main.cpp

/\*

==============================================================================

This file contains the basic startup code for a JUCE application.

==============================================================================

\*/

#include <JuceHeader.h>

#include "MainComponent.h"

//==============================================================================

class NewProjectApplication : public juce::JUCEApplication

{

public:

//==============================================================================

NewProjectApplication() {}

const juce::String getApplicationName() override { return ProjectInfo::projectName; }

const juce::String getApplicationVersion() override { return ProjectInfo::versionString; }

bool moreThanOneInstanceAllowed() override { return true; }

//==============================================================================

void initialise (const juce::String& commandLine) override

{

// This method is where you should put your application's initialisation

//code..

mainWindow.reset (new MainWindow (getApplicationName()));

}

void shutdown() override

{

// Add your application's shutdown code here..

mainWindow = nullptr; // (deletes our window)

}

//==============================================================================

void systemRequestedQuit() override

{

// This is called when the app is being asked to quit: you can ignore this

// request and let the app carry on running, or call quit() to allow the app

//to close.

quit();

}

void anotherInstanceStarted (const juce::String& commandLine) override

{

// When another instance of the app is launched while this one is running,

// this method is invoked, and the commandLine parameter tells you what

// the other instance's command-line arguments were.

}

//==============================================================================

/\*

This class implements the desktop window that contains an instance of

our MainComponent class.

\*/

class MainWindow : public juce::DocumentWindow

{

public:

MainWindow (juce::String name)

: DocumentWindow (name,

juce::Desktop::getInstance().getDefaultLookAndFeel()

.findColour (juce::ResizableWindow::backgroundColourId),

DocumentWindow::allButtons)

{

setUsingNativeTitleBar (true);

setContentOwned (new MainComponent(), true);

#if JUCE\_IOS || JUCE\_ANDROID

setFullScreen (true);

#else

setResizable (true, true);

centreWithSize (getWidth(), getHeight());

#endif

setVisible (true);

}

void closeButtonPressed() override

{

// This is called when the user tries to close this window. Here, we'll

//just

// ask the app to quit when this happens, but you can change this to do

// whatever you need.

JUCEApplication::getInstance()->systemRequestedQuit();

}

/\* Note: Be careful if you override any DocumentWindow methods - the base

class uses a lot of them, so by overriding you might break its

functionality.

It's best to do all your work in your content component instead, but if

you really have to override any DocumentWindow methods, make sure your

subclass also calls the superclass's method.

\*/

private:

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR (MainWindow)

};

private:

std::unique\_ptr<MainWindow> mainWindow;

};

//==============================================================================

// This macro generates the main() routine that launches the app.

START\_JUCE\_APPLICATION (NewProjectApplication)

MainComponent.h

#pragma once

#include <JuceHeader.h>

#include "DJAudioPlayer.h"

#include "DeckGUI.h"

#include "PlaylistComponent.h"

#include "FileBrowserComponent.h"

//==============================================================================

/\*

This component lives inside our window, and this is where you should put all

your controls and content.

\*/

class MainComponent : public juce::AudioAppComponent

{

public:

//==============================================================================

MainComponent();

~MainComponent() override;

//==============================================================================

void paint (juce::Graphics&) override;

void resized() override;

// AudioAppComponent methods

void prepareToPlay(int samplesPerBlockExpected, double sampleRate) override;

void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill) override;

void releaseResources() override;

private:

//==============================================================================

// Your private member variables go here...

juce::AudioFormatManager formatManager;

juce::AudioThumbnailCache thumbCache{ 100 };

DJAudioPlayer player1{formatManager};

DeckGUI deckGUI1{&player1, formatManager, thumbCache};

DJAudioPlayer player2{ formatManager };

DeckGUI deckGUI2{&player2, formatManager, thumbCache};

juce::MixerAudioSource mixerSource;

PlaylistComponent playlistComponent {&deckGUI1, &deckGUI2};

FileBrowserComponent fileBrowser { playlistComponent };

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR (MainComponent)

};

MainComponent.cpp

#include "MainComponent.h"

//==============================================================================

MainComponent::MainComponent()

{

setSize (800, 500);

setAudioChannels(0, 2);

addAndMakeVisible(deckGUI1);

addAndMakeVisible(deckGUI2);

addAndMakeVisible(playlistComponent);

addAndMakeVisible(fileBrowser);

fileBrowser.setBounds(getLocalBounds());

formatManager.registerBasicFormats();

}

MainComponent::~MainComponent()

{

shutdownAudio();

}

//==============================================================================

void MainComponent::paint (juce::Graphics& g)

{

// (Our component is opaque, so we must completely fill the background with a solid colour)

g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

}

void MainComponent::resized()

{

int width = getWidth();

int height = getHeight();

int halfWidth = width / 2;

int halfHeight = height / 2;

deckGUI1.setBounds(0, 0, halfWidth, halfHeight);

deckGUI2.setBounds(halfWidth, 0, halfWidth, halfHeight);

int thirdWidth = width / 3;

fileBrowser.setBounds(0, halfHeight, thirdWidth, halfHeight);

playlistComponent.setBounds(thirdWidth, halfHeight, 2 \* thirdWidth, halfHeight);

