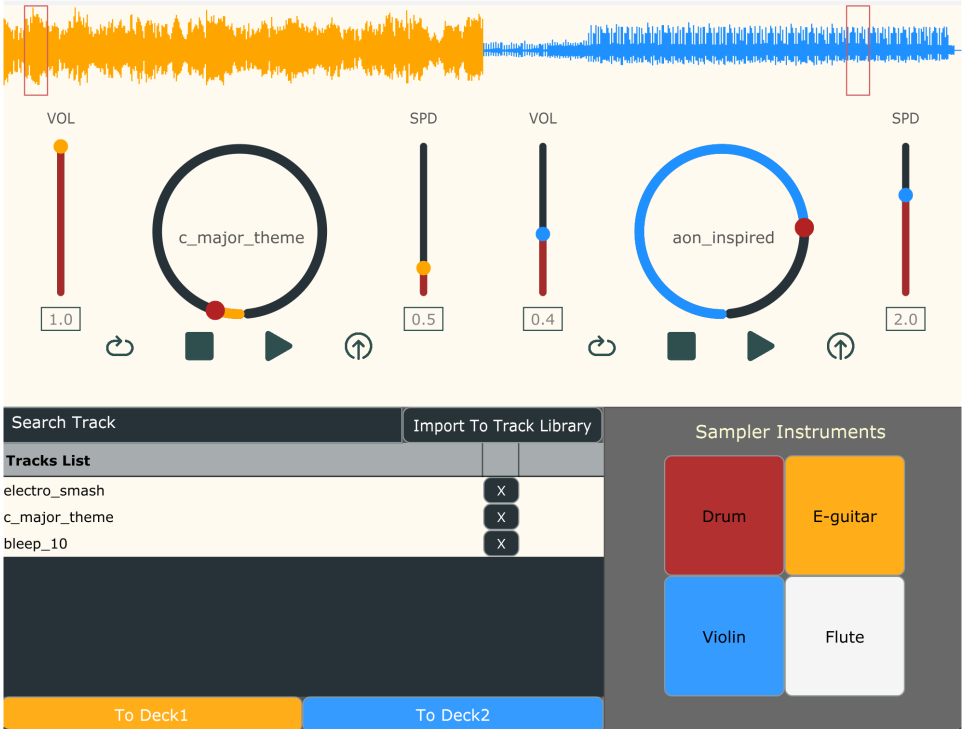
CM2005 Object-Oriented Programming Project Report – JUCE DJ Application

R1: The application should contain all the basic functionality shown in class:



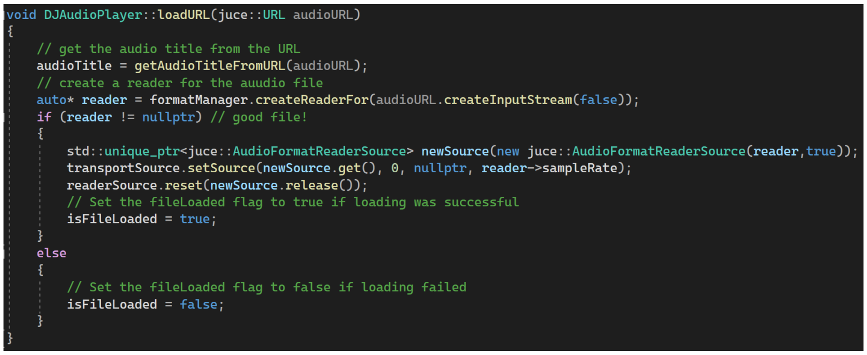
Tracks List		
electro_smash	X	
c_major_theme	X	
bleep_10	X	

R1A: can load audio files into audio players

In the application, there are two ways to load audio files into audio players. First, users can click the load button icon, which is one of the player controls, via the DeckGUI object. Second, users can click either the "To Deck1" or "To Deck2" button in the PlaylistComponent when selecting music that has already been added to the tracks list, to load it into the desired player.

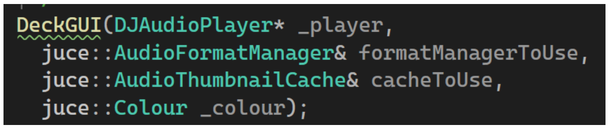
The process of loading audio files is facilitated by three objects: DJAudioPlayer, DeckGUI and PlaylistComponent. DJAudioPlayer primarily manages the loading of audio files. DeckGUI, on the other hand, utilizes the DJAudioPlayer object to load audio files into the player. Additionally, DeckGUI serves as a reference for the PlaylistComponent object, enabling PlaylistComponent to load the relevant audio file to the corresponding audio players.

`loadURL()` function of DJAudioPlayer manages the loading of audio files.



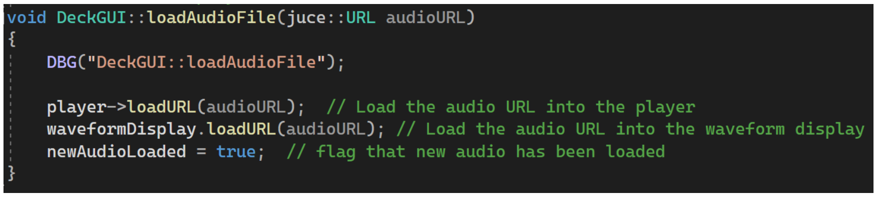
```
void DJAudioPlayer::loadURL(juce::URL audioURL)
{
    // get the audio title from the URL
    audioTitle = getAudioTitleFromURL(audioURL);
    // create a reader for the auudio file
    auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));
    if (reader != nullptr) // good file!
    {
        std::unique_ptr<juce::AudioFormatReaderSource> newSource(new juce::AudioFormatReaderSource(reader,true));
        transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
        readerSource.reset(newSource.release());
        // Set the fileLoaded flag to true if loading was successful
        isFileLoaded = true;
    }
    else
    {
        // Set the fileLoaded flag to false if loading failed
        isFileLoaded = false;
    }
}
```

