Orbit

Faculty of Environment and Technology

University of the West of England

Software Development for Audio

Year 3

26th of January 2017

Index

1. Introduction

Orbit is a flexible 8 step sequencer that combines the percussive sounds of drum kits and bass-led melodies. Orbit was inspired by the Novation Launchpad and is a tool that allows users from any background to interact and create new loops, melodies and sequences without needing a deep understanding of music or technology.

2. User Manual

2.1 Orbit's Interface

When you open Orbit, you will be faced with an interface like the one illustrated in figure 1.

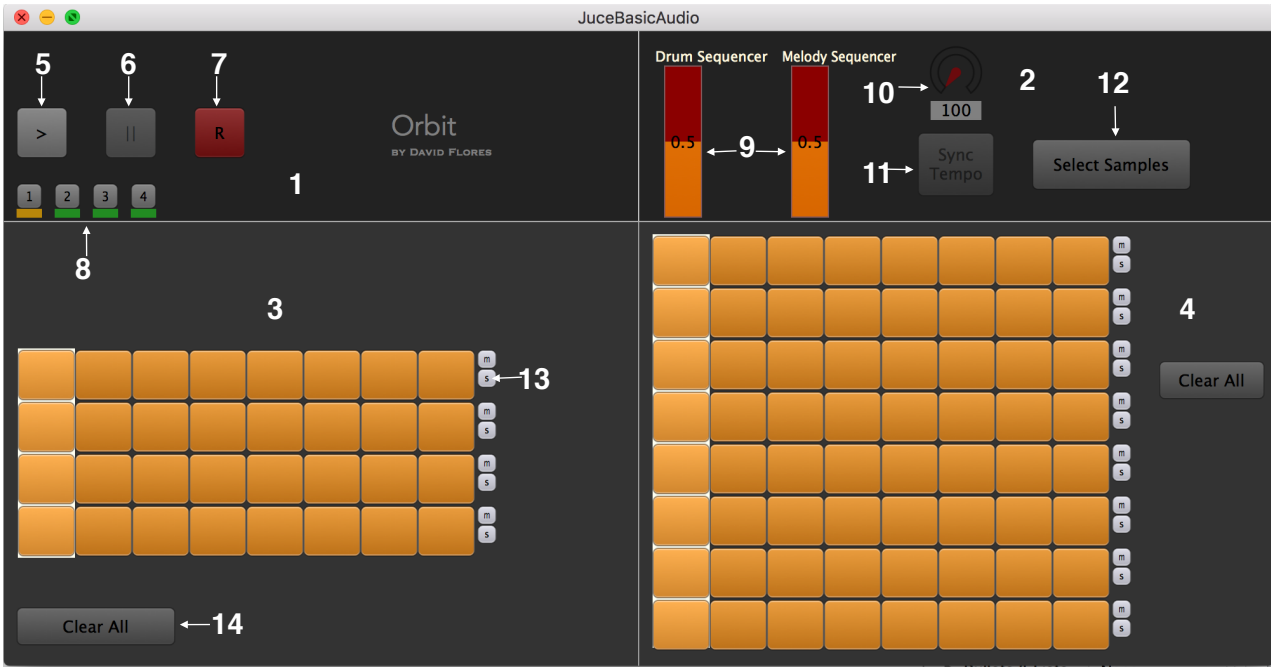

Figure 1. Orbit's interface

The interface is composed by the following:

1. Controls: In here you can play, pause, record or change the scenes of your sequencer.
2. Mixer: In here you can control the level of the drums and the melody, adjust the tempo and select the samples that you want to use.
3. Drum Sequencer: This is where the drum sounds will be played, the orange squares are buttons that you can toggle in order to listen to the sounds, the top row is the kick, second is the snare, third Hi-hat and fourth clap or overheads.
4. Melody Sequencer: Similar to the Drum Sequencer, it is tuned to the scale of E minor. Like a guitar, the lowest note is at the top and the highest at the bottom.
5. Play Button: Press this button to start the sequencer.
6. Pause Button: Press this button to pause the sequencer.

7. Record Button: Press this button to start recording. For more information about recording go to section 2.5.
8. Scenes Button: Toggle these buttons to change between the different scenes. For more information about scenes go to section 2.3.
9. Level faders: These faders control the gain of each sequencer.
10. Tempo knob: Use this knob to adjust the tempo.
11. Sync Tempo Button: Press this button to change the tempo of the sequencer to the value provided by the tempo knob.
12. Select Samples Button: Press this button to select the samples that will be used by the sequencer, for more information on selecting samples go to section 2.2.
13. Mute and Solo Buttons: These buttons will mute (button with letter ‘M’) or solo (button with letter ’S’) that row of the sequencer.
14. Clear All Button: Erases the information in that sequencer.

