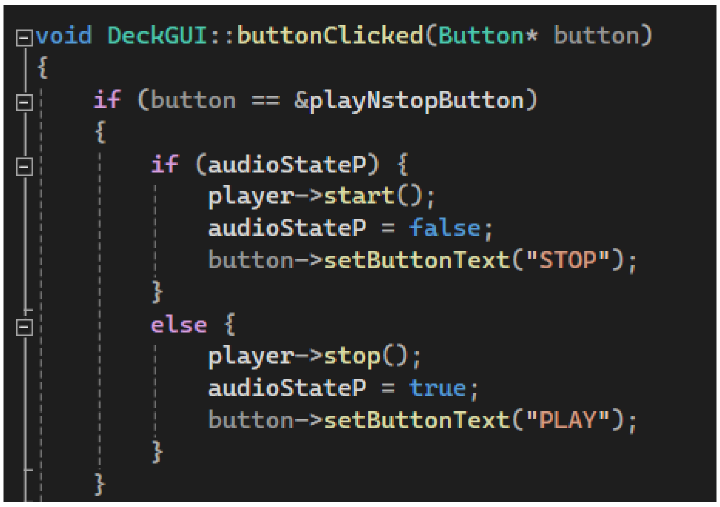
Therefore to improve the user experience, I decided to remove the numerical box beside the posSlider, make the slider completely transparent, and then move it over the waveform. I removed the numerical box because it does not contribute anything to us and the number on it can be misleading. I made the slider transparent and placed it over the waveform so the slider would not block the waveform, and the user could skip to their desired section of the song by clicking on the waveform, instead of approximating where the user would need to be with the old slider's design. This is done with 'posSlider.setBounds(getWidth() / 4, getHeight() * 2 / 3, getWidth()*3/4, getHeight() / 3);'



```
    posSlider.setRange(0.0, 1.0);
    posSlider.setSliderSnapsToMousePosition(true);
    posSlider.setSliderStyle(juce::Slider::LinearBar);
    posSlider.setTextBoxStyle(juce::Slider::NoTextBox, true, 0, 0);
    posSlider.setColour(juce::Slider::trackColourId, juce::Colours::transparentWhite);
```

Modification of posSlider's UI in DeckGUI's setup function

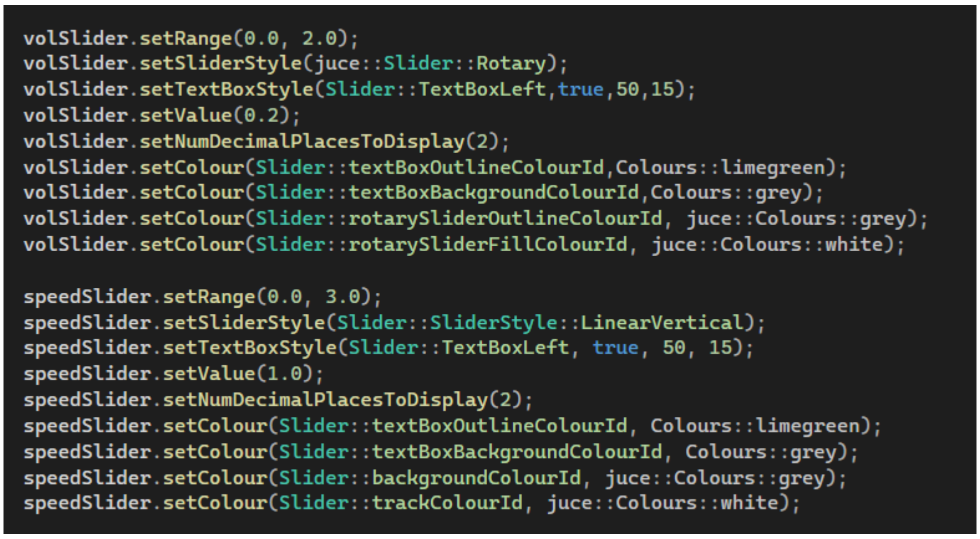
Having two separate buttons to stop and play the song takes up space unnecessarily. Therefore, I've made them into a single button instead by utilising setButtonText(), a bool variable 'audioStateP', and a conditional. Therefore, the button now both plays and stops the song while being able to change the text on it so the user won't be confused.



```
void DeckGUI::buttonClicked(Button* button)
{
    if (button == &playNstopButton)
    {
        if (audioStateP) {
            player->start();
            audioStateP = false;
            button->setButtonText("STOP");
        }
        else {
            player->stop();
            audioStateP = true;
            button->setButtonText("PLAY");
        }
    }
```

