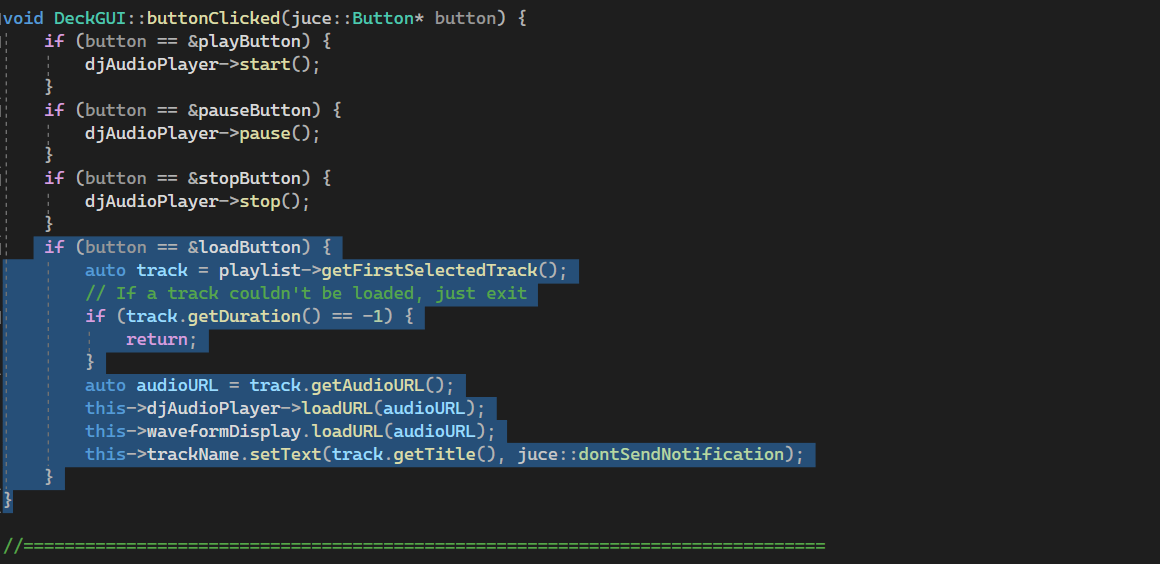
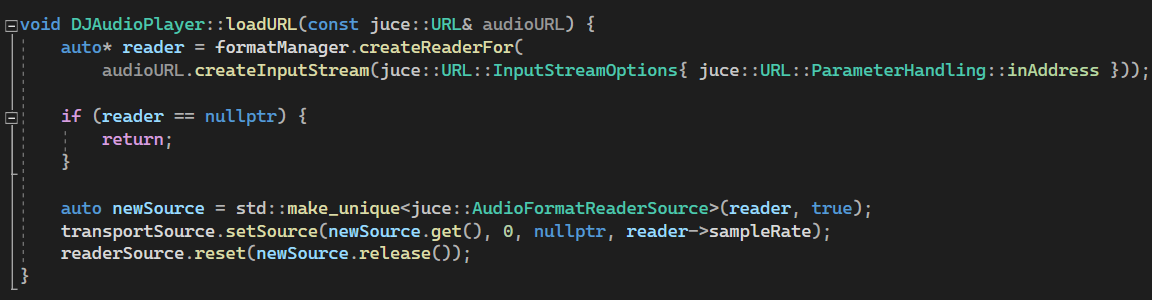
**OOP Final Report**

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

* R1A: Can load audio files into audio players

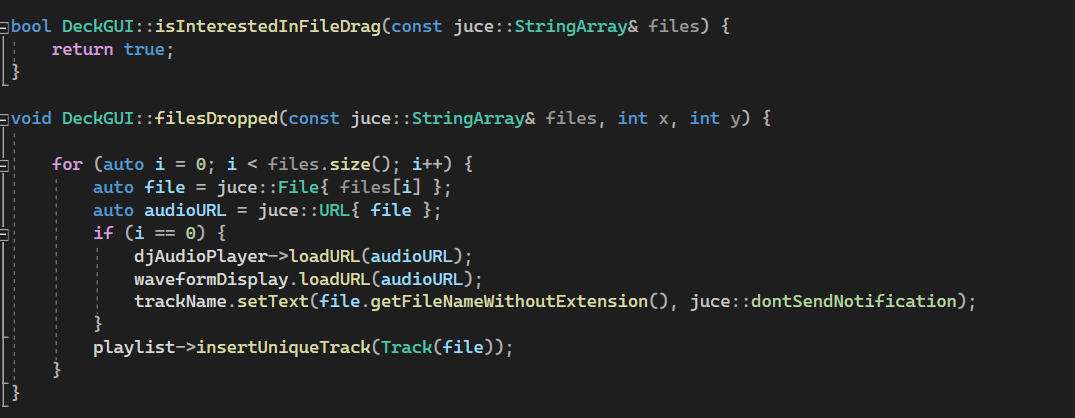


In the “DeckGUI.CPP” file, under void DeckGUI::buttonClicked, there is &loadbutton function. With this feature, users are being prompted to select a song from their computer, in which it will be loaded into the player. The purpose of the loadButton is to give the user the ability to load files into the app. The user is supposed to click on the button, and from there choose any audio file.



