SDA ASSIGNMENT REPORT

UFCF94-15-3

Software Development for Audio

João Maurício - Student number 18030724

INTRODUCTION

This report explores the development of an audio application for guitar players to practice with, which includes different guitar pedals, as well as an amplifier section and a speaker cabinet simulator.

The main motivations behind this idea were to explore the DSP classes within JUCE and how they can be used, as well implementing as many techniques as possible from the Software and Development for Audio lectures in a greater scale than in the exercises we were given.

The report will present a basic end-user manual, which will be proceeded by an explanation of the structure of the app aimed at other developers. Following from that, the conclusion will allow for some reflection on what was created and mention some future improvements to the application.

TABLE OF CONTENTS

USER MANUAL

The user interface consists of three main parts, top to bottom: **Audio settings**, **Pedal slots** and **Amp/Cabinet Selector.** It also includes an input level meter on top of the Pedal slots and one output meter below the Amp/Cabinet Selector (refer to figure 1).

In the **Audio Settings**, the user can choose the input and outputs that they wish to use, as well as changing the sample rate and buffer size of the audio device.

Following the signal path of the signal, the **Pedal Slots** section is where the user is presented with a selection of guitar pedals they can choose from, simply by selecting them from the four combo boxes on top of the pedals. If the user prefers not to have any pedal on a specific slot, simply choose “No Pedal” from the respective combo box.

In the **Amp** area, the user is presented with a simple amplifier, with the ability to boost or decrease the volume from the pedals, as well as controlling three filters (bass, mids and treble).

The last main section of the interface is the **Cab Simulator**, which is presented to the user as a combo box below the Amp in which they can choose from four different speaker cabinet profiles, altering the output of the processing chain.

Figure 1 - User Interface

SYSTEM DOCUMENTATION

This application is heavily based on an object oriented structure, and it further extends the use of polymorphism that was explored in the lectures.

As suggested in the feedback from the presentation on the 27th January 2021, the GUI and Audio parts of the application are separated and are now easier to maintain and alter without affecting each other.

Starting with MainComponent (GUI), in Figure 2 we can distinguish the three different main sections mentioned in the User Manual, being that AudioDeviceSelectorComponent and AudioDeviceInfoDisplay make up the Audio Settings. MainComponent also has one instance of CabSimulatorUI and AmpUI, and both of these classes have a reference to their respective Audio counterpart (CabSimulatorAudio and AmpAudio).

Regarding the guitar pedals, MainComponent has four instances of pedalUI (actual class named Pedal) that each hold a reference to a pedalAudio (actual class named dsp::ProcessorBase). Both of these classes are polymorphic, being that they can easily take the form of different pedals regarding the UI and the Audio, so that the user is able to choose different pedals in the interface.