Event of the playNstopButton in DeckGUI's buttonClicked()

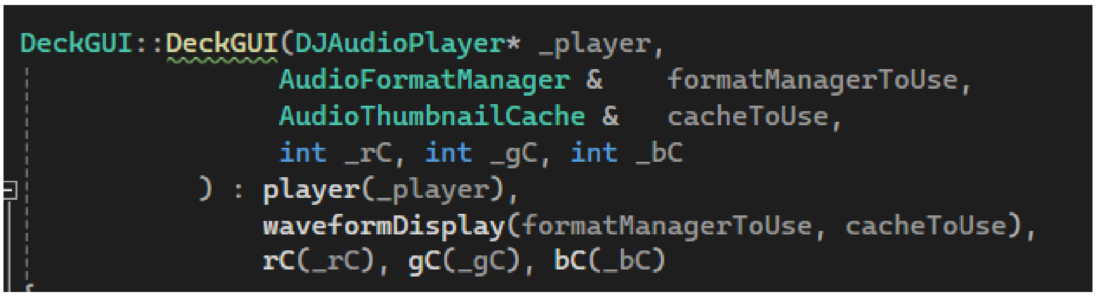
The default volume slider (volSlider) and speed slider (speedSlider) have some issues. Firstly, there is no reason to have numerical values that utilise a number with seven decimal places. Secondly, without defining the initial values, the song will play at its 100% volume (which can be very loud for headphone users) and sometimes cause the program to bug out. Thirdly, specifically the speedSlider, there's no reason to let a song speed up to 1000%.
To improve the user experience, I decided to limit the numerical value in the box to two decimal places, changed the sliders into their respective appropriate style for their purpose, and defined an initial value for both of the sliders.



```
volSlider.setRange(0.0, 2.0);
volSlider.setSliderStyle(juce::Slider::Rotary);
volSlider.setTextBoxStyle(Slider::TextBoxLeft,true,50,15);
volSlider.setValue(0.2);
volSlider.setNumDecimalPlacesToDisplay(2);
volSlider.setColour(Slider::textBoxOutlineColourId,Colours::limegreen);
volSlider.setColour(Slider::textBoxBackgroundColourId,Colours::grey);
volSlider.setColour(Slider::rotarySliderOutlineColourId, juce::Colours::grey);
volSlider.setColour(Slider::rotarySliderFillColourId, juce::Colours::white);

speedSlider.setRange(0.0, 3.0);
speedSlider.setSliderStyle(Slider::SliderStyle::LinearVertical);
speedSlider.setTextBoxStyle(Slider::TextBoxLeft, true, 50, 15);
speedSlider.setValue(1.0);
speedSlider.setNumDecimalPlacesToDisplay(2);
speedSlider.setColour(Slider::textBoxOutlineColourId, Colours::limegreen);
speedSlider.setColour(Slider::textBoxBackgroundColourId, Colours::grey);
speedSlider.setColour(Slider::backgroundColourId, juce::Colours::grey);
speedSlider.setColour(Slider::trackColourId, juce::Colours::white);
```

Modification of the UI of the volSlider and the speedSlider, from DeckGUI's setup function

I implemented a crossfader into this project. To improve the visibility of the crossfader to the user and its functionality more apparent, colour code the application. To make each deck more distinct, I change their background colour to red and blue to differentiate them. With the decks having different colours, I can set it such that the background of the crossfader will change its colour into either red or blue depending on the slider. For example, moving the crossfader up will isolate the upper deck with the red background, which will change the crossfader's background colour into red as well. Additionally, to make up for the dark colours of the decks, I will change the other listeners' colours (buttons and sliders) into brighter contrasting colours to improve visibility. This can be seen by making the sliders' text box outline green and their track white.



