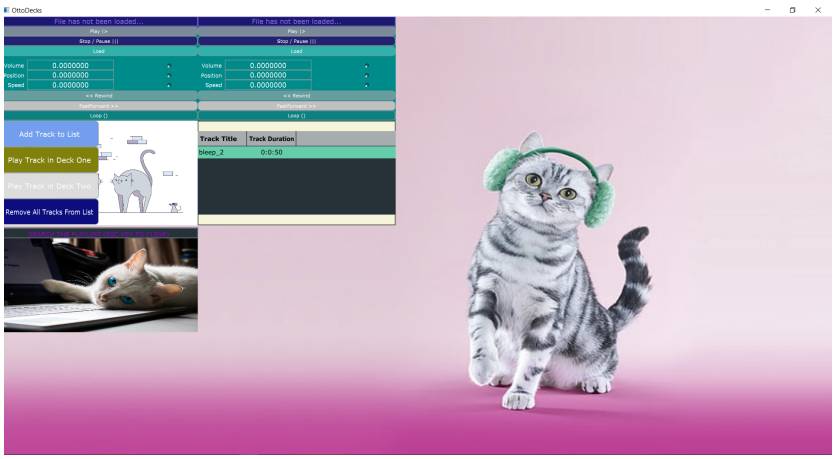
OTTO DECKS: A DJ APPLICATION WRITTEN IN JUCE



The following image depicts the final version of the OttoDecks application. It went through a couple of iterations before the cat theme was chosen as the final look. Why? Because cats are *awesome*.

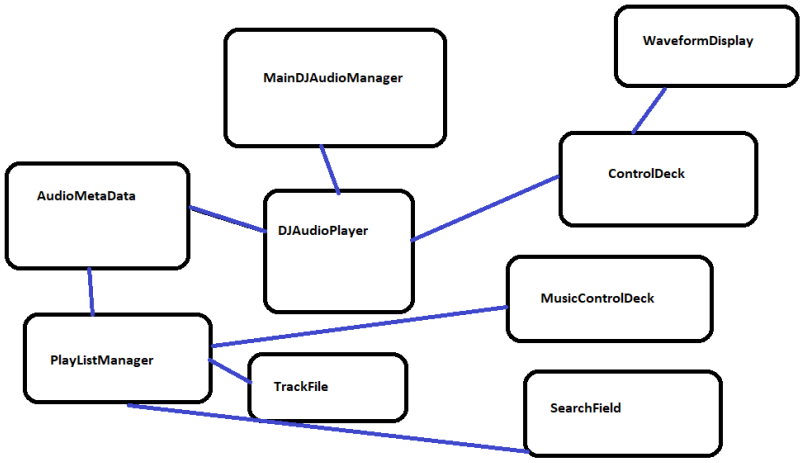
Purpose of the Application

Otto Decks is an audio application that was created using the JUCE framework: the underlying programming language of the framework is C++. More information on the Juce framework can be found here.

The main purpose of Otto Decks is to simulate the real behaviour of a DJ. This means that audio tracks can be loaded into players, their volumes adjusted and that a custom playlist can be created and saved for later use. In the same vein, the contents of the playlist can also be deleted, although more advanced features such as deleting one track separately have not been implemented at the current time of writing.

Class Implementation Diagram

The diagram below showcases how each of the components and classes are dependent on one another. The main goal behind creating other components in relation to the main one is to keep their specific function-related logic isolated, so that the code can be reused for future projects by using the DRY principle.



Requirements Table

The following table provides references as to how each of the final project prerequisites have been met. The methods and classes should serve as a guideline as to where to find a given requirement.

