# CodeDump

## audioeffect.cpp

#include "audioeffect.h"  
  
AudioEffect::AudioEffect(QWidget \*parent)  
{  
 effectDialog = new AudioEffectWindow(parent);  
}  
  
std::shared\_ptr<GainNode> AudioEffect::getOutputNode() {  
 return outputNode;  
}  
  
std::shared\_ptr<GainNode> AudioEffect::getInputNode() {  
 return inputNode;  
}  
  
void AudioEffect::showEffectWindow() {  
 effectDialog->exec();  
}  
  
void AudioEffect::hideEffectWindow() {  
 effectDialog->hide();  
}  
  
std::string AudioEffect::getFriendlyName() {  
 return name;  
}

## audioeffect.h

#ifndef AUDIOEFFECT\_H  
#define AUDIOEFFECT\_H  
  
#include "LabSound/LabSound.h"  
#include <QDialog>  
  
class AudioEffectWindow;  
#include "gui/audioeffectwindow.h"  
  
  
using namespace lab;  
  
class AudioEffect  
{  
public:  
 AudioEffect(QWidget \*parent);  
  
 virtual std::shared\_ptr<GainNode> getOutputNode();  
 virtual std::shared\_ptr<GainNode> getInputNode();  
  
 virtual void showEffectWindow();  
 virtual void hideEffectWindow();  
  
 virtual std::string getFriendlyName();  
  
protected:  
 std::shared\_ptr<GainNode> outputNode;  
 std::shared\_ptr<GainNode> inputNode;  
  
 AudioEffectWindow \*effectDialog;  
 QWidget \*effectUI;  
  
 std::string name;  
};  
  
#endif // AUDIOEFFECT\_H

## audiomanager.cpp

#include "audiomanager.h"  
  
  
AudioManager::AudioManager(QWidget \*parent, Timeline &\_timeline)  
{  
 debug::out(3, "Timeline init");  
 timeline = &\_timeline;  
 stopTime = 0.0;  
 isPlaying = false;  
 currentGridTime = 1.0;  
 scheduled = false;  
 debug::out(3, "Starting audio engine...");  
  
 outputNode = std::make\_shared<GainNode>();  
 outputNode->gain()->setValue(1.0f);  
 initContext();  
  
  
  
 trackList = new std::vector<class Track \*>();  
 selectedTrackList = new std::vector<class Track \*>();  
  
 debug::out(3, "Loading metronome...");  
  
 metronome = new Metronome(outputNode, this);  
  
  
 debug::out(3, "Starting event loop...");  
 eventTimer = new TimerEX(parent, std::bind(&AudioManager::eventLoop, this));  
  
 session = new Session(parent, \*this);  
 rendering = false;  
  
 //eventTimer->start();  
 debug::out(3, "Audio engine started without any issues!");  
}  
  
void AudioManager::initContext() {  
 const auto defaultAudioDeviceConfigurations = GetDefaultAudioDeviceConfiguration();  
 context = lab::MakeRealtimeAudioContext(defaultAudioDeviceConfigurations.second, defaultAudioDeviceConfigurations.first);  
 context->connect(context->device(), outputNode);  
}  
  
std::shared\_ptr<AudioBus> AudioManager::MakeBusFromSampleFile(std::string fileName) {  
  
 std::shared\_ptr<AudioBus> bus = MakeBusFromFile(fileName, false);  
 if (!bus) {  
 debug::out(1, "COULD NOT OPEN FILE: " + fileName);  
 return nullptr;  
 } else {  
 debug::out(3, "Loaded audio file" + fileName);  
 }  
 return bus;  
}  
  
void AudioManager::play() {  
 if (isPlaying == false) {  
 startTime = context->currentTime();  
 updateMetSchedule();  
 scheduleTracks();  
 isPlaying = true;  
 if (!rendering) {  
 eventTimer->start();  
 }  
 }  
}  
  
void AudioManager::pause() {  
 if (isPlaying == true) {  
 isPlaying = false;  
 cancelTrackPlayback();  
 if (!rendering) {  
 eventTimer->stop();  
 }  
 stopTime = getCurrentRelativeTime();  
 }  
}  
  
void AudioManager::stop() {  
  
 if (isPlaying == true) {  
 isPlaying = false;  
 cancelTrackPlayback();  
 if (!rendering) {  
 eventTimer->stop();  
 }  
 }  
 stopTime = 0.0;  
 currentGridTime = 1.0;  
}  
  
void AudioManager::setLookAhead(double \_value) {  
 lookAhead = \_value;  
}  
  
void AudioManager::updateMetSchedule() {  
 metronome->schedulePrimary(floor(currentGridTime) + 1);  
 double divGrid = 1.00 / division;  
 std::vector<double> scheduleQueue;  
 for (int i = 1; i < division; i++) {  
 scheduleQueue.insert(scheduleQueue.end(), (floor(currentGridTime) + 1) + (i \* divGrid));  
  
 }  
  
 metronome->scheduleSecondary(scheduleQueue);  
  
}  
  
void AudioManager::updateSchedule() {  
  
 //double toNearestBar = (floor(currentGridTime) + 1) - currentGridTime;  
 //if (toNearestBar < lookAhead || currentGridTime == 0) {  
 //metPrimaryNode->start(floor(currentGridTime));  
 //debug::out(3, "Buffered Primary Met");  
 //if (toNearestBar < 0.01) {  
  
 // if (scheduled == true) {  
  
 // scheduled = false;  
 // }  
  
 //} else {  
 // if (scheduled == false) {  
 //updateMetSchedule();  
 //debug::out(3, "Scheduling...");  
 // scheduled = true;  
 // }  
 // }  
  
 // }  
  
  
 //metronome->update();  
  
}  
  
void AudioManager::eventLoop() {  
 float relativeTime = (context->currentTime() - startTime) + stopTime;  
 currentGridTime = ((relativeTime / beatLength) / division) + 1.0;  
  
 if (rendering == true) {  
 dialogs::ProgressDialog::updateValue(int(context->currentTime()));  
 }  
 //updateSchedule();  
}  
  
void AudioManager::setDivision(int \_division) {  
 division = \_division;  
 barLength = bpm \* division;  
}  
  
void AudioManager::setBPM(double \_beatsPerMinuet) {  
 bpm = \_beatsPerMinuet;  
 beatLength = 60.00 / bpm;  
 barLength = bpm \* division;  
  
 for (int t = 0; t < int(trackList->size()); ++t) {  
 Track \*track = trackList->at(t);  
 for (int ar = 0; ar < track->getAudioRegionListCount(); ar++) {  
 AudioRegion \*audioRegion = track->getAudioRegionByIndex(ar);  
 audioRegion->updateGridLength();  
 audioRegion->getRegionGraphicItem()->setGridLength(audioRegion->getGridLength());  
 }  
 }  
}  
  
double AudioManager::getBPM() {  
 return bpm;  
}  
  
float AudioManager::getCurrentGridTime() {  
 return currentGridTime;  
}  
  
double AudioManager::gridTimeToContextSeconds(float \_gridTime) {  
 double secondsTime = ((\_gridTime - 1.0) \* beatLength) \* division;  
 return startTime + secondsTime;  
}  
  
double AudioManager::gridTimeToSeconds(float \_gridTime) {  
 double secondsTime = ((\_gridTime) \* beatLength) \* division;  
 return secondsTime;  
}  
  
float AudioManager::secondsToGridTime(double \_seconds) {  
 double gridTime = ((\_seconds / beatLength) / division) + 1.0;  
 return gridTime;  
}  
  
float AudioManager::getCurrentRelativeTime() {  
 float relativeTime = (context->currentTime() - startTime) + stopTime;  
 return relativeTime;  
}  
  
Track\* AudioManager::addTrack(std::string trackUUID) {  
 debug::out(3, "Creating new track...");  
 Track \*newTrack = new Track(\*timeline, \*this, trackUUID);  
 debug::out(3, "Pushing to list...");  
 trackList->push\_back(newTrack);  
 debug::out(3, "Setting index");  
 newTrack->setIndex(trackList->size() - 1);  
  
 debug::out(3, "Dispatching to UI...");  
 return newTrack;  
}  
  
void AudioManager::removeTrack(Track \*track) {  
 debug::out(3, "Deleting track");  
  
 auto iterator = std::find(trackList->begin(), trackList->end(), track);  
 if (iterator != trackList->end()) {  
 int index = std::distance(trackList->begin(), iterator);  
 trackList->erase(trackList->begin() + index);  
 }  
 delete track;  
}  
  
Track\* AudioManager::getTrackByIndex(int index) {  
 return trackList->at(index);  
}  
  
Track\* AudioManager::getSelectedTrack(int index) {  
 if (selectedTrackList->size() != 0) {  
 return selectedTrackList->at(index);  
 } else {  
 return nullptr;  
 }  
  
