OOP END-TERM ASSIGNMENT

Contents

Table of Requirements

Requirements	Done?
R1	
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 track*	✓
R2	
R2A: *Component has custom graphics implemented in a paint function*	✓
R2B: *Component enables the user to control the playback of a deck somehow*	✓
R3	
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*	✓
R4	
R4A: *GUI layout is significantly different from the basic DeckGUI shown in class, with extra control*	✓
R4B: *GUI layout includes the custom Component from R2*	✓
R4C: *GUI layout includes the music library component for R3*	✓

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

R1A: can load audio files into audio players

I have removed the original load button and used a popup menu to load a file. I have kept the FileDragAndDropTarget.

If there is only 1 file selected or dropped, we call *loadURL()* for both player and waveformDisplay to load the audio file.

```
/*Allow File Drag from Desktop */
bool DeckGUI::isInterestedInFileDrag(const StringArray &files)
{
    // only allow 1 audio file to be dragged
    if (files.size() != 1) return false;

    return true;
}
/* Receive files from desktop drag-and-drop*/
void DeckGUI::filesDropped(const StringArray &files, int x, int y)
{
    // player and waveformDisplay load url
    player->loadURL(URL{File{files[0]}});
    waveformDisplay.loadURL(URL{File{files[0]}});
```

Fig 1.0 loadURL() when file drop into a deck

When the user right clicks in a deck, the *MouseDown()* will generate a popup menu that links to a file dialog for user to choose a file.

```
void DeckGUI::mouseDown(const MouseEvent &e)
{
    /* user right-click at deckGUI and not within waveformDisplay */
    if (e.mods.isRightButtonDown() && Desktop::getInstance().findComponentAt(e.getMouseDownScreenPosition()) != &waveformDisplay)
    {
        PopupMenu mainMenu;
        mainMenu.addItem(1, "Load File from Directory");
        mainMenu.showMenuAsync(PopupMenu::Options(),
            [&](int result)
            {
                if (result == 1)
                {
                    // send to deck 1
                    if (fChooser.browseForFileToOpen(nullptr))
                    {
                        if (fChooser.getResults().size() == 1)
                        {
                            player->loadURL(URL{File{fChooser.getResult()}});
                            waveformDisplay.loadURL(URL{File{fChooser.getResult()}});
```

Fig 1.1 loadURL() when user selects the option to load file from directory in popupMenu

R1B: can play two or more tracks

No changes are made to this aspect from the base code.

The mixerSource allows multiple players to play audio at once, provided that the players have called *prepareToPlay()* and *releaseResources()* methods before and after adding them to the mixer.

```
void MainComponent::prepareToPlay (int samplesPerBlockExpected, double sampleRate)
{
    player1.prepareToPlay(samplesPerBlockExpected, sampleRate);
    player2.prepareToPlay(samplesPerBlockExpected, sampleRate);

    mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

    mixerSource.addInputSource(&player1, false);
    mixerSource.addInputSource(&player2, false);

}
void MainComponent::getNextAudioBlock (const AudioSourceChannelInfo& bufferToFill)
{
    mixerSource.getNextAudioBlock(bufferToFill);
}

void MainComponent::releaseResources()
{
    player1.releaseResources();
    player2.releaseResources();
    mixerSource.releaseResources();
}
```

Fig 1.2 add players to mixerSource

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

Each decks have their own slider to control the volume level. The sliders can adjust the gain level of the audio between 0 to 1. By default, the volume value is 1.

```
void DeckGUI::sliderValueChanged (Slider *slider)
{
    if (slider == &volSlider)
    {
        player->setGain(slider->getValue());
    }
```

Fig 1.3 setGain of player based on value of slider in DeckGUI

```
////////////////////////////////////////////////////////////////////////////////
void DJAudioPlayer::setGain(double gain)
{
    if (gain < 0 || gain > 1.0) {
        DBG("DJAudioPlayer::setGain gain should be between 0 and 1");
    }
    else {
        transportSource.setGain(gain);
    }

}
```

Fig 1.4 setGain of transportSource in DJAudioPlayer

R1D: can speed up and slow down the track

Each decks have their own slider to control the playback speed. This is achieved by changing the *ResamplingRatio()* of the player. By default, the speed/ratio is 1.

```
void DJAudioPlayer::setSpeed(double ratio)
{
    if (ratio < 0 || ratio > 5.0) {
        DBG("DJAudioPlayer::setSpeed ratio should be between 0 and 5");
    }
    else {
        resampleSource.setResamplingRatio(ratio);
    }
}
```

Fig 1.5 setResamplingRatio to adjust playback speed in DJAudioPlayer

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.