Requirement	(Y/N)	Relevant Methods / Classes

R1A: can load audio files into audio players	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::loadURL(URL audioURL)
R1B: can play two or more tracks	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::playSong()
R1C: can mix the tracks by varying each of their volumes	Y	void ControlDeck::sliderValueChanged(Slider* slider) void DJAudioPlayer::setSpeed(double ratio)
R1D: can speed up and slow down the tracks	Y	void ControlDeck::sliderValueChanged(Slider* slider) void DJAudioPlayer::setGain(double gain)
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.	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::fastForwardSong() void DJAudioPlayer::rewindSong() void DJAudioPlayer::startLoop()
R2A: Component has custom graphics implemented in a paint function.	Y	void MainDJAudioManager::paint (juce::Graphics& graphics) void MainDJAudioManager::resized() void ControlDeck::paint(juce::Graphics& graphics) void ControlDeck::repaintButtons() void ControlDeck::repaintSliders()
R2B: Component enables the user to control the playback of a deck somehow.	Y	void ControlDeck::buttonClicked(Button* button) void DJAudioPlayer::fastForwardSong() void DJAudioPlayer::rewindSong() void DJAudioPlayer::startLoop()
R3A: Component allows the user to add files to their library.	Y	void MusicControlDeck::buttonClicked(Button* button) void MusicControlDeck::populateTracksFile() MusicControlDeck::checkIfTrackAlreadyLoaded(TrackFile& trackFile) juce::Array<juce::File> MusicControlDeck::loadInTracks() void MusicControlDeck::removeAllContentsFromFile() void PlayListManager::addTrackToTracksVector(TrackFile& trackFile) void PlayListManager::deleteTracksFromTable()
R3B: Component parses and displays meta data such as filename and song length.	Y	juce::String TrackFile::getFileName() void TrackFile::setFileLength(juce::String length) uce::String AudioMetaData::getAudioTrackLength(juce::URL& audioURL)

		double DJAudioPlayer::getFileLengthSeconds() juce::String AudioMetaData::getFormattedAudioString(double& seconds)
R3C: Component allows the user to search for files.	Y	void PlayListManager::searchThePlaylist(juce::String inputtedText) void SearchField::changeTextWhenSearching() void SearchField::setUpSearchFieldProperties() void SearchField::clearTextAfterSearch()
R3D: Component allows the user to load files from the library into a deck.	Y	void PlayListManager::addTrackToPlayerOne() void PlayListManager::addTrackToPlayerTwo()
R3E: The music library persists so that it is restored when the user exits then restarts the application.	Y	void PlayListManager::loadTracksFile() void PlayListManager::saveTracksFile()
R4A: GUI layout is significantly different from the basic DeckGUI shown in class, with extra controls.	Y	void MainDJAudioManager::paint (juce::Graphics& graphics) void MainDJAudioManager::resized() void MusicControlDeck::paint(juce::Graphics& graphics) void MusicControlDeck::resized() void MusicControlDeck::repaintButtons()
R4B: GUI layout includes the custom Component from R2.	Y	void ControlDeck::paint(juce::Graphics& graphics) void ControlDeck::resized() void ControlDeck::repaintButtons() void ControlDeck::repaintSliders() void MusicControlDeck::paint(juce::Graphics& graphics) void MusicControlDeck::resized() void MusicControlDeck::repaintButtons()
R4C: GUI layout includes the music library component from R3.		void PlayListManager::paint(juce::Graphics& graphics) void PlayListManager::resized() void SearchField::paint (juce::Graphics& graphics) void SearchField::resized()

Basic Functionality

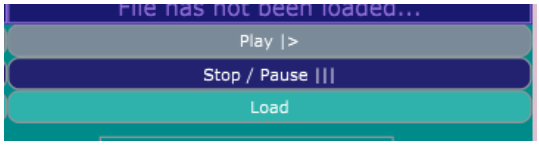
The following section provides an overview of the basic functionality that was worked on during the lecture; it also includes additional features created for the final project.

High Level Description of the Features

The basic functionality of the Otto Decks application can be broken down into three buttons and sliders. Each of these buttons and sliders have specific tasks associated with them as discussed in the paragraphs below.

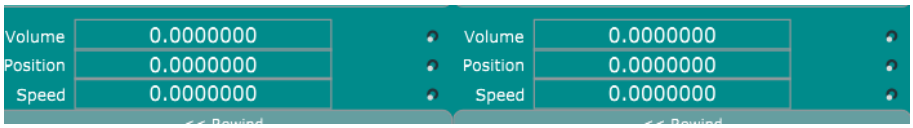
The Buttons

The three buttons are in control of:



- ★ Playing or resuming the audio of a track.
- ★ Stopping or pausing a track's audio.
- ★ Uploading an audio file into one of the two available players.

The Sliders



In addition to the buttons, the sliders, positioned underneath them, are responsible for:

- ★ The volume of the audio.
- ★ The position of the audio.
- ★ The speed of the audio.

Apart from the button and sliders, two components play an important role in terms of the basic functionality, namely the ControlDeck and DJAudioPlayer.

The ControlDeck Component

This component is in charge of mapping each button to the appropriate function call that deals with the relevant audio effect. This means that:

- ★ If the play button is clicked, it triggers the playSong function in DJAudioPlayer.
- ★ If the load button is clicked, it calls the loadURL function in the DJAudioPlayer.
- ★ If the stop button is clicked, it kicks off the stopSong function in DJAudioPlayer.
- ★ Each time a slider's value is changed, it communicates the desired audio effect to the relevant function in DJAudioPlayer.

The DJAudioPlayer Component

This component deals with the audio and the various ways its playback can be controlled / manipulated. In other words, it:

- ★ Loads a track from a given url which is often a directory.
- ★ Plays and stops a track.
- ★ Sets the volume, speed and gain of a track.
- ★ Deals with all the required audio processing.

Low Level Description of the Features

Loading a File

As stated above, the DJAudioPlayer component is responsible for the audio and playback processes of the tracks. For example, the loading of the tracks is dealt with in the following function below.

```
void DJAudioPlayer::loadURL(URL& audioURL)
{
	auto* reader = formatManager.createReaderFor(audioURL.createInputStream(false));

	if (reader != nullptr)
	{
		std::unique_ptr<AudioFormatReaderSource> newSource(new
AudioFormatReaderSource(reader, true));
		transportSource.setSource(newSource.get(), 0, nullptr, reader->sampleRate);
		readerSource.reset(newSource.release());
	}
}
```

The ControlDeck, meanwhile, matches each audio-related function to its corresponding button; the code snippet below is part of the buttonClicked method which is an override of Button::Listener. It uses a pointer to the DJAudioPlayer component to refer to methods from it. In the case below, it calls the loadURL function from DJAudioPlayer.

```
if (button == &loadButton)
   {
      auto dlgFlags = juce::FileBrowserComponent::openMode | juce::FileBrowserComponent::canSelectFiles;
      this->chooser.launchAsync(dlgFlags, [this](const juce::FileChooser& chooser)
         {
            auto fileUri = chooser.getURLResult();
            player->loadURL(fileUri);
            waveformDisplay.loadURL(fileUri);
         });

      paused = false;
   }
```

Playing or Stopping a Song

Functions such as playing or pausing the audio are carried out by functions such as playSong. An example is displayed below.

```
void DJAudioPlayer::playSong()
{
	transportSource.start();
}

void DJAudioPlayer::stopSong()
{
	transportSource.stop();
}
```

Yet again, the buttonClicked function is used to call upon the relevant DJAudioPlayer methods for stop and play. A bool value called paused is used to differentiate between the various states of the track; this is important for the text display of the play button.

```
if (button == &playButton)
   {
      player->setPosition(0);
      player->playSong();
      paused = false;
   }

   if (button == &stopButton)
   {
      player->stopSong();
      paused = true;
   }
```

Changing The Play Button Text To Resume

Another important part of the audio-related tasks is to distinguish the various states of the play button. For example – when the audio has been paused – the text of the play button changes to 'Resume' as shown below.

```
void ControlDeck::displayPlayButtonText(bool pauseButtonStatus)
{
   std::string playButtonText = "";
   switch (pauseButtonStatus)
   {

   case true:
      playButtonText = "Resume";
      break;

   case false:
      playButtonText = "Play";
      break;
   }

   playButton.setButtonText(playButtonText);
}
```

This function takes a bool parameter as an argument that determines which text will be displayed for the play button in the wake of others having been clicked.

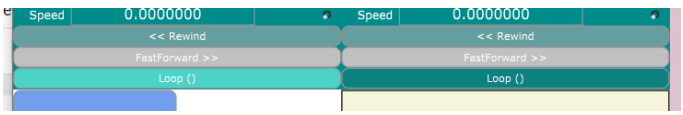
Custom Deck Control

This section deals with the custom control decks that were created for the final project.

High Level Description of the Features

In addition to the basic controls, three new buttons have been added.

The Buttons



Building upon the design of the original buttons, they contribute to the Otto Decks application as listed below:

- ★ The rewinding of a track.
- ★ The fastforwarding of a track.

- ★ The looping of a track.

As in the case of the basic functionality, the ControlDeck and DJAudioPlayer components also play an important role in the advanced audio playback features.

The ControlDeck Component

In addition to the buttons above, the ControlDeck also contains:

- ★ The ability to change the colours of the sliders and buttons depending on their state of activity.
- ★ Custom designs such as making the slides rotary and adding labels next to them.

The DJAudioPlayer Component

Beyond the basic functionality, this component also deals with the following audio-related tasks:

- ★ Rewinding or fast forwarding a song via the help of the track's current position and then adding / subtracting from it.
- ★ Looping a track in the case that the relevant button has been clicked.
- ★ Getting the file length in seconds, which is important for displaying the file length of a track in the playlist.

Low Level Description of the Features

Rewinding / Fastforwarding a Song

The function below takes advantage of the track's current position and calculates a point at which the song can be rewinded from.

```
void DJAudioPlayer::rewindSong()
{
        int currentPosition = transportSource.getCurrentPosition();

        if (currentPosition > 0.5)
        {
                setPosition(currentPosition - 5.0);
        }

        else
        {
                setPosition(0.0);
        }
}
```

Like with the previous basic playback methods, this function is called by the ControlDeck's buttonClicked function as shown in the code snippet below.

```
if (button == &rewindButton)
   {
      player->rewindSong();
      paused = false;
   }
```

Similarly to the rewindSong function above, fastforwarding a track also takes advantage of its current audio position in order to calculate a point from which to move the track along as shown below.

```
void DJAudioPlayer::fastForwardSong()
{
	double lastPositionHeld = transportSource.getLengthInSeconds();
	int currentPosition = transportSource.getCurrentPosition();

	if (currentPosition + 1.5 != lastPositionHeld && currentPosition + 1.5 > lastPositionHeld)
	{
		transportSource.setPosition(currentPosition + 2.0);
	}
}
```

This function is also referenced by the buttonClicked method in order to call fastForwardSong.

```
if (button == &fastForwardButton)
   {
      player->fastForwardSong();
      paused = false;
   }
```

Looping a Track

Looping a track checks with a boolean variable called buttonIsOn as to whether the relevant loop button has been clicked. The code snippet below shows the interplay between the DJAudioPlayer and ControlDeck in terms of how this is accomplished.

```
void DJAudioPlayer::startLoop(bool buttonIsOn)
{
	if (readerSource != nullptr)
	{
		readerSource->setLooping(buttonIsOn);
	}
}

void ControlDeck::buttonClicked(Button* button)
{...
  // only loop when the loop button has been pressed / is on
   if (button == &loopButton)
   {
      paused = false;
      player->startLoop(loopButton.getToggleState());
   }
…
}
```

Repainting Buttons and Sliders

The ControlDeck component also includes a way to repaint the button depending on their state: the function is called repaintButtons which is called in the paint function. Its implementation details are shown below.

```
void ControlDeck::repaintButtons()
{
```

```
    // set the colour for the play button if mouse over OR it has stopped playing
    if (playButton.isOver() || playButton.isMouseOver())
    {
        playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::darkblue);
    }

    // set the colour of the play button when the mouse is not hovering over it
    else
    {
        playButton.setColour(juce::TextButton::buttonColourId, juce::Colours::lightslategrey);
    }

    // set the colour of the stop button if mouse is over OR it has stopped playing
    if (stopButton.isOver() || stopButton.isMouseOver())
    {
        stopButton.setColour(juce::TextButton::buttonColourId, juce::Colours::aquamarine);
    }
```

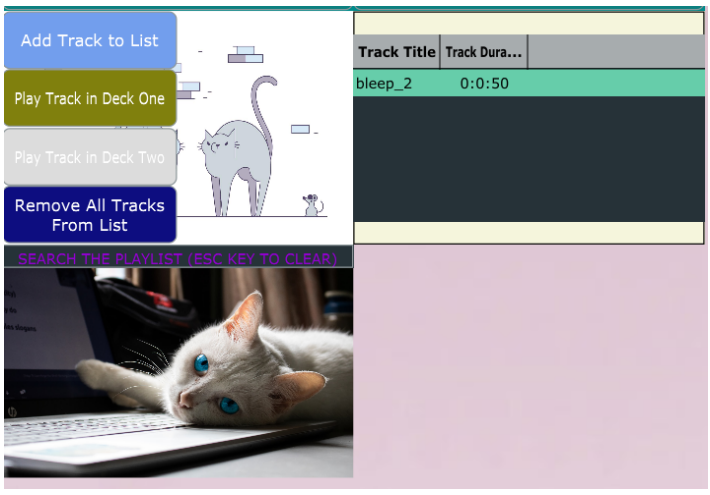
A variation of the repaintButtons method is reused for the sliders as shown below.

```
void ControlDeck::repaintSliders()
{
    if (volumeSlider.isMouseOver() || positionSlider.isMouseOver() || speedSlider.isMouseOver())
    {
        getLookAndFeel().setColour(juce::Slider::thumbColourId, juce::Colours::mistyrose);
    }

    else
    {
        getLookAndFeel().setColour(juce::Slider::thumbColourId, juce::Colours::gainsboro);
    }
}
```

The Playlist

This section is concerned with the playlist and the manner in which it has been implemented.



High Level Description of the Features

The implementation of the playlist is one of the most complex in the entire Otto Decks application; it contains three components and two classes.

The classes serve as placeholders for specific audio-related properties, such as the track file length or the audio metadata information that can be retrieved from the player. Meanwhile, the components are representations of various GUI parts.

The MusicControlDeck Component

The main objective of this component is the creation of the tracks.txt file; it is also in charge of assigning each button to a specific role. More specifically, it:

- ★ Saves the tracks to a vector which is used as a container to store them in one place.
- ★ Loads the tracks from a txt file into the application.
- ★ Saves the tracks into a txt file if the application has been closed.
- ★ Determines which audio player a track is assigned to.
- ★ Searches the track vector for a given input string.
- ★ Deals with all table-related properties, such as painting cells on the screen.

The PlayListManager Component

The following component is tasked with handling the contents of the playlist table. It also has four buttons that deliver the following duties:

- ★ Adding a track to the playlist table via loading it from a directory.
- ★ Playing a selected track in the playlist in either of the available audio players.
- ★ Removing all tracks from the playlist.

Beyond that, the PlayListManager also:

- ★ Determines how buttons are displayed depending on their state of activity.
- ★ Populates the tracks by writing them into a txt file.
- ★ Paints the buttons and backgrounds onto the screen

The TrackFile class

This class deals with tasks such as the retrieval of track file specific properties like the file name. Moreover, the TrackFile also contains the following information about a track:

- ★ The file length, url and name in order to use them for the playlist.
- ★ Specific file properties such as the entire file path.

The AudioMetaData class

The AudioMetaData class supervises the retrieval of audio-related mediadata such as the length of the file. To clarify this further, this means that this class can:

- ★ Get the audio track length by using a pointer to the DJAudioPlayer component in order to call the relevant function.
- ★ Format the audio track into the desired string representation.

The SearchField component

The SearchField component displays the search field and deals with how the search result is depicted on the table. It also:

- ★ Retrieves the result of a search and then displays it on the table by selecting the relevant row.
- ★ Sets up general search field related properties such as the font size.
- ★ Paints the custom background image onto the screen.

Low Level Description of the Features

Populating a Tracks Text File

The creation of the txt file and / or writing of a single track into is achieved as shown in the function below.

```
void MusicControlDeck::populateTracksFile()
{
    std::ofstream fileList;

    // append the track list into the existing playlist
    fileList.open("tracks.txt", std::fstream::app);

    juce::Array<juce::File> files = loadInTracks();

    for (File& file : files)
    {
        TrackFile trackFile{file};

        if (!checkIfTrackAlreadyLoaded(trackFile))
        {
            filesAlreadyLoaded.push_back(trackFile);
            fileList << trackFile.getTrackFileProperties().getFullPathName() << "\n";
            playList->addTrackToTracksVector(trackFile);
        }
    }

    fileList.close();
}
```

In relation to the above, the method checkIfTrackAlreadyLoaded() contains a check for whether a certain track has already been added to the tracks.txt file. The way this is done is displayed in the table below.

```
bool MusicControlDeck::checkIfTrackAlreadyLoaded(TrackFile& trackFile)
{
    bool trackAlreadyLoaded = false;

    for (TrackFile& existingTrackFile : filesAlreadyLoaded)
    {
        if (existingTrackFile.getFileName() == trackFile.getFileName())
        {
            trackAlreadyLoaded = true;
        }
    }

    return trackAlreadyLoaded;
}
```

Loading a Track File Into a Tracks Vector

While the code snippet below looks trivial, it performs the important task of adding each track file to a vector that serves as a basis for displaying, searching and saving the playlist.

```
void PlayListManager::addTrackToTracksVector(TrackFile& trackFile)
{
```

```
    File file = trackFile.getTrackFileProperties();
    juce::URL audioURL{ file };
    trackFile.setFileLength(audioMetaData->getAudioTrackLength(audioURL));
    trackList.push_back(trackFile);
}
```

Searching the Playlist for a Track

Searching the playlist for a track is achieved by looping through the tracks list and assigning any filename that contains the inputted text to a variable called matchingTrackTitleId. This variable is used to highlight the given result in the playlist table.

```
void PlayListManager::searchThePlaylist(juce::String& inputtedText)
{
    int matchingTrackTitleId = 0;

    for (unsigned int i = 0; i < trackList.size(); ++i)
    {
        if (trackList[i].getFileName().contains(inputtedText))
        {
            matchingTrackTitleId = i;
        }

        playListTable.selectRow(matchingTrackTitleId);
    }
}
```

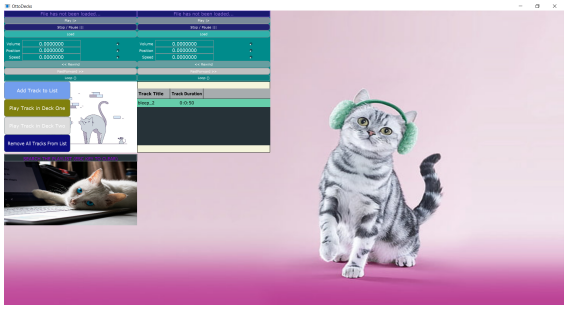
The SearchField component calls upon the method above by relying on a pointer to the PlayListManager component as showcased below.

```
void SearchField::changeTextWhenSearching()
{
    searchField.onTextChange = [this] {playListManager->searchThePlaylist(searchField.getText());};
}
```

Custom GUI

This section deals with the custom GUI that was created for the final project.

High Level Description of the Features



The Otto Deck layout consists of the following building blocks, which each contribute to the whole look of the application:

★ The two control decks, positioned next to each other.
★ The MusicControlDeck and the playlist table, which are underneath the two decks.
★ A search field that is displayed underneath the MusicControlDeck.
★ A custom background that displays a cat with headphones.
★ A custom background incorporated within the MusicControlDeck that shows an angry cat.
★ A custom background displayed within the search field that shows a white cat and a laptop.

Low Level Description of the Features

The section below provides some code snippets of a few of the features that were used to produce the custom GUI.

Custom Backgrounds

The implementation of the background images was achieved by creating a cache of the relevant image in the MainDJAudioManager.cpp constructor and then referring to it in the paint method. The table below showcases how this was accomplished.

```
// add background image
    // credit @ https://unsplash.com/@markusspiske
    backgroundImage = ImageCache::getFromMemory(BinaryData::background_png,
BinaryData::background_pngSize);

void MainDJAudioManager::paint (juce::Graphics& graphics)
{
    graphics.drawImage(backgroundImage, getLocalBounds().toFloat());
}
```

Customising Each Component

In terms of customising each specific component, the paint and resized functions were utilised as shown in the example for the MusicControlDeck below.

```
void MusicControlDeck::paint(juce::Graphics& graphics)
{
    graphics.fillAll(juce::Colours::whitesmoke);
    graphics.drawImage(backgroundImage, getLocalBounds().toFloat());

    // A call to the buttons and sliders painting functions
    repaintButtons();
}

void MusicControlDeck::resized()
{
    double rowH = getHeight() / 4.0;

    // play, stop and load button positions
    loadTrack.setBounds(0, 0, getWidth() / 2.0, rowH * 1.0);
    loadToDeckOne.setBounds(0, rowH * 1.0, getWidth() / 2.0, rowH * 1.0);
    loadToDeckTwo.setBounds(0, rowH * 2.0, getWidth() / 2.0, rowH * 1.0);
    removeTrack.setBounds(0, rowH * 3.0, getWidth() / 2.0, rowH * 1.0);
}
```

The resized method was of particular importance for the ControlDeck because of the positioning of the label buttons that had to be aligned in the same area as the rotary sliders. Therefore, as can be seen below, the x coordinate of location had to be moved by 50 centimetres in order to create space for the label buttons.

```
void ControlDeck::resized()
{
    double rowH = getHeight() / 10.5;
```

```
    // Waveform Display
    waveformDisplay.setBounds(0, 0, getWidth(), rowH * 1);

    // Play, Stop and Load button positions
    playButton.setBounds(0, rowH * 1.0, getWidth(), rowH * 1.0);
    stopButton.setBounds(0, rowH * 2.0, getWidth(), rowH * 1.0);
    loadButton.setBounds(0, rowH * 3.0, getWidth(), rowH * 1.0);

    // Slider Positions
    volumeSlider.setBounds(50, rowH * 4.2, getWidth(), rowH * 1.5);
    positionSlider.setBounds(50, rowH * 5.2, getWidth(), rowH * 1.5);
    speedSlider.setBounds(50, rowH * 6.2, getWidth(), rowH * 1.5);

    // Rewind, Fastforward and Loop buttons
    rewindButton.setBounds(0, rowH * 7.5, getWidth(), rowH * 1.0);
    fastForwardButton.setBounds(0, rowH * 8.5, getWidth(), rowH * 1.0);
    loopButton.setBounds(0, rowH * 9.5, getWidth(), rowH * 1.0);
}
```

Bringing It Altogether

The biggest challenge in terms of the GUI was getting all the custom components to be displayed in tandem. While the layout is not the most user-friendly as its buttons are small, it is a labour of love, with a lot of thought of having gone into how each part would fit into the whole picture.

```
void MainDJAudioManager::resized()
{
    /**
    *  DECK 1 | DECK 2
    | MUSIC CONTROL DECK | PLAYLIST |
    | SEARCH FIELD
    **/

    // DECKS
    deck1.setBounds(0, 0, getWidth() / 4.2, getHeight() / 4.2);
    deck2.setBounds(getWidth() / 4.2, 0, getWidth() / 4.2, getHeight() / 4.2);

    // MUSIC CONTROL DECK
    musicControlDeck.setBounds(0, getHeight() / 4.2, getWidth() / 4.2, getHeight() / 4.2);

    // PLAYLIST
    playListComponent.setBounds(getWidth() / 4.2, getHeight() / 4.2, getWidth() / 4.2, getHeight() / 4.2);

    // SEARCH FIELD
    searchField.setBounds(0, getHeight() - 420, getWidth() / 4.2, getHeight() / 4.2);
}
```