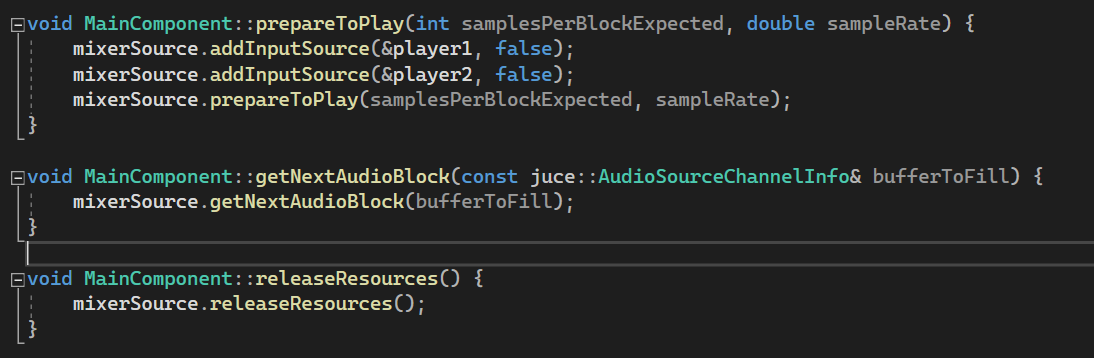
In the “DJAudioPlayer.cpp, void DJAudioPlayer :: loadURL function will direct the user to their computer’s file section where they can choose their song/track to load into the deck.

For the player, when the loadURL() method is called an input stream is created, and a reader is created for it, from there the audio source can be played, stopped, etc.



Under “DeckGUI.CPP” both of these above functions allows the user to drag and drop their track/song into the decks from their file manager.

* R1B: can play two or more tracks

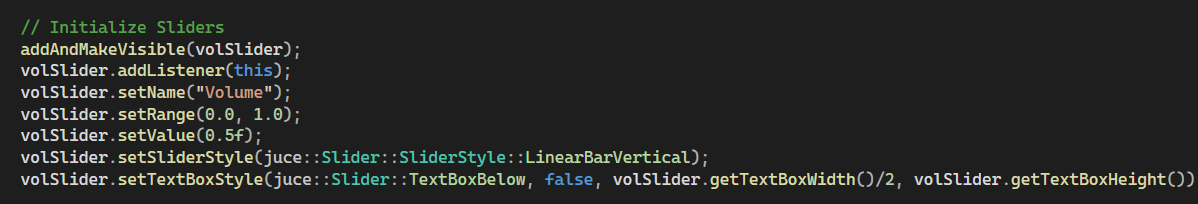


In order to play two or more tracks / songs, I created a MixerAudioSource function. The MixerAudioSource is an AudioSource that mixes together the output of a group of other AudioSources. Therefore, it gives the program the ability to play two or more tracks at one particular time. In the “MainComponent.cpp” file, void MainComponent::prepareToPlay()function, mixerSource calls prepareToPlay(), which tells the source to prepare for playing. Within the same prepareToPlay () function mixerAudio calls the addInputSource () method, twice, once for each player as shown in the codes above.

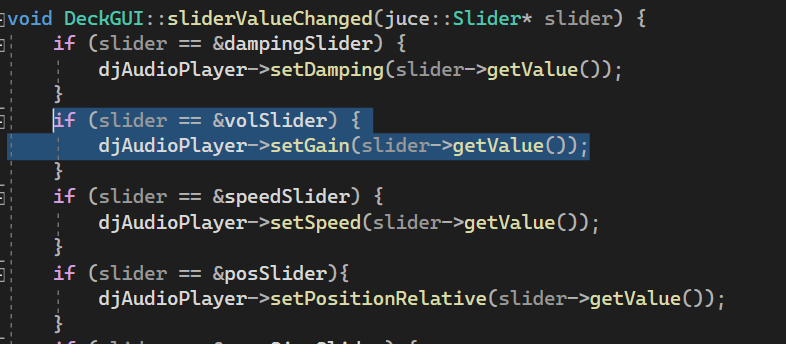
Under “MainComponent.cpp” I created the MainComponent::getNextAudioBlock() function. In this function, mixerSource calls getNextAudioBlock(), which is a function that is called to retrieve successive blocks of audio data. Once prepareToPlay() is called, getNextAudioBlock() continues to be called while an audio block of data is needed. Within the MainComponent::releaseResources() function, mixerSource calls releaseResources(), this method allows the source to release anything that it no longer needs, once playback is complete.