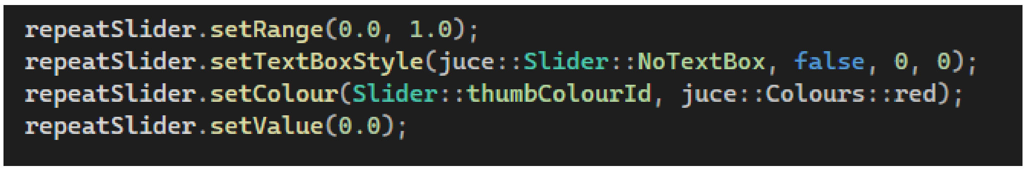
```
DeckGUI::DeckGUI(DJAudioPlayer* _player,
                 AudioFormatManager &    formatManagerToUse,
                 AudioThumbnailCache &   cacheToUse,
                 int _rC, int _gC, int _bC
             ) : player(_player),
                 waveformDisplay(formatManagerToUse, cacheToUse),
                 rC(_rC), gC(_gC), bC(_bC)
```

The inputs of the DeckGUI class, especially the rC, gC, and bC

◦R2B: GUI code has at least one event listener that was not in the original codebase seen in class

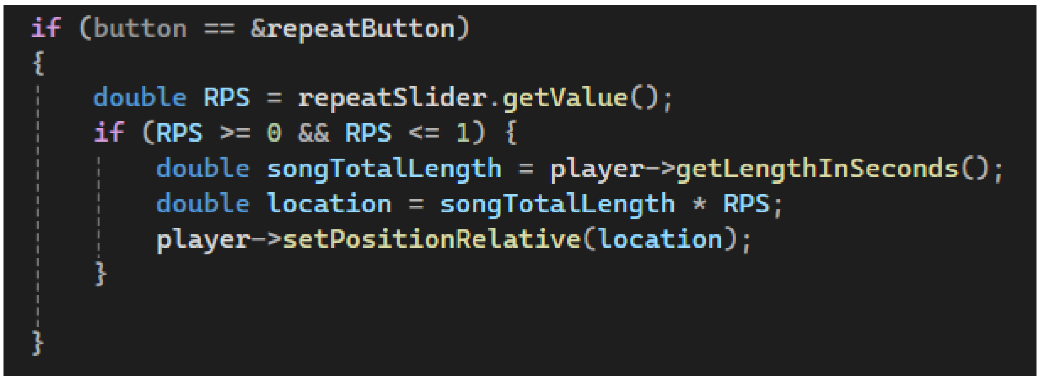
The event listeners that I added are the checkpoint button(repeatButton), the checkpoint slider(repeatSlider), and the crossfader(CFSlider in CrossFader.cpp). I will go into the crossfader in the third section of this report.

The checkpoint function's purpose is to allow the user to jump to a section of a song more accurately and consistently. It works by first moving the small red thumb slider (repeatSlider) above the waveform. The red thumb will have a red vertical line below it (overlay the waveform) to indicate to the user which section of the song the deck would skip to if they press the checkpoint button (the button with the text "JUMP TO CHECKPOINT"). By relying on a button to skip to the checkpoint, the user doesn't need to move their cursor to their desired song's section's approximate position and press the left mouse button, which are the factors that might accidentally make the user jump to the undesired section. Additionally, beside the checkpoint button will be a text that displays the timestamp of where the checkpoint is respective to the song, accurately showing where the user is jumping to.