DeckGUI holds a pointer that points to DJAudioPlayer



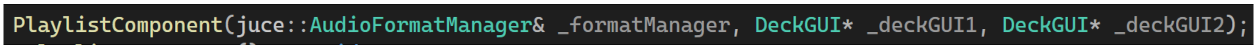
```
DeckGUI(DJAudioPlayer* _player,
  juce::AudioFormatManager& formatManagerToUse,
  juce::AudioThumbnailCache& cacheToUse,
  juce::Colour _colour);
```

Way1: to load audio files into audio player via `loadAudioFile()` of DeckGUI



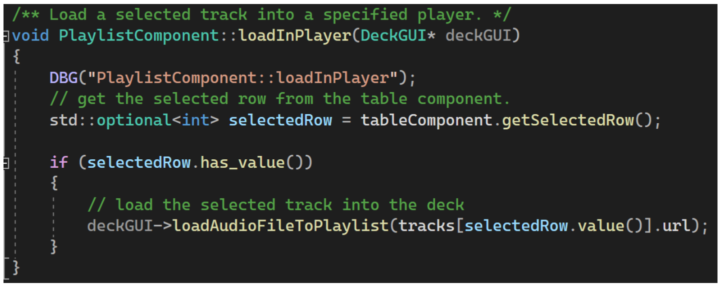
```
void DeckGUI::loadAudioFile(juce::URL audioURL)
{
    DBG("DeckGUI::loadAudioFile");

    player->loadURL(audioURL);  // Load the audio URL into the player
    waveformDisplay.loadURL(audioURL); // Load the audio URL into the waveform display
    newAudioLoaded = true;  // flag that new audio has been loaded
}
```

DeckGUI serves as a reference for the PlaylistComponent object



```
PlaylistComponent(juce::AudioFormatManager& _formatManager, DeckGUI* _deckGUI1, DeckGUI* _deckGUI2);
```

Way2: to load audio files into audio player via `loadInPlayer()` function of PlaylistComponent



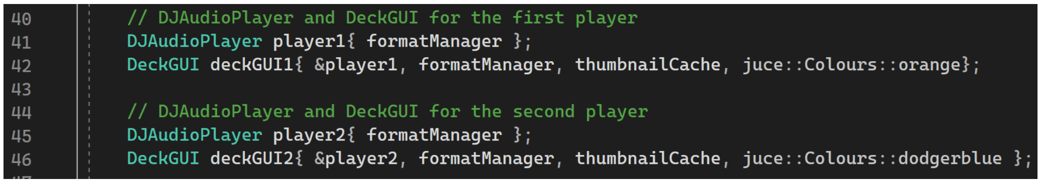
```
/** Load a selected track into a specified player. */
void PlaylistComponent::loadInPlayer(DeckGUI* deckGUI)
{
    DBG("PlaylistComponent::loadInPlayer");
    // get the selected row from the table component.
    std::optional<int> selectedRow = tableComponent.getSelectedRow();

    if (selectedRow.has_value())
    {
        // load the selected track into the deck
        deckGUI->loadAudioFileToPlaylist(tracks[selectedRow.value()].url);
    }
}
```

R1B: can play two or more tracks

The application features two independent audio players, positioned at the top of the program, with one on the left and one on the right. These players are represented by the DeckGUI object, and each of them is equipped with a play button for initiating track playback. The presence of two players, the application can simultaneously play two tracks.

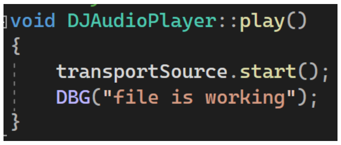
DeckGUI includes functionality for rendering the play button as a user interface element, allowing users to click and play tracks. Similarly, when it comes to loading audio files into an audio player, the DJAudioPlayer takes charge of handling the selected track for playback. Furthermore, it acts as a reference point for the DeckGUI object, allowing to play the relevant audio files on the corresponding DeckGUI components.

Two tracks are created in `MainComponent.h` object



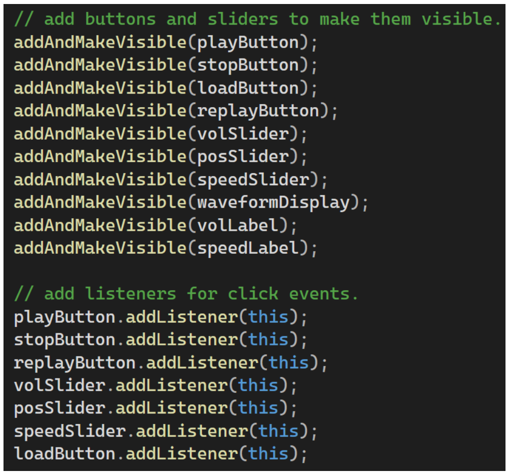
```
    // DJAudioPlayer and DeckGUI for the first player
    DJAudioPlayer player1{ formatManager };
    DeckGUI deckGUI1{ &player1, formatManager, thumbnailCache, juce::Colours::orange};

    // DJAudioPlayer and DeckGUI for the second player
    DJAudioPlayer player2{ formatManager };
    DeckGUI deckGUI2{ &player2, formatManager, thumbnailCache, juce::Colours::dodgerblue };
```

`play()` function of DJAudioPlayer manages the playback function.