I have made changes to how the position is set (in WaveFormDisplay), and added a loop button and cue button in DeckGUI.

R2A: Component has custom graphics implemented in a paint function

I have customized the WaveFormDisplay to show the playhead in a form of vertical line

I have also added a layer of grey shade from the beginning to the current position of the waveform to show a contrast.

Fig 1.6 contrast between audio played and not yet played.

```
// 'play' playhead Pos
g.setColour(Colour::fromRGB(255, 128, 0));
g.drawRect(position * getWidth(), 0, 1, getHeight());

// 'cue' playhead Pos
g.setColour(Colour::fromRGB(169, 156, 145));
g.drawRect(cuePos * getWidth(), 0, 1, getHeight());
g.setFont(15.0f);
g.setColour(Colour::fromRGB(243, 243, 243));
g.drawText("C", Rectangle(int(cuePos * getWidth()) + 5, 5, 10, 10), Justification::topRight, true);

// fill passed time with grey colour shade
if (position > 0)
{
  g.setColour(Colour::fromRGB(21, 21, 21));
  g.setOpacity(0.6);
  g.fillRect(0, 0, position * getWidth(), getHeight());
}

// show a playhead when user is using mouse to set pos of audio
if (isMouseMoving)
{
  g.setColour(Colour::fromRGBA(255, 128, 0,200));
  g.drawRect(mouseXPos, 0, 1, getHeight());

  g.setColour(Colour::fromRGB(243, 243, 243));
  g.setFont(15.0f);
  // display time on current MouseX position
  //shift text to the left side when we are reaching around 80% of wfd width
  if (int(mouseXPos > getWidth() - getWidth()/5))
  {
      g.drawText(currentTime, Rectangle(int(mouseXPos) - 100, getHeight() / 2 - 25, 40, 10), Justification::topLeft, true);
      g.drawText(remainingTime, Rectangle(int(mouseXPos) - 100, getHeight() / 2 - 25, 90, 10), Justification::topRight, true);
      g.drawText(timeFromPlayHead, Rectangle(int(mouseXPos) - 100, getHeight() / 2, 90, 10), Justification::topRight, true);
  }
  else {
      g.drawText(currentTime, Rectangle(int(mouseXPos) + 10, getHeight() / 2 - 25, 40, 10), Justification::topLeft, true);
      g.drawText(remainingTime, Rectangle(int(mouseXPos) + 10, getHeight() / 2 - 25, 90, 10), Justification::topRight, true);
      g.drawText(timeFromPlayHead, Rectangle(int(mouseXPos) + 10, getHeight() / 2, 60, 10), Justification::topLeft, true);
  }
```

Fig 1.7 WaveFormDisplay Paint() function to display playheads

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

These are the new functionalities implemented:

1) Click / dragging to set the audio playhead position
2) Cue function
3) Loop function

R2B (i): Click / dragging to set the audio playback position

The original feature was using the posSlider to set the position of the audio. It was clunky and inconvenient, so I have used the Mouse events to allow precise placement.

```
/* Mouse events to set Position*/
void WaveformDisplay::mouseDown(const MouseEvent &e)
{
  mouseXPos = e.getPosition().getX();
  isMouseMoving = true;
  repaint();
}

void WaveformDisplay::mouseDrag(const MouseEvent &e)
{
  // mouseDown position is beyond waveFormDisplay width
  if (e.getPosition().getX() < 0 || e.getPosition().getX() > getWidth())
  {
   return;
  }

  mouseXPos = e.getPosition().getX();

  isMouseMoving = true;
  repaint();
}

void WaveformDisplay::mouseUp(const MouseEvent &event)
{

  isMouseMoving = false;
  repaint();
}
```

Fig 1.8 WaveFormDisplay Mouse functions to set position of playhead

The isMouseMoving variable is to paint the playhead. Refer to *fig 1.7*.