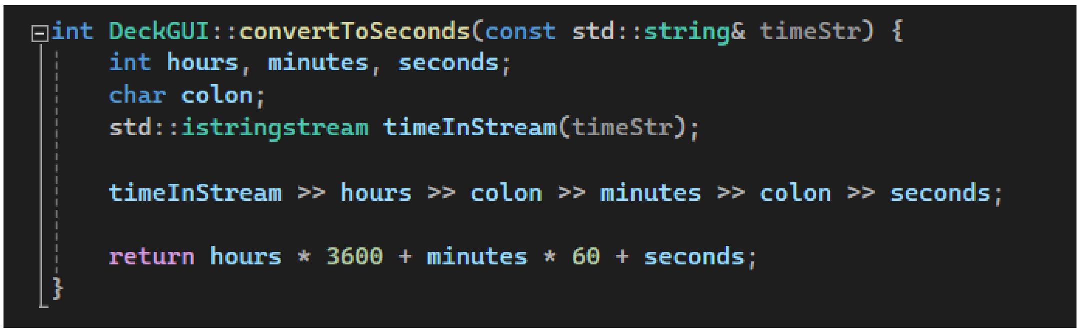
```
repeatSlider.setRange(0.0, 1.0);
repeatSlider.setTextBoxStyle(juce::Slider::NoTextBox, false, 0, 0);
repeatSlider.setColour(Slider::thumbColourId, juce::Colours::red);
repeatSlider.setValue(0.0);
```

Checkpoint function's slider UI, in DeckGUI's setup function



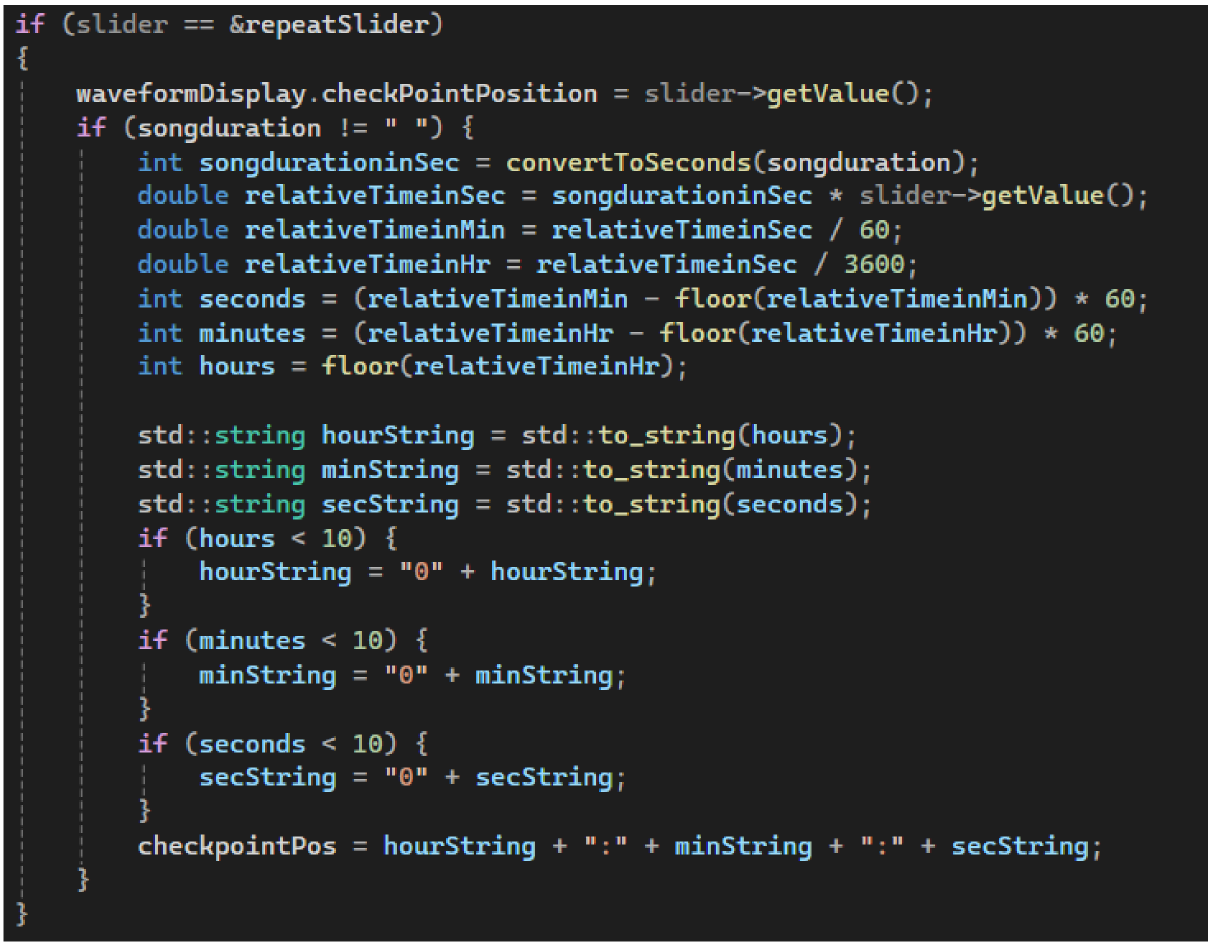
```
if (button == &repeatButton)
{
    double RPS = repeatSlider.getValue();
    if (RPS >= 0 && RPS <= 1) {
        double songTotalLength = player->getLengthInSeconds();
        double location = songTotalLength * RPS;
        player->setPositionRelative(location);
    }

}
```