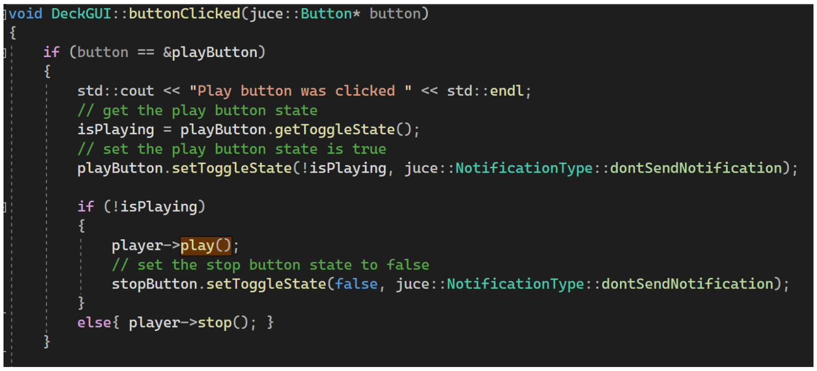
```
void DJAudioPlayer::play()
{
    transportSource.start();
    DBG("file is working");
}
```

DeckGUI handles the function of rendering playback button



```
// add buttons and sliders to make them visible.
addAndMakeVisible(playButton);
addAndMakeVisible(stopButton);
addAndMakeVisible(loadButton);
addAndMakeVisible(replayButton);
addAndMakeVisible(volSlider);
addAndMakeVisible(posSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(waveformDisplay);
addAndMakeVisible(volLabel);
addAndMakeVisible(speedLabel);

// add listeners for click events.
playButton.addListener(this);
stopButton.addListener(this);
replayButton.addListener(this);
volSlider.addListener(this);
posSlider.addListener(this);
speedSlider.addListener(this);
loadButton.addListener(this);
```

`player` is assigned to the DJAudioPlayer object, which acts as a reference point for the DeckGUI object. In the `buttonClicked()` function of DeckGUI, when the "play" button is clicked, it initiates the playback of the music.

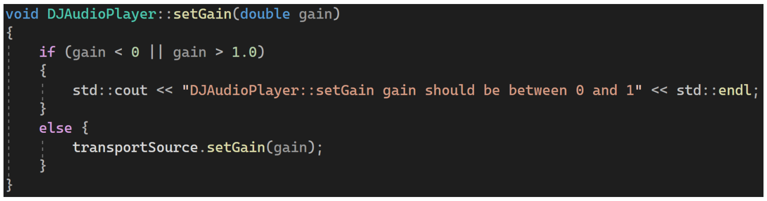


```
void DeckGUI::buttonClicked(juce::Button* button)
{
    if (button == &playButton)
    {
        std::cout << "Play button was clicked " << std::endl;
        // get the play button state
        isPlaying = playButton.getToggleState();
        // set the play button state is true
        playButton.setToggleState(!isPlaying, juce::NotificationType::dontSendNotification);

        if (!isPlaying)
        {
            player->play();
            // set the stop button state to false
            stopButton.setToggleState(false, juce::NotificationType::dontSendNotification);
        }
        else{ player->stop(); }
    }
```

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

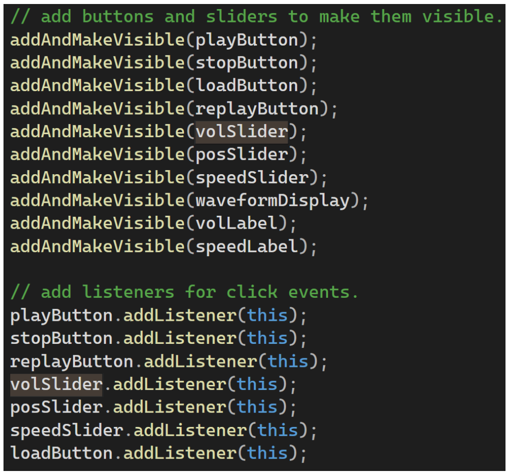
The volume slider is one of the player controls located on the left side of each player. DeckGUI includes functionality for rendering the slider as a user interface element, while the task of adjusting the volumes of track is handled by DJAudioPlayer. Similarly, the DeckGUI object establishes a connection with DJAudioPlayer, enabling the volume slider to perform its function properly. Each player has their own volume slider, so they can vary each of their volumes of track. This arrangement enables independent volume control for each player, allowing them to adjust the volume of their respective tracks.

`setGain()` function of DJAudioPlayer manages the volume of the track.



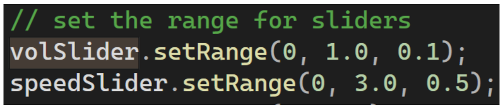
```
void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0)
    {
        std::cout << "DJAudioPlayer::setGain gain should be between 0 and 1" << std::endl;
    }
    else {
        transportSource.setGain(gain);
    }
}
```

DeckGUI is responsible for rendering the volume slider



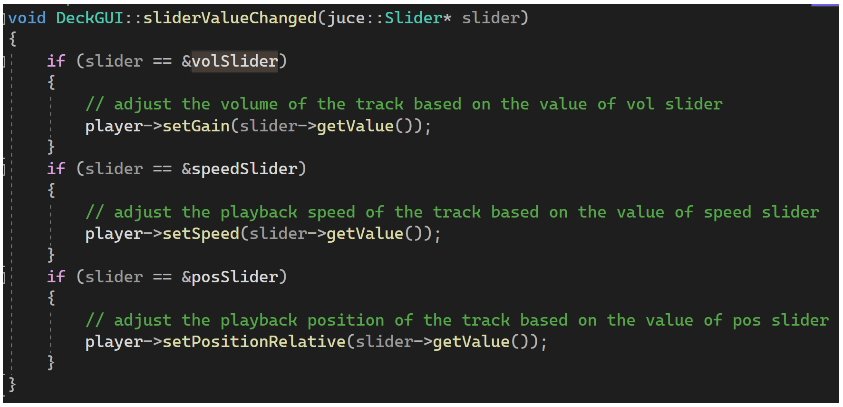
```
// add buttons and sliders to make them visible.
addAndMakeVisible(playButton);
addAndMakeVisible(stopButton);
addAndMakeVisible(loadButton);
addAndMakeVisible(replayButton);
addAndMakeVisible(volSlider);
addAndMakeVisible(posSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(waveformDisplay);
addAndMakeVisible(volLabel);
addAndMakeVisible(speedLabel);

// add listeners for click events.
playButton.addListener(this);
stopButton.addListener(this);
replayButton.addListener(this);
volSlider.addListener(this);
posSlider.addListener(this);
speedSlider.addListener(this);
loadButton.addListener(this);
```

Set the range of volslider



```
// set the range for sliders
volSlider.setRange(0, 1.0, 0.1);
speedSlider.setRange(0, 3.0, 0.5);
```

`player` is assigned to the DJAudioPlayer object, serving as a reference point for the DeckGUI object. In the `sliderValueChanged()` function of DeckGUI, when the "volSlider" value is adjusted, it controls the playback volume of the track.