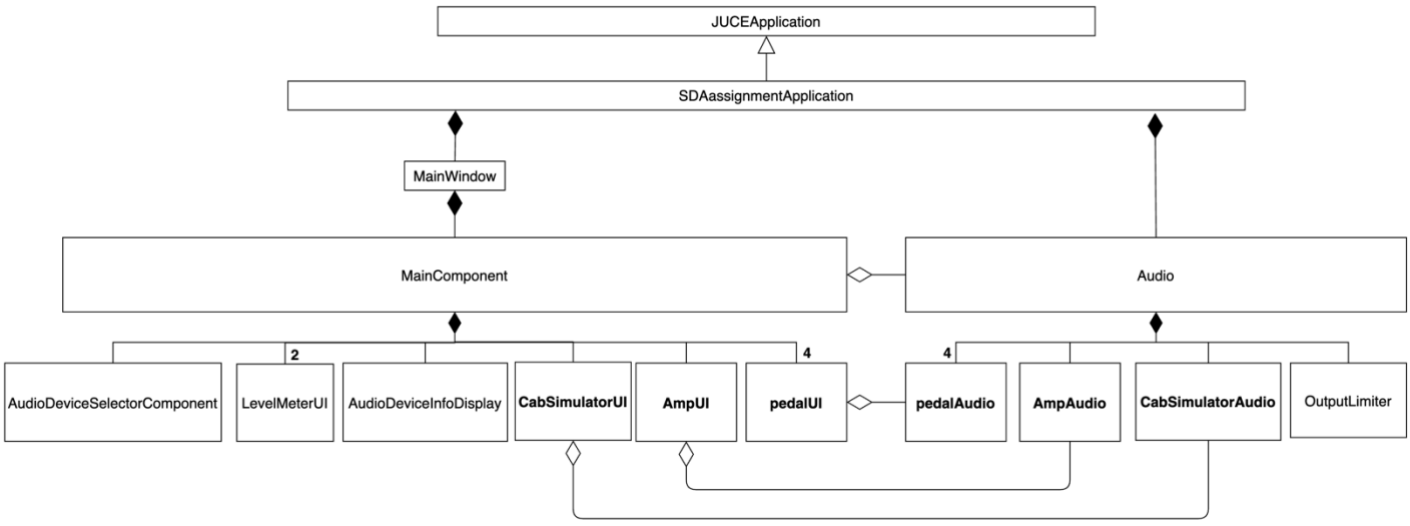

Figure 2 - Overview of system structure

As evidenced in Figure 3, we can see that all of the pedal UIs inherit from the Pedal class, as most of them are very similar to each other, with a few sliders and a toggle. This makes it very simple to have four different pedal slots in the interface, each being able to take the form of any of pedal UIs through polymorphism. It also includes a BlankPedalUI which doesn't have a reference to any audio counterpart, as it's simply used to display a blank UI whenever the user chooses not to have a pedal in a specific pedal slot.

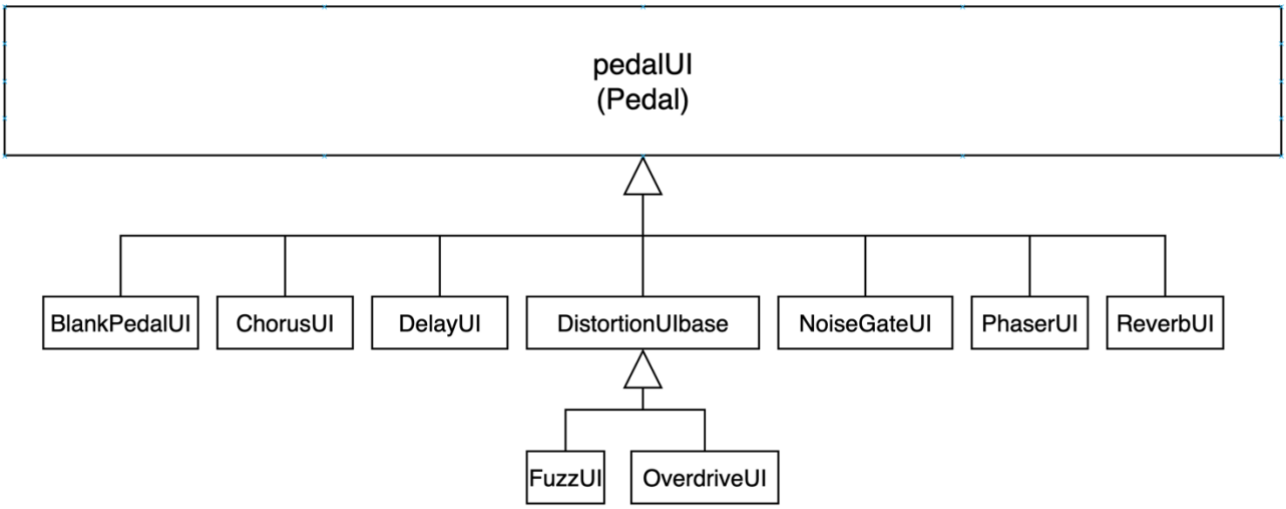

Figure 3 - Pedal UI inheritance

The same principle applies to pedalAudio (actual class named dsp::ProcessorBase) as all of the audio pedals inherit from it, as seen in Figure 4. This allows for the UI pedal changes to also change the signal path of the audio (e.g. reverb before or after overdrive).