All the above functions work collectively, allowing more than 1 track / song to be played.

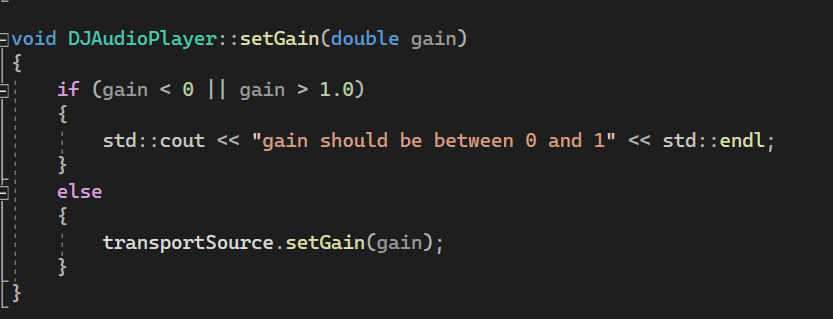
* R1C: can mix the tracks by varying each of their volumes



The volume of the tracks can be adjusted using a volume slider. In order to achieve this, under “DeckGui.cpp” I have juce::Slider::Listener function which receives call-backs from a Slider. When the user moves the slider up and down, it changes the value of the slider; an addListener () method is called on the volume slider, so that the program knows when the slider’s value has changed.

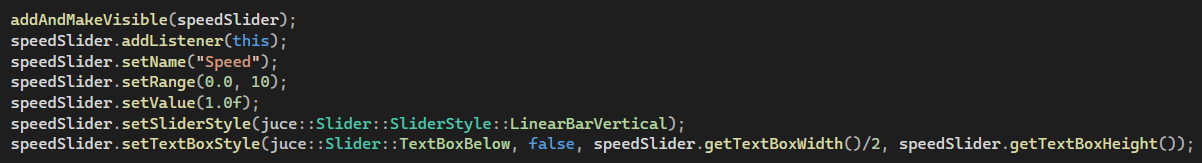


The user could move the slider as much as they want, but their actions would have no effect on the volume. Within the DeckGUI::sliderValueChanged() function, there is an “if(slider== &volslider) statement is used as a check, so the program knows which slider the user has used. If the address of the slider is that of the volume slider, then the player calls setGain(), and passes the slider’s getValue() method, as a parameter.

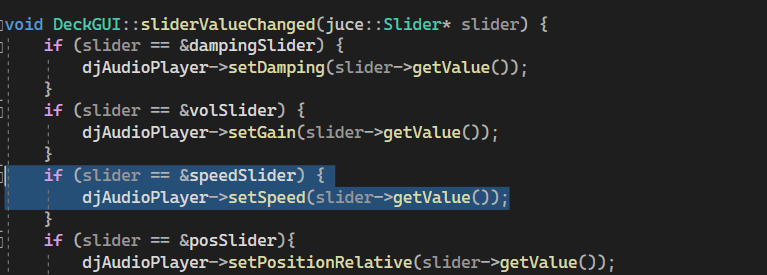


In the “DJAudioPlayer.cpp”, in the DJAudioPlayer::setGain() function, transportSource calls setGain(), and then passes the value it received from the DJAudioPlayer::setGain() function as a parameter. As a result, when two tracks are playing simultaneously, the user is able to adjust their volumes. This added functionality allows the user to fade tracks in and out, blend the sounds, and easily allow one track to be dominant over another, and vice versa- all giving a wonderful mixing effect.

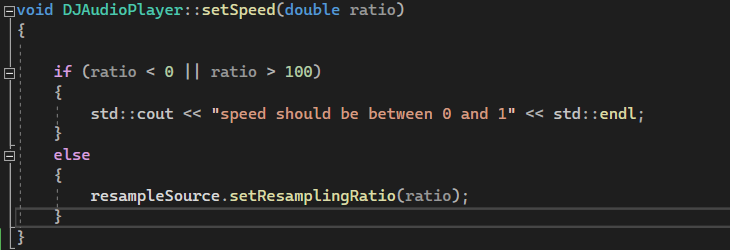
* R1D: can speed up and slow down the tracks



Like R1C for volume, speed works the same way. The user can adjust the speed using a slider. This is accomplished by first going through the typical motions: creating a Slider and naming it speedSlider; calling addAndMakeVisible() on it; and then adding a listener to it.