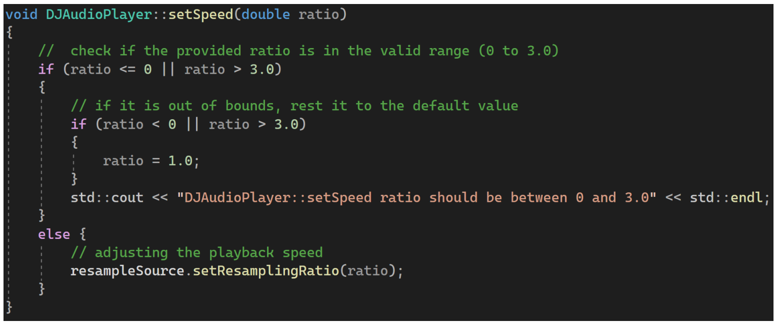


```
void DeckGUI::sliderValueChanged(juce::Slider* slider)
{
    if (slider == &volSlider)
    {
        // adjust the volume of the track based on the value of vol slider
        player->setGain(slider->getValue());
    }
    if (slider == &speedSlider)
    {
        // adjust the playback speed of the track based on the value of speed slider
        player->setSpeed(slider->getValue());
    }
    if (slider == &posSlider)
    {
        // adjust the playback position of the track based on the value of pos slider
        player->setPositionRelative(slider->getValue());
    }
}
```

R1D: can speed up and slow down the tracks

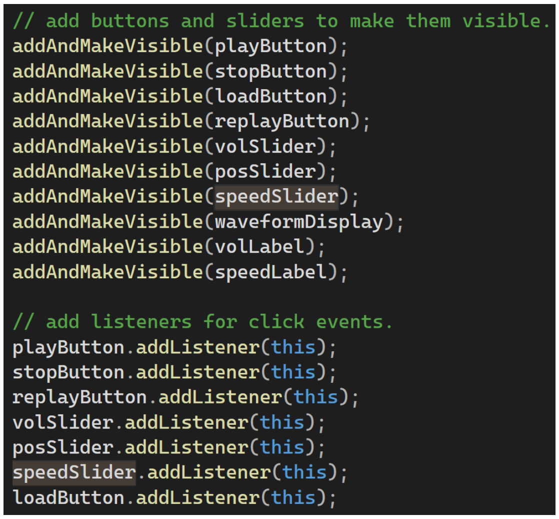
Each player can speed up and slow down the tracks using a speed slider located on the right side of the player. Similar to R1C, DeckGUI is responsible for rendering the speed slider as a user interface element, while the actual speed adjustment functionality is managed by DJAudioPlayer. The DJAudioPlayer serves as a reference to DeckGUI, ensuring that the speed slider functions properly. The speed of a track can be increased in increments of 0.5, with a maximum speed of triple the original playback speed.

`setSpeed()` function of DJAudioPlayer manages the speed of track.



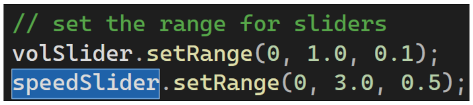
```
void DJAudioPlayer::setSpeed(double ratio)
{
    //  check if the provided ratio is in the valid range (0 to 3.0)
    if (ratio <= 0 || ratio > 3.0)
    {
        // if it is out of bounds, rest it to the default value
        if (ratio < 0 || ratio > 3.0)
        {
            ratio = 1.0;
        }
        std::cout << "DJAudioPlayer::setSpeed ratio should be between 0 and 3.0" << std::endl;
    }
    else {
        // adjusting the playback speed
        resampleSource.setResamplingRatio(ratio);
    }
}
```

DeckGUI is responsible for rendering the speed slider



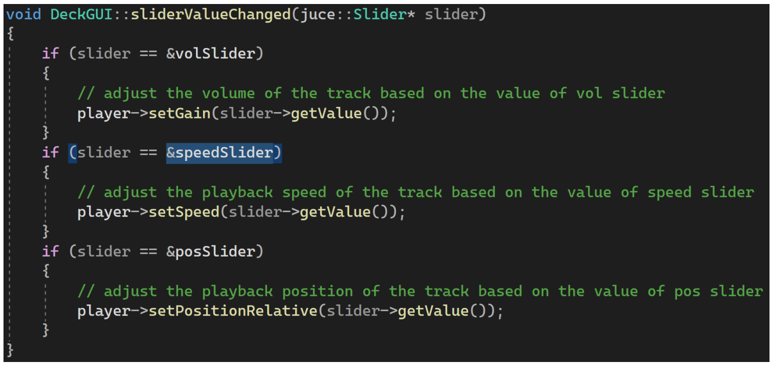
```
// add buttons and sliders to make them visible.
addAndMakeVisible(playButton);
addAndMakeVisible(stopButton);
addAndMakeVisible(loadButton);
addAndMakeVisible(replayButton);
addAndMakeVisible(volSlider);
addAndMakeVisible(posSlider);
addAndMakeVisible(speedSlider);
addAndMakeVisible(waveformDisplay);
addAndMakeVisible(volLabel);
addAndMakeVisible(speedLabel);

// add listeners for click events.
playButton.addListener(this);
stopButton.addListener(this);
replayButton.addListener(this);
volSlider.addListener(this);
posSlider.addListener(this);
speedSlider.addListener(this);
loadButton.addListener(this);
```