}

void MainComponent::prepareToPlay(int samplesPerBlockExpected, double sampleRate)

{

// This method is where you should set up any resources you need before playback

//starts.

player1.prepareToPlay(samplesPerBlockExpected, sampleRate);

mixerSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

mixerSource.addInputSource(&player1, false);

mixerSource.addInputSource(&player2, false);

}

void MainComponent::getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)

{

mixerSource.getNextAudioBlock(bufferToFill);

}

void MainComponent::releaseResources()

{

// This method is where you should release any resources you no longer need.

player1.releaseResources();

player1.releaseResources();

mixerSource.releaseResources();

}

DJAudioPlayer.h

#pragma once

#include "JuceHeader.h"

// DJAudioPlayer class handles audio playback, including loading, playing, and controlling audio files.

class DJAudioPlayer : public juce::AudioSource

{

public:

DJAudioPlayer(juce::AudioFormatManager& \_formatManager);

~DJAudioPlayer() override;

/\*\* Prepares the audio player for playback.\*/

void prepareToPlay(int samplesPerBlockExpected, double sampleRate) override;

/\*\* Renders the next audio block to be played.\*/

void getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill) override;

/\*\* Releases the audio player's resources.\*/

void releaseResources() override;

/\*\* Sets the gain (volume) of the audio player.\*/

void setGain(double newGain);

/\*\* Sets the speed (playback rate) of the audio player.\*/

void setSpeed(double ratio);

/\*\* Sets the playback position in seconds.\*/

void setPosition(double posInSecs);

/\*\* Sets the playback position relative to the total length.\*/

void setPositionRelative(double posInSecs);

/\* Starts audio playback. \*/

void start();

/\* Stops audio playback. \*/

void stop();

/\* Checks if the audio player is currently playing. \*/

bool isPlaying();

/\* Gets the relative position of the playhead. \*/

double getPositionRelative();

/\* Gets the playback position in seconds. \*/

double getPositionInSeconds();

void loadURL(const juce::URL& audioURL);

private:

// Reference to the audio format manager

juce::AudioFormatManager& formatManager;

// Source for reading audio data

std::unique\_ptr<juce::AudioFormatReaderSource> readerSource;

// Source for transporting audio data

juce::AudioTransportSource transportSource;

// Source for resampling audio data

juce::ResamplingAudioSource resampleSource{ &transportSource, false, 2 };

};

DJAudioPlayer.cpp

#include "DJAudioPlayer.h"

DJAudioPlayer::DJAudioPlayer(juce::AudioFormatManager& \_formatManager) : formatManager(\_formatManager)

{ }

DJAudioPlayer::~DJAudioPlayer()

{ }

void DJAudioPlayer::prepareToPlay(int samplesPerBlockExpected, double sampleRate)

{

formatManager.registerBasicFormats();

transportSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

resampleSource.prepareToPlay(samplesPerBlockExpected, sampleRate);

}

void DJAudioPlayer::getNextAudioBlock(const juce::AudioSourceChannelInfo& bufferToFill)

{

resampleSource.getNextAudioBlock(bufferToFill);

}

void DJAudioPlayer::releaseResources()

{

transportSource.releaseResources();

resampleSource.releaseResources();

}

void DJAudioPlayer::loadURL(const juce::URL& audioURL)

{

formatManager.registerBasicFormats();

std::unique\_ptr<juce::InputStream> inputStream(audioURL.createInputStream(false));

if (inputStream != nullptr)

{

auto\* reader = formatManager.createReaderFor(std::move(inputStream));

if (reader != nullptr)

{

std::unique\_ptr<juce::AudioFormatReaderSource> newSource(

new juce::AudioFormatReaderSource(reader, true));

transportSource.setSource(newSource.get(), 0, nullptr,

reader->sampleRate);

readerSource.reset(newSource.release());

transportSource.start();

}

}

}

void DJAudioPlayer::setGain(double newGain)

{

if (newGain < 0 || newGain > 100)

{

juce::Logger::writeToLog("DJAudioPlayer::setGain gain should be

beetween 0 and 100");

}

else

{

transportSource.setGain(newGain / 100);

}

}

void DJAudioPlayer::setSpeed(double ratio)

{

if (ratio < 0 || ratio > 1)

{

juce::Logger::writeToLog("DJAudioPlayer::setSpeed ratio gain should be

beetween 0 and 100");

}

else

{

resampleSource.setResamplingRatio(ratio);

}

}

void DJAudioPlayer::setPosition(double posInSecs)

{

transportSource.setPosition(posInSecs);

}

void DJAudioPlayer::setPositionRelative(double posInSecs)

{

if (posInSecs < 0 || posInSecs > 1)

{

juce::Logger::writeToLog("DJAudioPlayer::setPositionRelative pos gain

should be beetween 0 and 1");

}

else

{

double posInSec = transportSource.getLengthInSeconds() \* posInSecs;

setPosition(posInSec);

}

}

void DJAudioPlayer::start()

{

transportSource.start();

}

void DJAudioPlayer::stop()

{

transportSource.stop();

}

bool DJAudioPlayer::isPlaying()

{

return transportSource.isPlaying();

}

double DJAudioPlayer::getPositionRelative()

{

return transportSource.getCurrentPosition() /

transportSource.getLengthInSeconds();