The deckGUI will intercept the *mouseUp()* to run some code to set the position in player and then waveformDisplay.

```
////////////////////////////////////////////////////////////////////////////////////////////////
/* set audio Position of player and waveformdisplay */
void DeckGUI::mouseUp(const MouseEvent &e)
{
    // clicked within waveformDisplay
    if (Desktop::getInstance().findComponentAt(e.getMouseDownScreenPosition()) == &waveformDisplay && e.mods.isLeftButtonDown())
    {
        if (!player->isPlayingAudio())
        {
            playButton.setButtonText("PLAY");
        }
        // get mouseX value within waveFormDisplay
        double mouseX = e.getPosition().getX();
        double w = waveformDisplay.getWidth();

        double pos = mouseX / w;

        player->setPositionRelative(pos);
        waveformDisplay.setPositionRelative(
            player->getPositionRelative());
    }
}
```

Fig 1.9 MouseUp() function in DeckGUI to setPosition

This will be how the mouseXPos playhead looks like when user is dragging the mouse.
In this image, the mouse is at time *4* seconds and *-4* seconds behind the 'play' playhead.

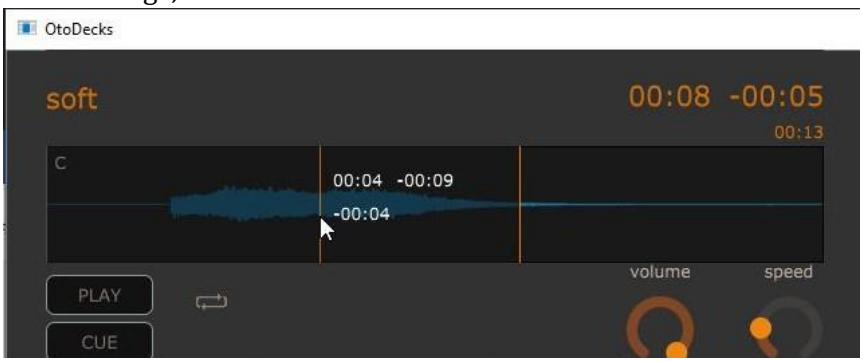

Fig 2.0 WaveFormDisplay mouseX position playhead with time

R2B (ii): Cue function

The cue button allows you to choose the point of the track that you want to start playing from with a single click.

You can simply play the track or drag until a specific point that you want to save as cue point, stop the audio and then click the cue button to save it. Now, you can click on the cue button while playing to return to the cue point.

You can also hold down on the cue button to play the track from the cue point. Once you release the button, the audio will jump back to the saved cue point.

The Cue point is a grey playhead with a 'C'

Fig 2.1 snippet of cue point in waveformDisplay GUI

In code,
The *ButtonClicked()* is called when the cue button is clicked. If audio is playing, it will stop the audio and set the position of player and waveformDisplay to the cue Position.
If the audio is already stopped, the cue position will be set to the player position instead.

The *buttonStateChanged()* will be helpful to determine whether the user held down the cue button. If the mouseDown() is true and audio is not playing, this will start playing the audio. When mouseDown is released, the audio will jump back to the cue position and player will stop.

R2B (iii): Loop function

If you want to replay the track, you can toggle the loop button.

When user clicks the loop button, the *ButtonClicked()* function will toggle the button state

Looping is achieved by checking in the *timerCallback()* whether the audioStream has finished and the loop button toggleState is true, and simply run *player->start()* again.

The white colour image means it is disabled, while the orange-coloured image means it is enabled.

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

My music library consists of 2 *tableListBoxModel* components named FolderComponent and PlaylistComponent.

FolderComponent contains rows of playlist names that will show the playlist according to the row selected.

PlaylistComponent will contains rows of tracks that can be loaded into the DeckGUIs etc.

They get the data from the class AudioPlaylists via pass-by-reference.

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

Within the PlaylistComponent, I have implemented popup menu that will display various options, including adding files to the library when the user right clicks.

The fchooser will open a dialog window that allows you to choose multiple files. The *fchooser.getResults()* will be looped to load every file into the playlistComponent.

The *cellClicked()*, *backgroundClicked()* and *MouseDown()* function each contains a popup menu.

The figure below is an example of how the popupMenu works

```
if (e.mods.isRightButtonDown())
{
    PopupMenu backgroundClickedMenu;
    backgroundClickedMenu.addItem(1, "Load File from directory");

    backgroundClickedMenu.showMenuAsync(PopupMenu::Options(),
                [&](int result)
                {
                    if (result == 1)
                    {
                        //open file dialog
                        if (fChooser.browseForMultipleFilesToOpen(nullptr))
                        {
                            for (auto i = 0; i < fChooser.getResults().size(); ++i)
                            {
                                insertNewTracks(fChooser.getResults()[i]);
                            }
```

Fig 2.2 snippet of popupMenu in PlaylistComponent BackgroundClicked()

It is also possible to drag and drop files into the PlaylistComponent.