Checkpoint's button event, in DeckGUI's buttonClicked()



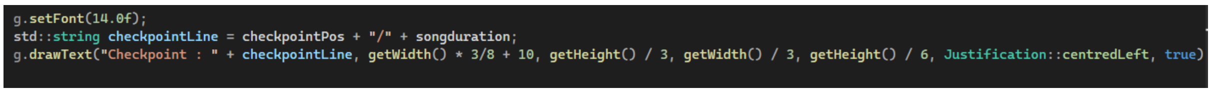
```
int DeckGUI::convertToSeconds(const std::string& timeStr) {
    int hours, minutes, seconds;
    char colon;
    std::istringstream timeInStream(timeStr);

    timeInStream >> hours >> colon >> minutes >> colon >> seconds;

    return hours * 3600 + minutes * 60 + seconds;
}
```

A function to convert eg the string "00:01:30" into the int 90



```cpp
if (slider == &repeatSlider)
{
    waveformDisplay.checkPointPosition = slider->getValue();
    if (songduration != " ") {
        int songdurationinSec = convertToSeconds(songduration);
        double relativeTimeinSec = songdurationinSec * slider->getValue();
        double relativeTimeinMin = relativeTimeinSec / 60;
        double relativeTimeinHr = relativeTimeinSec / 3600;
        int seconds = (relativeTimeinMin - floor(relativeTimeinMin)) * 60;
        int minutes = (relativeTimeinHr - floor(relativeTimeinHr)) * 60;
        int hours = floor(relativeTimeinHr);

        std::string hourString = std::to_string(hours);
        std::string minString = std::to_string(minutes);
        std::string secString = std::to_string(seconds);
        if (hours < 10) {
            hourString = "0" + hourString;
        }
        if (minutes < 10) {
            minString = "0" + minString;
        }
        if (seconds < 10) {
            secString = "0" + secString;
        }
        checkpointPos = hourString + ":" + minString + ":" + secString;
    }
}
```

To calculate the time point where the checkpoint is placed, at DeckGUI's sliderValueChanged()