}

double DJAudioPlayer::getPositionInSeconds()

{

return transportSource.getCurrentPosition();

}

DeckGUI.h

#pragma once

#include <JuceHeader.h>

#include "DJAudioPlayer.h"

#include "WaveformDisplay.h"

#include "InfiniteRotarySliderComponent.h"

class DeckGUI : public juce::Component,

public juce::Button::Listener,

public juce::Slider::Listener,

public juce::FileDragAndDropTarget,

public juce::Timer

{

public:

DeckGUI(DJAudioPlayer\* player,

juce::AudioFormatManager & formatManagerToUse,

juce::AudioThumbnailCache & cacheToUse);

~DeckGUI() override;

void paint (juce::Graphics&) override;

void resized() override;

/\*\* implement Button::Listener \*/

void buttonClicked(juce::Button\*) override;

/\*\* implement Slider::Listener \*/

void sliderValueChanged(juce::Slider\* slider) override;

void loadTrack(const juce::URL& audioURL);

/\*\* Checks if the component is interested in file drag events. \*/

bool isInterestedInFileDrag(const juce::StringArray& files) override;

/\*\* Handles files dropped onto the component. \*/

void filesDropped(const juce::StringArray& files, int x, int y) override;

/\*\* Timer callback for updating the component. \*/

void timerCallback();

private:

// Wheel rotation speed

const double speed = 0.8f;

InfiniteRotarySliderComponent infiniteRotarySlider;

/\*\* Sets the color of the play button based on the player's state. \*/

void playButtonSetColor();

// UI components

juce::TextButton playButton{ "PLAY" };

juce::TextButton stopButton{ "STOP" };

juce::TextButton loadButton{ "LOAD" };

juce::Slider volSlider;

juce::Slider speedSlider;

juce::Slider posSlider;

juce::Label volLabel;

juce::Label speedLabel;

DJAudioPlayer\* player;

juce::FileChooser fChooser{ "Select a file..." };

WaveformDisplay waveformDisplay;

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR (DeckGUI)

};

DeckGUI. cpp

#include <JuceHeader.h>

#include "DeckGUI.h"

// Constructor: Initializes the DeckGUI with the given player, format manager, and cache.

DeckGUI::DeckGUI(DJAudioPlayer\* player,

juce::AudioFormatManager& formatManagerToUse,

juce::AudioThumbnailCache& cacheToUse)

: player(player),

waveformDisplay(formatManagerToUse, cacheToUse)

{

// Initialize and add UI components

addAndMakeVisible(playButton);

addAndMakeVisible(volSlider);

addAndMakeVisible(speedSlider);

addAndMakeVisible(infiniteRotarySlider);

addAndMakeVisible(waveformDisplay);

// Add listeners

volSlider.addListener(this);

speedSlider.addListener(this);

posSlider.addListener(this);

playButton.addListener(this);

playButtonSetColor();

// Set slider ranges and initial values

speedSlider.setRange(0, 1);

volSlider.setRange(0, 100);

posSlider.setRange(0, 1);

volSlider.setValue(50);

speedSlider.setValue(1);

// Set labels

volLabel.setText("Volume", juce::dontSendNotification);

volLabel.setJustificationType(juce::Justification::centred);

addAndMakeVisible(volLabel);

speedLabel.setText("Speed", juce::dontSendNotification);

speedLabel.setJustificationType(juce::Justification::centred);

addAndMakeVisible(speedLabel);

// Set slider styles

volSlider.setSliderStyle(juce::Slider::LinearVertical);

volSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, true, 100, 20); // Размещение текстового поля

volSlider.setColour(juce::Slider::textBoxOutlineColourId, juce::Colours::green); // Цвет текстового поля

volSlider.setNumDecimalPlacesToDisplay(0);

volSlider.setColour(juce::Slider::textBoxOutlineColourId, juce::Colours::transparentBlack); // Убираем обводку текстового поля

speedSlider.setSliderStyle(juce::Slider::LinearVertical);

speedSlider.setNumDecimalPlacesToDisplay(3);

speedSlider.setColour(juce::Slider::textBoxOutlineColourId, juce::Colours::green);

speedSlider.setTextBoxStyle(juce::Slider::TextBoxBelow, true, 100, 20);

speedSlider.setColour(juce::Slider::textBoxOutlineColourId, juce::Colours::transparentBlack); // Убираем обводку текстового поля

posSlider.setSliderStyle(juce::Slider::Rotary);

startTimer(40);

// Set callbacks for waveform display and infinite rotary slider

waveformDisplay.onPositionChanged = [this, player](double newPosition) {

player->setPositionRelative(newPosition);

};

// Set the onTrackPositionChange callback for the infiniteRotarySlider

infiniteRotarySlider.onTrackPositionChange = [this, player]() {

double newPosition = infiniteRotarySlider.getValue() / (180 \* speed);

player->setPosition(newPosition);

};

}

// Destructor: Cleans up resources.