}  
  
std::vector<class Track\*>\* AudioManager::getSelectedTracks() {  
 return selectedTrackList;  
}  
  
std::shared\_ptr<GainNode> AudioManager::getOutputNode() {  
 return outputNode;  
}  
  
void AudioManager::setTrackSelected(Track \*track, bool selected) {  
 if (selected == true) {  
 debug::out(3, "Pushing track to vector...");  
 for (int i = 0; i < int(selectedTrackList->size()); i++ ) {  
 setTrackSelected(selectedTrackList->at(i), false);  
 }  
 selectedTrackList->clear();  
 selectedTrackList->push\_back(track);  
 debug::out(3, "Setting as selected...");  
 track->setSelected(true);  
 } else {  
 auto iterator = std::find(selectedTrackList->begin(), selectedTrackList->end(), track);  
  
 if (iterator != selectedTrackList->end()) {  
 int index = std::distance(selectedTrackList->begin(), iterator);  
 selectedTrackList->erase(selectedTrackList->begin() + index);  
 track->setSelected(false);  
 }  
 }  
}  
  
void AudioManager::setTrackRangeSelected(Track \*firstTrack, Track \*lastTrack) {  
  
 for (int i = 0; i < int(selectedTrackList->size()); i++ ) {  
 setTrackSelected(selectedTrackList->at(i), false);  
 }  
 selectedTrackList->clear();  
  
 auto firstIterator = std::find(trackList->begin(), trackList->end(), firstTrack);  
 auto lastIterator = std::find(trackList->begin(), trackList->end(), lastTrack);  
  
 int firstIndex;  
 int lastIndex;  
  
 if (firstIterator != trackList->end()) {  
 firstIndex = std::distance(trackList->begin(), firstIterator);  
 } else {  
 return;  
 }  
  
 if (lastIterator != trackList->end()) {  
 lastIndex = std::distance(trackList->begin(), lastIterator);  
 } else {  
 return;  
 }  
  
 for (int i = firstIndex; i < lastIndex; i++) {  
 selectedTrackList->push\_back(trackList->at(i));  
 trackList->at(i)->setSelected(true);  
 }  
}  
  
int AudioManager::getTrackListCount() {  
 return trackList->size();  
}  
  
void AudioManager::scheduleTracks() {  
 for (int i = 0; i < int(trackList->size()); i++) {  
 trackList->at(i)->scheduleAudioRegions();  
 debug::out(3, "Scheduled a track...");  
 }  
}  
  
void AudioManager::cancelTrackPlayback() {  
 for (int i = 0; i < int(trackList->size()); i++) {  
 trackList->at(i)->cancelAudioRegions();  
 debug::out(3, "Cancelling track...");  
 }  
}  
  
void AudioManager::setCurrentGridTime(float \_value) {  
 currentGridTime = \_value;  
}  
  
std::vector<const float \*> AudioManager::getPeaks(std::shared\_ptr<AudioBus> bus) {  
  
 std::vector<const float \*> channelSamples = {};  
  
 std::cout << "Max size" << channelSamples.max\_size();  
  
 for (int channelIdx = 0; channelIdx < (int)bus->numberOfChannels(); channelIdx++) {  
 channelSamples.push\_back(bus->channel(channelIdx)->data());  
 }  
  
 std::cout << "Actual size" << channelSamples.size();  
  
 return channelSamples;  
}  
  
void AudioManager::engageSolo() {  
 soloEnabled = true;  
 for (int i = 0; i < int(trackList->size()); i++) {  
 if (trackList->at(i)->getSolo() == false) {  
 trackList->at(i)->getTrackOutputNode()->gain()->setValue(0.0f);  
 }  
  
 }  
}  
  
void AudioManager::disengageSolo() {  
 soloEnabled = false;  
 for (int i = 0; i < int(trackList->size()); i++) {  
 if (trackList->at(i)->getSolo() == false) {  
 trackList->at(i)->getTrackOutputNode()->gain()->setValue(0.0f);  
 }  
  
 }  
}  
  
void AudioManager::clearAll() {  
 for (auto p : \*trackList) {  
 delete p;  
 }  
 trackList->clear();  
 selectedTrackList->clear();  
 //selectedRegionList->clear();  
}  
  
Track\* AudioManager::getTrackByUUID(QString uuid) {  
 for (int ti= 0; ti < this->getTrackListCount(); ti++) {  
 Track \*track = this->getTrackByIndex(ti);  
 if (track->getUUID() == uuid.toStdString()) {  
 return track;  
 }  
 }  
 return nullptr;  
}  
  
AudioRegion\* AudioManager::getAudioRegionByUUID(QString uuid) {  
 for (int ti= 0; ti < this->getTrackListCount(); ti++) {  
 Track \*track = this->getTrackByIndex(ti);  
 for (int ri = 0; ri < track->getAudioRegionListCount(); ri++) {  
 AudioRegion \*audioRegion = track->getAudioRegionByIndex(ri);  
 if (audioRegion->getUUID() == uuid.toStdString()) {  
 return audioRegion;  
 }  
 }  
 }  
 return nullptr;  
}  
  
void AudioManager::moveRegion(QString uuid, double gridLocation) {  
 AudioRegion \*audioRegion = getAudioRegionByUUID(uuid);  
 if (this->isPlaying == true) {  
 audioRegion->schedule();  
 }  
 audioRegion->setGridLocation(gridLocation);  
 audioRegion->getRegionGraphicItem()->setGridLocation(gridLocation);  
 audioRegion->getRegionGraphicItem()->update();  
}  
  
void AudioManager::setTrackMute(QString uuid, bool mute) {  
 Track \*track = getTrackByUUID(uuid);  
 track->setMute(mute);  
}  
  
void AudioManager::setTrackPan(QString uuid, float pan) {  
 Track \*track = getTrackByUUID(uuid);  
 track->setPan(pan);  
}  
  
void AudioManager::setTrackGain(QString uuid, float gain) {  
 Track \*track = getTrackByUUID(uuid);  
 track->setGain(gain);  
}  
  
void AudioManager::renderAudio(QObject \*parent, std::string fileName, int sampleRate, int channels) {  
  
 qDebug() << "Rendering...";  
 AudioStreamConfig offlineConfig;  
 offlineConfig.device\_index = 0;  
 offlineConfig.desired\_samplerate = sampleRate;  
 offlineConfig.desired\_channels = channels;  
  
 qDebug() << "Config set";  
 rendering = true;  
 stop();  
 eventTimer->start();  
 qDebug() << "Started event timer";  
  
 FileRendering \*fileRendering = new FileRendering(parent, [this] {  
 rendering = false;  
 stop();  
 initContext();  
 dialogs::ProgressDialog::close();  
 dialogs::MessageDialog::show("Done!", "The project has been rendered successfully.", dialogs::MessageDialog::info, dialogs::MessageDialog::okOnly);  
 });  
 dialogs::ProgressDialog::show(0, 60, "Rendering Audio...");  
 fileRendering->operate(this, offlineConfig, fileName);  
 //context.swap(offlineContext);  
}

## audiomanager.h

#ifndef AUDIOMANAGER\_H  
#define AUDIOMANAGER\_H  
  
  
#include <iostream>  
#include <stdio.h>  
#include <chrono>  
#include <ratio>  
#include <thread>  
#include <math.h>  
  
#include "LabSound/LabSound.h"  
  
class Metronome;  
  
#include "metronome.h"  
  
  
  
  
//#include "timer.h"  
  
#include "track.h"  
#include "region.h"  
#include "audioregion.h"  
  
#include "gui/timeline.h"  
#include "gui/mixer.h"  
  
#include "common/audioutil.h"  
#include "common/timer.h"  
#include "common/debug.h"  
#include "common/dialogs.h"  
#include "common/util.h"  
#include "filerendering.h"  
#include "common/timerex.h"  
  
#include "network/session.h"  
  
#include <iostream>  
#include <stdio.h>  
#include <QtGui>  
#include <chrono>  
#include <ratio>  
#include <thread>  
#include <memory>  
  
#include <QJsonObject>  
#include <QJsonDocument>  
  
#include <QThread>  
#include <QUuid>  
  
//class AudioTrackManager;  
//class Track;  
  
class Session;  
class AudioRegion;  
  
using namespace lab;  
//using namespace std::chrono\_literals;  
  
class AudioManager  
{  
public:  
 AudioManager(QWidget \*parent, Timeline &\_timeline);  
  
 void initContext();  
  
 void play();  
 void pause();  
 void stop();  
  
 bool isPlaying;  
  
 void updateSchedule();  
  
 void setDivision(int \_division);  
 void setBPM(double \_beatsPerMinuet);  
 double getBPM();  
 void setLookAhead(double \_value);  
  
 float getCurrentGridTime();  
 void setCurrentGridTime(float \_value);  
  
 double gridTimeToContextSeconds(float \_gridTime);  
 double gridTimeToSeconds(float \_gridTime);  
 float secondsToGridTime(double \_seconds);  
 float getCurrentRelativeTime();  
  