Set the range of speed slider



```
// set the range for sliders
volSlider.setRange(0, 1.0, 0.1);
speedSlider.setRange(0, 3.0, 0.5);
```

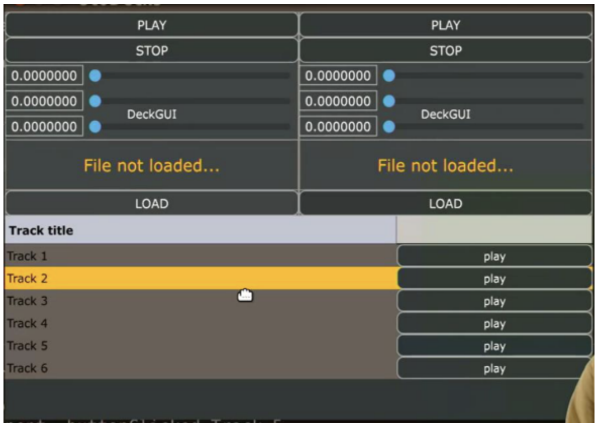
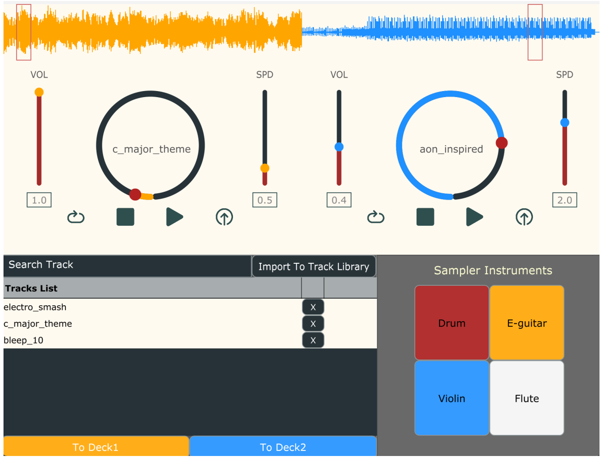
`player` is assigned to the DJAudioPlayer object, serving as a reference point for the DeckGUI object. In the `sliderValueChanged()` function of DeckGUI, when the "speedSlider" value is adjusted, it controls the playback speed of the track.



```
void DeckGUI::sliderValueChanged(juce::Slider* slider)
{
    if (slider == &volSlider)
    {
        // adjust the volume of the track based on the value of vol slider
        player->setGain(slider->getValue());
    }
    if (slider == &speedSlider)
    {
        // adjust the playback speed of the track based on the value of speed slider
        player->setSpeed(slider->getValue());
    }
    if (slider == &posSlider)
    {
        // adjust the playback position of the track based on the value of pos slider
        player->setPositionRelative(slider->getValue());
    }
}
```

R2: Customize the user interface (UI): i.e., change the colors, and change the layout.

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

Original DJ Application layout	Redesigned DJ Application layout
 PLAY STOP 0.0000000 DeckGUI File not loaded... LOAD Track title Track 1 Track 2 Track 3 Track 4 Track 5 Track 6 play	 VOL SPD c_major_theme aon_inspired 1.0 0.5 0.4 2.0 Search Track Import To Track Library Tracks List electro_smash c_major_theme bleep_10 X Sampler Instruments Drum E-guitar Violin Flute To Deck1 To Deck2

As evident from the comparison between the original and the redesigned DJ Applications layouts, there have been significantly changed in the GUI design. The design concept of the new DJ Application layout is inspired from the "DJ Mixer Player - Music DJ app" by Sky Made, which is available on the Google App Store(`https://play.google.com/store/apps/details?id=com.sky.djmixer.musicplayer.djmusic&hl=en_US&pli=1`).

The application has been changed to a lighter theme, adopting a color palette that includes shades of orange, dodgerblue, firebrick, dark gray and bridal heath.

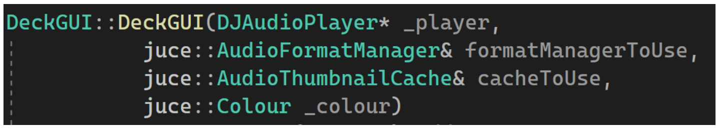
To implement these design changes, the `OtherLookAndFeel` object was created to customize the application's user interface. This object serves as a pointer for "DeckGUI` and applies the desired customization to the deck player control elements.

The appearance of the deck control buttons has been transformed. Playback, pause, replay and load buttons are now represented as image buttons instead of text buttons. This change aligns the application more closely with the real DJ application and enhances user-friendliness. Slider controls, on the other hand, have also been revamped. Volume and speed control sliders are now horizontal, while the audio playback position slider has been redesigned as a rotary control, simulating a clock-like interface. Notably, the position slider has been configured to behave as a complete circle, akin to a clock, rather than an incomplete circle by default.

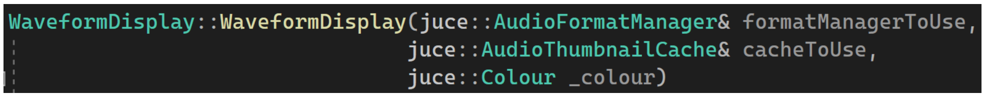
Furthermore, each of the two players has been assigned a specific color scheme. The left deck primarily features an orange color, while the right deck uses a dodge blue color. I have added a new `color` parameter in the `DeckGUI`, `Waveform` and `OtherLookAndFeel` objects, allowing for color assignment. This ensures that the display color of the waveform and the configuration settings in the "OtherLookAndFeel" object are synchronized with the color initialized in the "DeckGUI" object, creating a cohesive and visually appealing interface for each deck.