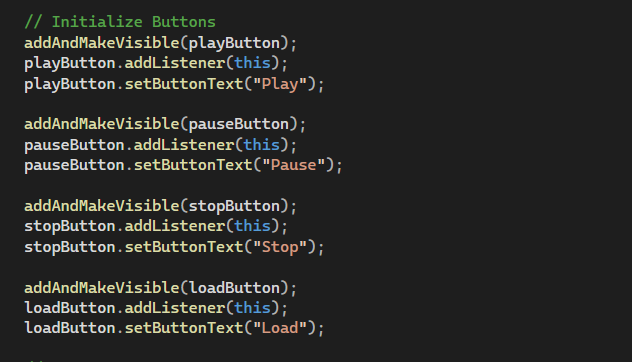
Once those three tasks are complete, within the DeckGUI::sliderValueChanged() function, I added the following “if(slider== speedSlider” statement for speed as shown above.

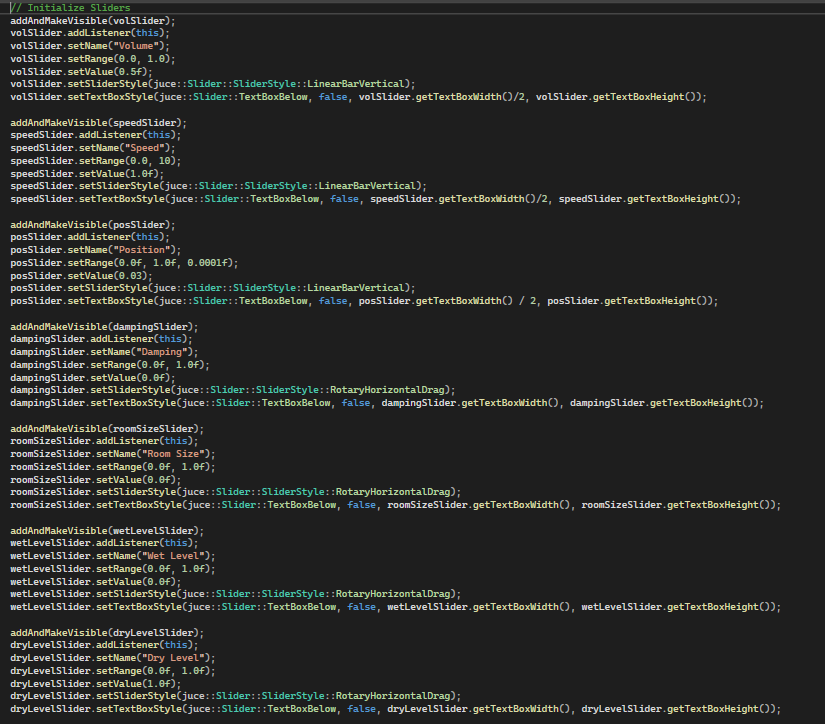


The “if(slider== speedSlider” statement, first checks to see whether the slider in use, is indeed the speedSlider (by comparing the address), and if so the player calls the setSpeed() method, and passes the slider’s new value to it. In the “DJAudioplayer.cpp” the DJAudioPlayer::setSpeed() function, resampleSource calls setResamplingRatio(), which changes the resampling ratio, which in turn changes the speed to the users’ desired level.

R2: Customise the user interface (UI): i.e., change the colours, and change the layout.

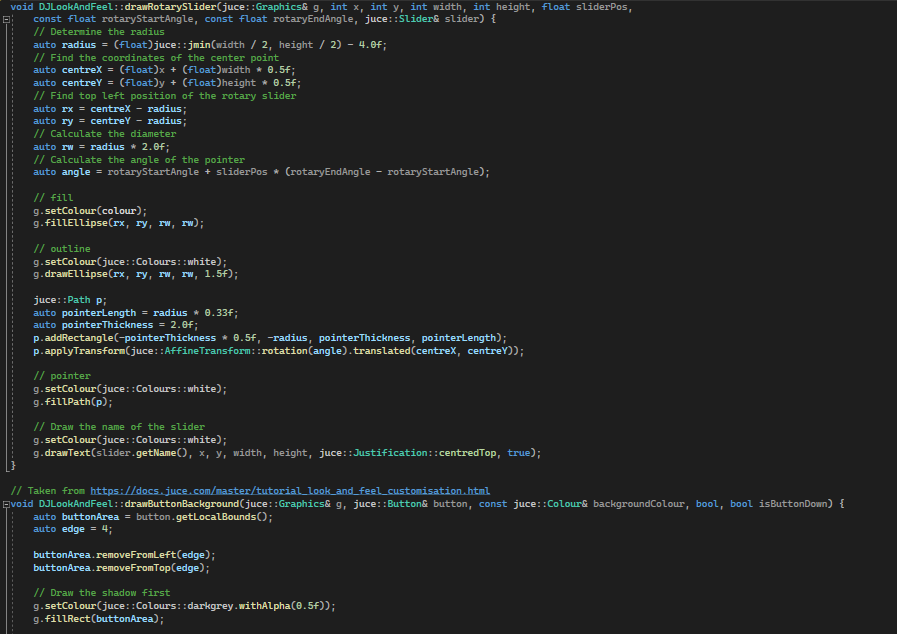
* R2A: GUI layout is significantly different from the basic DeckGUI shown in class