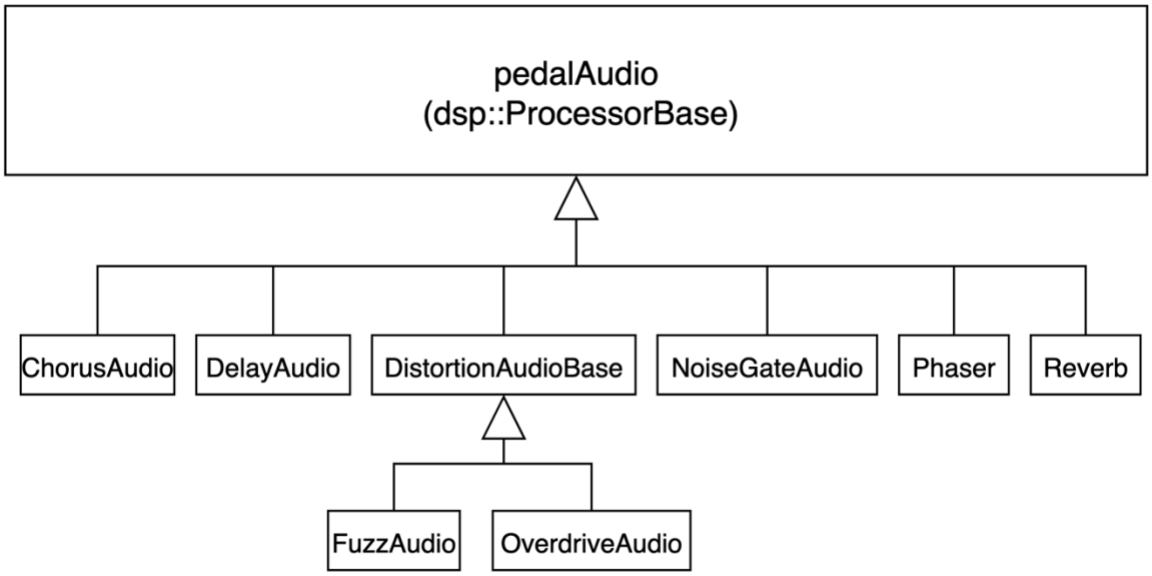

Figure 4 - dsp::ProcessorBase inheritance

CONCLUSION

This assignment proved to be incredibly beneficial for my learning experience with both C++ and JUCE, given that the majority of the material from the lectures was included in the final code (e.g. polymorphism, templates, atomics, smart pointers, etc.) alongside parts of JUCE that were not used in the practicals (DSP classes, OwnedArrays, BinaryData, etc.).

Learning how to properly use the DSP processors within JUCE was definitely a challenge as it did not seem clear how to implement them in getNextAudioBlock(). However, the JUCE forum proved to be a great resource.

Also, being that the prototype was mainly based around the AudioAppComponent, separating GUI and Audio proved to be a very lengthy process after that, as a lot of the work needed to be undone. Nonetheless, it proved to be an advantage in the end as a lot more material from the lectures was allowed to be covered.

Overall, it proved to be a very successful project given the amount of work that needed to be done, which allowed for a steep learning curve with JUCE and C++.

FUTURE DEVELOPMENTS

- Being able to drag pedals in and out of the UI pedal slots, making it easier for the user to change the signal path;
- Possibly re-arranging the project in a different way so that it doesn't need these many files in the source folder;
- Fixing two remaining race conditions which were detected through Xcode's Thread Sanitizer, occurring because of AudioDeviceInfoDisplay and CabSimulator.

APPENDIX

Class Documentation

AmpAudio Class Reference

Detailed Description

Class for the audio processing that the Amp performs.

It uses a dsp::ProcessorChain that consists of gain and three filters.

Member Function Documentation

void AmpAudio::prepare (const dsp::ProcessSpec & *spec*)

Initialises the dsp processors in the dsp::ProcessorChain object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void AmpAudio::process (dsp::ProcessContextReplacing< float > *context*)

Processes the input samples supplied in the context through the processing chain (gain, low shelf filter, mids peak filter and high shelf filter)

Parameters

context	contains the input and output samples to be processed

void AmpAudio::reset ()

Resets the internal state of the dsp processors in the processor chain.

AmpUI Class Reference

Detailed Description

Class for the UI of the Amp, which changes the parameters of a **AmpAudio** object that it's assigned to control.

See also

setAmpToControl

Member Function Documentation

void AmpUI::setAmpToControl (AmpAudio * *amp*)

Sets the **AmpAudio** object that the **AmpUI** will control.