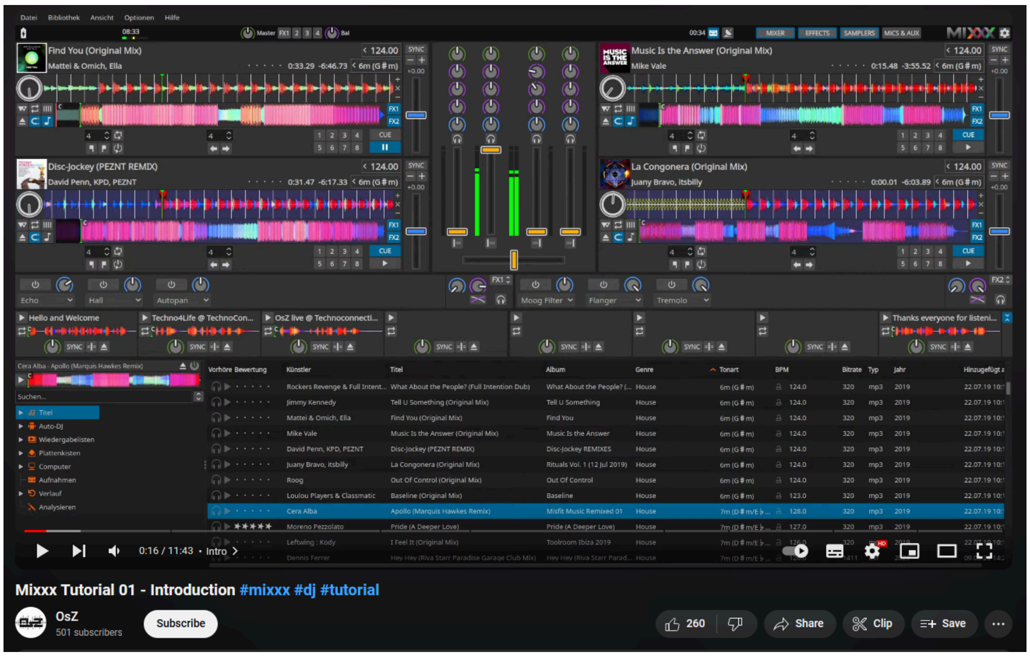


```cpp
g.setFont(14.0f);
std::string checkpointLine = checkpointPos + "/" + songduration;
g.drawText("Checkpoint : " + checkpointLine, getWidth() * 3/8 + 10, getHeight() / 3, getWidth() / 3, getHeight() / 6, Justification::centredLeft, true)
```

To display the checkpoint's time point, in DeckGUI's paint()

3. Research and technical information covering how you identified, analysed and implemented a new feature that you found in another DJ application (R3)

◦**research DJ applications and identify a feature to implement**



Layout of Mixxx

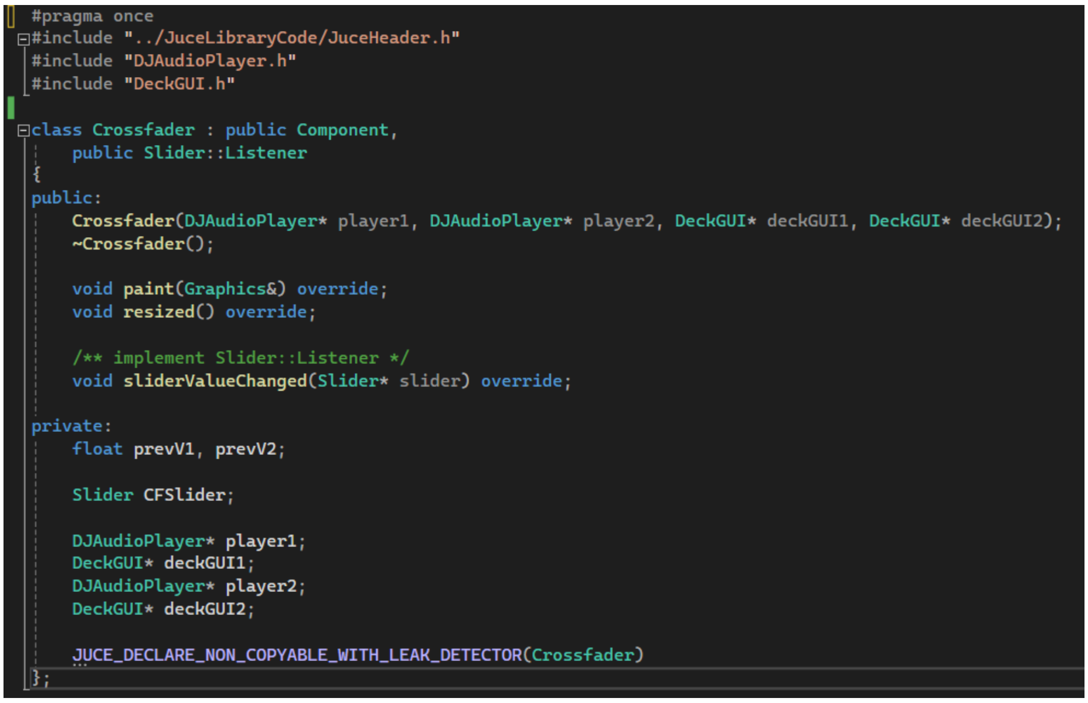
The DJ application I decided to research is Mixxx, a free and open-source DJ software. I know about Mixxx because I have a friend whose hobby is mixing songs, and from its UI, it seems to be a great option to research functionalities to implement into my project.