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

The File object contains methods to retrieve metadata. I have implemented a method called *insertNewTracks()* which takes in a File and returns a custom TrackData object containing metadata of the track such as name, duration, file type and file location.

```
class TrackData
{
public:
    TrackData(std::string _playListName,
              std::string _trackTitle,
              std::string _duration,
              std::string _type,
              std::string _fileLocation
        );

    std::string playListName;
    std::string trackTitle;
    std::string duration;
    std::string type;
    std::string fileLocation;

};
```

Fig 2.3 TrackData class

```
//increment rowNumber by 1 and convert to string
std::string rowNum = std::to_string(indxV[rowNumber] + 1);
g.setColour(Colour::fromRGBA(194, 179, 165, 255));
if (rowIsSelected)
{
    g.setColour(Colours::white);
}
if (columnId == 1)
{
    // row number
    g.drawText(rowNum,
               2, 0,
               width - 4, height,
               Justification::centredLeft,
               true);
}
else if (columnId == 2)
{
    // trackTitle
    g.drawText(currentPlaylist[indxV[rowNumber]].trackTitle,
               2, 0,
               width - 4, height,
               Justification::centredLeft,
               true);
}
else if (columnId == 3)
```

Fig 2.4 snippet of paintCell() in playlistComponent

R3C: Component allows the user to search for files

The searchBar is a TextEditor in FolderComponent and we catch changes in *textEditorTextChanged()* function. The new text is saved in a variable.

A command is sent to MainComponent to get the new text using the *getSearchBarText()* and update the PlaylistComponent.

Filtering the table in PlaylistComponent is quite tricky. Rather than changing the original vector, it is more efficient to create a vector<int> to represent the index of each element in the playlist and filter this. In addition, this also helps in table sorting (see *sortOrderChanged()*).

```
// get current playlist
currentPlaylist = playlists->getPlaylist();

// fill the indx vector
indxV.resize(currentPlaylist.size());
std::iota(indxV.begin(), indxV.end(), 0);
```

Fig 2.5 filling an index Vector

```
void PlaylistComponent::setSearchBarText(std::string text)
{
    if (text.empty())
    {
        indxV.resize(currentPlaylist.size());
        std::iota(indxV.begin(), indxV.end(), 0);
        tableComponent.updateContent();
        return;
    }

    indxV.clear();

    for (auto i = 0; i < currentPlaylist.size(); ++i)
    {
        // push the index of TrackData that contains the substr of the text in the trackTitle
        if (currentPlaylist[i].trackTitle.find(text) != std::string::npos)
        {
            indxV.push_back(i);
        }
    }
    tableComponent.updateContent();
    repaint();
}
```

Fig 2.6 function to filter rows by searchBar text

d

NewPlaylist

#	Title	Duration	Type	Location
2	aon_inspired	01:34	mp3	D:\UoL\CM2005 OOP\uol_oop_...
3	bad_frog	08:00	mp3	D:\UoL\CM2005 OOP\uol_oop_...
8	fast_melody_regular_...	01:24	mp3	D:\UoL\CM2005 OOP\uol_oop_...
9	fast_melody_thing	01:20	mp3	D:\UoL\CM2005 OOP\uol_oop_...
10	hard	00:12	mp3	D:\UoL\CM2005 OOP\uol_oop_...
16	twindrive	06:50	mp3	D:\UoL\CM2005 OOP\uol_oop_...

Fig 2.7 snippet of searchBar filtering the table

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

There are three methods to load files from the PlaylistComponent into a deck

1. Popup Menu
2. Double clicking the row
3. Dragging the row into the deck

Method 1 and 2 requires calling a deckGUI function *sendURL()* which takes in the URL of the current row.

```
PopupMenu subMenu;
subMenu.addItem(1, "Deck 1");
subMenu.addItem(2, "Deck 2");

PopupMenu mainMenu;
mainMenu.addSubMenu("Load to Deck", subMenu);
mainMenu.addItem(3, "Remove from playlist");
mainMenu.addItem(4, "Load File from directory");

mainMenu.showMenuAsync(PopupMenu::Options(),
            [&](int result)
            {
                if (result == 1)
                {
                    // send to deck 1
                    deckGUI1->sendURL(getFileLocationURL(currentRowNumber));
                }
                else if (result == 2)
                {
                    // send to deck 2
                    deckGUI2->sendURL(getFileLocationURL(currentRowNumber));
                }
```