Additionally, the playlist has been resized to accommodate the new feature, the `sampler instrument` component, seamlessly integrating these elements into the application.

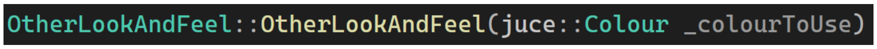
A new `color` parameter in the `DeckGUI`, `Waveform` and `OtherLookAndFeel` objects.



```
DeckGUI::DeckGUI(DJAudioPlayer* _player,
           juce::AudioFormatManager& formatManagerToUse,
           juce::AudioThumbnailCache& cacheToUse,
           juce::Colour _colour)
```

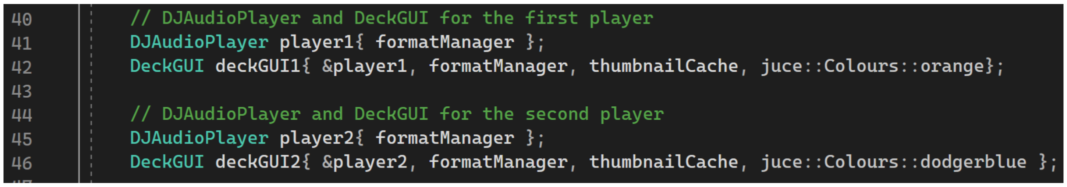


```
WaveformDisplay::WaveformDisplay(juce::AudioFormatManager& formatManagerToUse,
                                 juce::AudioThumbnailCache& cacheToUse,
                                 juce::Colour _colour)
```



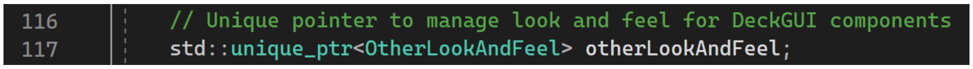
```
OtherLookAndFeel::OtherLookAndFeel(juce::Colour _colourToUse)
```

In `MainComponent.h`
the color is initialized in the `DeckGUI` object.

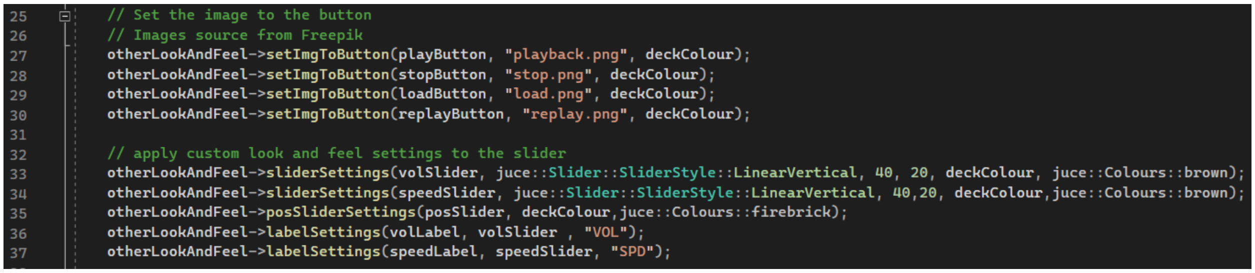


```
    // DJAudioPlayer and DeckGUI for the first player
    DJAudioPlayer player1{ formatManager };
    DeckGUI deckGUI1{ &player1, formatManager, thumbnailCache, juce::Colours::orange};

    // DJAudioPlayer and DeckGUI for the second player
    DJAudioPlayer player2{ formatManager };
    DeckGUI deckGUI2{ &player2, formatManager, thumbnailCache, juce::Colours::dodgerblue };
```

`OtherLookAndFeel` object serves as a pointer in the `DeckGUI` class, allowing it to applies the desired customization to the deck player control elements - sliders and buttons



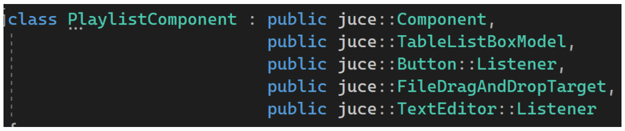
```
    // Unique pointer to manage look and feel for DeckGUI components
    std::unique_ptr<OtherLookAndFeel> otherLookAndFeel;
```



```
    // Set the image to the button
    // Images source from Freepik
    otherLookAndFeel->setImgToButton(playButton, "playback.png", deckColour);
    otherLookAndFeel->setImgToButton(stopButton, "stop.png", deckColour);
    otherLookAndFeel->setImgToButton(loadButton, "load.png", deckColour);
    otherLookAndFeel->setImgToButton(replayButton, "replay.png", deckColour);

    // apply custom look and feel settings to the slider
    otherLookAndFeel->sliderSettings(volSlider, juce::Slider::SliderStyle::LinearVertical, 40, 20, deckColour, juce::Colours::brown);
    otherLookAndFeel->sliderSettings(speedSlider, juce::Slider::SliderStyle::LinearVertical, 40,20, deckColour,juce::Colours::brown);
    otherLookAndFeel->posSliderSettings(posSlider, deckColour,juce::Colours::firebrick);
    otherLookAndFeel->labelSettings(volLabel, volSlider , "VOL");
    otherLookAndFeel->labelSettings(speedLabel, speedSlider, "SPD");
```

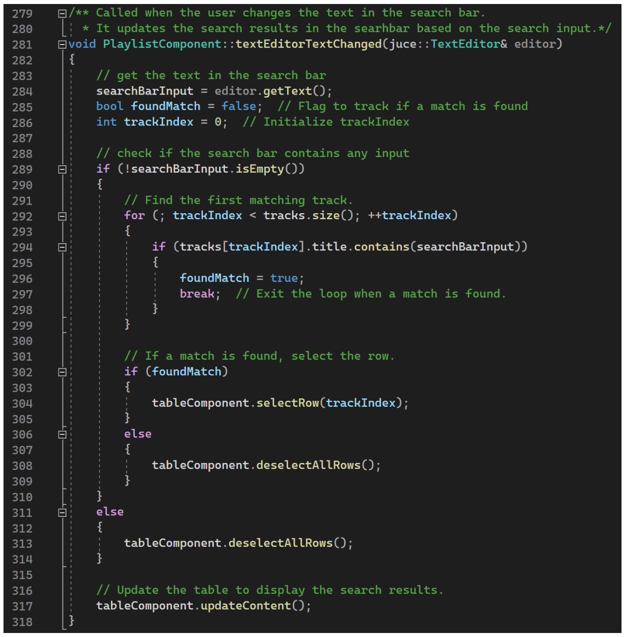
R2B: GUI code has at least one event listener that was not in the original codebase seen in class.