 Track\* addTrack(std::string trackUUID);  
 void removeTrack(Track \*track);  
 Track\* getTrackByIndex(int index);  
  
 Track\* getSelectedTrack(int index);  
 std::vector<class Track\*>\* getSelectedTracks();  
 void setTrackSelected(Track \*track, bool selected);  
 void setTrackRangeSelected(Track \*firstTrack, Track \*lastTrack);  
  
 int getTrackListCount();  
 void scheduleTracks();  
  
 std::shared\_ptr<GainNode> getOutputNode();  
  
 std::shared\_ptr<AudioBus> MakeBusFromSampleFile(std::string fileName);  
  
  
 float startTime;  
 float stopTime;  
  
 std::vector<const float \*> getPeaks(std::shared\_ptr<AudioBus> bus);  
  
 std::shared\_ptr<AudioContext> context;  
  
 void engageSolo();  
 void disengageSolo();  
  
 bool soloEnabled;  
  
 void clearAll();  
  
 Session \*session;  
  
 void moveRegion(QString uuid, double gridLocation);  
 void setTrackMute(QString uuid, bool mute);  
 void setTrackPan(QString uuid, float pan);  
 void setTrackGain(QString uuid, float gain);  
  
 Track\* getTrackByUUID(QString uuid);  
 AudioRegion\* getAudioRegionByUUID(QString uuid);  
  
 void renderAudio(QObject \*parent, std::string fileName, int sampleRate, int channels);  
  
 bool rendering;  
  
 void eventLoop();  
  
  
private:  
 QObject \*parent;  
  
 std::shared\_ptr<GainNode> outputNode;  
  
  
 std::vector<class Track \*> \*trackList;  
 std::vector<class Track \*> \*selectedTrackList;  
  
 std::vector<class Region \*> \*selectedRegionList;  
  
 Metronome \*metronome;  
  
 Timeline \*timeline;  
  
 double bpm;  
 double beatLength;  
 double barLength;  
  
  
 TimerEX \*eventTimer;  
 bool quitLoop;  
  
  
 int division;  
 int currentPos;  
 double lookAhead;  
  
 float currentGridTime;  
 bool scheduled;  
  
 void updateMetSchedule();  
 void cancelTrackPlayback();  
  
  
};  
  
#endif // AUDIOMANAGER\_H

## audioregion.cpp

#include "audioregion.h"  
  
  
AudioRegion::AudioRegion(Timeline \*\_timeline, Track \*\_track, std::string \_uuid) : Region(\_timeline, \_track, \_uuid)  
{  
 debug::out(3, "Audio region added");  
}  
  
void AudioRegion::loadFile(std::string fileName, bool \_progressDialog) {  
  
 debug::out(3, "Begining file loading...");  
  
 progressDialog = \_progressDialog;  
 if (progressDialog == true) {  
 dialogs::ProgressDialog::show(0, 0, "Loading Audio file...");  
 }  
  
  
  
 QFileInfo fileInfo(QString::fromStdString(fileName));  
 setRegionName(fileInfo.fileName().toStdString());  
  
 loadedFileName = fileName;  
  
 fileLoading = new FileLoading(nullptr, std::bind(&AudioRegion::loadedFileCallBack, this));  
 debug::out(3, "Spawining thread...");  
  
 fileLoading->operate(track->getAudioManager(), QString::fromStdString(loadedFileName));  
  
}  
  
void AudioRegion::loadedFileCallBack() {  
 audioClipBus = fileLoading->bus;  
 audioClipNode = fileLoading->node;  
  
 track->getAudioManager()->context->connect(outputNode, audioClipNode);  
  
 updateGridLength();  
 debug::out(3, "Length calculated");  
  
 regionGraphicsItem->setGhost(false);  
 regionGraphicsItem->setGridLength(length);  
 regionGraphicsItem->setWaveform(audioClipBus);  
  
  
  
 if (progressDialog == true) {  
 dialogs::ProgressDialog::close();  
 } else {  
 if (dialogs::ProgressDialog::getValue() + 1 == dialogs::ProgressDialog::getMax()) {  
 dialogs::ProgressDialog::close();  
 } else {  
 qDebug() << "Updating progress..." << dialogs::ProgressDialog::getValue() << dialogs::ProgressDialog::getMax();  
 dialogs::ProgressDialog::updateValue(dialogs::ProgressDialog::getValue() + 1);  
 }  
 }  
  
  
 if (length > timeline->barCount) {  
 timeline->setBarAmount(ceil(length));  
 }  
 timeline->updateViewports();  
 debug::out(3, "Successfully Loaded File!");  
}  
  
void AudioRegion::schedule() {  
 float timeEnd = length + gridLocation;  
  
 {  
 ContextRenderLock r(track->getAudioManager()->context.get(), "Horizon");  
 audioClipNode->reset(r);  
 }  
  
 audioClipNode->initialize();  
  
 audioClipNode->gain()->setValue(1.0f);  
  
 if (track->getAudioManager()->getCurrentGridTime() > gridLocation && track->getAudioManager()->getCurrentGridTime() < timeEnd) {  
  
 debug::out(3, "Scheduled region during playhead");  
 float playheadDiff = track->getAudioManager()->getCurrentGridTime() - gridLocation;  
 audioClipNode->startGrain(track->getAudioManager()->context->currentTime(), track->getAudioManager()->gridTimeToSeconds(playheadDiff));  
 return;  
 }  
  
 if (track->getAudioManager()->getCurrentGridTime() <= gridLocation ) {  
 debug::out(3, "Scheduled region ahead of playhead");  
 double timeToGo = track->getAudioManager()->context->currentTime() + (track->getAudioManager()->gridTimeToSeconds(gridLocation - track->getAudioManager()->getCurrentGridTime()));  
 audioClipNode->start(timeToGo);  
  
 return;  
 }  
}  
  
void AudioRegion::cancelSchedule() {  
  
 audioClipNode->gain()->setValue(0.0f);  
  
 audioClipNode->stop(track->getAudioManager()->context->currentTime());  
 {  
 ContextRenderLock r(track->getAudioManager()->context.get(), "Horizon");  
 audioClipNode->reset(r);  
 }  
  
  
}  
  
void AudioRegion::disconnectTrack() {  
 cancelSchedule();  
 debug::out(3, "Audio Region Disconnect Called --------------");  
 Region::disconnectTrack();  
}  
  
void AudioRegion::setTrack(Track \*\_track) {  
  
  
 {  
 ContextRenderLock r(track->getAudioManager()->context.get(), "Horizon");  
 audioClipNode->reset(r);  
  
 }  
  
 debug::out(3, "Switching Tracks...");  
 //outputNode->uninitialize();  
  
 track->getTrackInputNode()->input(0)->junctionDisconnectAllOutputs();  
  
 \_track->getAudioManager()->context->connect(\_track->getTrackInputNode(), outputNode);  
  
  
  
 audioClipNode->initialize();  
 debug::out(3, "Connected to track");  
 setGain(gain);  
  
 track = \_track;  
}  
  
std::string AudioRegion::getLoadedFileName() {  
 return loadedFileName;  
}  
  
  
void AudioRegion::switchContext(AudioContext \*context) {  
  
}  
  
  
void AudioRegion::updateGridLength() {  
 length = track->getAudioManager()->secondsToGridTime(audioClipNode->duration()) - 1;  
}

## audioregion.h

#ifndef AUDIOREGION\_H  
#define AUDIOREGION\_H  
  
  
  
//class Timeline;  
//class Track;  
//class Region;  
//class RegionGraphicItem;  
  
//#include "LabSound/LabSound.h"  
//#include "track.h"  
#include "track.h"  
//class Region;  
#include "region.h"  
#include "fileloading.h"  
class FileLoading;  
//class Timeline;  
  
//#include <QThread>  
  
using namespace lab;  
  
  
class AudioRegion : public Region  
{  
public:  
 AudioRegion(Timeline \*\_timeline, Track \*\_track, std::string uuid);  
  
 void loadFile(std::string fileName, bool progressDialog);  
 void schedule() override;  
 void cancelSchedule();  
  
 void disconnectTrack() override;  
 void setTrack(Track \*\_track) override;  
  
 void switchContext(AudioContext \*context);  
  
 std::string getLoadedFileName();  
 std::string preLoadedFile;  
  
 void updateGridLength();  
  
private:  
  
 std::shared\_ptr<AudioBus> audioClipBus;  
 std::shared\_ptr<SampledAudioNode> audioClipNode;  
  
 std::string loadedFileName;  
  
  
 //void loadFileThread(std::function<void()> callback);  
 void loadedFileCallBack();  
  
 std::queue<std::function<void()>> callbackQueue;  
  
 FileLoading \*fileLoading;  
  
 bool progressDialog;  
  
 //double duration;  
  
};  
  
  
  
#endif // AudioRegion\_H

## effecttypes.h

#ifndef EFFECTTYPES\_H  
#define EFFECTTYPES\_H  
  
#include "audio\_effects/reverbeffect.h"  
#include "audio\_effects/compressoreffect.h"  
  
enum effectType {reverb, utility, compressor};  
  