DeckGUI::~DeckGUI()

{

stopTimer();

}

// Paints the component

void DeckGUI::paint (juce::Graphics& g)

{

g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId)); // clear the background

g.setColour (juce::Colours::grey);

g.drawRect (getLocalBounds(), 1); // draw an outline around the component

}

void DeckGUI::resized()

{

// colums

double colW = getWidth() / 2;

double rowH = getHeight() / 8;

// colum 1

double sliderW = colW / 3;

waveformDisplay.setBounds(0, 0, getWidth(), rowH \* 2);

volSlider.setBounds(0, rowH \* 3, sliderW, rowH \* 5);

speedSlider.setBounds(sliderW \* 1, rowH \* 3, sliderW, rowH \* 5);

// colum 2

double buttonW = colW / 4;

playButton.setBounds(colW + buttonW \* 2, rowH \* 7, buttonW, rowH);

infiniteRotarySlider.setBounds(colW, rowH \* 3, colW, rowH \* 3);

volLabel.setBounds(0, rowH \* 2, sliderW, rowH \* 0.5);

speedLabel.setBounds(sliderW \* 1, rowH \* 2, sliderW, rowH \* 0.5);

}

// Sets the color of the play button based on the player's state.

void DeckGUI::playButtonSetColor()

{

playButton.setButtonText(player->isPlaying() ? "Stop" : "Play");

playButton.setColour(juce::TextButton::buttonColourId, player->isPlaying() ? juce::Colour(0xffed797f) : juce::Colour(0xff00bd16));

}

// Handles button click events.

void DeckGUI::buttonClicked(juce::Button\* button)

{

if (button == &playButton)

{

juce::Logger::writeToLog("Play button was clicked");

if (player->isPlaying()) {

player->stop();

}

else {

player->start();

}

playButtonSetColor();

}

}

// Handles slider value change events.

void DeckGUI::sliderValueChanged(juce::Slider\* slider)

{

if (slider == &volSlider)

{

player->setGain(volSlider.getValue());

}

if (slider == &speedSlider)

{

player->setSpeed(speedSlider.getValue());

}

if (slider == &posSlider)

{

player->setPositionRelative(posSlider.getValue());

}

}

// Checks if the component is interested in file drag events.

bool DeckGUI::isInterestedInFileDrag(const juce::StringArray& files)

{

juce::Logger::writeToLog("DeckGUI::isInterestedInFileDrag");

return true;

}

// Handles files dropped onto the component.

void DeckGUI::filesDropped(const juce::StringArray& files, int x, int y)

{

juce::Logger::writeToLog("DeckGUI::filesDropped");

if (files.size() == 1)

{

loadTrack(juce::URL{ juce::File{files[0]} });

}

}

// Timer callback for updating the component.

void DeckGUI::timerCallback()

{

double positionRelative = player->getPositionRelative();

waveformDisplay.setPositionRelative(positionRelative);

if (player->isPlaying())

{

double positionInSeconds = player->getPositionInSeconds();

infiniteRotarySlider.setValue(static\_cast<int>(positionInSeconds \* speed \* 180));

// Calculate the angle based on the relative position

float angle = static\_cast<float>(positionInSeconds \* juce::MathConstants<double>::pi \* speed);

infiniteRotarySlider.setAngle(angle);

}

}

// Loads a track from the given URL

void DeckGUI::loadTrack(const juce::URL& audioURL)

{

player->loadURL(audioURL);

waveformDisplay.loadURL(audioURL);

playButtonSetColor();

}

PlaylistComponent.h

#pragma once

#include <JuceHeader.h>

#include <vector>

#include <string>

#include "DeckGUI.h"

// Structure to hold track information

struct TrackInfo

{

juce::String path;

juce::String title;

double duration = 0.0; // Duration of the track in seconds

};

// PlaylistComponent class handles the playlist, including displaying tracks and handling user interactions.

class PlaylistComponent : public juce::Component,

public juce::TableListBoxModel,

public juce::Button::Listener

{

public:

PlaylistComponent(DeckGUI\* deckGUI1, DeckGUI\* deckGUI2);

~PlaylistComponent() override;

void paint (juce::Graphics&) override;

void resized() override;

/\*\* Returns the number of rows in the playlist.\*/

int getNumRows() override;

/\*\* Paints the background of a row in the playlist.\*/

void paintRowBackground ( juce::Graphics& g,

int rowNumber,

int width,

int height,

bool rowIsSelected) override;

/\*\* Paints a cell in the playlist.\*/

void paintCell( juce::Graphics& g,

int rowNumber,

int columnId,

int width,

int height,

bool rowIsSelected) override;

/\*\* Refreshes the component for a cell in the playlist.\*/

juce::Component\* refreshComponentForCell( int rowNumber,

int columnId,

bool isRowSelected,

juce::Component\* existingComponentToUpdate) override;

/\*\* Handles button click events.\*/

void buttonClicked(juce::Button\* button) override;

/\*\* Adds a track to the playlist.\*/

void addTrack(const juce::String& trackPath);

private:

juce::TableListBox tableComponent; // Table component to display the playlist

std::vector<TrackInfo> tracks; // Vector to hold track information

DeckGUI\* deckGUI1; // Pointer to the first DeckGUI instance

DeckGUI\* deckGUI2; // Pointer to the second DeckGUI instance

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR (PlaylistComponent)

};

PlaylistComponent. cpp

#include <JuceHeader.h>

#include "PlaylistComponent.h"