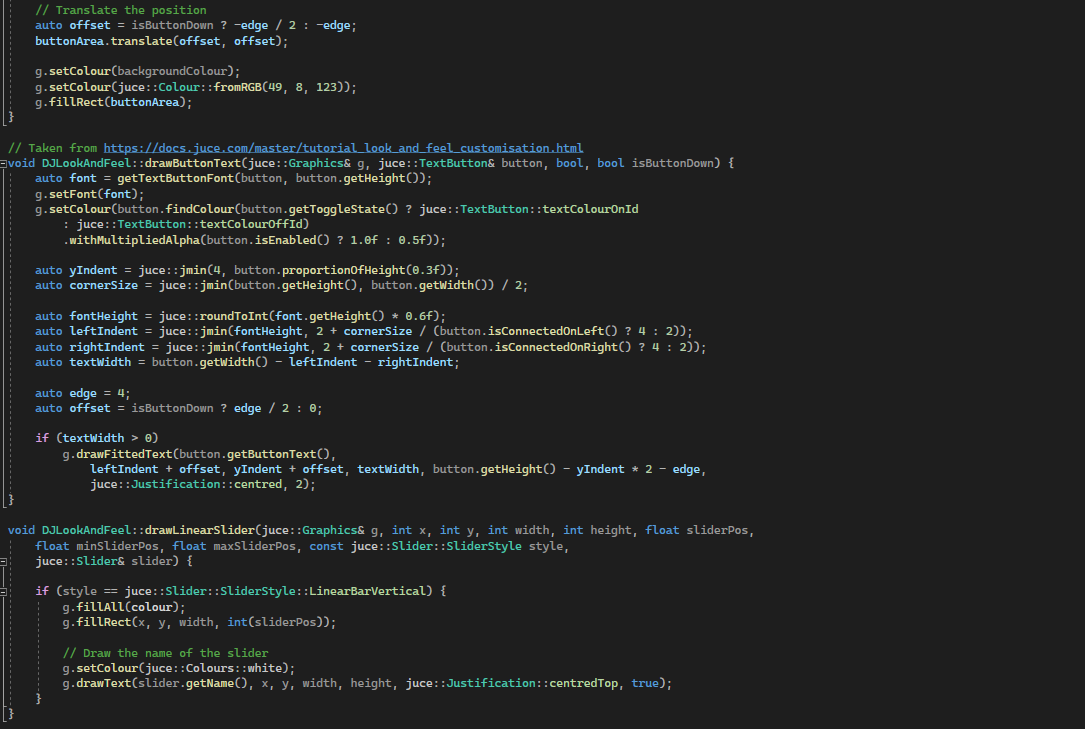




In the “DeckGUI.cpp”, to showcase a more creative layout, I first initialize the buttons and sliders as shown above. The user can play, pause, load and stop the tracks they want using buttons. This is accomplished by first going through the typical motions: creating a various buttons and naming it playButton, pause Button, stopButton and loadButton and most importantly calling addAndMakeVisible() on it; and then adding a listener to it.

For the sliders, I wanted position, speed and volume represented as a simple Linear Vertical Bar whereas for my R3 Features (Room Size, Damping, Wet & Dry Level) I wanted a rotary slider. For all the 7 sliders, I set the range and set values to be shown in the Otodecks App.



