CM2005: Object-Oriented Programming

End-of-Term Assignment

Arjun Muralidharan
University of London

19th September 2020

Contents

List of Figures

List of Listings

1 Introduction

This report documents how the OtoDecks application was extended to fulfil the requirements for the end-of-term assignment. The application uses the starter code from Week 19 in the module.

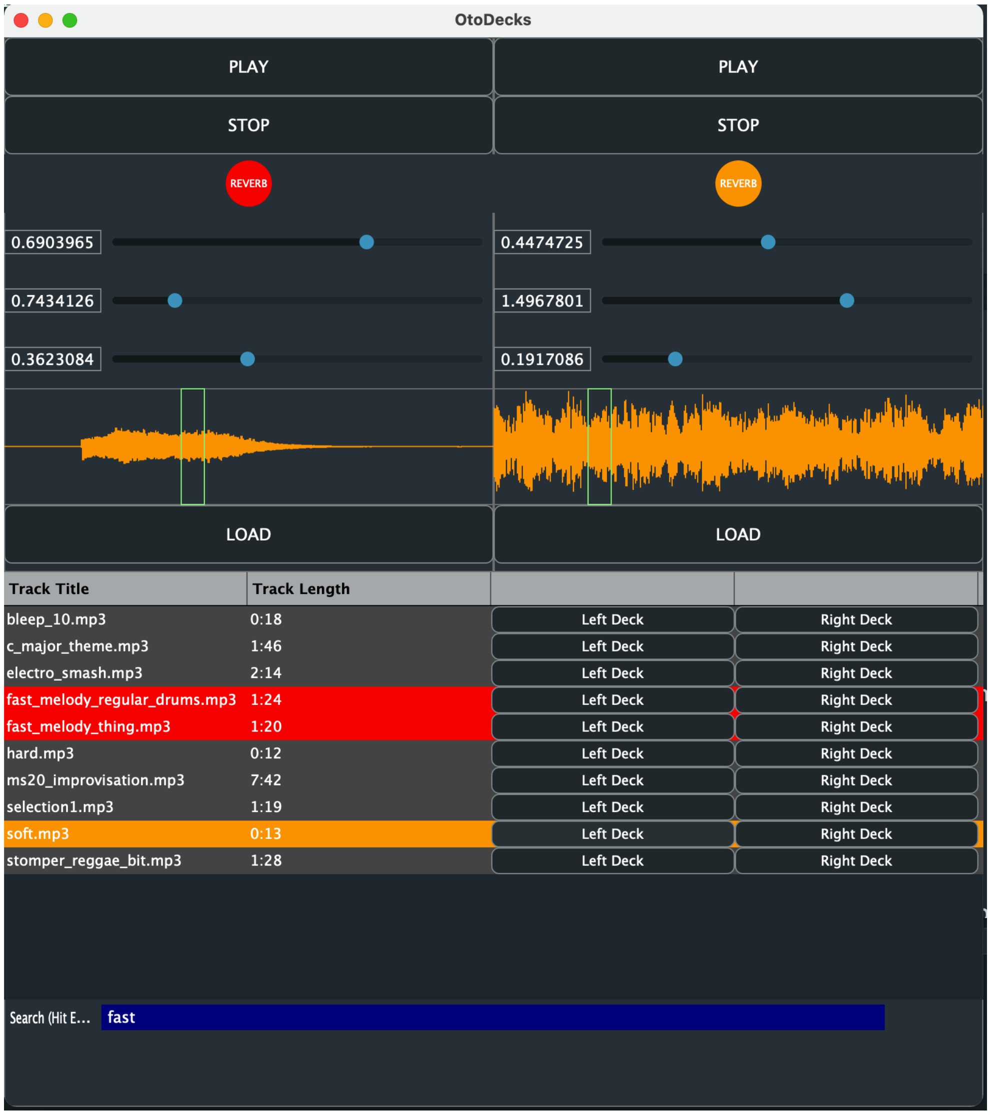


Figure 1. *Completed OtoDecks Application*

2 R1: Custom Deck Control

The custom deck control implemented in this application is a button to toggle a reverb effect. It uses a built-in JUCE reverb class. In order to fulfill the requirement of custom graphics, a separate custom JUCE component was create.

2.1 R1A: Custom Graphics

Custom graphics were achieved by creating a separate, custom `CustomDeckControl` class which has its own `paint()` function. As the class inherits from `TextButton`, the `paint()` function was replaced by a `paintButton()` function as recommended by the JUCE documentation. This allowed reacting to user interaction with the button.