//==============================================================================

PlaylistComponent::PlaylistComponent(DeckGUI\* deckGUI1, DeckGUI\* deckGUI2)

: deckGUI1(deckGUI1), deckGUI2(deckGUI2)

{

tableComponent.getHeader().addColumn("Title", 1, 150);

tableComponent.getHeader().addColumn("Duration", 2, 60);

tableComponent.getHeader().addColumn("Play L", 3, 60);

tableComponent.getHeader().addColumn("Play R", 4, 60);

tableComponent.getHeader().addColumn("Delete", 5, 60);

tableComponent.setModel(this);

addAndMakeVisible(tableComponent);

}

PlaylistComponent::~PlaylistComponent()

{

}

void PlaylistComponent::paint(juce::Graphics& g)

{

// clear the background g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));

g.setColour(juce::Colours::grey);

g.drawRect(getLocalBounds(), 1); // draw an outline around the component

g.setColour(juce::Colours::white);

g.setFont(juce::FontOptions(14.0f));

g.drawText("PlaylistComponent", getLocalBounds(),

juce::Justification::centred, true); // draw some placeholder text

}

void PlaylistComponent::resized()

{

tableComponent.setBounds(0, 0, getWidth(), getHeight());

}

int PlaylistComponent::getNumRows()

{

return tracks.size();

}

void PlaylistComponent::paintRowBackground(juce::Graphics& g,

int rowNumber,

int width,

int height,

bool rowIsSelected)

{

if (rowIsSelected)

{

g.fillAll(juce::Colours::orange);

}

else

{

g.fillAll(juce::Colours::darkgrey);

}

}

void PlaylistComponent::paintCell(juce::Graphics& g,

int rowNumber,

int columnId,

int width,

int height,

bool rowIsSelected)

{

if (rowNumber >= tracks.size()) return;

g.setColour(juce::Colours::white);

juce::String text;

if (columnId == 1) text = tracks[rowNumber].title;

else if (columnId == 2) text = juce::String(tracks[rowNumber].duration, 2)

+ " s";

g.drawText(text, 2, 0, width - 4, height, juce::Justification::centredLeft,

true);

}

juce::Component\* PlaylistComponent::refreshComponentForCell(int rowNumber,

int columnId,

bool isRowSelected,

juce::Component\* existingComponentToUpdate)

{

if (columnId == 3 || columnId == 4 || columnId == 5)

{

if (existingComponentToUpdate == nullptr)

{

juce::TextButton\* btn = new juce::TextButton(columnId == 3 ? "Play L" :

(columnId == 4 ? "Play R" : "Delete"));

std::string id = std::to\_string(rowNumber) + (columnId == 3 ? "\_L" :

(columnId == 4 ? "\_R" : "\_D"));

btn->setComponentID(id);

btn->addListener(this);

existingComponentToUpdate = btn;

}

}

return existingComponentToUpdate;

}

void PlaylistComponent::buttonClicked(juce::Button\* button)

{

std::string id = button->getComponentID().toStdString();

size\_t underscorePos = id.find('\_');

int rowNumber = std::stoi(id.substr(0, underscorePos));

char action = id[underscorePos + 1];

if (action == 'L' || action == 'R')

{

juce::URL audioURL{ juce::File{ tracks[rowNumber].path } };

if (action == 'L')

{

deckGUI1->loadTrack(audioURL);

}

else if (action == 'R')

{

deckGUI2->loadTrack(audioURL);

}

}

else if (action == 'D')

{

// Remove the track from the playlist

tracks.erase(tracks.begin() + rowNumber);

tableComponent.updateContent();

}

}

void PlaylistComponent::addTrack(const juce::String& trackPath)

{

// Check if the track already exists in the playlist

for (const auto& track : tracks)

{

if (track.path == trackPath)

{

juce::Logger::writeToLog("Track already exists in the playlist: " +

trackPath);

return; // Track already exists, so do not add it again

}

}

juce::File file(trackPath);

if (!file.existsAsFile())

return;

juce::AudioFormatManager formatManager;

formatManager.registerBasicFormats();

std::unique\_ptr<juce::AudioFormatReader> reader(

formatManager.createReaderFor(file));

TrackInfo track;

track.path = trackPath;

track.title = file.getFileNameWithoutExtension();

if (reader)

{

track.duration = reader->lengthInSamples / reader->sampleRate;

}

// Add the track to the list of tracks

tracks.push\_back(track);

tableComponent.updateContent();

}

FileBrowserComponent.h

#pragma once

#include <JuceHeader.h>

#include "PlaylistComponent.h"

/\* FileBrowserComponent class handles the file browsing functionality, allowing users to navigate directories and add

\*/

class FileBrowserComponent : public juce::Component,

private juce::FileBrowserListener

{

public:

FileBrowserComponent(PlaylistComponent& playlist);

void resized() override;

private:

// Called when the file selection changes.

void selectionChanged() override;

// Called when a file is clicked.

void fileClicked(const juce::File& file, const juce::MouseEvent& event) override;

// Called when a file is double-clicked.

void fileDoubleClicked(const juce::File& file) override;

// Called when the root directory of the browser changes.

void browserRootChanged(const juce::File& newRoot) override;

// Scans the given directory and adds files to the playlist.

void scanAndAddFiles(const juce::File& directory);

// Thread for handling file I/O operations

juce::TimeSliceThread thread{ "File Browser Thread" };

// List of directory contents

juce::DirectoryContentsList directoryList{ nullptr, thread };

// Component for displaying the file tree

juce::FileTreeComponent fileTreeComp{ directoryList };

PlaylistComponent& playlistComponent; // Reference to the PlaylistComponent

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR(FileBrowserComponent)

};

FileBrowserComponent. cpp

#include "FileBrowserComponent.h"

FileBrowserComponent::FileBrowserComponent(PlaylistComponent& playlist)

: playlistComponent(playlist)

{

// Start the thread for handling file I/O operations.

thread.startThread();

// Add the file tree component to the UI.

addAndMakeVisible(fileTreeComp);

// Set the initial directory to the user's music directory.

directoryList.setDirectory(juce::File::getSpecialLocation(

juce::File::userMusicDirectory), true, true);

// Set the title and background color for the file tree component.

fileTreeComp.setTitle("Files");

fileTreeComp.setColour(juce::FileTreeComponent::backgroundColourId,

juce::Colours::lightgrey.withAlpha(0.6f));

// Add this component as a listener to the file tree component.

fileTreeComp.addListener(this);

}

void FileBrowserComponent::resized()

{

fileTreeComp.setBounds(getLocalBounds());

}

void FileBrowserComponent::selectionChanged()

{

}

void FileBrowserComponent::fileClicked(const juce::File& file, const juce::MouseEvent& event)

{

// If the clicked file is a directory, scan and add its files to the playlist.

if (file.isDirectory())

{

scanAndAddFiles(file);

}

}

void FileBrowserComponent::fileDoubleClicked(const juce::File& file)

{

}

void FileBrowserComponent::browserRootChanged(const juce::File& newRoot)

{

juce::Logger::writeToLog("browserRootChanged called: " +

newRoot.getFullPathName());

scanAndAddFiles(newRoot);

}

void FileBrowserComponent::scanAndAddFiles(const juce::File& directory)

{

if (!directory.isDirectory())

return;

// Find all audio files in the directory.

juce::Array<juce::File> files = directory.findChildFiles(juce::File::findFiles,

false, "\*.mp3;\*.wav;\*.flac");

// Add each found file to the playlist.

for (const auto& file : files)

{

playlistComponent.addTrack(file.getFullPathName());

}

}

WaveformDisplay.h

#pragma once

#include <JuceHeader.h>

#include <functional>

class WaveformDisplay : public juce::Component,

public juce::ChangeListener

{

public:

WaveformDisplay(juce::AudioFormatManager& formatManagerToUse,

juce::AudioThumbnailCache& cacheToUse);

~WaveformDisplay() override;

void paint (juce::Graphics&) override;

void resized() override;

/\*\* Called when there is a change in the audio thumbnail.\*/

void changeListenerCallback(juce::ChangeBroadcaster\* source) override;

/\*\* Loads an audio file from the given URL and generates its waveform.\*/

void loadURL(juce::URL audioURL);

/\*\* Set the relative position of the playhead \*/

void setPositionRelative(double pos);

// Callback function to notify the player about the new position

std::function<void(double)> onPositionChanged;

private:

// Handles mouse drag events to update the playhead position.

void mouseDrag(const juce::MouseEvent& e) override;

juce::AudioThumbnail audioThumb; // Audio thumbnail for generating the waveform

bool fileLoaded; // Flag indicating whether a file is loaded

double position; // Current position of the playhead

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR(WaveformDisplay)

};

WaveformDisplay.cpp

#include <JuceHeader.h>

#include "WaveformDisplay.h"

WaveformDisplay::WaveformDisplay(juce::AudioFormatManager& formatManagerToUse,

juce::AudioThumbnailCache& cacheToUse) :

audioThumb(1000, formatManagerToUse,

cacheToUse),

fileLoaded(false),

position(0)

{

// In your constructor, you should add any child components, and

// initialise any special settings that your component needs.

audioThumb.addChangeListener(this);

}

WaveformDisplay::~WaveformDisplay()

{

}

void WaveformDisplay::paint (juce::Graphics& g)

{

g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId)); // clear the background

g.setColour (juce::Colours::grey);

g.drawRect (getLocalBounds(), 1); // draw an outline around the component

g.setColour (juce::Colours::white);

if (fileLoaded)

{

audioThumb.drawChannel(g,

getLocalBounds(),

0,

audioThumb.getTotalLength(),

0,

1.0f

);

g.setColour(juce::Colours::lightgreen);

g.fillRect(static\_cast<float> (position \* getWidth()), 0.0f,

1.0f, static\_cast<float> (getHeight()));

}

else

{

g.setFont (juce::FontOptions (20.0f));

g.drawText ("File not loaded...", getLocalBounds(),

juce::Justification::centred, true); // draw some placeholder text

}

}

void WaveformDisplay::resized()

{

// This method is where you should set the bounds of any child

// components that your component contains..

