Object Oriented Programming Coursework for

Endterm: Otodecks

Introduction:

As part of this coursework, I created Otodecks, a simple DJ application. A custom deck control component and music library component were added, and these were then integrated into a new GUI design I created for the application.

Requirements:

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

R1A: can load audio files into audio players

R1B: can play two or more tracks

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

R1D: can speed up and slow down the tracks

Figure 1: Custom Deck GUI

Figure 1 shows the deck GUI layout of the DJ Application. Here, it can be seen that users can load audio files into the audio players. They can also pay and mix tracks by varying their volume and speed.

```
private:
    AudioFormatManager& formatManager;
    std::unique_ptr<AudioFormatReaderSource> readerSource;
    AudioTransportSource transportSource;
    ResamplingAudioSource resampleSource{ &transportSource, false, 2 };
};
```

Figure 2: AudioTransportSource Object

An AudioTransportSource object called transportSource is created in the class DJAudioPlayer while inheriting the juce library AudioSource. This gives us the ability to assign actions to an audio source, such starting or pausing music tracks.

The DeckGUI class is used to implement an interface and function as an audio player when playing tracks. Two audio decks have been developed that can play two distinct tracks either concurrently or independently.

```
void DeckGUI::resized()
{
    auto labelposition = 55;//the sapce for the label
    double rowH = getHeight() / 8.0;
    playButton.setBounds(0, 0, getWidth(), rowH);
    stopButton.setBounds(0, rowH, getWidth(), rowH);
    resumeButton.setBounds(0, rowH * 2, getWidth(), rowH - 10);
    volSlider.setBounds(0, rowH * 2 + 40, getWidth() / 2, rowH * 4);
    speedSlider.setBounds(getWidth() / 2, rowH * 2 + 40, getWidth() / 2, rowH * 4);
    posSlider.setBounds(labelposition, rowH * 7, getWidth() - 10 - labelposition, rowH);

    //waveFormDisplay->setBounds(0, rowH * 5, getWidth(), rowH * 2);
}
```

Figure 3: resized() function

The addAndMakeVisible() function in the DeckGUI cpp file enables the volume slider to be shown on the audio deck. The volume slider on the audio player deck is given a size and location using setBounds in the resized() function. The DeckGUI class initializes the volume slider's range so that it goes from 0.0, the least volume, to 1.0, the maximum volume.

Hence, all the requirements R1 is being met. Users are able to laod and play audio tracks simultaneously. They can also mix the tracks by varying the volumes and can increase or decrease the speed of the tracks.

R2: Implementation of a custom deck control Component with custom graphics which allows the user to control deck playback in some way that is more advanced than stop/ start.

R2A: Component has custom graphics implemented in a paint function

R2B: Component enables the user to control the playback of a deck somehow

```
    addAndMakeVisible(playButton);
    addAndMakeVisible(stopButton);
    addAndMakeVisible(resumeButton);

    addAndMakeVisible(volSlider);
    addAndMakeVisible(speedSlider);
    addAndMakeVisible(posSlider);
    volSlider.setSliderStyle(Slider::SliderStyle::Rotary);
    speedSlider.setSliderStyle(Slider::SliderStyle::Rotary);
    volSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 80, 25);
    speedSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 80, 25);
    speedSlider.setValue(1.0);
    volSlider.setValue(1.0);

    addAndMakeVisible(volume);
    addAndMakeVisible(position);
    addAndMakeVisible(speed);

    getLookAndFeel().setColour(Slider::thumbColourId, Colour::fromRGB(38, 246, 198));
    getLookAndFeel().setColour(Slider::rotarySliderFillColourId, Colour::fromRGB(178, 102, 255));
    getLookAndFeel().setColour(TextButton::buttonColourId, Colour::fromRGB(38, 246, 198));
    getLookAndFeel().setColour(TextButton::textColourOffId, Colour::fromRGB(0,0,0));
    getLookAndFeel().setColour(Slider::trackColourId, Colour::fromRGB(178, 102, 255));

    volume.setText("volume", juce::dontSendNotification);
    position.setText("Position", juce::dontSendNotification);
    speed.setText("Speed", juce::dontSendNotification);

    volume.setJustificationType(Justification::centredBottom);
    speed.setJustificationType(Justification::centredBottom);

    volume.attachToComponent(&volSlider, false);
    position.attachToComponent(&posSlider, true);
    speed.attachToComponent(&speedSlider, false);
```

Figure 4: Custom Graphics

The function setSliderStyle() in the DeckGUI cpp file is used to convert the volume and position slider objects from linear sliders to rotational sliders by inheriting the SliderStyle property from the slider class. As a result, the sliders for volume and speed are of rotary type.