#endif // EFFECTTYPES\_H

## fileloading.cpp

#include "fileloading.h"  
  
Q\_DECLARE\_SMART\_POINTER\_METATYPE(std::shared\_ptr)  
Q\_DECLARE\_METATYPE(std::shared\_ptr<AudioBus>)  
Q\_DECLARE\_METATYPE(std::shared\_ptr<SampledAudioNode>)  
Q\_DECLARE\_METATYPE(std::vector<const float \*>)  
  
void FileLoadingThread::doWork(AudioManager \*audioManager, QString loadedFileName) {  
 debug::out(3, "Spawned file handling thread");  
 debug::out(3, "Starting file loading...");  
 std::shared\_ptr<AudioBus> audioClipBus = audioManager->MakeBusFromSampleFile(loadedFileName.toStdString());  
  
 std::shared\_ptr<SampledAudioNode> audioClipNode = std::make\_shared<SampledAudioNode>();  
 {  
 ContextRenderLock r(audioManager->context.get(), "Horizon");  
 audioClipNode->setBus(r, audioClipBus);  
 }  
  
  
 debug::out(3, "Loaded audio, running callback function...");  
  
 emit resultReady(audioClipBus, audioClipNode, audioManager->getPeaks(audioClipBus));  
}  
  
  
  
FileLoading::FileLoading(QObject \*parent, std::function<void()> \_callback) : QObject(parent)  
{  
 qDebug() << "Init file loading";  
  
 qRegisterMetaType<std::shared\_ptr<AudioBus>>();  
 qRegisterMetaType<std::shared\_ptr<SampledAudioNode>>();  
 qRegisterMetaType<std::vector<const float \*>>();  
 callback = \_callback;  
 FileLoadingThread \*flt = new FileLoadingThread;  
 flt->moveToThread(&workerThread);  
 connect(&workerThread, &QThread::finished, flt, &QObject::deleteLater);  
 connect(this, &FileLoading::operate, flt, &FileLoadingThread::doWork);  
 connect(flt, &FileLoadingThread::resultReady, this, &FileLoading::handleResults, Qt::QueuedConnection);  
 workerThread.start();  
}  
  
FileLoading::~FileLoading() {  
 workerThread.quit();  
 workerThread.wait();  
}  
  
void FileLoading::handleResults(std::shared\_ptr<AudioBus> \_bus, std::shared\_ptr<SampledAudioNode> \_node, std::vector<const float \*> \_peaks) {  
 bus = \_bus;  
 node = \_node;  
 peaks = \_peaks;  
 callback();  
}

## fileloading.h

#ifndef FILELOADING\_H  
#define FILELOADING\_H  
  
#include <QObject>  
#include <QThread>  
#include "audiomanager.h"  
  
  
class FileLoadingThread : public QObject {  
 Q\_OBJECT  
  
public slots:  
 void doWork(AudioManager \*audioManager, QString loadedFileName);  
signals:  
 void resultReady(std::shared\_ptr<AudioBus> bus, std::shared\_ptr<SampledAudioNode> node, std::vector<const float \*> peaks);  
};  
  
  
class FileLoading : public QObject  
{  
 Q\_OBJECT  
  
public:  
 FileLoading(QObject \*parent = nullptr, std::function<void()> callback = nullptr);  
 ~FileLoading();  
 std::shared\_ptr<AudioBus> bus;  
 std::shared\_ptr<SampledAudioNode> node;  
 std::vector<const float \*> peaks;  
private:  
 std::function<void()> callback;  
 QThread workerThread;  
public slots:  
 void handleResults(std::shared\_ptr<AudioBus> \_bus, std::shared\_ptr<SampledAudioNode> \_node, std::vector<const float \*> \_peaks);  
signals:  
 void operate(AudioManager \*audioManager, QString loadedFileName);  
};  
  
#endif // FILELOADING\_H

## filerendering.cpp

#include "filerendering.h"  
  
Q\_DECLARE\_SMART\_POINTER\_METATYPE(std::shared\_ptr)  
Q\_DECLARE\_METATYPE(std::shared\_ptr<AudioContext>)  
Q\_DECLARE\_METATYPE(AudioStreamConfig)  
Q\_DECLARE\_METATYPE(std::string);  
  
void FileRenderingThread::doWork(AudioManager \*audioMan, AudioStreamConfig config, std::string fileName) {  
 debug::out(3, "Spawned file render thread");  
 debug::out(3, "Starting file Rendering...");  
  
  
 audioMan->context = lab::MakeOfflineAudioContext(config, 60000.f);  
  
 auto recorder = std::make\_shared<RecorderNode>(config);  
  
 audioMan->context->addAutomaticPullNode(recorder);  
 recorder->startRecording();  
  
  
 debug::out(3, "Starting offline playback...");  
 audioMan->context->connect(recorder, audioMan->getOutputNode());  
  
 audioMan->play();  
  
  
  
 audioMan->context->offlineRenderCompleteCallback = [&recorder, config, this, audioMan, fileName] {  
 recorder->stopRecording();  
 audioMan->context->removeAutomaticPullNode(recorder);  
 //context->removeAutomaticPullNode(recorder);  
 debug::out(3, "Writing to wav file...");  
 recorder->writeRecordingToWav(fileName);  
 debug::out(3, "All done!");  
  
 emit this->resultReady();  
 };  
  
 audioMan->context->startOfflineRendering();  
}  
  
  
  
FileRendering::FileRendering(QObject \*parent, std::function<void()> \_callback) : QObject(parent)  
{  
 qRegisterMetaType<std::shared\_ptr<AudioContext>>();  
 qRegisterMetaType<AudioStreamConfig>();  
 qRegisterMetaType<std::string>();  
  
 callback = \_callback;  
 FileRenderingThread \*rt = new FileRenderingThread;  
 rt->moveToThread(&workerThread);  
  
 connect(&workerThread, &QThread::finished, rt, &QObject::deleteLater);  
 connect(this, &FileRendering::operate, rt, &FileRenderingThread::doWork);  
 connect(rt, &FileRenderingThread::resultReady, this, &FileRendering::handleResults, Qt::QueuedConnection);  
 workerThread.start();  
}  
  
FileRendering::~FileRendering() {  
 workerThread.quit();  
 workerThread.wait();  
}  
  
void FileRendering::handleResults() {  
 callback();  
}

## filerendering.h

#ifndef FILERENDERING\_H  
#define FILERENDERING\_H  
  
#include <QObject>  
#include <QThread>  
#include "audiomanager.h"  
  
  
class FileRenderingThread : public QObject {  
 Q\_OBJECT  
  
public slots:  
 void doWork(AudioManager \*audioMan, AudioStreamConfig config, std::string fileName);  
signals:  
 void resultReady();  
};  
  
  
class FileRendering : public QObject  
{  
 Q\_OBJECT  
  
public:  
 FileRendering(QObject \*parent = nullptr, std::function<void()> callback = nullptr);  
 ~FileRendering();  
  
private:  
 std::function<void()> callback;  
 QThread workerThread;  
public slots:  
 void handleResults();  
signals:  
 void operate(AudioManager \*audioMan, AudioStreamConfig config, std::string fileName);  
};  
  
#endif // FILERendering\_H

## indexingthread.cpp

#include "indexingthread.h"  
  
IndexingThread::IndexingThread(QObject \*parent, QDir dir, bool \_topLevelSpecial) : QThread(parent)  
{  
 parentDir = dir;  
 topLevelSpecial = \_topLevelSpecial;  
}  
  
void IndexingThread::run() {  
 isTopLevelSet = false;  
 QTreeWidgetItem \*widgetItem = scanDir(parentDir);  
 emit resultReady(widgetItem);  
}  
  
QTreeWidgetItem\* IndexingThread::scanDir(QDir dir) {  
 QTreeWidgetItem \*folder = new QTreeWidgetItem();  
 if (topLevelSpecial && topLevelSpecial) {  
 isTopLevelSet = true;  
 folder->setIcon(0, QIcon(samplesIcon));  
 } else {  
 folder->setIcon(0, QIcon(folderIcon));  
 }  
  
 folder->setText(0, dir.dirName());  
  
  
  
 dir.setFilter(QDir::AllDirs | QDir::NoDotAndDotDot | QDir::NoSymLinks);  
 QStringList dirList = dir.entryList();  
 foreach(QString dirName, dirList) {  
 QString newPath = QString("%1/%2").arg(dir.absolutePath()).arg(dirName);  
 folder->addChild(scanDir(QDir(newPath)));  
 }  
  
 dir.setNameFilters(QStringList() << "\*.mp3" << "\*.wav");  
 dir.setFilter(QDir::Files | QDir::NoDotAndDotDot | QDir::NoSymLinks);  
  
 foreach(QString filename, dir.entryList()) {  
  
 QFileInfo fileInfo(dir.path() + "/" + filename);  
 QTreeWidgetItem \*audioFile = new QTreeWidgetItem();  
 audioFile->setText(0, fileInfo.fileName());  
 audioFile->setText(1, fileInfo.filePath());  
  
 QMimeDatabase db;  
 QMimeType mime = db.mimeTypeForFile(fileInfo.filePath(), QMimeDatabase::MatchContent);  
  
 if (mime.preferredSuffix() == "wav") {  
 audioFile->setIcon(0, QIcon(wavIcon));  
 } else if (mime.preferredSuffix() == "mp3") {  
 audioFile->setIcon(0, QIcon(mp3Icon));  
 }  
 folder->addChild(audioFile);  
 }  
  
 return folder;  
}

## indexingthread.h

#ifndef INDEXINGTHREAD\_H  
#define INDEXINGTHREAD\_H  
  
#include <QObject>  
#include <QThread>  
#include <QDir>  
#include <QTreeWidgetItem>  
#include <QMimeDatabase>  
  
  
class IndexingThread : public QThread  
{  
 Q\_OBJECT  
public:  
 explicit IndexingThread(QObject \*parent = nullptr, QDir dir = QDir(), bool topLevelSpecial = false);  
 void run() override;  
 QTreeWidgetItem\* scanDir(QDir dir);  
  
 QString wavIcon;  
 QString mp3Icon;  
 QString folderIcon;  
 QString samplesIcon;  
  
 QDir parentDir;  
  
  
  
signals:  
 void resultReady(QTreeWidgetItem \*treeWidgetItem);  
private:  
 bool topLevelSpecial;  
 bool isTopLevelSet;  
};  
  
#endif // INDEXINGTHREAD\_H

## metronome.cpp

#include "metronome.h"  
  
Metronome::Metronome(std::shared\_ptr<GainNode> \_outputNode, AudioManager \*\_audioMan)  
{  
 outputNode = \_outputNode;  
 audioMan = \_audioMan;  
  
 #ifndef \_WIN32  
 metPrimaryBus = audioMan->MakeBusFromSampleFile(util::getResourceBundle() + "/core/metronome/Primary.wav");  
 metSecondaryBus = audioMan->MakeBusFromSampleFile(util::getResourceBundle() + "/core/metronome/Secondary.wav");  
 #else  
 metPrimaryBus = audioMan->MakeBusFromSampleFile(util::getInstallDir() + "/core/metronome/Primary.wav");  
 metSecondaryBus = audioMan->MakeBusFromSampleFile(util::getInstallDir() + "/core/metronome/Secondary.wav");  
 #endif  
 metPrimaryNode = std::make\_shared<SampledAudioNode>();  
 {  
 ContextRenderLock r(audioMan->context.get(), "horizon");  
 metPrimaryNode->setBus(r, metPrimaryBus);  
 }  
  
 metSecondaryNode = std::make\_shared<SampledAudioNode>();  
 {  
 ContextRenderLock r(audioMan->context.get(), "horizon");  
 metSecondaryNode->setBus(r, metSecondaryBus);  
 }  
  
 audioMan->context->connect(outputNode, metPrimaryNode);  
 audioMan->context->connect(outputNode, metSecondaryNode);  
}  
  
  
void Metronome::schedulePrimary(double when) {  
 //metPrimaryNode->start(audioMan->gridTimeToContextSeconds(when));  
}  
  
void Metronome::scheduleSecondary(std::vector<double> \_scheduleQueue) {  
 scheduleQueue = \_scheduleQueue;  
 scheduleQueueTimes.clear();  
  
 for (std::vector<double>::size\_type i = 0; i != scheduleQueue.size(); i++) {  
 scheduleQueueTimes.insert(scheduleQueueTimes.end(), audioMan->gridTimeToContextSeconds(scheduleQueue.at(i)));  
 }  
 //qDebug() << scheduleQueueTimes;  
  
  
}  
  
void Metronome::update() {  
 if (scheduleQueueTimes.size() != 0) {  
 double timeTillClick = scheduleQueueTimes.at(0) - audioMan->context->currentTime();  
 if (timeTillClick < 0.1) {  
 if (nextSchedule == true) {  
 double time = scheduleQueueTimes.at(0);  
 //metSecondaryNode->start(time);  
 scheduleQueueTimes.erase(scheduleQueueTimes.begin());  
 nextSchedule = false;  
 }  
 } else {  
 nextSchedule = true;  
 }  
 //qDebug() << timeTillClick;  
 }  
}

## metronome.h

#ifndef METRONOME\_H  
#define METRONOME\_H  
  
class AudioManager;  
  
#include "audiomanager.h"  
  
using namespace lab;  
  
  
class Metronome  
{  
public:  
 Metronome(std::shared\_ptr<GainNode> \_outputNode, AudioManager \*audioMan);  
 bool scheduled;  
 void schedulePrimary(double when);  
 void scheduleSecondary(std::vector<double> \_scheduleQueue);  
  
 void update();  
  
private:  
 std::shared\_ptr<AudioBus> metPrimaryBus;  
 std::shared\_ptr<SampledAudioNode> metPrimaryNode;  
  
 std::shared\_ptr<AudioBus> metSecondaryBus;  
 std::shared\_ptr<SampledAudioNode> metSecondaryNode;  
  
  
 std::shared\_ptr<GainNode> outputNode;  
  
 AudioManager \*audioMan;  
  
 std::vector<double> scheduleQueue;  
 std::vector<double> scheduleQueueTimes;  
  
 bool nextSchedule;  
  
  
};  
  
#endif // METRONOME\_H

## projectserialization.cpp

#include "projectserialization.h"  
  
ProjectSerialization::ProjectSerialization()  
{  
 tempFileList = {};  
}  
  
  
std::string ProjectSerialization::serialize(AudioManager &audioMan, bool epoch) {  
  
 QJsonDocument jsonDocument;  
 QJsonObject root;  
  
 debug::out(3, "Starting deserialisation...");  
 root.insert("Application", "Horizon");  
 root.insert("tempo", audioMan.getBPM());  
  
 std::chrono::milliseconds ms = std::chrono::duration\_cast< std::chrono::milliseconds >(  
 std::chrono::system\_clock::now().time\_since\_epoch()  
 );  
 if (epoch == true) {  
 root.insert("TS\_EPOCH", QString::fromStdString(std::to\_string(ms.count())));  
 }  
  
  
 QJsonArray trackArray;  
  
 for(int i = 0; i < audioMan.getTrackListCount(); i++) {  
  
 QJsonObject trackObject;  
 Track \*track = audioMan.getTrackByIndex(i);  
 trackObject.insert("type", "track");  
 trackObject.insert("uuid", QString::fromStdString(track->getUUID()));  
 trackObject.insert("index", track->getIndex());  
 trackObject.insert("mute", track->getMute());  
 trackObject.insert("gain", QString::number(track->getGain()));  
 trackObject.insert("pan", QString::number(track->getPan()));  
 trackObject.insert("color", track->getColor().name(QColor::HexRgb));  
  
 QJsonArray audioRegionArray;  
  
 for(int j = 0; j < track->getAudioRegionListCount(); j++) {  
  
 QJsonObject audioRegionObject;  
 AudioRegion \*audioRegion = track->getAudioRegionByIndex(j);  
 audioRegionObject.insert("type", "audioRegion");  
 audioRegionObject.insert("uuid", QString::fromStdString(audioRegion->getUUID()));  
 audioRegionObject.insert("gridLocation", QString::number(audioRegion->getGridLocation()));  
 if (copyToTemp == true) {  
  
 QByteArray byteArray = fileChecksum(QString::fromStdString(audioRegion->getLoadedFileName()), QCryptographicHash::Sha1);  
 QString checkSUM = QString::fromUtf8(byteArray.toHex());  
 QString tempFilePath = "/" + sessionID + "/" + checkSUM + "/" + QFileInfo(QString::fromStdString(audioRegion->getLoadedFileName())).fileName();  
  
 bool exists = false;  
 for (int i = 0; i < int(tempFileList.size()); i++) {  
 if (tempFileList.at(i).at(1) == checkSUM) {  
 exists = true;  
 }  
 }  
 if (!exists) {  
 QString tempDir = QStandardPaths::writableLocation(QStandardPaths::MusicLocation) + "/Horizon";  
  
 QString dirPath = QStandardPaths::writableLocation(QStandardPaths::MusicLocation) + "/Horizon/" + sessionID + "/" + checkSUM;  
 QDir dir(dirPath);  
 dir.mkpath(dirPath);  
  
  
 if (QFile::copy(QString::fromStdString(audioRegion->getLoadedFileName()), tempDir + tempFilePath)) {  
 debug::out(3, "Coppied source file to temp session directory");  
 } else {  
 debug::out(1, "Could not copy source file to temp session directory");  
 }  
  
 QList<QString> list;  
 list.append(tempFilePath);  
 list.append(checkSUM);  
 tempFileList.push\_back(list);  
 }  
 audioRegionObject.insert("filePath", tempFilePath);  
 audioRegionObject.insert("tempLocation", true);  
 } else {  
 audioRegionObject.insert("filePath", QString::fromStdString(audioRegion->getLoadedFileName()));  
 audioRegionObject.insert("tempLocation", false);  
 }  
  
 audioRegionArray.append(audioRegionObject);  
 }  
  
 trackObject.insert("audioRegions", audioRegionArray);  
 trackArray.append(trackObject);  
 }  
  
 root.insert("tracks", trackArray);  
 jsonDocument.setObject(root);  
  
 return jsonDocument.toJson().toStdString();  
  
}  
  
void ProjectSerialization::deSerialize(std::string json, AudioManager &audioMan) {  
 QJsonDocument jsonDocument = QJsonDocument::fromJson(QString::fromStdString(json).toUtf8());  
 QJsonObject root = jsonDocument.object();  
  
 audioMan.setBPM(root.value("tempo").toDouble());  
  
 for (int i = 0; i < root.value("tracks").toArray().size(); i++) {  
  
 QJsonObject trackJSON = root.value("tracks").toArray().at(i).toObject();  
 if (trackJSON.value("type") == "track") {  
 debug::out(3, "Adding track");  
 QString trackUuid;  
 if (trackJSON.value("uuid").toString() == "") {  
 trackUuid = QUuid::createUuid().toString();  
 } else {  
 trackUuid = trackJSON.value("uuid").toString();  
 }  
 Track \*track = audioMan.addTrack(trackUuid.toStdString());  
  
 for (int ar = 0; ar < trackJSON.value("audioRegions").toArray().size(); ar++) {  
 QJsonObject audioRegionJSON = trackJSON.value("audioRegions").toArray().at(ar).toObject();  
  
 if (audioRegionJSON.value("type").toString() == "audioRegion") {  
 debug::out(3, "Adding audio region");  
 QString regionUuid;  
 if (audioRegionJSON.value("uuid").toString() == "") {  
 regionUuid = QUuid::createUuid().toString();  
 } else {  
 regionUuid = audioRegionJSON.value("uuid").toString();  
 }  
 AudioRegion \*audioRegion = track->addAudioRegion(regionUuid.toStdString());  
 audioRegion->setGridLocation(std::stod(audioRegionJSON.value("gridLocation").toString().toStdString()));  
 qDebug() << QString::fromStdString(audioRegionJSON.value("filePath").toString().toStdString());  
 if (audioRegionJSON.value("tempLocation").toBool()) {  
 QString tempDir = QStandardPaths::writableLocation(QStandardPaths::MusicLocation) + "/Horizon";  
 audioRegion->preLoadedFile = (tempDir + audioRegionJSON.value("filePath").toString()).toStdString();  
 } else {  
 audioRegion->preLoadedFile = audioRegionJSON.value("filePath").toString().toStdString();  
 }  
  
 }  
 }  
  
 track->setGain(std::stof(trackJSON.value("gain").toString().toStdString()));  
 track->setPan(std::stof(trackJSON.value("pan").toString().toStdString()));  
 track->setMute(trackJSON.value("mute").toBool());  
 QColor color;  
 color.setNamedColor(trackJSON.value("color").toString());  
 track->setColor(color);  
 }  
 }  
}  
  
bool ProjectSerialization::compaire(std::string a, std::string b) {  
 QJsonDocument aJSON = QJsonDocument::fromJson(QString::fromStdString(a).toUtf8());  
 QJsonDocument bJSON = QJsonDocument::fromJson(QString::fromStdString(b).toUtf8());  
  
 QString aCompact = aJSON.toJson(QJsonDocument::Compact);  
 QString bCompact = bJSON.toJson(QJsonDocument::Compact);  
  
 return aCompact == bCompact;  
}  
  
QByteArray ProjectSerialization::fileChecksum(const QString &fileName,  
 QCryptographicHash::Algorithm hashAlgorithm)  
{  
 QFile f(fileName);  
 if (f.open(QFile::ReadOnly)) {  
 QCryptographicHash hash(hashAlgorithm);  
 if (hash.addData(&f)) {  
 return hash.result();  
 }  
 }  
 return QByteArray();  
}

## projectserialization.h

#ifndef PROJECTSERIALIZATION\_H  
#define PROJECTSERIALIZATION\_H  
  
#include <QObject>  
#include <QJsonDocument>  
#include <QJsonObject>  
#include <QCryptographicHash>  
#include <chrono>  
#include "app/audiomanager.h"  
  
class ProjectSerialization : public QObject  
{  
 Q\_OBJECT  
public:  
 ProjectSerialization();  
 std::string serialize(AudioManager &audioMan, bool epoch);  
 void deSerialize(std::string json, AudioManager &audioMan);  
  
 bool compaire(std::string a, std::string b);  
  
 bool copyToTemp = false;  
  
 QList<QList<QString>> tempFileList;  
  
 QString sessionID = "";  
  
private:  
  
 QByteArray fileChecksum(const QString &fileName, QCryptographicHash::Algorithm hashAlgorithm);  
};  
  
#endif // PROJECTSERIALIZATION\_H

## region.cpp

#include "region.h"  
  
  
Region::Region(Timeline \*\_timeline, Track \*\_track, std::string \_uuid)  
{  
 uuid = \_uuid;  
 timeline = \_timeline;  
 track = \_track;  
 outputNode = std::make\_shared<GainNode>();  
 setGain(1.0f);  
 track->getAudioManager()->context->connect(track->getTrackInputNode(), outputNode);  
 gridLocation = 1;  
 selected = false;  
}  
  
Region::~Region() {  
 delete regionGraphicsItem;  
 track->getAudioManager()->context->disconnect(track->getTrackOutputNode(), outputNode);  
}  
  
Track\* Region::getTrack() {  
 return track;  
}  
  
Timeline\* Region::getTimeline() {  
 return timeline;  
}  
  
RegionGraphicItem\* Region::getRegionGraphicItem() {  
 return regionGraphicsItem;  
}  
  
void Region::setRegionGraphicItem(RegionGraphicItem \*rgi) {  
 regionGraphicsItem = rgi;  
}  
  
void Region::disconnectTrack() {  
 track->getAudioManager()->context->disconnect(track->getTrackInputNode(), outputNode);  
 debug::out(3, "Disconnected from track");  
}  
  
void Region::setTrack(Track \*\_track) {  
 track->getAudioManager()->context->connect(\_track->getTrackInputNode(), outputNode);  
 debug::out(3, "Connected to track");  
 setGain(gain);  
 track = \_track;  
}  
  
double Region::getGridLocation() {  
 return gridLocation;  
}  
  
void Region::setGridLocation(double time) {  
 gridLocation = time;  
 debug::out(3, "Grid location just set!");  
}  
  
double Region::getGridLength() {  
 return length;  
}  
  
void Region::setGridLength(double value) {  
 length = value;  
}  
  
void Region::schedule() {  
  
}  
  
void Region::setGain(float \_gain) {  
 gain = \_gain;  
 outputNode->gain()->setValue(\_gain);  
}  
  
float Region::getGain() {  
 return gain;  
}  
  
std::shared\_ptr<GainNode> Region::getOutputNode() {  
 return outputNode;  
}  
  
std::string Region::getRegionName() {  
 return regionName;  
}  
  
void Region::setRegionName(std::string \_name) {  
 regionName = \_name;  
}  
  
std::string Region::getUUID() {  
 return uuid;  
}  
  
bool Region::getSelected() {  
 return selected;  
}  
  
void Region::setSelected(bool \_selected) {  
 selected = \_selected;  
 if (selected) {  
 qDebug() << "REGION IS SELECTED";  
 }  
 regionGraphicsItem->setSelected(\_selected);  
}

## region.h

#ifndef REGION\_H  
#define REGION\_H  
  
//#include <QGraphicsItem>  
  
  
#include "LabSound/LabSound.h"  
//#include <QString>  
//#include "audioregion.h"  
  
  
//class Timeline;  
//class Track;  
//class Region;  
//class RegionGraphicItem;  
//class AudioManager;  
  
//#include "track.h"  
  
class AudioManager;  
//#include "audiomanager.h"  
class Track;  
  
  
class Timeline;  
class RegionGraphicItem;  
//#include "regiongraphicitem.h"  
  
using namespace lab;  
  
class Region  
{  
public:  
 Region(Timeline \*\_timeline, Track \*\_track, std::string uuid);  
 ~Region();  
  
 virtual Track\* getTrack();  
 virtual Timeline\* getTimeline();  
 virtual RegionGraphicItem\* getRegionGraphicItem();  
 virtual void setRegionGraphicItem(RegionGraphicItem \*rgi);  
 virtual void setTrack(Track \*\_track);  
  
 virtual void disconnectTrack();  
  
 virtual void setGridLocation(double time);  
 virtual double getGridLocation();  
  
 virtual void setGridLength(double value);  
 virtual double getGridLength();  
  
 virtual void schedule();  
  
 virtual float getGain();  
 virtual void setGain(float \_gain);  
  
 virtual std::string getRegionName();  
 virtual void setRegionName(std::string \_name);  
  
 virtual std::shared\_ptr<GainNode> getOutputNode();  
  
 virtual std::string getUUID();  
  
 virtual bool getSelected();  
 virtual void setSelected(bool \_selected);  
  
protected:  
  
  
  
 Track \*track;  
 Timeline \*timeline;  
 RegionGraphicItem \*regionGraphicsItem;  
  
 std::shared\_ptr<GainNode> outputNode;  
  
 double gridLocation;  
 double length;  
  
 std::string regionName;  
  
 float gain;  
  
 std::string uuid;  
  
 bool selected;  
  
};  
  
#include "track.h"  
  
#endif // REGION\_H

## track.cpp

#include "track.h"  
  
Track::Track(Timeline &\_timeLine, AudioManager &\_audioMan, std::string \_uuid) {  
  
 debug::out(3, "Creating track");  
 audioMan = &\_audioMan;  
 debug::out(3, "setting timeline");  
 timeline = &\_timeLine;  
 debug::out(3, "setting input node");  
 trackInputNode = std::make\_shared<GainNode>();  
 debug::out(3, "setting output node");  
  
 uuid = \_uuid;  
  
 trackOutputNode = std::make\_shared<GainNode>();  
 pannerNode = std::make\_shared<StereoPannerNode>();  
 Lanalyser = std::make\_shared<AnalyserNode>();  
 Ranalyser = std::make\_shared<AnalyserNode>();  
  
 Lanalyser->setSmoothingTimeConstant(0.0);  
 Ranalyser->setSmoothingTimeConstant(0.0);  
  
 channelSplitter = std::make\_shared<ChannelSplitterNode>(2);  
 channelMerger = std::make\_shared<ChannelMergerNode>(2);  
  
  
 trackInputNode->gain()->setValue(1.0f);  
 trackOutputNode->gain()->setValue(1.0f);  
  
 audioMan->context.get()->connect(trackOutputNode, trackInputNode);  
 audioMan->context.get()->connect(pannerNode, trackOutputNode);  
 audioMan->context.get()->connect(channelSplitter, pannerNode);  
  
 //channelSplitter->addOutputs(2);  
  
 audioMan->context.get()->connect(Lanalyser, channelSplitter, 0, 0);  
 audioMan->context.get()->connect(Ranalyser, channelSplitter, 0, 1);  
  
 audioMan->context.get()->connect(channelMerger, Lanalyser, 0, 0);  
 audioMan->context.get()->connect(channelMerger, Ranalyser, 1, 0);  
  
  
 audioMan->context->connect(audioMan->getOutputNode(), channelMerger);  
  
  
  
 selected = false;  
 regionList = new std::vector<class Region \*>;  
 selectedRegionList = new std::vector<class Region \*>;  
  
 //setGain(0.0f);  
 // setPan(0.0f);  
 //gain = 1.0f;  
 peakdB = -100;  
 setMute(false);  
 setGain(0.0);  
 setPan(0.0);  
  
}  
  
  
  
Track::~Track() {  
  
 for (auto r : \*regionList) {  
 delete r;  
 }  
 delete trackControlWidget;  
 delete mixerChannelWidget;  
 delete trackGraphicItem;  
  
 audioMan->context->disconnect(audioMan->getOutputNode(), trackOutputNode);  
 audioMan->context->disconnect(trackInputNode, trackOutputNode);  
 qDebug() << "distroying";  
}  
  
void Track::setTrackControlsWidget(TrackControlsWidget \*\_tcw) {  
 trackControlWidget = \_tcw;  
  
}  
  
void Track::setMixerChannelWidget(MixerChannelWidget \*\_mcw) {  
 mixerChannelWidget = \_mcw;  
}  
  
void Track::setTrackGraphicsItem(TrackGraphicItem \*\_tgi) {  
 trackGraphicItem = \_tgi;  
}  
  
void Track::setHScaleFactor(int \_hScaleFactor) {  
  
}  
  
AudioRegion\* Track::addAudioRegion(std::string regionUUID) {  
  
 AudioRegion \*audioRegion = new AudioRegion(timeline, this, regionUUID);  
 regionList->push\_back(audioRegion);  
 return audioRegion;  
}  
  
void Track::setRegion(Region \*\_region) {  
  
 regionList->insert(regionList->end(), \_region);  
}  
  
void Track::removeRegion(Region \*\_region) {  
  
 qDebug() << "Removing Region... IDX" << index;  
 regionList->erase(regionList->begin() + getIndexByRegion(\_region));  
 trackInputNode->uninitialize();  
  
 Lanalyser->uninitialize();  
 Ranalyser->uninitialize();  
  
 qDebug() << "Track connections before:" << trackInputNode->numberOfInputs();  
 audioMan->context->disconnect(trackInputNode, \_region->getOutputNode());  
  
 qDebug() << "Track connections after:" << trackInputNode->numberOfInputs();  
 trackInputNode->initialize();  
  
 Lanalyser->initialize();  
 Ranalyser->initialize();  
}  
  
AudioManager\* Track::getAudioManager() {  
 return audioMan;  
}  
  
int Track::getIndex() {  
 return index;  
}  
  
void Track::setIndex(int \_index) {  
 index = \_index;  
}  
  
void Track::setSelected(bool \_selected) {  
 selected = \_selected;  
 trackControlWidget->setSelected(selected);  
 mixerChannelWidget->setSelected(selected);  
}  
  
bool Track::getSelected() {  
 return selected;  
}  
  
int Track::getIndexByRegion(Region \*region) {  
 auto iterator = std::find(regionList->begin(), regionList->end(), region);  
  
 if (iterator != regionList->end()) {  
 return std::distance(regionList->begin(), iterator);  
 } else {  
 return -1;  
 }  
  
}  
  
std::shared\_ptr<GainNode> Track::getTrackInputNode() {  
 return trackInputNode;  
}  
  
std::shared\_ptr<GainNode> Track::getTrackOutputNode() {  
 return trackOutputNode;  
}  
  
void Track::scheduleAudioRegions() {  
 for (int i = 0; i < int(regionList->size()); i++) {  
 AudioRegion\* audioRegion = dynamic\_cast<AudioRegion\*>(regionList->at(i));  
 //double contextLocation = audioMan->gridTimeToContextSeconds(audioRegion->getGridLocation()) - audioMan->getCurrentRelativeTime();  
 audioRegion->schedule();  
 debug::out(3, "Scheduled a region...");  
 }  
}  
  
void Track::cancelAudioRegions() {  
 for (int i = 0; i < int(regionList->size()); i++) {  
 AudioRegion\* audioRegion = dynamic\_cast<AudioRegion\*>(regionList->at(i));  
 audioRegion->cancelSchedule();  
 debug::out(3, "Cancelling a region...");  
 }  
}  
  
//void Track::removeRegion(int position) {  
// std::vector<class Region \*>::iterator it = std::find(regionList->begin(), regionList->end(), \_region);  
// if (it != regionList->end()) {  
//  
// regionList->erase(std::distance(regionList->begin(), it));  
// }  
//}  
  
void Track::setGain(float \_value) {  
 gain = pow(10, (\_value / 20));  
 gainNonLog = \_value;  
 qDebug() << "Setting Gain" << gain;  
 if (!mute) {  
 trackOutputNode->gain()->setValue(gain);  
 }  
  
}  
  
float Track::getGain() {  
  
 return gainNonLog;  
}  
  
void Track::setPan(float \_value) {  
 pan = \_value;  
 pannerNode->pan()->setValue(\_value);  
}  
  
float Track::getPan() {  
 return pan;  
}  
  
void Track::setMute(bool \_mute) {  
 mute = \_mute;  
 if (mute == true) {  
 trackOutputNode->gain()->setValue(0.0f);  
 } else {  
 trackOutputNode->gain()->setValue(gain);  
 }  
}  
  
void Track::setSolo(bool \_solo) {  
 solo = \_solo;  
 if (solo == true) {  
  
 }  
}  
  
bool Track::getMute() {  
 return mute;  
}  
  
bool Track::getSolo() {  
 return solo;  
}  
  
QColor Track::getColor() {  
 return color;  
}  
  
void Track::setColor(QColor \_color) {  
 color = \_color;  
}  
  
void Track::updateColor(QColor \_color) {  
 color = \_color;  
 trackControlWidget->updateColor();  
 mixerChannelWidget->updateColor();  
}  
  
std::vector<int> Track::getLMeterData() {  
  
 std::vector<float> buffer(2048);  
  
 Lanalyser->getFloatTimeDomainData(buffer);  
  
 //analyser->getFloatFrequencyData(buffer);  
 // qDebug() << "BUFFER" << buffer[0];  
  
 float sumOfSquares = 0;  
 for (int i = 0; i < (int)buffer.size(); i++) {  
 sumOfSquares += pow(buffer[i], 2);  
  
 }  
 //qDebug() << "SOS" << sumOfSquares;  
  
 float avgPowerDecibels = 10 \* log10(sumOfSquares / buffer.size());  
 int peakInstantaneousPower = 0;  
  
 for (int i = 0; i < (int)buffer.size(); i++) {  
 int power = pow(buffer[i], 2);  
 peakInstantaneousPower = max(power, peakInstantaneousPower);  
 }  
  
 float peakInstantaneousPowerDecibels = 10 \* log10(peakInstantaneousPower);  
  
 if (avgPowerDecibels >= peakdB) {  
 peakdB = std::ceil(avgPowerDecibels \* 100.0) / 100.0;  
  
 }  
  
  
  
 return std::vector<int> {static\_cast<int>(round(avgPowerDecibels)), static\_cast<int>(round(peakInstantaneousPowerDecibels))};  
  
}  
  
std::vector<int> Track::getRMeterData() {  
  
 std::vector<float> buffer(2048);  
  
 Ranalyser->getFloatTimeDomainData(buffer);  
  
 //analyser->getFloatFrequencyData(buffer);  
 // qDebug() << "BUFFER" << buffer[0];  
  
 float sumOfSquares = 0;  
 for (int i = 0; i < (int)buffer.size(); i++) {  
 sumOfSquares += pow(buffer[i], 2);  
  
 }  
 //qDebug() << "SOS" << sumOfSquares;  
  
 float avgPowerDecibels = 10 \* log10(sumOfSquares / buffer.size());  
 int peakInstantaneousPower = 0;  
  
 for (int i = 0; i < (int)buffer.size(); i++) {  
 int power = pow(buffer[i], 2);  
 peakInstantaneousPower = max(power, peakInstantaneousPower);  
 }  
  
 float peakInstantaneousPowerDecibels = 10 \* log10(peakInstantaneousPower);  
  
 if (avgPowerDecibels >= peakdB) {  
 peakdB = std::ceil(avgPowerDecibels \* 100.0) / 100.0;  
  
 }  
 return std::vector<int> {static\_cast<int>(round(avgPowerDecibels)), static\_cast<int>(round(peakInstantaneousPowerDecibels))};  
  
}  
  
  
int Track::getAudioRegionListCount() {  
 return regionList->size();  
}  
  
AudioRegion\* Track::getAudioRegionByIndex(int index) {  
 return dynamic\_cast<AudioRegion\*>(regionList->at(index));  
}  
  
std::string Track::getUUID() {  
 return uuid;  
}  
  
void Track::uiUpdate() {  
 trackControlWidget->uiUpdate();  
 mixerChannelWidget->uiUpdate();  
}  
  
Region\* Track::getSelectedRegion(int index) {  
 if (selectedRegionList->size() != 0) {  
 return selectedRegionList->at(index);  
 } else {  
 return nullptr;  
 }  
}  
  
void Track::setRegionSelected(Region \*region, bool selected) {  
 if (selected == true) {  
 debug::out(3, "Pushing region to vector...");  
 for(int t = 0; t < audioMan->getTrackListCount(); t++) {  
 Track \*track = audioMan->getTrackByIndex(t);  
 for (int i = 0; i < int(track->selectedRegionList->size()); i++ ) {  
 track->setRegionSelected(track->selectedRegionList->at(i), false);  
 }  
 }  
  
 selectedRegionList->clear();  
 selectedRegionList->push\_back(region);  
 debug::out(3, "Setting region as selected...");  
 region->setSelected(true);  
 } else {  
 auto iterator = std::find(selectedRegionList->begin(), selectedRegionList->end(), region);  
  
 if (iterator != selectedRegionList->end()) {  
 int index = std::distance(selectedRegionList->begin(), iterator);  
 selectedRegionList->erase(selectedRegionList->begin() + index);  
 region->setSelected(false);  
 }  
 }  
}  
  
AudioEffect\* Track::addAudioEffect(effectType type, std::string uuid) {  
 if (uuid == "") {  
 uuid = "testUUID";  
 }  
  
 if (type == effectType::compressor) {  
 CompressorEffect \*compressorEffect = new CompressorEffect(mixerChannelWidget);  
 audioEffectChain.push\_back(compressorEffect);  
 compressorEffect->showEffectWindow();  
 }  
}

## track.h

#ifndef TRACK\_H  
#define TRACK\_H  
  
//#include "timeline.h"  
#include <QWidget>  
#include <QGraphicsScene>  
#include <QGraphicsView>  
#include <QTransform>  
#include <QDebug>  
#include <QMouseEvent>  
#include <QKeyEvent>  
#include <QDebug>  
#include <QPen>  
#include <QBrush>  
#include <QGraphicsItem>  
#include <QPoint>  
#include <QColor>  
#include <iostream>  
  
  
#include <vector>  
#include <memory>  
#include <cstdio>  
#include <fstream>  
#include <cassert>  
#include <functional>  
#include <math.h>  
#include "LabSound/LabSound.h"  
  
//#include "audioregion.h"  
//class Region;  
//class AudioRegion;  
  
class AudioManager;  
class Region;  
#include "audiomanager.h"  
#include "audioeffect.h"  
  
#include "effecttypes.h"  
  
class AudioRegion;  
  
class TrackControlsWidget;  
class MixerChannelWidget;  
class TrackGraphicItem;  
class Timeline;  
  
//#include "trackcontrolswidget.h"  
//#include "trackgraphicitem.h"  
  
//#include "LabSound/LabSound.h"  
  
using namespace lab;  
  
class Track  
{  
public:  
 Track(Timeline &\_timeLine, AudioManager &\_audioMan, std::string uuid);  
 ~Track();  
  
 void setSelected(bool \_selected);  
 bool getSelected();  
  
 void setTrackControlsWidget(TrackControlsWidget \*\_tcw);  
 void setMixerChannelWidget(MixerChannelWidget \*\_mcw);  
 void setTrackGraphicsItem(TrackGraphicItem \*\_tgi);  
  
 QColor getColor();  
 void setColor(QColor \_color);  
 void updateColor(QColor \_color);  
  
 // void setColorTheme(QColor primaryColor);  
 void setHScaleFactor(int \_hScaleFactor);  
  
 AudioRegion\* addAudioRegion(std::string regionUUID);  
 void setRegion(Region \*\_region);  
 void removeRegion(Region \*\_region, Track \*newTrack);  
  
 int getIndex();  
 void setIndex(int \_index);  
  
 AudioRegion\* getAudioRegionByIndex(int index);  
 int getAudioRegionListCount();  
  
  
 int getIndexByRegion(Region \*region);  
  
 void removeRegion(Region \*\_region);  
  
 void scheduleAudioRegions();  
 void cancelAudioRegions();  
  
 std::shared\_ptr<GainNode> getTrackInputNode();  
 std::shared\_ptr<GainNode> getTrackOutputNode();  
  
  
  
  
 AudioManager\* getAudioManager();  
  
 std::string getUUID();  
  
 void setMute(bool \_mute);  
 void setSolo(bool \_solo);  
  
 bool getMute();  
 bool getSolo();  
  
 void setGain(float \_value);  
 float getGain();  
  
 void setPan(float \_value);  
 float getPan();  
  
 std::vector<int> getLMeterData();  
 std::vector<int> getRMeterData();  
  
 float peakdB;  
  
 void uiUpdate();  
  
 Region\* getSelectedRegion(int index);  
 void setRegionSelected(Region \*region, bool selected);  
  
  
  
 AudioEffect\* addAudioEffect(effectType type, std::string uuid = "");  
 void showEffectWindow(AudioEffect \*effect);  
  
private:  
 bool selected;  
 int index;  
  
  
 std::vector<class Region \*> \*regionList;  
 std::vector<class Region \*> \*selectedRegionList;  
  
 std::vector<class AudioEffect \*> audioEffectChain;  
  
 std::string uuid;  
  
 std::shared\_ptr<GainNode> trackInputNode;  
 std::shared\_ptr<GainNode> trackOutputNode;  
  
 std::shared\_ptr<AnalyserNode> Lanalyser;  
 std::shared\_ptr<AnalyserNode> Ranalyser;  
  
 std::shared\_ptr<ChannelSplitterNode> channelSplitter;  
 std::shared\_ptr<ChannelMergerNode> channelMerger;  
  
 std::shared\_ptr<StereoPannerNode> pannerNode;  
  
 AudioManager \*audioMan;  
  
 Timeline \*timeline;  
  
 TrackControlsWidget \*trackControlWidget;  
 MixerChannelWidget \*mixerChannelWidget;  
 TrackGraphicItem \*trackGraphicItem;  
  
 QColor color;  
  
  
  
 bool mute = false;  
 bool solo;  
  
 float gain = 0.0f;  
 float gainNonLog = 0.0f;  
 float pan = 0.0f;  
  
  
};  
  
#endif // TRACK\_H