When my friend was mixing songs in the Discord voice chat, one functionality I noticed he used frequently was a Crossfader. Crossfader (which is located at the centre of its UI, with an orange rectangular thumb) is a slider used to blend multiple audio sources by varying their volumes. From my perspective, it seems to play a crucial role in mixing songs for a DJ, thus, I will implement that.

◦**Implement a DJ-related feature that you have seen in another DJ program**

To start, I used the DeckGUI files and copied them into a new set of files named CrossFader as a base considering I will be implementing a slider that will change a deck's volume level. Considering that a crossfader will need to be able to change the volume of both decks and know the current volume values, I will need to take in player1, player2, deckGUI1, and deckGUI2 from the mainComponent files.

My crossfader will require a slider with a range from 0 to 1. How this will work is that I will multiply the current volume of a player with the value of the slider, while the other player will be timed with the value of the slider after getting minus by 1. For example, by setting the crossfader slider at 0.7, player1 will be playing its song at 70% of its current volume while player2 will be playing its song at 30% of its current volume.



```
#pragma once
#include "../JuceLibraryCode/JuceHeader.h"
#include "DJAudioPlayer.h"
#include "DeckGUI.h"

class Crossfader : public Component,
    public Slider::Listener
{
public:
    Crossfader(DJAudioPlayer* player1, DJAudioPlayer* player2, DeckGUI* deckGUI1, DeckGUI* deckGUI2);
    ~Crossfader();

    void paint(Graphics&) override;
    void resized() override;

    /** implement Slider::Listener */
    void sliderValueChanged(Slider* slider) override;

private:
    float prevV1, prevV2;

    Slider CFSlider;

    DJAudioPlayer* player1;
    DeckGUI* deckGUI1;
    DJAudioPlayer* player2;
    DeckGUI* deckGUI2;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(Crossfader)
};
```

CrossFader.h

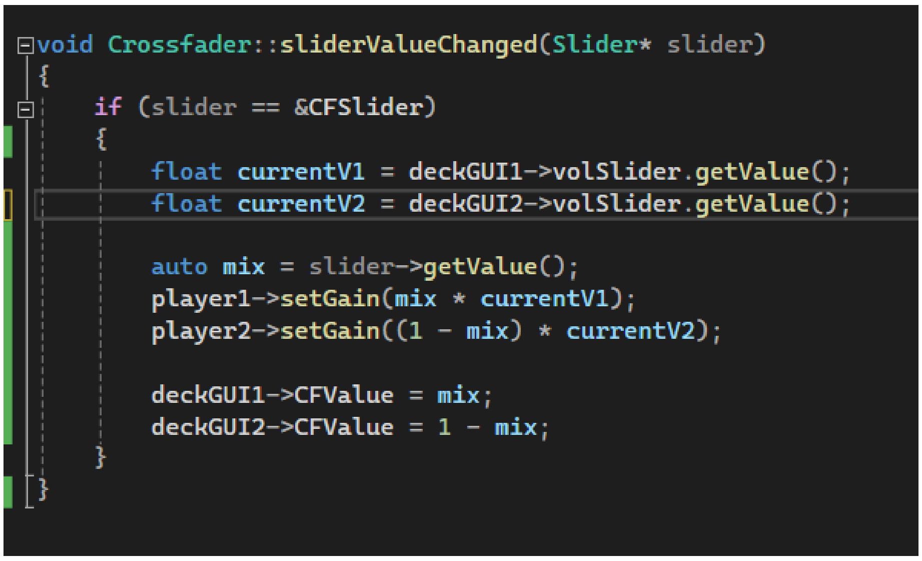
By taking in both players as the class's inputs, I can access their 'setGain()' to change the players' volume. Originally with the Crossfader class's inputs only consisting of player1 and player2, I used 'getGain()' and 'setGain()' to modify a player's volume. For example, 'player1->setGain(slider->getValue() * mix);'.