The graphical aspect was achieved by drawing an ellipse with dedicated colour states for mouse hovering and mouse clicks. A toggle capability was implemented by introducing a public data member on the class, `toggle`, storing the state of the toggle as a boolean value. If the component is clicked, the toggle value is flipped. The actual toggling happens in the `DeckGUI` class, which has the listener for this custom component.

Listing 1 Toggling the graphical state in DeckGUI

```
// Toggle the reverb audio effect
if (button == &reverbButton)
{
        player->toggleReverb();
        reverbButton.toggle = !reverbButton.toggle;
}
```

The final result is a toggle button that was placed right below the play and stop buttons.

Figure 2. *Custom Deck Control with Custom Graphics*

2.2 R1B: Playback Control

To actually apply a reverb effect, the function `toggleReverb` in the `DJAudioPlayer` class was created. It is called as shown in Listing 1 and the function itself toggles a private boolean data member in the `DJAudioPlayer` class. In order to handle the reverb, a separate `AudioTransportSource` was needed based on JUCE's `ReverbAudioSource` class. Based on the toggle, either this transport source will be used, or the regular `resampleSource`. This logic is implemented in the `getNextAudioBlock` function of the audio player as shown in Listing 2.

Listing 2 Switching the audio source for reverb

```
void DJAudioPlayer::getNextAudioBlock(const AudioSourceChannelInfo& bufferToFill)
{
        // Switch to the reverb source if reverb is active
        if (reverb == true)
        {
                reverbSource.getNextAudioBlock(bufferToFill);
        }
        // Otherwise use the regular resample source
        else
        {
                resampleSource.getNextAudioBlock(bufferToFill);
        }
}
```

3 R2: Music Library

3.1 R2A: Add Files

Adding files to the music library has been implemented using file drag and drop capabilities in JUCE. All of this functionality was implemented in the `PlaylistComponent` class. All tracks are stored by file path in the `trackTitles` array, as shown in Listing 3. The tracks are added using the `paintCell` function to the first columns of the table constructed in the constructor function of `PlaylistComponent`.

Listing 3 Adding files to the application

```
void PlaylistComponent::filesDropped(const StringArray &files, int x, int y)
{
        // Iterate over all dropped files and store the track titles and lengths
        ↪  into vectors
        for (String file : files)
        {
                trackTitles.push_back(file);
        }

        // Refresh the table to show the newly added files
        tableComponent.updateContent();
}

/************************/

void PlaylistComponent::paintCell(Graphics &g,
                                        int rowNumber,
                                        int columnId,
                                        int width,
                                        int height,
                                        bool rowIsSelected)
{

        // Insert the track titles into the first column of the library
        if (columnId == 1)
        {
                g.drawText(File{trackTitles[rowNumber]}.getFileName(),
                                   2, 0,
                                   width - 4, height,
                                   Justification::centredLeft,
                                   true);
        }

}
```

3.2 R2B: Meta Data

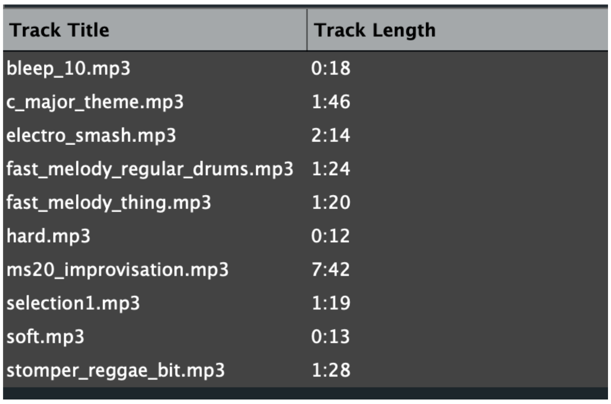
As shown in Listing 3, only the track file names (without the full path) are shown to the user. We are parsing this information by running the path names through the `getFileName()` function. Additionally, we are calculating the time length of specific track in the `getLengthOfTrack` function that was custom implemented. This function receives a file name and returns the length as a string in "MM:SS" format. This procedure is shown in Listing 4.