}

void WaveformDisplay::loadURL(juce::URL audioURL) {

juce::Logger::writeToLog("wfd: loadURL");

audioThumb.clear();

fileLoaded = audioThumb.setSource(new juce::URLInputSource(audioURL));

if (fileLoaded) {

juce::Logger::writeToLog("wfd: loaded!");

repaint();

}

else

{

juce::Logger::writeToLog("wfd: not loaded!");

}

}

void WaveformDisplay::changeListenerCallback(juce::ChangeBroadcaster\* source)

{

juce::Logger::writeToLog("wfd: change recevied!");

repaint();

}

void WaveformDisplay::setPositionRelative(double pos)

{

if (pos != position) {

position = pos;

repaint();

}

}

void WaveformDisplay::mouseDrag(const juce::MouseEvent& e)

{

if (fileLoaded)

{

double newPosition = juce::jmax(static\_cast<double> (e.x), 0.0) / getWidth();

// Ensure newPosition is between 0 and 1

newPosition = juce::jlimit(0.0, 1.0, newPosition);

setPositionRelative(newPosition);

// Notify the player about the new position

if (onPositionChanged)

{

onPositionChanged(newPosition);

}

}

}

InfiniteRotarySliderComponent.h

#pragma once

#include <JuceHeader.h>

/\* InfiniteRotarySliderComponent class provides a custom rotary slider with infinite rotation capability. \*/

class InfiniteRotarySliderComponent : public juce::Component

{

public:

// Constructor: Initializes the InfiniteRotarySliderComponent.

InfiniteRotarySliderComponent();

// Destructor: Cleans up resources.

~InfiniteRotarySliderComponent() override;

// Returns the current value of the slider.

float getValue();

// Sets the range of the slider.

void setRange(double newMinimum, double newMaximum);

// Sets the value of the slider.

void setValue(double newValue);

// Sets the angle of the slider.

void setAngle(float newAngle);

// Sets the minimum value for one revolution of the slider.

void setOneRevolutionSliderMin(float newMin);

// Sets the maximum value for one revolution of the slider.

void setOneRevolutionSliderMax(float newMax);

// Sets the end angle of the slider.

void setEndAngle(float newEndAngle);

// Sets the start value of the slider.

void setStartValue(float newStartValue);

// Sets whether the start value needs to be updated.

void setStartValueNeedUpdate(bool newStartValueNeedUpdate);

// Sets the step value for the slider.

void setStep(float newStep);

// Sets the adjusted value of the slider.

void setAdjustedValue(float newAdjustedValue);

void paint(juce::Graphics&) override;

void resized() override;

// Callback function to notify when the value changes.

std::function<void()> onValueChange;

// Callback function to notify when the track position changes.

std::function<void()> onTrackPositionChange;

// Underlying juce::Slider component is exposed in order to allow the user to //set styles etc.

juce::Slider slider;

// Returns the current angle of the slider.

float getAngle();

// Returns the adjusted value of the slider.

float getAdjustedValue();

private:

// Processes the slider value change.

void m\_processSliderValue(float value);

// Sets the flag to update the start value.

void m\_setValueStartNeedUpdate();

// Sets the end angle of the slider.

void m\_setAdjustedAngleEnd();

// Returns the range of the slider.

float m\_getSliderRange();

float m\_oneRevolutionSliderMin{ 0.0f }; // Minimum value for one revolution

float m\_oneRevolutionSliderMax{ 360.0f }; // Maximum value for one revolution

float m\_angle{ 0.0f }; // Current angle of the slider

float m\_endAngle{ 0.0f }; // End angle of the slider

float m\_startValue{ 0.0f }; // Start value of the slider

bool m\_startValueNeedUpdate{ true }; // Flag indicating whether the start value // needs to be updated

float m\_value{ 0.0f }; // Current value of the slider

float m\_step{ 1.0f }; // Step value for the slider

float m\_adjustedValue{ 0.0f }; // Adjusted value of the slider

float m\_knobMin{ 0.0f }; // Minimum value of the knob

float m\_knobMax{ 100.0f \* 360 }; // Maximum value of the knob

JUCE\_DECLARE\_NON\_COPYABLE\_WITH\_LEAK\_DETECTOR(InfiniteRotarySliderComponent)

};

InfiniteRotarySliderComponent.cpp

#include <JuceHeader.h>

#include "InfiniteRotarySliderComponent.h"

InfiniteRotarySliderComponent::InfiniteRotarySliderComponent()

{

slider.setSliderStyle(juce::Slider::Rotary);

/\*\* Hide underlying slider UI. TODO - let the user use lookAndFeel methods for that \*/

slider.setColour(juce::Slider::backgroundColourId, juce::Colours::transparentBlack);

slider.setColour(juce::Slider::thumbColourId, juce::Colours::transparentBlack);

slider.setColour(juce::Slider::trackColourId, juce::Colours::transparentBlack);

slider.setColour(juce::Slider::rotarySliderFillColourId, juce::Colours::transparentBlack);

slider.setColour(juce::Slider::rotarySliderOutlineColourId, juce::Colours::transparentBlack);

/\*\* TODO Make configurable by the user \*/

slider.setTextBoxStyle(juce::Slider::TextEntryBoxPosition::NoTextBox, true, 0, 0);

slider.setRotaryParameters(0.0f, juce::MathConstants<float>::twoPi, false);

slider.setRange(m\_oneRevolutionSliderMin, m\_oneRevolutionSliderMax);

slider.onValueChange = [this] {

m\_processSliderValue(slider.getValue());

};

slider.onDragStart = [this] {

m\_setValueStartNeedUpdate();

};

slider.onDragEnd = [this] {

m\_setAdjustedAngleEnd();

};

addAndMakeVisible(slider);

}

InfiniteRotarySliderComponent::~InfiniteRotarySliderComponent()

{ }

void InfiniteRotarySliderComponent::paint (juce::Graphics& g)

{

float lineThickness = 4.0f;

float radius = juce::jmin(getWidth(), getHeight()) \* 0.5f - lineThickness;

float rectW = 2.0f;

float rectH = radius \* 2 / 3; // New length

juce::Path p;

p.addRectangle((getWidth() - rectW) \* 0.5f,

getHeight() \* 0.5f - radius,

rectW,

rectH);

/\*\* Rotate the mark to show slider's rotation \*/

p.applyTransform(

juce::AffineTransform::rotation(

getAngle(),

getWidth() \* 0.5,

getHeight() \* 0.5));

// Set the color to white and fill the path, which creates the white line

g.setColour(juce::Colours::white);

g.fillPath(p);

/\*\* Add knob handle (just a spot) \*/

float spotR = 10.0f;

g.setColour(juce::Colours::orange);

g.drawEllipse(getWidth() \* 0.5 - radius,

getHeight() \* 0.5 - radius,

radius \* 2,

radius \* 2,

lineThickness);

g.setColour(juce::Colours::grey);

g.fillEllipse(getWidth() \* 0.5 - radius / 3,

getHeight() \* 0.5 - radius / 3,

radius \* 2 / 3,

radius \* 2 / 3);

}

void InfiniteRotarySliderComponent::resized()

{

/\*\* Occupy all available space \*/

slider.setBounds(0, 0, getWidth(), getHeight());

}

void InfiniteRotarySliderComponent::m\_processSliderValue(float value) {

/\*\* UI part \*/

if (m\_startValueNeedUpdate) {

m\_startValue = value;

m\_startValueNeedUpdate = false;

}

else {

m\_angle = m\_endAngle + (value - m\_startValue) / m\_getSliderRange() \* juce::MathConstants<float>::twoPi;

}

/\*\* Data part \*/

if (abs(value - m\_startValue) > m\_step) {

if (m\_value > value) {

m\_adjustedValue -= m\_step;

}

else if (value - m\_value > m\_step) {

if ((value - m\_value) > (m\_oneRevolutionSliderMax - m\_oneRevolutionSliderMin) \* 0.9f) {

m\_adjustedValue -= m\_step;

}

else {

m\_adjustedValue += m\_step;

}

}

}

m\_value = (float)((int)value / m\_step);

m\_adjustedValue = juce::jmin(m\_adjustedValue, m\_knobMax);

m\_adjustedValue = juce::jmax(m\_adjustedValue, m\_knobMin);

// Notify the track position change

if (onTrackPositionChange) {

onTrackPositionChange();

}

}

void InfiniteRotarySliderComponent::m\_setValueStartNeedUpdate() {

m\_startValueNeedUpdate = true;

}

void InfiniteRotarySliderComponent::m\_setAdjustedAngleEnd() {

m\_endAngle = m\_angle;

}

float InfiniteRotarySliderComponent::m\_getSliderRange() {

return m\_oneRevolutionSliderMax - m\_oneRevolutionSliderMin;

}

float InfiniteRotarySliderComponent::getValue() {

return m\_adjustedValue;

}

float InfiniteRotarySliderComponent::getAngle() {

return m\_angle;

}

float InfiniteRotarySliderComponent::getAdjustedValue() {

return m\_adjustedValue;

}

void InfiniteRotarySliderComponent::setRange(double newMinimum, double newMaximum) {

m\_knobMin = (float)newMinimum;

m\_knobMax = (float)newMaximum;

}

void InfiniteRotarySliderComponent::setValue(double newValue) {

m\_adjustedValue = newValue;

}

void InfiniteRotarySliderComponent::setAngle(float newAngle)

{

m\_angle = newAngle;

m\_endAngle = newAngle; // Update the end angle to match the new angle

repaint(); // Trigger a repaint to update the visual representation

}

void InfiniteRotarySliderComponent::setOneRevolutionSliderMin(float newMin) {

m\_oneRevolutionSliderMin = newMin;

}

void InfiniteRotarySliderComponent::setOneRevolutionSliderMax(float newMax) {

m\_oneRevolutionSliderMax = newMax;

}

void InfiniteRotarySliderComponent::setEndAngle(float newEndAngle) {

m\_endAngle = newEndAngle;

}

void InfiniteRotarySliderComponent::setStartValue(float newStartValue) {

m\_startValue = newStartValue;

}

void InfiniteRotarySliderComponent::setStartValueNeedUpdate(bool newStartValueNeedUpdate) {

m\_startValueNeedUpdate = newStartValueNeedUpdate;

}

void InfiniteRotarySliderComponent::setStep(float newStep) {

m\_step = newStep;

}

void InfiniteRotarySliderComponent::setAdjustedValue(float newAdjustedValue) {

m\_adjustedValue = newAdjustedValue;

}