Parameters

amp	pointer to the **AmpAudio** object to control

Audio Class Reference

Detailed Description

Brings all of the audio processing together in the getNextAudioBlock() callback.

Current processing chain is pedal1, pedal2, pedal3, pedal4, amp, cab simulator and output limiter

Public Types

- enum { **NumberOfPedals** = 4 }
  Number of pedals to process the audio input through.
- enum **PedalsAvailable** { **Chorus**, **Reverb**, **Overdrive**, **NoiseGate**, **Delay**, **Fuzz**, **Phaser** }
  Pedals available to process audio in this project.

Public Attributes

- std::atomic< bool > **isPedalToggleOn** [**NumberOfPedals**]
  Holds the toggle state of each of the 4 pedals in the interface, to determine if a specific slot should be bypassed in the audio processing.

- AudioDeviceManager & **deviceManager**

Member Enumeration Documentation

anonymous enum

Number of pedals to process the audio input through.

Enumerator:

NumberOfPedals	

enum Audio::PedalsAvailable

Pedals available to process audio in this project.

Enumerator:

Chorus	
Reverb	

Overdrive	
NoiseGate	
Delay	
Fuzz	
Phaser	

Member Function Documentation

AmpAudio * Audio::getAmp ()

Returns the audio object for the **AmpUI** to control.

Returns

pointer to the **AmpAudio** object

CabSimulatorAudio * Audio::getCabSimulator ()

Returns the audio object for the **CabSimulatorUI** to control.

Returns

pointer to the **CabSimulatorAudio** object

ChorusAudio * Audio::getChorusAudio ()

Returns the audio object for the **ChorusUI** to control.

Returns

pointer to the **ChorusAudio** object

DelayAudio * Audio::getDelay ()

Returns the audio object for the **DelayUI** to control.

Returns

pointer to the **DelayAudio** object

FuzzAudio * Audio::getFuzzDistortionAudio ()

Returns the audio object for the **FuzzUI** to control.

Returns

pointer to the **FuzzAudio** object

NoiseGateAudio * Audio::getNoiseGate ()

Returns the audio object for the **NoiseGateUI** to control.

Returns

pointer to the **NoiseGateAudio** object

OverdriveAudio * Audio::getOverdriveDistortionAudio ()

Returns the audio object for the **OverdriveUI** to control.

Returns
pointer to the **OverdriveAudio** object

PhaserAudio * Audio::getPhaserAudio ()

Returns the audio object for the **PhaserUI** to control.

Returns
pointer to the **PhaserAudio** object

ReverbAudio * Audio::getReverbAudio ()

Returns the audio object for the **ReverbUI** to control.

Returns
pointer to the **ReverbAudio** object

void Audio::setPedal (int *pedalIndexToChange*, PedalsAvailable *pedalToChangeTo*)

Sets the pedalAudio[pedalIndexToChange] to be another pedalAudio instead.

Parameters

pedalIndexToChange	index of the pedalAudio to change
pedalToChangeTo	type of audio pedal to change pedalAudio[pedalIndexToChange] to

---

Member Data Documentation

AudioDeviceManager& Audio::deviceManager

std::atomic<bool> Audio::isPedalToggleOn[NumberOfPedals]

Holds the toggle state of each of the 4 pedals in the interface, to determine if a specific slot should be bypassed in the audio processing.

AudioDeviceInfoDisplay Class Reference

Detailed Description

Class adapted from JUCE's tutorial "Tutorial: The AudioDeviceManagerClass to display the current settings of the AudioDeviceManager it mirrors.

Member Function Documentation

void AudioDeviceInfoDisplay::dumpDeviceInfo ()

Outputs the AudioDeviceManager's current settings as logMessages.
It should be called in a changeListenerCallback

void AudioDeviceInfoDisplay::setDeviceManager (AudioDeviceManager * *deviceManagerToDisplay*)

Sets the AudioDeviceManager object that the **AudioDeviceInfoDisplay** will mirror.

Parameters

deviceManagerToDisplay	pointer to the **ChorusAudio** object to display

BlankPedalUI Class Reference

Detailed Description

Class for the UI of a blank pedal, for when the user decides not to choose a pedal in a pedal slot.