Listing 4 Calculating track lengths

```cpp
String PlaylistComponent::getLengthOfTrack(String file)
{
        auto duration{0};

        // Calculate the duration of the track by using an AudioFormatReader on the audio file
        std::unique_ptr<juce::AudioFormatReader> reader(formatManager.createReaderFor(File{file}));
        if (reader.get() != nullptr)
        {
                // Formula to calculate length based on sample size and sample rate
                duration = (float)reader->lengthInSamples / reader->sampleRate;
        }

        // Calculate minutes and seconds
        String minutes = std::to_string(duration / 60);
        String seconds = std::to_string(duration % 60);

        // If the track doesn't have any seconds beyond the full minute, we need to append another 0
        // Otherwise tracks that are e.g. 5 minutes long show up as "5:0" instead of "5:00".
        if (seconds == "0")
        {
                seconds = "00";
        }

        // Return the combined track length in MM:SS format
        return minutes + ":" + seconds;
}
```

These track lengths are retrieved at the moment when files are dropped into the library and stored in a vector, similar to the track titles. They are then drawn to a second column in the table. This is identical handling to the track titles; therefore additional code samples are omitted.

The library with tracks and metadata is shown in Figure 3.



Track Title	Track Length
bleep_10.mp3	0:18
c_major_theme.mp3	1:46
electro_smash.mp3	2:14
fast_melody_regular_drums.mp3	1:24
fast_melody_thing.mp3	1:20
hard.mp3	0:12
ms20_improvisation.mp3	7:42
selection1.mp3	1:19
soft.mp3	0:13
stomper_reggae_bit.mp3	1:28

Figure 3. *Library with tracks titles and lengths*

Listing 5 Initialising the search box

```
    searchTitle.setText("Search (Hit Enter)", juce::dontSendNotification);
    searchTitle.attachToComponent(&searchInput, true);
    searchTitle.setJustificationType(juce::Justification::verticallyCentred);
    searchInput.setEditable(true);
    searchInput.setColour(juce::Label::backgroundColourId,
    ↪  juce::Colours::darkblue);

    // Lambda function to handle text inputs into the search box
    searchInput.onTextChange = [this] {
    ↪  searchLibraryFor(searchInput.getText()); };
```

3.3 R2C: Search

Search functionality was implemented by adding a `Label` to enter text at the bottom of the player. Graphically, this was solved in the `PlaylistComponent` class with two labels, one for the title of the search box, and another for the search box itself. These are initialised in the constructor function as shown in Listing 5 and drawn with the usual `paint()` and `resize()` functions. A lambda function is attached to listen for changes to the search box contents.

The actual search functionality is triggered in the `searchLibraryFor` function by traversing the vector of track titles and comparing the search term to those. JUCE provides a variety of classes to make this string search easier out of the box, as shown in Listing 6. The results are pushed to a new vector by flagging each vector position with a `true` or `false` value.

This approach is used to highlight the results when drawing the library table. Each row's ID is compared to the results vector. If it is true, then the row is painted in a different color. The result is that matching search results are shown with a red background once the users hits enter after entering the search term, as shown in Figure 4.

Listing 6 Running a search

```
        void PlaylistComponent::searchLibraryFor(const String &searchTerm)
{
        searchResults.clear();

        // User needs to type at least two characters for the search to execute.
        // The search works by looking for a string in each element of the track titles vector.
        // We maintain a vector "searchResults" that is of equal length as track titles. If the track title matches our
        // search term, we mark our result vector as "true" in the same index position as the track title.
        // Later, we can check each track title against the results vector to decide how to show the user that a specific
        // track was a match (or not)
        if (searchTerm.length() > 1)
        {
                // Iterate over the track titles and look for the search term in each
                for (int i = 0; i < trackTitles.size(); ++i)
                {
                        if (trackTitles[i].contains(searchTerm))
                        {
                                // Hit - mark this track as a result
                                searchResults.push_back(true);
                        }
                        else
                        {
                                // Miss - mark this track as not a result
                                searchResults.push_back(false);
                        }
                }
        }
```

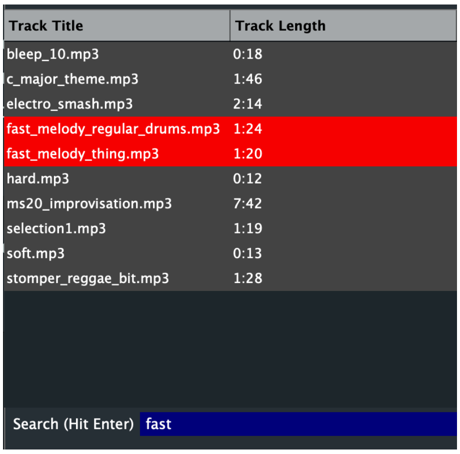


Figure 4. *Search results highlighted*

3.4 R2D: Deck Playback

The library was given a second column with additional buttons to address the left and right decks. In order to access the decks, the constructor of the library had to be extended to receive pointers to both decks, which are stored as private members of the `PlaylistComponent`. This allows us to then push a track to the respective deck on a button event as shown in Listing 7. An additional `loadURL()` function was provided in the `DeckGUI` class, which in turn address the respective `loadURL()` interfaces of the `WaveformDisplay` and `DJAudioPlayer` classes. This retains proper abstraction from the music library to the underlying components of the deck. The left and right decks are addressed respectively by identifying the buttons with a component name of "Left" and "Right", respectively. This allows us to retain the component ID for both buttons on a single row, but still differentiate between the left and right button to know which deck to target.