Fig 2.8 Method1 Popup menu in PlaylistComponent

```
////////////////////////////////////////////////////////////////////////////////////////////////////
void PlaylistComponent::cellDoubleClicked(int rowNumber,
                                          int columnId,
                                          const MouseEvent &event)
{
    currentRowNumber = indxV[rowNumber];
    if (event.mods.isLeftButtonDown())
    {
        URL fileURL = getFileLocationURL(currentRowNumber);
        //by default send to deck1 if it is not playing any audio
        if (!deckGUI1->isPlayingAudio())
        {
            deckGUI1->sendURL(fileURL);
        }
        //else, send to deck2 if it is not playing any audio
        else if (!deckGUI2->isPlayingAudio())
        {
            deckGUI2->sendURL(fileURL);
        }
    }
    repaint();
}
```

Fig 2.9 Method2 cellDoubleClicked() to load audio in deck 1 or deck2

```
/* Receive url from playlistComponent */
void DeckGUI::sendURL(URL url)
{
    //get full path from url
    String path = URL::removeEscapeChars(url.getDomain() + "/" + url.getSubPath());

    trackDragComponent.setCurrentAudioFileString(path);
    player->loadURL(url);
    waveformDisplay.loadURL(url);

    playButton.setToggleState(false, dontSendNotification);
    playButton.setButtonText("PLAY");

    cuePos = 0.0;
    repaint();
}
```

Fig 3.0 sendURL function in deckGUI to load URL from playlistComponent

Method 3 uses the DragAndDropContainer and DragAndDropTarget classes.

The DragAndDropContainer is the source of the dragging and DragAndDropTarget will be the receiving end. Thus, DeckGUI inherits DragAndDropTarget and PlaylistComponent inherits the other. It is also possible for a component to inherit both.

The 2 main methods of the DragAndDropTarget that I will use is the *isInterestedInDragSource()* and *itemDropped().*

```
/* Allow drag-and-drop from any Juce component except itself*/
bool DeckGUI::isInterestedInDragSource(const SourceDetails &dragSourceDetails)
{
    if (dragSourceDetails.sourceComponent->getParentComponent() == this) return false;

    if (dragSourceDetails.description.equals(""))
        return false;

    return true;
}
/* drop dragSource*/
void DeckGUI::itemDropped(const SourceDetails &dragSourceDetails)
{
    File path = File{dragSourceDetails.description.toString()};
    trackDragComponent.setCurrentAudioFileString(path.getFullPathName());

    // Load url
    player->loadURL(URL{File{dragSourceDetails.description.toString()}});
    waveformDisplay.loadURL(URL{File{dragSourceDetails.description.toString()}});

    // reset playButton state after new audio is loaded
    playButton.setToggleState(false, dontSendNotification);
    playButton.setButtonText("PLAY");

    // reset Cue position to start
    cuePos = 0.0;
    cueReady = true;
    repaint();
    DBG("DeckGUI::itemDropped");
}
```

Fig 3.1 DragAndDropTarget methods in deckGUI

The DragAndDropContainer uses the *mouseDrag()* function to initiate dragging. Make sure the MainComponent is also a DragAndDropContainer to set it as the parentComponent for access to the DeckGUI.

TableListBoxModel has a *getDragSourceDescription()* to retrieve data from rows.

```
var PlaylistComponent::getDragSourceDescription(const SparseSet<int> &currentlySelectedRows)
{
    //currentlySelectedRows will only have 1 row
    return currentPlaylist[currentlySelectedRows[0]].fileLocation;
}

void PlaylistComponent::mouseDrag(const MouseEvent &e)
{
    if (getNumRows() == 0)
        return;
    if (e.mods.isRightButtonDown())
        return;

    DragAndDropContainer *dragR =
        DragAndDropContainer::findParentDragContainerFor(this);

    if (dragR)
    {
        if (!dragR->isDragAndDropActive())
        {
            SparseSet<int> selectedRows;
            currentRowNumber = indxV[tableComponent.getSelectedRow()];

            //only add the currentRowNumber into selectedRows
            selectedRows.addRange(Range<int>(currentRowNumber, currentRowNumber + 1));
            dragR->startDragging(getDragSourceDescription(selectedRows), this, ImageCache::getFromMemory(BinaryData::audioLogo_png, BinaryData::audioLogo_pngSize));
        }
    }
    else
    {
            //  DBG("PlaylistComponent::mouseDrag(): can't find parent drag container");

    }
}
```

Fig 3.2 Implement row Dragging in deckGUI

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

Using what I have learnt from MerkelMain, I have created a CSVReader Class to handle csv file operations.