However, it will permanently modify the players' volume and retrieve the modified volume to repeat the cycle continuously. For example, by sliding the CFSlider upward fully, I will mute the bottom player. However, the moment I move the CFSlider down, the bottom player is still mute. This is so because by maxing out the CFSlider, I would have set the player2's volume to 0%. Therefore, any future changes to CFSlider will not change player2's volume since the CFSlider's value times zero equals zero.
To mitigate this issue, I need to implement such that the Crossfader class can record the current volumeSlider of both decks. Thus, I implemented additional input for the Crossfader class (deckGUI1 and deckGUI2), as well as the two variables currentV1 and currentV2 (currentVs = current volume).

Knowing that the deck's volumeSlider's value does not change from the players' 'setGain()'. This ensures that every time I modify the volume of both players with the CFSlider, the volume of a player is dependent on a static variable. So if the CFSlider's value is 0, I would not be applying 'setGain(player's volume x 0)' in which the player's volume changes continuously, permanently muting the player. Instead, it would be 'setGain(deck's volumeSlider value * 0)' in which the volumeSlider's value can't be changed without the user's interaction, allowing the user to 'un-mute' the player when CFSlider is more than zero.

Example:
O Without currentVs;
player1 ‘s volume = 0.7
setGain(0.7 * 0) → player1’s volume = 0
setGain(0 * 0.8) → player1’s volume = 0

O With currentVs;
player1 ‘s volume = 0.7
setGain(0.7 * 0) → player1’s volume = 0
setGain(0.7 * 0.8) → player1’s volume = 0.56



```cpp
void Crossfader::sliderValueChanged(Slider* slider)
{
    if (slider == &CFSlider)
    {
        float currentV1 = deckGUI1->volSlider.getValue();
        float currentV2 = deckGUI2->volSlider.getValue();

        auto mix = slider->getValue();
        player1->setGain(mix * currentV1);
        player2->setGain((1 - mix) * currentV2);

        deckGUI1->CFValue = mix;
        deckGUI2->CFValue = 1 - mix;
    }
}
```

Crossfader’s sliderValueChanged()

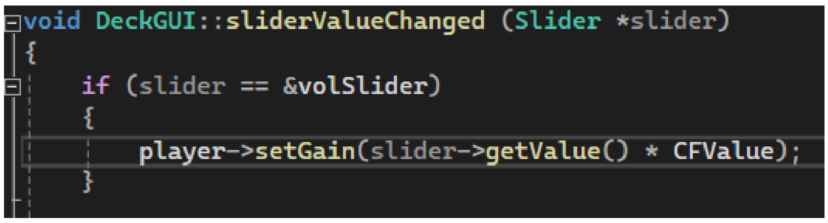
The core functionality of a crossfader is to modify the volumes of both songs **while** limiting the volume if the user decides to change the volume of a deck independently. I used to face an issue after setting the CFSlider, such that it got completely overridden when I changed the player’s volume through the deck afterwards.
For example, I set the CFSlider as 0.7 when player1’s volumeSlider is 0.3, which will make player1’s song to play at 21%. However, by sliding up the volumeSlider of deck1 to 0.9, player1’s song will now be played at 90% instead of 63% (0.7 x 0.9 = 0.63).
Therefore, I need to add a new variable to the deckGUI files to include the value of the crossfader slider (CFSlider).

This will allow any changes to the volumeSlider, after moving the CFSlider, to consider the CFSlider’s value as well. This prevents any unintended increases of the player’s volume.

Example:
O Without CFValue;
deck1’s volumeSlider = 0.3
Crossfaders.cpp :: setGain(0.3 * 0.5) → player1’s volume = 15%
deck1’s volumeSlider = 0.7

DeckGUI.cpp :: setGain(0.7) → player1’s volume = 70%

O With CFValue;
deck1’s volumeSlider = 0.3
Crossfaders.cpp :: setGain(0.3 * 0.5) → player1’s volume = 15%
deck1’s volumeSlider = 0.7
DeckGUI.cpp :: setGain(0.7 * 0.5) → player1’s volume = 45%



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

DeckGUI’s sliderValueChanged()