I used the setText() function to take in a text parameter for each label that will serve as the name of each slider in the program. "Volume," "Speed," and "Position" are the names of the corresponding sliders for these three. Another parameter that setText() accepts is one that is configured to disable user notification push. Each label is positioned so that it is in the center of each slider using the setJustificationType() function. The setColour() function accepts an integer for the textColourId and uses the juce class colors to paint. Font type and font size are parameters that the setFont() function accepts.

```
void DJAudioPlayer::setPositionRelative(double pos)
{
    if (pos < 0 || pos > 1.0)
    {
        std::cout << "DJAudioPlayer::setPositionRelative pos should be between 0 and 1" << std::endl;
    }
    else
    {
        double posInSecs = transportSource.getLengthInSeconds() * pos;
        setPosition(posInSecs);
    }
}
```

Figure 5: setPositionRelative()

The function sliderValueChanged() implements this by retrieving the slider's value using getValue (). The DJAudioPlayer class's setRelativePosition() function, which we developed in the second figure above, then takes this value and multiplies it by the transportSource's getLengthInSeconds function's calculation of the total duration of the track that is presently being played (). It is then used to invoke the setPosition() function, which sets the playback position to the corresponding timing in the track or transportSource. It is saved in a new double variable called posInSecs. Hence, the playback control is put into practice.

R3: Implementation of a music library component which allows the user to manage their music library

R3A: Component allows the user to add files to their library

R3B: Component parses and displays meta data such as filename and song length

R3C: Component allows the user to search for files

R3D: Component allows the user to load files from the library into a deck

R3E: The music library persists so that it is restored when the user exits then restarts the application

```
    tableComponent.getHeader().addColumn("Track List", 1, 400);
    tableComponent.getHeader().addColumn("Load", 2, 100);
    tableComponent.getHeader().addColumn("Deck 1", 3, 100);
    tableComponent.getHeader().addColumn("Deck 2", 4, 100);
    tableComponent.getHeader().addColumn("Delete", 5, 100);
    tableComponent.setModel(this);
    addAndMakeVisible(tableComponent);
    addAndMakeVisible(searchingTextBox);
    addAndMakeVisible(saveButton);
    addAndMakeVisible(loadButton);

    saveButton.addListener(this);
    loadButton.addListener(this);
```

Figure 6: tableComponent()

The constructed load button can be seen on the tableComponent thanks to the code mentioned above, which is located in the cpp file. The load button in the tableComponent is given a size and a location using setBounds in the resized() function.

```
    searchingTextBox.setTextToShowWhenEmpty("search your music", juce::Colours::black);
    searchingTextBox.onReturnKey = [this] {searching(searchingTextBox.getText()); };
    tableComponent.setMultipleSelectionEnabled(true);
    illustration();

    getLookAndFeel().setColour(TextEditor::textColourId, Colour::fromRGB(255, 204, 153));
    getLookAndFeel().setColour(TextEditor::backgroundColourId, Colour::fromRGB(255,255,255));

    tableComponent.setColour(ListBox::backgroundColourId, Colour::fromRGB(0,0,0));
}
```

Figure 7: searchingTextBox()

The PlaylistComponent class' header file implements a function with the name searchingTextBox (). A new variable called searchingTextBox is generated from the TextEditor class from the juce library in the header file for the PlaylistComponent to build a search box that the user can use to look for tunes.

R4: Implementation of a complete custom GUI

R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls

R4B: GUI layout includes the custom Component from R2

R4C: GUI layout includes the music library component from R3

```
void DeckGUI::buttonClicked(Button* button)
{
    if (button == &playButton)
    {
        std::cout << "Play button was clicked " << std::endl;
        playButton.setEnabled(false);
        stopButton.setEnabled(true);
        player->start();
    }
    if (button == &stopButton)
    {
        std::cout << "Stop button was clicked " << std::endl;
        stopButton.setEnabled(false);
        playButton.setEnabled(true);
        player->stop();

    }
    if (button == &resumeButton) {
        stopButton.setEnabled(true);
        playButton.setEnabled(true);
        player->stop();
        posSlider.setValue(0.0);
        player->resume();
    }
```

Figure 8: resumeButton()

Using the positionSlider as its parameters, the code above adds a button listener to the resumeButton. Also, the deckGUI is able to resume the current track by using the start() function in the DJAudioPlayer class, which is invoked and loaded into the player.

The codes from both the DeckGUI class and the PlaylistComponent class were used to create the music library component, tableComponent. Each newly introduced feature is placed such that it has a certain GUI layout.