1. For reading a csv File, the *readCSV()* function is ran in the constructor of AudioPlaylists that holds the vector of TrackData.

   All the playlists are stored into a single csv file named "playlists.csv". If the file does not exist, it will return an empty playlist.

```
// set default directory for saving csv
std::string csvFile = File::getCurrentWorkingDirectory().getFullPathName().toStdString() + "\\playlists.csv";
AudioPlaylists playlists{csvFile};

PlaylistComponent playlistComponent{&deckGUI1, &deckGUI2, &playlists, formatManager};
FolderComponent folderComponent{&playlists};
```

Fig 3.3 Initializing the AudioPlaylists to read a csv file. Pass-by-reference to components

```
AudioPlaylists::AudioPlaylists(std::string _csvFile)
    : csvFile(_csvFile)
{
    playlists = CSVReader::readCSV(csvFile);
```

Fig 3.4 readCSV() in constructor

2. The method to handle writing to a csv is *insertToCSV().*

   The *insertToCSV()* function is called in the destructor of AudioPlaylists so that it will only insert the vector into the csv when the user closes the app. If the csv file does not exist, it will automatically be created and inserted into.

```
AudioPlaylists::~AudioPlaylists() {

    CSVReader::insertToCSV(csvFile, playlists);
}
```

Fig 3.5 insertToCSV() in destructor

```
void CSVReader::insertToCSV(std::string csvFilename, std::vector<TrackData> playlists)
{
    std::ofstream myfile;
    //truncate file to remove existing data
    myfile.open(csvFilename, std::ofstream::out | std::ofstream::trunc);
    for (TrackData& p : playlists)
    {
        std::string row = p.playListName + "," + p.trackTitle + "," + p.duration + "," + p.type + "," + p.fileLocation + "\n";
        myfile << row;
    }
    myfile.close();
}
```

Fig 3.6 insertToCSV() function

R4: Implementation of a complete custom GUI

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

The DeckGUI is split into 3 main rows.

The topmost row is the TrackDragComponent where it displays the audio name and time.

The middle row is reserved for the WaveFormDisplay

The bottom row is a mixture of buttons and sliders. Buttons on the left and sliders on the right

Fig 3.7 DeckGUI final custom layout

R4B: GUI layout includes the custom Component from R2

Refer to *Fig 3.7* which also showed the cue Button and loop Button.

R4C: GUI layout includes the music library component for R3

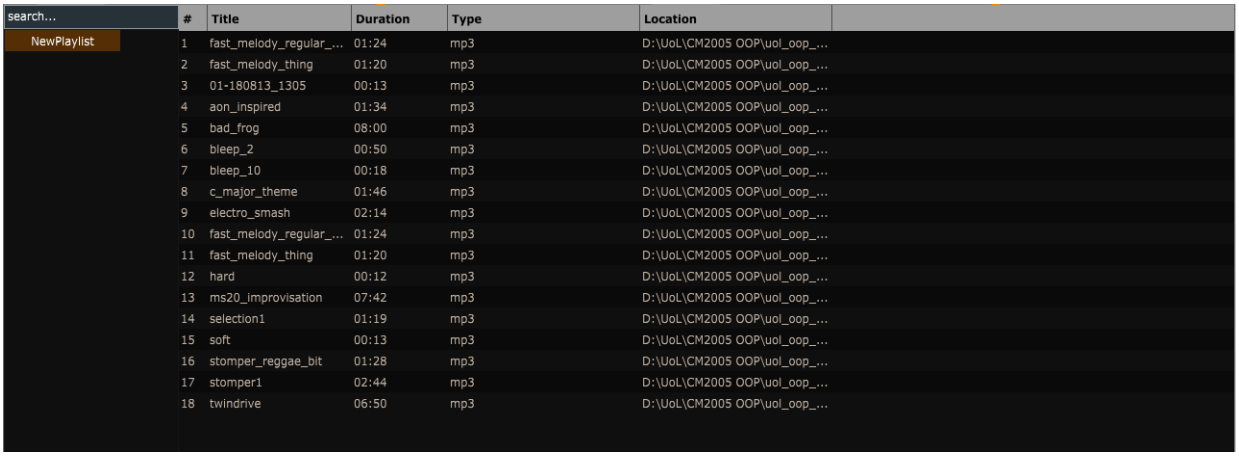

Fig 3.8 FolderComponent and PlaylistComponent GUI