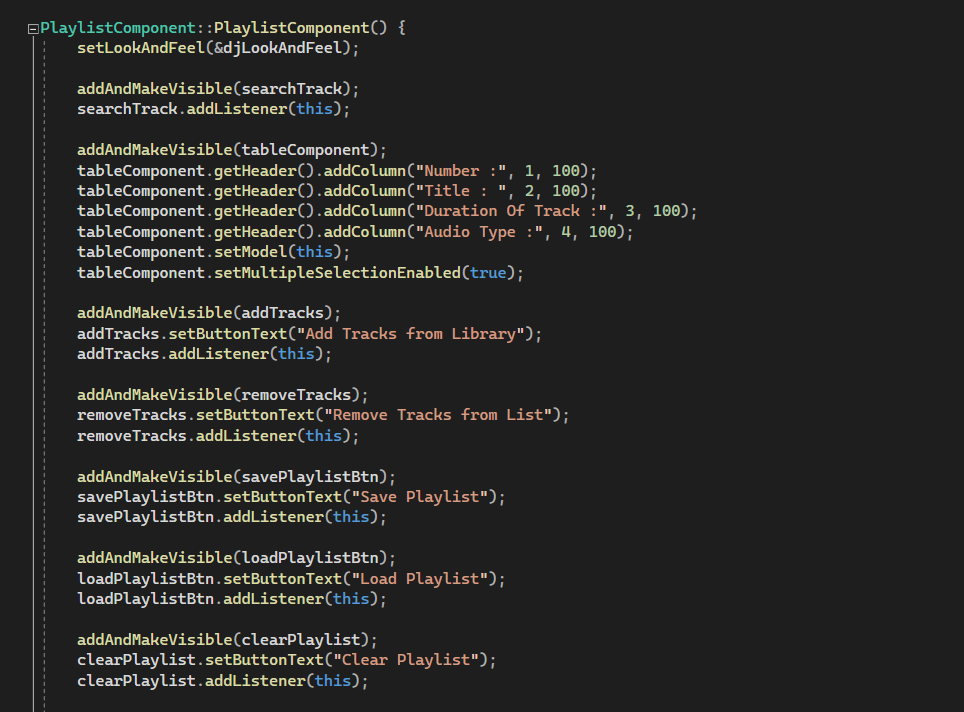


I also created “DJLookAndFeel.cpp” file as shown in the 2 screenshots shown above. Next, within the DeckGUI’s header file, I created an instance of the DJLookAndFeel class, which I named AS DJLookAndFeel. In the DeckGUI’s cpp file, within the constructor I passed setLookAndFeel as a parameter into setLookAndFeel(&customLookAndFeel);function. Thus changing the default-look-and-feel. I use the DJLookAndFeel.cpp to present my general codes for Linear and Rotary Sliders, Button Text (font size etc). I also created codes for other functions such as ellipse and rectangle width/height for the rotary and linear respectively in this cpp file.

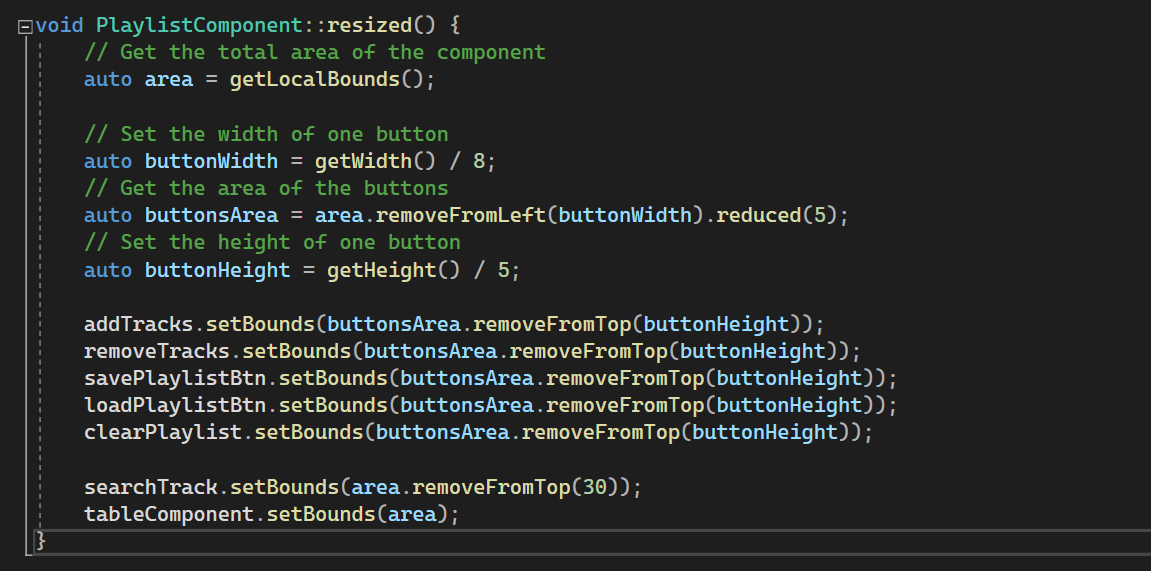
All the combined codes from DeckGUI and DJLookAndFeel files work together to produce my enhanced Otodecks Player as shown below.