The completed music library is shown in Figure 5.

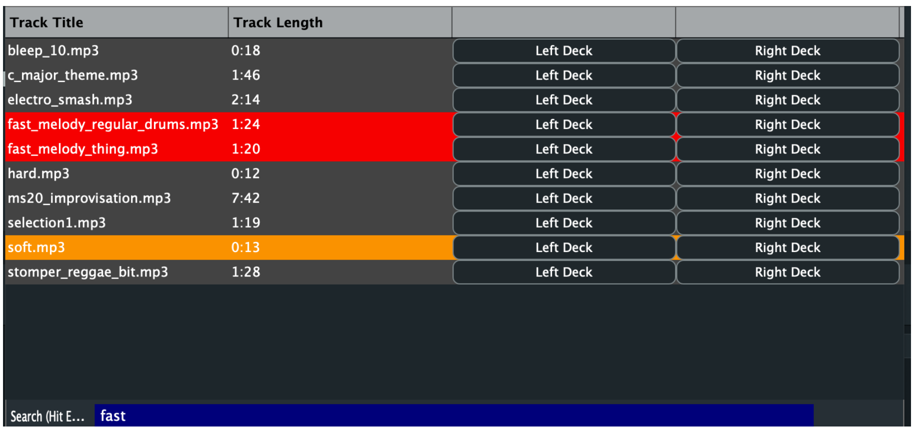


Figure 5. *Finished music library*

Listing 7 Pushing tracks to the decks

```
void PlaylistComponent::buttonClicked(Button *button)
{
        // Convert the buttons component ID to an integer, so we can use it as the
        ↪  index of our track titles array
        int id = std::stoi(button->getComponentID().toStdString());

        // If the left button was clicked, push the track to the left deck
        if (button->getName() == "Left")
        {
                deck1->loadURL(URL{File{trackTitles[id]}});
        }

        // If the right button was clicked, push the track to the left deck
        if (button->getName() == "Right")
        {
                deck2->loadURL(URL{File{trackTitles[id]}});
        }
}
```

3.5 R2E: Library Persistence

To achieve persistence of the library, the tracks are stored out to a text file as soon as they are dropped into the application. This is done by opening a `FileOutputStream` and streaming the vector of tracks to that text file. The file is stored in the user's music folder of the operating system. JUCE provides abstractions for addressing this folder in an OS-agnostic fashion.

Similarly, a `FileInputStream` is used in the constructor function of `PlaylistComponent` to read in the contents of a library file, if it already exists. JUCE provides functions to traverse a file line-by-line, and this was used to extract each line back into the track vector. Both reading and writing is shown in Listing 8.

4 Conclusion

There is vast scope to build upon the application to add better UX, more functionality and more robust handling of files. This assignment was helpful in learning about object-oriented principles and the power of a highly object-oriented framework such as JUCE, which helps adopt these principles by enforcement.

References

[1] Horton, I., & van Weert, P. (2018). *Beginning C++17: From Novice to Professional*. Berkeley, CA: Apress.

[2] Raw Material Software Limited, *JUCE Class Reference List*. Retrieved 19 September 2020 from `https://docs.juce.com/master/index.html`.

Listing 8 Reading and writing the music library

```
        /****** WRITING ******/
        // Store the music library to a text file on disk so we can retrieve it again
        // Create an output stream
        FileOutputStream libStream(library);
        // Write each loaded track to a new line in the library
        for (const String &trackTitle : trackTitles)
        {
                libStream << trackTitle << newLine;
        }

        /****** READING ******/

        if (library.existsAsFile())
        {
                FileInputStream libStream(library);
        trackTitles.clear();
        while (!libStream.isExhausted())
                {
                        String currentTrack = libStream.readNextLine();
                        trackTitles.push_back(currentTrack);
                        trackLengths.push_back(getLengthOfTrack(currentTrack));
                        tableComponent.updateContent();
                }
        }

```