CabSimulatorAudio Class Reference

Detailed Description

Class for the audio processing that the CabSimulator performs, through dsp::Convolution.
It has pointers to 4 files from BinaryData that correspond to 4 different Impulse Responses

Member Function Documentation

void CabSimulatorAudio::setImpulseResponse (int *fileIndexChosen*)

Sets the impulse response to use.

Parameters

fileIndexChosen	index from the 4 impulse responses available to use

CabSimulatorUI Class Reference

Detailed Description

Class for the UI of a cab simulator, which changes the impulse response that a **CabSimulatorAudio** object is using through a comboBox.

Member Function Documentation

void CabSimulatorUI::setCabSimulatorToControl (CabSimulatorAudio * *cabSimulator*)

Sets the **CabSimulatorAudio** object that the **CabSimulatorUI** will control.

Parameters

cabSimulator	pointer to the **CabSimulatorAudio** object to control

ChorusAudio Class Reference

Detailed Description

Class for the audio processing that the Chorus pedal performs, using an instance of dsp::Chorus.

Member Function Documentation

void ChorusAudio::prepare (const dsp::ProcessSpec & *spec*) `[override]`

Initialises the dsp::Chorus object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void ChorusAudio::process (const dsp::ProcessContextReplacing< float > & *context*) `[override]`

Processes the input samples supplied in the context through the dsp::Chorus object.

Parameters

context	contains the input and output samples to be processed

void ChorusAudio::reset () `[override]`

Resets the dsp::Chorus instance internal state.

Member Data Documentation

std::atomic<float> ChorusAudio::depthValue

std::atomic<float> ChorusAudio::mixValue

std::atomic<float> ChorusAudio::rateValue

ChorusUI Class Reference

Detailed Description

Class for the UI of the chorus pedal, which changes the parameters of a **ChorusAudio** object that it's assigned to control.

Member Function Documentation

void ChorusUI::setChorusToControl (ChorusAudio * *chorus*)

Sets the **ChorusAudio** object that the **ChorusUI** will control.

Parameters

chorus	pointer to the **ChorusAudio** object to control

DelayAudio Class Reference

Detailed Description

Class for the audio processing that the Delay pedal performs, using an instance of dsp::DelayLine.

Member Function Documentation

void DelayAudio::prepare (const dsp::ProcessSpec & *spec*) `[override]`

Initialises the dsp::DelayLine object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void DelayAudio::process (const dsp::ProcessContextReplacing< float > & *context*) `[override]`

Processes the input samples supplied in the context sample-by-sample through a delay line.

Parameters

context	contains the input and output samples to be processed

void DelayAudio::reset () `[override]`

Resets the dsp::DelayLine instance internal state.

Member Data Documentation

std::atomic<int> DelayAudio::delayInSamples

std::atomic<float> DelayAudio::feedbackAmount

DelayUI Class Reference

Detailed Description

Class for the UI of the delay pedal, which changes the parameters of a **DelayAudio** object that it's assigned to control.

Member Function Documentation

void DelayUI::setDelayToControl (DelayAudio * *delay*)

Sets the **DelayAudio** object that the **DelayUI** will control.

Parameters

delay	pointer to the **DelayAudio** object to control

DistortionAudioBase Class Reference

Detailed Description

Base class for the audio processing of the two distortion pedals used (Overdrive and Fuzz).

Its processing chain consists of two dsp::Gain objects (preGain and postGain) and an instance of dsp::WaveShaper

---

Member Function Documentation

void DistortionAudioBase::prepare (const dsp::ProcessSpec & *spec*)`[override]`

Initialises the DSP objects.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void DistortionAudioBase::process (const dsp::ProcessContextReplacing< float > & *context*)`[override]`

Processes the input samples supplied in the context through the processing chain (preGain, waveshaper and postGain)

Parameters

context	contains the input and output samples to be processed

void DistortionAudioBase::reset () `[override]`

Resets the DSP instances internal state.

Member Data Documentation

std::atomic<float> DistortionAudioBase::postGainValue

std::atomic<float> DistortionAudioBase::preGainValue

dsp::WaveShaper<float> DistortionAudioBase::waveshaper

DistortionUIbase Class Reference

Detailed Description

Base class for the UI of the two distortion pedals (Overdrive and Fuzz), which will both have two similar sliders(preGain and outputGain.

FuzzAudio Class Reference

Detailed Description

Class for the audio processing of the Fuzz pedal.

Inherits from **DistortionAudioBase** and changes the transfer function that the waveshaper from **DistortionAudioBase** will use

---

FuzzUI Class Reference

Detailed Description

Class for the UI of the fuzz pedal, which changes the parameters of a **FuzzAudio** object that it's assigned to control.

Member Function Documentation

void FuzzUI::setDistortionToControl (FuzzAudio * *fuzz*)

Sets the **FuzzAudio** object that the **FuzzUI** will control.

Parameters

fuzz	pointer to the **FuzzAudio** object to control

LevelMeter Class Reference

Public Types

- enum **MeterOutputOrInput** { **Input**, **Output** }

Public Member Functions

- **LevelMeter** (AudioDeviceManager &m, **MeterOutputOrInput** outputOrInput)
  LevelMeter *constructor.*

Detailed Description

Class copied and extended from AudioDeviceSelectorComponent::SimpleDeviceManagerInputLevelMeter to also be able to display output levels from the AudioDeviceManager.

Constructor & Destructor Documentation

LevelMeter::LevelMeter (AudioDeviceManager & *m*, MeterOutputOrInput *outputOrInput*)

LevelMeter constructor.

Parameters

m	reference to AudioDeviceManager to use for metering
outputOrInput	will this meter display the AudioDeviceManager input or output

MainComponent Class Reference

Public Types

- enum { **NumberOfPedals** = 4 }
  Number of pedal slots to display.

Detailed Description

Brings together all of the components of the GUI.

Member Function Documentation

void MainComponent::buttonClicked (Button * *toggle*) `[override]`

Listens to the pedal toggles to turn their audio processing on/off.

void MainComponent::changeListenerCallback (ChangeBroadcaster *) `[override]`

Called when there is a change in the current AudioDeviceManager.

void MainComponent::comboBoxChanged (ComboBox * *comboBoxThatHasChanged*) `[override]`

Listens to changes in the pedal slots combo boxes to set pedalUI and pedalAudio to specific pedal types.

void MainComponent::handleAsyncUpdate () `[override]`, `[virtual]`

Used to work around the fact that the size of an AudioDeviceSelectorComponent changes its own size depending on some settings chosen.

void MainComponent::paint (Graphics & *g*) `[override]`

Paints the main UI.

void MainComponent::resized () `[override]`

Called when the size of the window changes.

Used to set the bounds of all components in **MainComponent**

NoiseGateAudio Class Reference

Detailed Description

Class for the audio processing that the NoiseGate pedal performs, using an instance of dsp::NoiseGate.

Member Function Documentation

void NoiseGateAudio::prepare (const dsp::ProcessSpec & *spec*) `[override]`

Initialises the dsp::NoiseGate object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void NoiseGateAudio::process (const dsp::ProcessContextReplacing< float > & *context*) `[override]`

Processes the input samples supplied in the context through the dsp::NoiseGate object.

Parameters

context	contains the input and output samples to be processed

void NoiseGateAudio::reset () `[override]`

Resets the dsp::NoiseGate instance internal state.

NoiseGateUI Class Reference

Detailed Description

Class for the UI of the noise gate pedal, which changes the parameters of a **NoiseGateAudio** object that it's assigned to control.

Member Function Documentation

void NoiseGateUI::setNoiseGateToControl (NoiseGateAudio * *noiseGate*)

Sets the **NoiseGateAudio** object that the **NoiseGateUI** will control.

Parameters

noiseGate	pointer to the **NoiseGateAudio** object to control

---

OutputLimiter Struct Reference

Simple hard limiter that inherits from dsp::WaveShaper to restrict the samples that it outputs to an amplitude of -1.0 to 1.0.

Inheritance diagram for OutputLimiter:

Public Member Functions

- **OutputLimiter** ()
  ChorusAudio *constructor*.

Detailed Description

Simple hard limiter that inherits from dsp::WaveShaper to restrict the samples that it outputs to an amplitude of -1.0 to 1.0.

OverdriveAudio Class Reference

Detailed Description

Class for the audio processing of the Overdrive pedal.

Inherits from **DistortionAudioBase** and changes the transfer function that the waveshaper from **DistortionAudioBase** will use

OverdriveUI Class Reference

Detailed Description

Class for the UI of the overdrive pedal, which changes the parameters of a **OverdriveAudio** object that it's assigned to control.

Member Function Documentation

void OverdriveUI::setDistortionToControl (OverdriveAudio * *overdrive*)

Sets the **OverdriveAudio** object that the **OverdriveUI** controls.

Parameters

overdrive	pointer to the **OverdriveAudio** object to control

Pedal Class Reference

Base class for the UI of all of the pedals.

`#include <Pedal.h>`

Inheritance diagram for Pedal:

Public Member Functions

- **Pedal** (int numOfSliders)
  Pedal *constructor, where the number of sliders needed for the pedal UI is requested.*

Public Attributes

- ToggleButton **pedalToggle**
- OwnedArray< Slider > **slider**
  Array for all of the sliders requested in ***Pedal****'s constructor.*

- OwnedArray< Label > **sliderLabel**

Detailed Description

Base class for the UI of all of the pedals.

Has a variable number of sliders (which the class inheriting from **Pedal** will determine) and a toggle that controls an assigned audio pedal class

Constructor & Destructor Documentation

Pedal::Pedal (int *numOfSliders*)

Pedal constructor, where the number of sliders needed for the pedal UI is requested.

Parameters

numOfSliders	number of sliders that the class inheriting from **Pedal** will need

Member Data Documentation

ToggleButton Pedal::pedalToggle

OwnedArray<Slider> Pedal::slider

Array for all of the sliders requested in **Pedal**'s constructor.

See also

Pedal(int numOfSliders)

OwnedArray<Label> Pedal::sliderLabel

PhaserAudio Class Reference

Detailed Description

Class for the audio processing that the Phaser pedal performs, using an instance of dsp::Phaser.

Member Function Documentation

void PhaserAudio::prepare (const dsp::ProcessSpec & *spec*) `[override]`

Initialises the dsp::Phaser object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void PhaserAudio::process (const dsp::ProcessContextReplacing< float > & *context*) `[override]`

Processes the input samples supplied in the context through the dsp::Phaser object.

Parameters

context	contains the input and output samples to be processed

void PhaserAudio::reset () `[override]`

Resets the dsp::Phaser instance internal state.

Member Data Documentation

std::atomic<float> PhaserAudio::depthValue

std::atomic<float> PhaserAudio::mixValue

std::atomic<float> PhaserAudio::rateValue

PhaserUI Class Reference

Detailed Description

Class for the UI of the phaser pedal, which changes the parameters of a **PhaserAudio** object that it's assigned to control.

Member Function Documentation

void PhaserUI::setPhaserToControl (PhaserAudio * *phaser*)

Sets the **PhaserAudio** object that the **PhaserUI** will control.

Parameters

phaser	pointer to the **PhaserAudio** object to control

ReverbAudio Class Reference

Detailed Description

Class for the audio processing that the Reverb pedal performs, using an instance of dsp::Reverb.

The reverb parameters (wet, room size, damping, stereo width) are held by an instance of Reverb::Parameters

---

Member Function Documentation

void ReverbAudio::prepare (const dsp::ProcessSpec & *spec*) [override]

Initialises the dsp::Reverb object.

Parameters

spec	holds information such as number of channels to process, sample rate and maximum block size

void ReverbAudio::process (const dsp::ProcessContextReplacing< float > & *context*) [override]

Processes the input samples supplied in the context through the dsp::Reverb object.

Parameters

context	contains the input and output samples to be processed

void ReverbAudio::reset () [override]

Resets the dsp::Reverb instance internal state.

Member Data Documentation

std::atomic<float> ReverbAudio::dampingValue

Reverb::Parameters ReverbAudio::reverbParameters

Holds all of the parameters for the dsp::Reverb object to use.

std::atomic<float> ReverbAudio::roomSizeValue

std::atomic<float> ReverbAudio::stereoWidthValue

std::atomic<float> ReverbAudio::wetValue

ReverbUI Class Reference

Detailed Description

Class for the UI of the reverb pedal, which changes the parameters of a **ReverbAudio** object that it's assigned to control.

Member Function Documentation

void ReverbUI::setReverbToControl (ReverbAudio * *reverb*)

Sets the **ReverbAudio** object that the **ReverbUI** will control.

Parameters

reverb	pointer to the **ReverbAudio** object to control