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



Under “PlaylistComponent.cpp” file, I added table component function to show the Number of each track, Title of each track, Duration of track and lastly the audio format of each track. Moreover, I also added addAndMakeVisible function with buttontext ( buttons ) to Save,Clear and Load Playlist into either right or left decks as well as adding tracks / songs from library and lastly to remove tracks from the list in the deck.



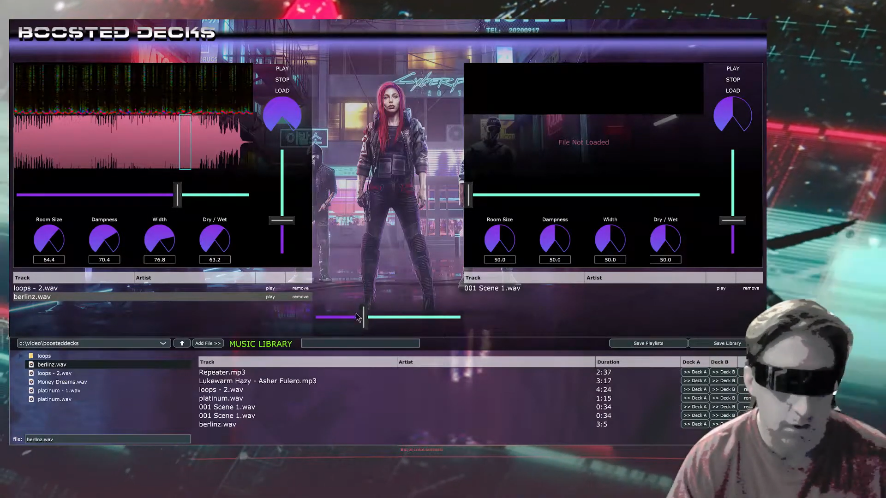
Later in the same playlist component file, within the void PlaylistComponent::resized function , I set bounds and height for the 5 buttons.

**Delving into specifics:**

For addTracks function, it allows user to add one and multiple tracks at the same time into the deck. For removeTracks function, in the deck, usrs can select one particular track and remove it completely from the deck. The savePlaylist function will allow users to save several tracks as a playlist into their own file manager on their computer. On the other hand, the loadPlaylist function aids the users by allowing them to load the whole saved playlist that they got from Otodecks ( the one they saved using the savePlaylist function on the deck ) into either decks. And last but not least, for clearPlaylist function it aims to clear the whole set of tracks from the deck so as to enable the user to create a different form of playlist or just include multiple tracks.

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

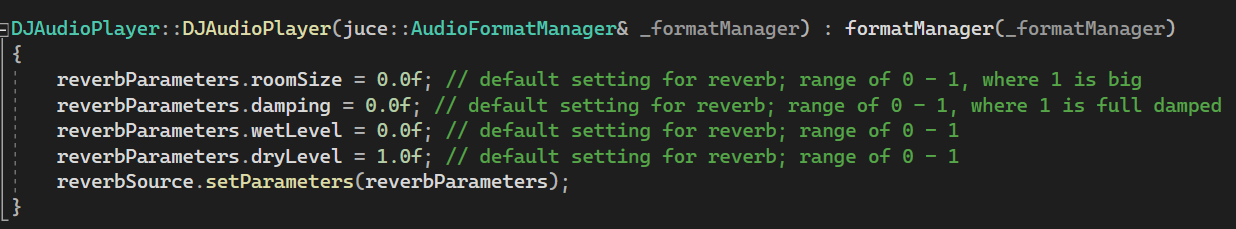
For R3, other than the basic features that was taught in class ( Position , Volume & Speed ) I decided that I wanted to try out these 4 features : Room Size , Damping , Wet and Dry Levels. I went to Youtube ( <https://youtu.be/0fvtOjzgx6A?si=SP3ecJ1Za9lt5k_B> )and liked this design as shown below:



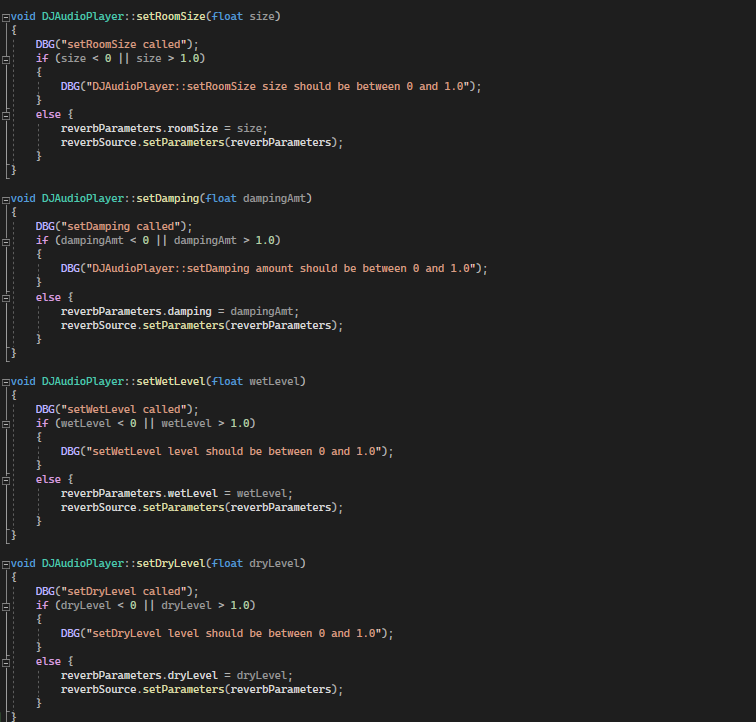
So as to research more, I went to Github and I found this person who did like damping factor for their Otodecks app [ Link : <https://github.com/brianvansteen/OtoDecks> ] . This is their output:



This is how I implemented the 4 features:



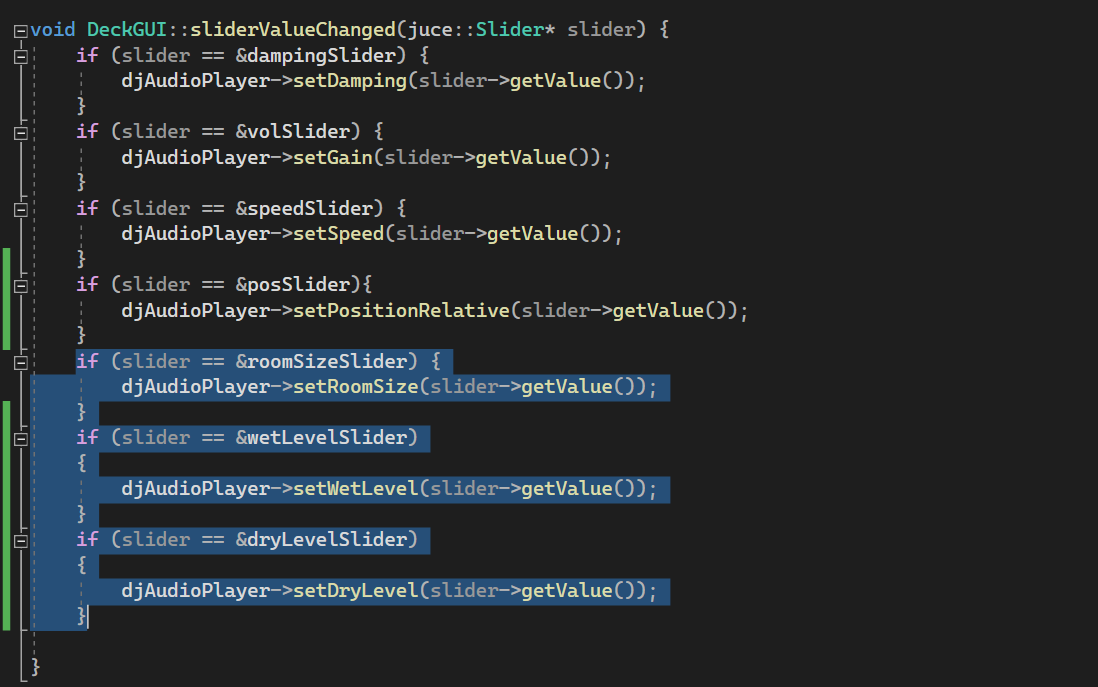
Firstly under “DJAudioPlayer.cpp”, I added DJAudioPlayer function I created reverbParameters for the 4 features and set the default values.



Then in the same DJAudioPlayer.cpp file , I created void DJAudioPlayer function to set all the levels as well as the range of each 4 features , which is between 0.0 to 1.0 for all the 4 features.



The room size, damping, wet and dry levels of the tracks can be adjusted using rotary sliders. In order to achieve this, under “DeckGUI.cpp” I have juce::Slider::Listener function which receives call-backs from a Slider. When the user moves the slider up and down, it changes the value of the slider; an addListener () method is called on the 4 sliders, so that the program knows when the slider’s value has changed.



Next in the “DeckGUI.cpp” I created the DeckGUI::sliderValueChanged() function as shown above , for all the 4 features. For all of them, the “if(slider==) statement is used as a check, so the program knows which slider the user has used. If the address of the slider is that of the wet level slider, then the player calls setWetLevel(), and passes the slider’s getValue() method, as a parameter.

Final Look:

