University of London
Object Oriented Programming CM2005
Prof. Matthew Yee-King

Name:
Flores, Kriss
Student Number:
190126209

OTODECKS

- **R1: 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.**

  I created a new component called LooperComponent. There is a header file LooperComponent.h and an implementation file LooperComponent.cpp. This new component is a child of the DeckGUI and it is located under the WaveformDisplayComponent. It holds a ToggleButton and a Label component. See Figure 1 for reference.

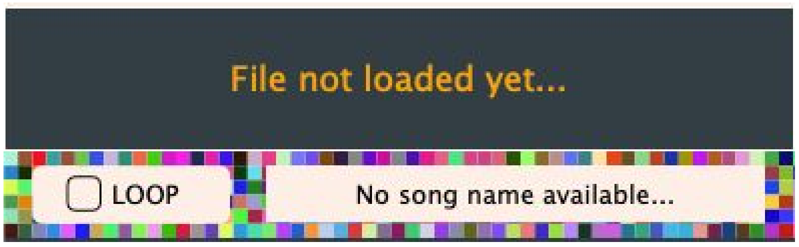


Figure 1. Looper Component (no song loaded)

  - **R1A Component has custom graphics implemented in a paint function:** As you can see in the LooperComponent.cpp file (lines 31-65) custom graphics were implemented for this component. The background of the component is filled with small rectangles of random colors, these colors change when a new song is uploaded into the DeckGUI or when the MainComponent is resized. Also a custom rounded rectangle was drawn behind the LOOP toggle button so that the tick and the letters in the button were clearly visible to the user.

  - **R1B Component enables the user to control the playback of a deck somehow:** When the user clicks the Toggle button inside the LooperComponent, the looping state of the DJAudioPlayer belonging to that deck changes: if the button is active the song will play indefinitely, if not, the song will stop playing as soon as there are no more buffers to send and the position of the DJAudioPlayer will be set to the start of the song. This can be achieved thanks to the *AudioFormatReaderSource::setLooping* function. Also, when a song is uploaded into the Deck, the name of the song will be displayed in the Label component located at the right side of the LooperComponent, as can be seen in Figure 2.

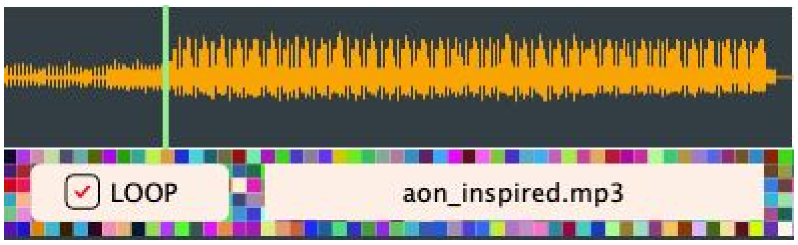


Figure 2. Looper Component (song loaded)

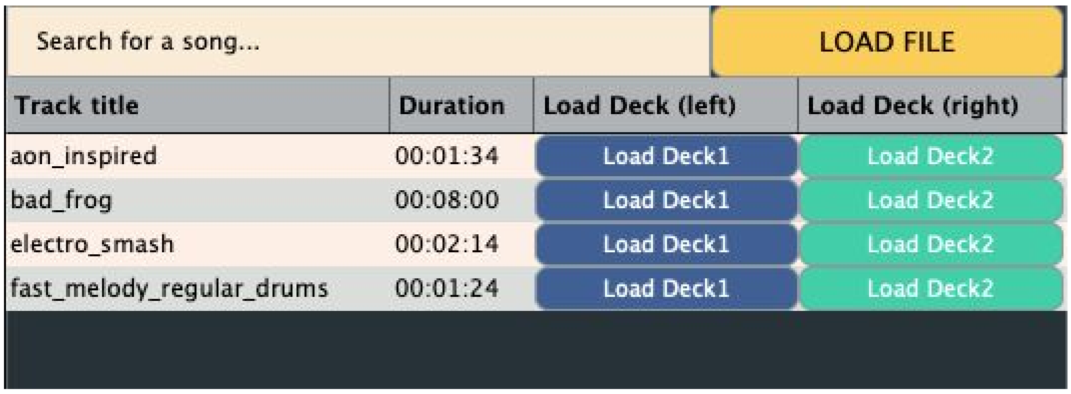
This is possible because when an instance of the LooperComponent is created inside the constructor of the DeckGUI, the DJAudioPlayer pointer passed as a parameter to the DeckGUI is also used as a parameter to initialize the LooperComponent, in this way the DeckGUI acts as an intermediary between these two different components.

When a new song is uploaded into a DeckGUI, either via the Load button inside the DeckGUI itself or by clicking one of the Load buttons in the PlaylistComponent, the *loadFile(URL songURL)* function of the DeckGUI is called. Here the public functions *updateSongName()* and *isLooping()* of the LooperComponent are used to update the song name displayed in the LooperComponent and also to check the current state of the toggle button "LOOP". If the button is active, the song is immediately set in a looping state. Please see lines 174-183 of the DeckGUI.cpp file, lines 92-100 of the LooperComponent.cpp file and lines 128-133 of the DJAudioPlayer.cpp file for more information on this section.

- **R2: Implementation of a music library component which allows the user to manage their music library.**

During the lectures we created the PlaylistComponent, this component was supposed to hold a playlist (a list of songs), however, at the end of the lectures this list just stored hardcoded names of songs. During this project new features were added to the PlaylistComponent to be able to comply with the requirements.

- **R2A Component allows the user to add files to their library:** This requirement was achieved. The PlaylistComponent now has a TextButton called loadFileButton (which displays “LOAD FILE” in the app) located at the top right of the component, which allows the user to add files into the library (see Figure 3).



Search for a song...		LOAD FILE	
Track title	Duration	Load Deck (left)	Load Deck (right)
aon_inspired	00:01:34	Load Deck1	Load Deck2
bad_frog	00:08:00	Load Deck1	Load Deck2
electro_smash	00:02:14	Load Deck1	Load Deck2
fast_melody_regular_drums	00:01:24	Load Deck1	Load Deck2

Figure 3. PlaylistComponent (loadFileButton is the yellow button at the top right corner of the component)

When the user clicks on a button inside the PlaylistComponent, the function *PlaylistComponent : : buttonClicked()* is called, here if the button clicked was the loadFileButton then a file chooser is opened, much like we did during the lectures with the Load button in the DeckGUI (see lines 138-145 of PlaylistComponent.cpp). When the user chooses a file, the private function *PlaylistComponent : : existInTrackList(fileName)* is called to check if a file with the same name is already included in the playlist. If such a file does not exist, then a new row is added to the playlist for this new song.

To keep track of the songs added to the playlist the PlaylistComponent has three private attributes, these are vectors named trackURLs, trackLengths and trackTitles. When a new song is going to be “added” to the playlist what the program does is:

- Append the name of the selected file into the *trackTiles* vector.
- Append the URL of the selected file into the trackURLs vector.
- Append the information of the duration of the song into the trackLengths vector.

These three vectors are the ones that actually store the state of the playlist and will later be used to save the playlist for future use. Please refer to lines

145-160 of the PlaylistComponet.cpp file for detailed information for this section.

Note: The user is still able to load a song into one of the Decks with the LOAD button (the yellow button located under the PLAY and PAUSE buttons in the DeckGUI as can be seen in Figure 5 at the end of this document) as it was done during the lectures. Nonetheless, when a song is added into the decks in this way the song is not added to the playlist so its name will now be shown in the PlaylistComponent.

- **R2B Component parses and displays meta data such as filename and song length:** This requirement was achieved as can be seen in Figure 3. When a new song is going to be added to the playlist (R2A requirement) the name of the file is extracted from the result of the choosing process (lines 143-144 in PlaylistComponent.cpp) and this name is added to the trackTitles vector which in turn is used by the PlayListComponent : : paintCell() function to display the names of each song in the playlist.

  To be able to get the song's duration it was necessary to create an AudioTransportSource instance when the row was going to be added to the playlist. This was necessary because a transportSource is needed to be able to use the *AudioTransportSource : : getLenghtInSeconds()* function. Once we obtain the duration of the song in seconds, the private function *PlayListComponent : : getTrackDuration(seconds)* is used to parse the length in seconds and transform it into a string of the form " hour: minutes: seconds" which then would be stored in the *trackLenghts* vector. Please see lines 153-159 and 285-308 of the PlaylistComponent.cpp file for detailed information on how the parsing of the song's duration is done within the app.

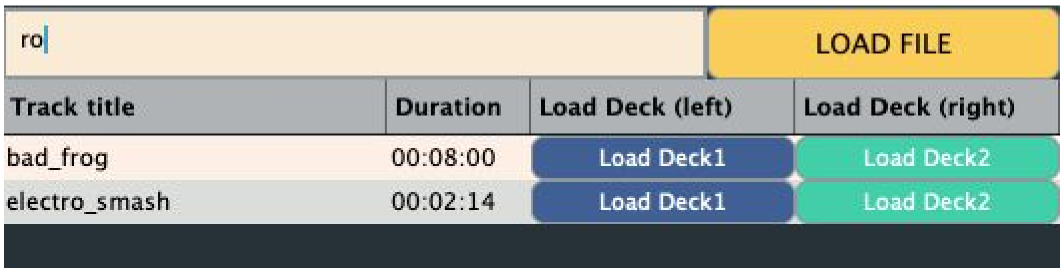
- **R2C Component allows the user to search for files:** This requirement was achieved. The PlaylistComponent now has a private TextEditor attribute called searchBar and it is also a TextEditor: : Listener. When the user writes or changes the text in the searchBar the function *PlaylistComponent : : textEditorTextChanged* is triggered; inside it the function *PlaylistComponent :* :searchMatchingIndexes(textToMatch) is called with the current contents of the searchBar.

  Since it was not clearly established in the requirements how the searching functionality should work within the app, I implemented a substring matching, this means that any song in the playlist whose name includes the content of the searchBar as a substring will be matched. This functionality is given by the *PlaylistComponent : : searchMatchingIndexes(strToMatch)* function.

The PlaylistComponent has another private member called *matchedIndexes*, this is a vector that stores the indexes of the songs whose names contain the string to match as a substring. The *PlaylistComponent : : searchMatchingIndexes* function clears the previous contents of the matchedIndexes vector and loops through the trackTitles vector and for each match it finds, a new index is added to the matchedIndexes vector and the *updateContent()* function is called on the tableComponent after all matching indexes are found(see lines 314-325 of PlaylistComponent.cpp file).
The matchedIndexes vector is important for two reasons. First, the PlaylistComponent::getNumRows() function returns the size of the *matchedIndexes* vector instead of the *trackLengths* vector. The PlaylistComponent always shows a list of songs that are a match; when there is nothing written in the searchBar the string to be matched is the empty string "", which matches with all the songs names in the playlist so all the contents are shown in this case.

Second, when the *PlaylistComponent : : paintCell* function is called (as many times as rows in the *matchedIndexes* vector) it doesn't use the *rowNumber* parameter directly to find the names and duration of the songs matched, instead this parameter is used to get the content of the *matchedIndexes* vector in that position, the index obtained in this way is then used to get the value inside trackTitles and trackLengths vectors respectively, this is how the app is able to show only the rows in the table that are a match. See lines 100-108 of the *PlaylistComponent.cpp* file for more detailed information of this process.

As an example, if we search for the string "ro" in the playlist displayed in Figure 3, only two songs are matched as can be seen on Figure 4.



ro

LOAD FILE

Track title	Duration	Load Deck (left)	Load Deck (right)
bad_frog	00:08:00	Load Deck1	Load Deck2
electro_smash	00:02:14	Load Deck1	Load Deck2

Figure 4. Matched songs in playlist for string “ro”

- **R2D Component allows the user to load files from the library into a deck:** This requirement was achieved. As you can see in Figure 3, each row of the table shown in the PlaylistComponent has two buttons, the left button loads the song into Deck 1 (the left DeckGUI) and the right button loads it into Deck 2 (the right DeckGUI). These buttons are created and updated in the *PlaylistComponent : : refreshComponentForCell()* function (much like we did during the lectures, the difference being that now there is one more column and two buttons instead of one).

  When the buttons are created/updated, an ID is assigned to them, this ID is a string of the form “rowNumber-columnNumber”. It is important to notice that the row number assigned to the ID comes from the matchedIndexes vector, instead of being directly the rowNumber parameter passed to the *refreshComponentForCell* function, this is because the ID of the button must be updated each time a new search/matching is made and these buttons should always be associated with the right song in the playlist, please see lines 114-131 of the PlaylistComponent.cpp file for detailed information about this section.

  For each button created this way, the PlaylistComponent is added as a listener, that is why when the user clicks on them the function *PlaylistComponent::buttonClicked(Button* button)* is triggered. Since these buttons are different from the Load button, the *else* case inside the *buttonClicked* function runs; here is where the ID given to each button shows its purpose: the column section informs which of the 2 buttons on the row was clicked (the left one or the right one) and this way we know to which DeckGUI we should send the URL of the song, if the column was 3 then the song is sent to DeckGUI 1 and if the column is 4 then it is sent to DeckGUI 2.

  The row number stored in the button’s ID will tell us which URL inside the *trackURLs* vector should be sent to the DeckGUI. At this point it is important to notice that the *PlaylistComponent.h* file contains the definition of a *Listener* (see lines 60-65) and also that the MainComponent was made a PlaylistComponent::Listener. This listener should implement one function:

void loadButtonClicked (URL trackURL, int deckNumber)

Continuing with the *PlaylistComponent : : buttonClicked()* function, once we obtained the *row* and *column* information, a callback function is called for every listener of this component (in this case only the MainComponent is a listener of the PlaylistComponent). This callback is the *loadButtonClicked()* function mentioned above (see lines 168-177 of the PlaylistComponent.cpp file).

The parameters passed to the *loadButtonClicked* function are the URL of the song located at position "*row*" in the trackURL vector and the column number of the button that was clicked. If the column number was 3 then the DeckGUI : : loadFile(trackURL) function is called on the DeckGUI located to the left of the app, otherwise, it is called on the DeckGUI to the right of the MainComponent (See lines 102 - 110 of the MainComponent.cpp file to get information about the *loadButtonClicked* function). In this way we see how the MainComponent acts as the bridge between the DeckGUIs and the PlaylistComponent since it is the only one who has access to both of them; this is the way by which we can load a song from the *playlist* into one of the *DeckGUIs*.

It must be noted that the user can still load songs into each Deck by using the LOAD buttons available in the DeckGUI, however, loading a song in this way does not create a new row in the playlist so this song will not be saved when the application closes and thus will not be available when the application restarts.

- **R2E The music library persists so that it is restored when the user exits then restarts the application:** When the application starts a new instance of the MainComponent is created, inside its constructor function a file named "OtoDecks_saved_Playlist.xml" is created inside the directory where the application is stored. If there is already a file with this name in this directory a new empty file is not created (please see documentation for the File::create() function on the JUCE library and lines 17-20 of the MainComponent.cpp file).

When the application is started for the first time (or if the file OtoDecks_saved_Playlist.xml is deleted from the directory), then a new file with this name is created, this file starts empty. The users can then add as many songs to the playlist as they wish using the LOAD button in the PlaylistComponent. When the user decides to close the application, the destructor function of the MainComponent is called and inside it the PlaylistComponent : : saveState(filePath) function is triggered (see lines 50 - 55 of the MainComponent.cpp file).

You must remember from previous sections of this report that the PlaylistComponenent has three private members that hold the information required to build each row in the table, these are trackTitles, trackLengths and trackURLs. The *PlaylistComponent : : saveState(filePath)* function does the following with these vectors:

- Creates a *FileOutputStream* object using as input the *filePath* parameter, this opens the file located in the given path.
- Clear the contents of the stream object (if it was not empty)
- Creates a *ValueTree* object with its respective *Identifier*.
- Creates three StringArray objects that hold the same information as the *trackTitles*, *trackLengths* and *trackURLs* vectors.
- Sets three properties for the ValueTree object using the StringArray objects previously generated.
- Writes the contents of the ValueTree into the stream.

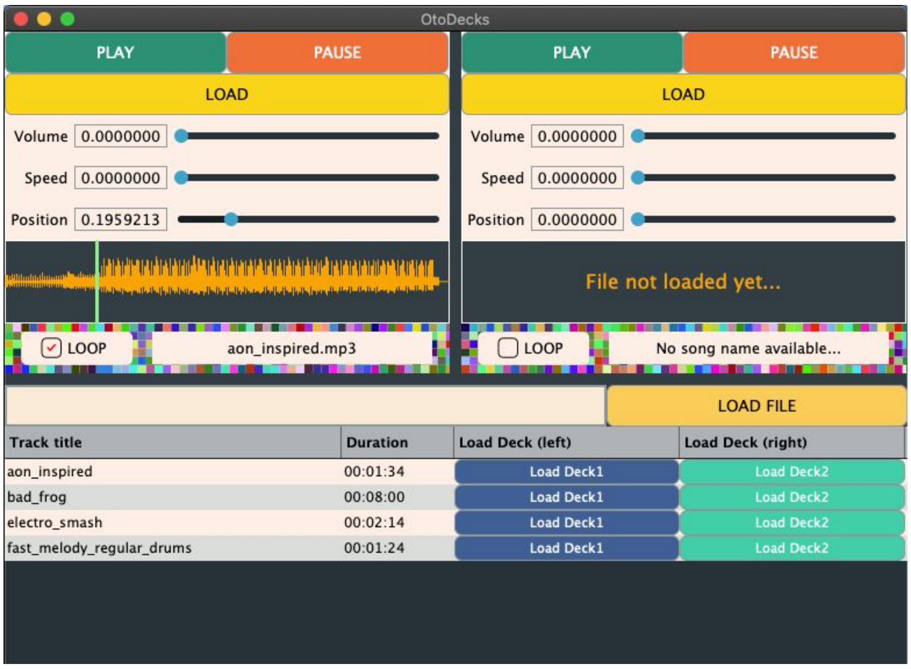
All these steps effectively open the file that was created in the constructor of the MainComponent and writes three properties of a ValueTree object into it. This is the way in which we can save the current state of the playlist. See lines 239-271 of the PlaylistComponent.cpp file for more information on how this application saves the state of the playlist.

Now that the file “OtoDecks_saved_Playlist.xml” holds some information, when the application is started again the previous state of the playlist can be loaded. This happens inside the constructor of the MainComponent on line 23, here the ***PlaylistComponent : : loadState(playListFile)*** function is called passing the “OtoDecks_saved_Playlist.xml” File object as a parameter.

Inside the *loadState()* function is implemented a process that is the reverse of that implemented in the *saveState()* function:

- Creates a *FileOutputStream* object using as input the *filePath* parameter, this opens the file located in the given path.
- Creates a ValueTree object and fills the ValueTree with the information from the stream using the ValueTree::readFromStream(stream) function.
- Read the properties of the ValueTree and obtain an Array<var>* for each of them.
- Loop through the Arrays obtained in the previous step and get the values stored in them. These values must be pushed back into the (now empty) trackTitles, trackLengths and trackURLs vectors.
- To be able to show the loaded contents, the function PlaylistComponent : : searchMatchingIndexes(strToMatch) must be called with the empty string "" as a parameter, this is because all the songs in the playlist will be a match for the empty string as was previously explained in this report.

This is the way by which this application is able to load the contents of a previously saved playlist. Please see lines 207-231 of the PlaylistComponent.cpp file for the complete description of the *loadState()* function. See Figure 5 for a full picture of the OtoDecks application developed for this project.



Volume 0.0000000
Speed 0.0000000
Position 0.1959213
☑ LOOP aon_inspired.mp3

Volume 0.0000000
Speed 0.0000000
Position 0.0000000
☐ LOOP No song name available...

Track title	Duration	Load Deck (left)	Load Deck (right)
aon_inspired	00:01:34	Load Deck1	Load Deck2
bad_frog	00:08:00	Load Deck1	Load Deck2
electro_smash	00:02:14	Load Deck1	Load Deck2
fast_melody_regular_drums	00:01:24	Load Deck1	Load Deck2

Figure 5. OtoDecks application showing all internal components