The application features an editable text box component, functioning as a search bar, enabling users to search for specific tracks already imported into the playlist. This search bar is a part of the PlaylistComponent. To facilitate the detection of the search bar's content, a text editor event listener has been introduced. This addition was not present in the original codebase seen in class.



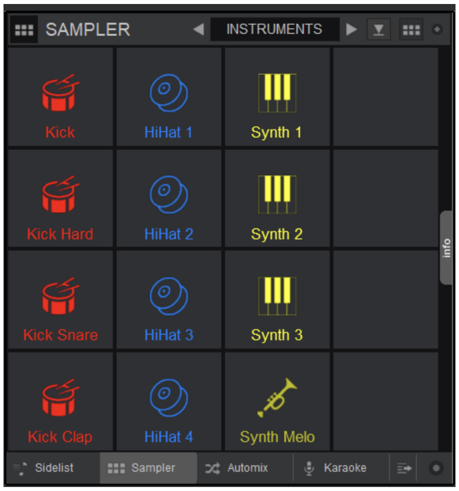
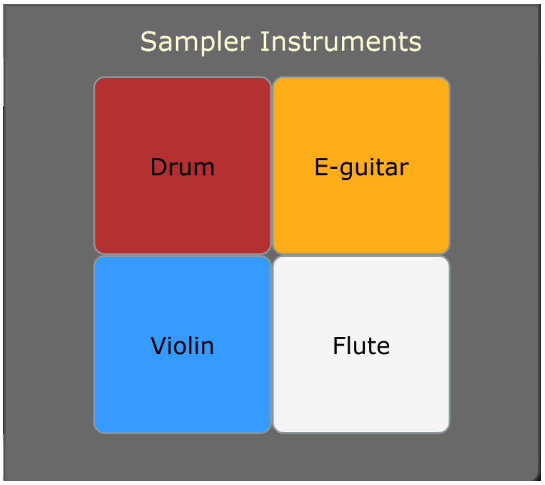
```
class PlaylistComponent : public juce::Component,
                          public juce::TableListBoxModel,
                          public juce::Button::Listener,
                          public juce::FileDragAndDropTarget,
                          public juce::TextEditor::Listener
```

The member function of the text editor event listener, textEditorTextChanged(), has been integrated into the code. It actively updates the search results within the search bar based on the user's search input.



```
/** Called when the user changes the text in the search bar.
  * It updates the search results in the searhbar based on the search input.*/
void PlaylistComponent::textEditorTextChanged(juce::TextEditor& editor)
{
    // get the text in the search bar
    searchBarInput = editor.getText();
    bool foundMatch = false;  // Flag to track if a match is found
    int trackIndex = 0;  // Initialize trackIndex

    // check if the search bar contains any input
    if (!searchBarInput.isEmpty())
    {
        // Find the first matching track.
        for (; trackIndex < tracks.size(); ++trackIndex)
        {
            if (tracks[trackIndex].title.contains(searchBarInput))
            {
                foundMatch = true;
                break;  // Exit the loop when a match is found.
            }
        }

        // If a match is found, select the row.
        if (foundMatch)
        {
            tableComponent.selectRow(trackIndex);
        }
        else
        {
            tableComponent.deselectAllRows();
        }
    }
    else
    {
        tableComponent.deselectAllRows();
    }

    // Update the table to display the search results.
    tableComponent.updateContent();
}
```

R3: Investigate and implement a new feature inspired by a real DJ program.

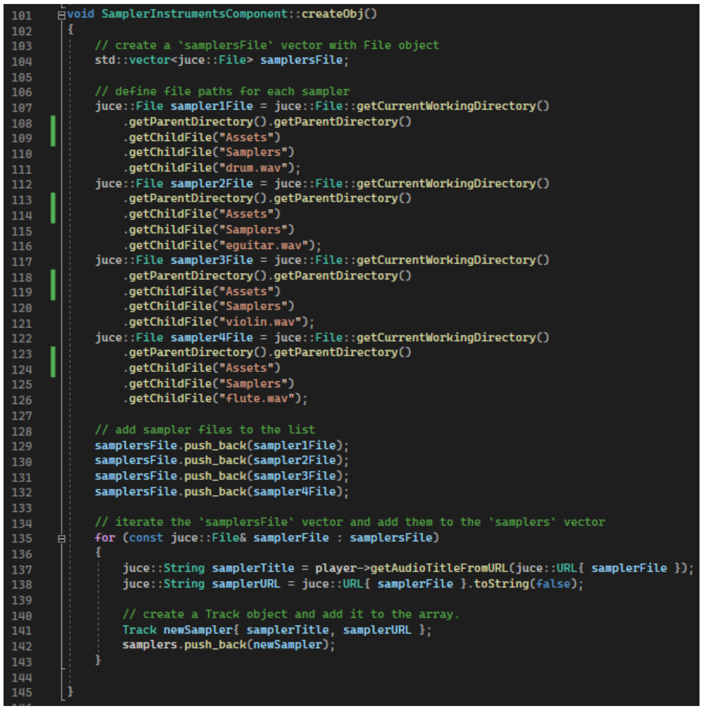
Reference of Sampler Instruments	DJ Application Sampler Instruments
	

The new feature I implemented to the DJ application is the `Sampler Instruments` feature. This feature draws inspiration from a component found in VirtualDJ, a virtual DJ application that I search online (`https://www.virtualdj.com/manuals/virtualdj/interface/browser/sideview/sampler.html`). Although my implementation is not as complicated as the VirtualDJ, it brings a new layer of versatility to the DJ application.

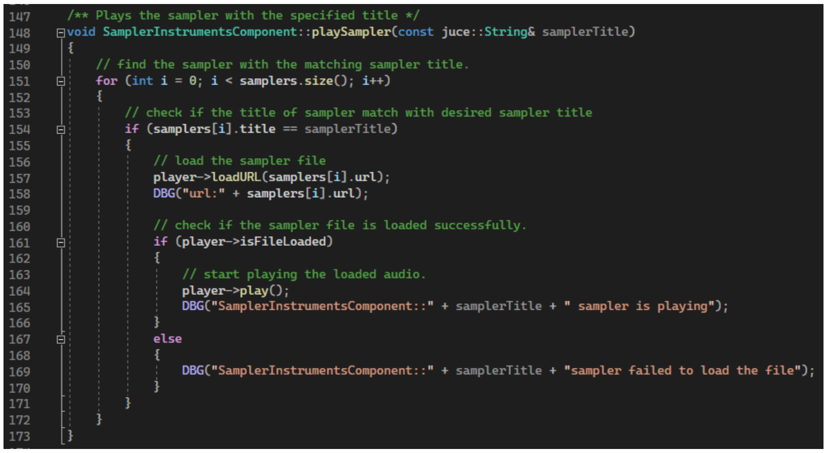
The sampler instrument component introduces four different instrument sounds buttons — each representing unique instrument effects: drum, electric guitar, violin and flute. Users can easily trigger these sampler instruments using the corresponding buttons, enabling them to mix different sound effects when playing tracks.

To implement this functionality, a `SamplerInstrumentComponent` object was created. It interfaces with the `DJAudioPlayer` object to access its functions for playing instrument sounds. The `createObj()` function within the `SamplerInstrumentComponent` serves to access the source of the instrument samplers. It initializes each sampler into a `Track` object, which is then stored into the `samplers` vector.

When a user clicks on a specific sampler button, the corresponding sound is played via the `playSampler()` function. This function searches for the sampler with a title matching the selected sampler button by iterating through the `sampler` vector, ensuring the desired instrument sound is triggered.



```cpp
void SamplerInstrumentsComponent::createObj()
{
    // create a `samplersFile` vector with File object
    std::vector<juce::File> samplersFile;

    // define file paths for each sampler
    juce::File sampler1File = juce::File::getCurrentWorkingDirectory()
        .getParentDirectory().getParentDirectory()
        .getChildFile("Assets")
        .getChildFile("Samplers")
        .getChildFile("drum.wav");
    juce::File sampler2File = juce::File::getCurrentWorkingDirectory()
        .getParentDirectory().getParentDirectory()
        .getChildFile("Assets")
        .getChildFile("Samplers")
        .getChildFile("eguitar.wav");
    juce::File sampler3File = juce::File::getCurrentWorkingDirectory()
        .getParentDirectory().getParentDirectory()
        .getChildFile("Assets")
        .getChildFile("Samplers")
        .getChildFile("violin.wav");
    juce::File sampler4File = juce::File::getCurrentWorkingDirectory()
        .getParentDirectory().getParentDirectory()
        .getChildFile("Assets")
        .getChildFile("Samplers")
        .getChildFile("flute.wav");

    // add sampler files to the list
    samplersFile.push_back(sampler1File);
    samplersFile.push_back(sampler2File);
    samplersFile.push_back(sampler3File);
    samplersFile.push_back(sampler4File);

    // iterate the `samplersFile` vector and add them to the `samplers` vector
    for (const juce::File& samplerFile : samplersFile)
    {
        juce::String samplerTitle = player->getAudioTitleFromURL(juce::URL{ samplerFile });
        juce::String samplerURL = juce::URL{ samplerFile }.toString(false);

        // create a Track object and add it to the array.
        Track newSampler{ samplerTitle, samplerURL };
        samplers.push_back(newSampler);
    }

}
```



```cpp
/** Plays the sampler with the specified title */
void SamplerInstrumentsComponent::playSampler(const juce::String& samplerTitle)
{
    // find the sampler with the matching sampler title.
    for (int i = 0; i < samplers.size(); i++)
    {
        // check if the title of sampler match with desired sampler title
        if (samplers[i].title == samplerTitle)
        {
            // load the sampler file
            player->loadURL(samplers[i].url);
            DBG("url:" + samplers[i].url);

            // check if the sampler file is loaded successfully.
            if (player->isFileLoaded)
            {
                // start playing the loaded audio.
                player->play();
                DBG("SamplerInstrumentsComponent::" + samplerTitle + " sampler is playing");
            }
            else
            {
                DBG("SamplerInstrumentsComponent::" + samplerTitle + "sampler failed to load the file");
            }
        }
    }
}
```

Reference:

1. Freepik (no date) *Stop Button Icon*, *Freepik*. Available at: https://www.freepik.com/search?format=search&last_filter=query&last_value=stop+button&query=stop+button&type=icon.
2. Freepik (no date) *Play Button Icons*, *Freepik*. Available at: https://www.freepik.com/search?format=search&last_filter=query&last_value=play+button&query=play+button&type=icon.
3. Gajah Mada (no date) *Upload Button Icon*, *Freepik*. Available at: https://www.freepik.com/search?format=search&last_filter=query&last_value=upload+button+&query=upload+button+&type=icon.
4. KP Arts (no date) *Replay Button Icon*. Available at: https://www.freepik.com/search?format=search&last_filter=page&last_value=1&page=1&query=loop+button&type=icon.