2.2 Getting Started

Before you start using Orbit, it is important to load the samples that the sequencer will use, in order to do that, follow the next steps:

1. Press the ‘Select Samples’ Button, you will be prompted with a screen like this:

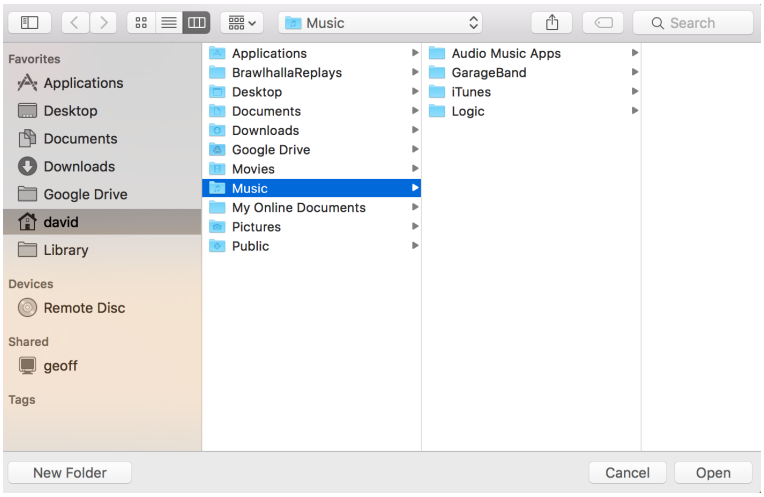

Figure 2: Browsing Screen

2. Browse through your files and select the folders where your samples are, Orbit provides two libraries of samples to get you started, ‘Rock’ and ‘EDM’. Click on the one that you prefer and select the ‘Sounds’ folder, press Open. If the loading was successful the ‘Select Samples’ button will be disabled.

2.3 Changing Scenes

Orbit allows you to store up to 4 different scenes in one session, to change between them just press the scene button with the number that you want to load. Underneath the scenes button there is a small light that lets you know if the scene has been used (red), if it is being used (yellow) or if it is empty (green).

2.4 Adjusting the Tempo

The tempo of the sequencer is variable from 100 BPM up to 500 BPM. If you want to adjust it use the ‘Tempo’ knob, when you are happy with it press the ‘Sync Tempo’ button and the sequencer will adjust the tempo.

2.5 Recording

Orbit allows you to record your sequence and save it as a wave file. To do this, press the ‘Record’ button, it will start flashing red and white. Orbit will record the next 30 seconds of your loop, after which you will be prompted with this screen:

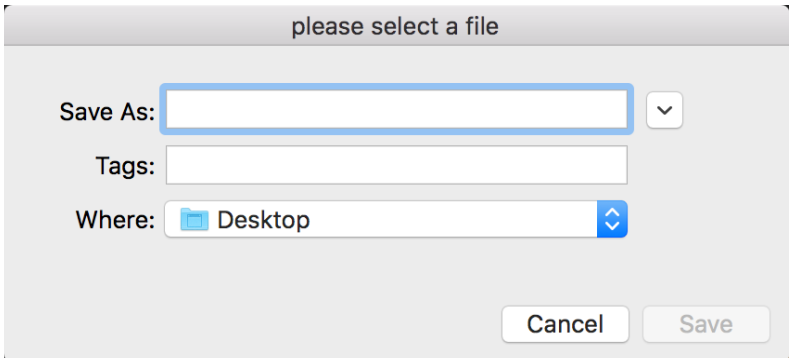

Figure 3: Saving Screen

Name your file and select where to save it. If the recording was successful the following message will show up on your screen:

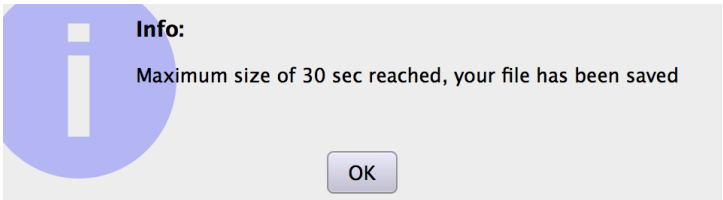

Figure 4: Info Message

That is all you need to know to get started with Orbit, enjoy.

3. System Documentation

Orbit was developed using the MVC design pattern by presenting the users with a friendly and self-intuitive interface that stores values and bool states that trigger sounds in a separate thread that is continuously updated by the user interaction. It also uses features of the observer pattern in order to send messages between the different classes.

3.1 Class Relationship Diagram

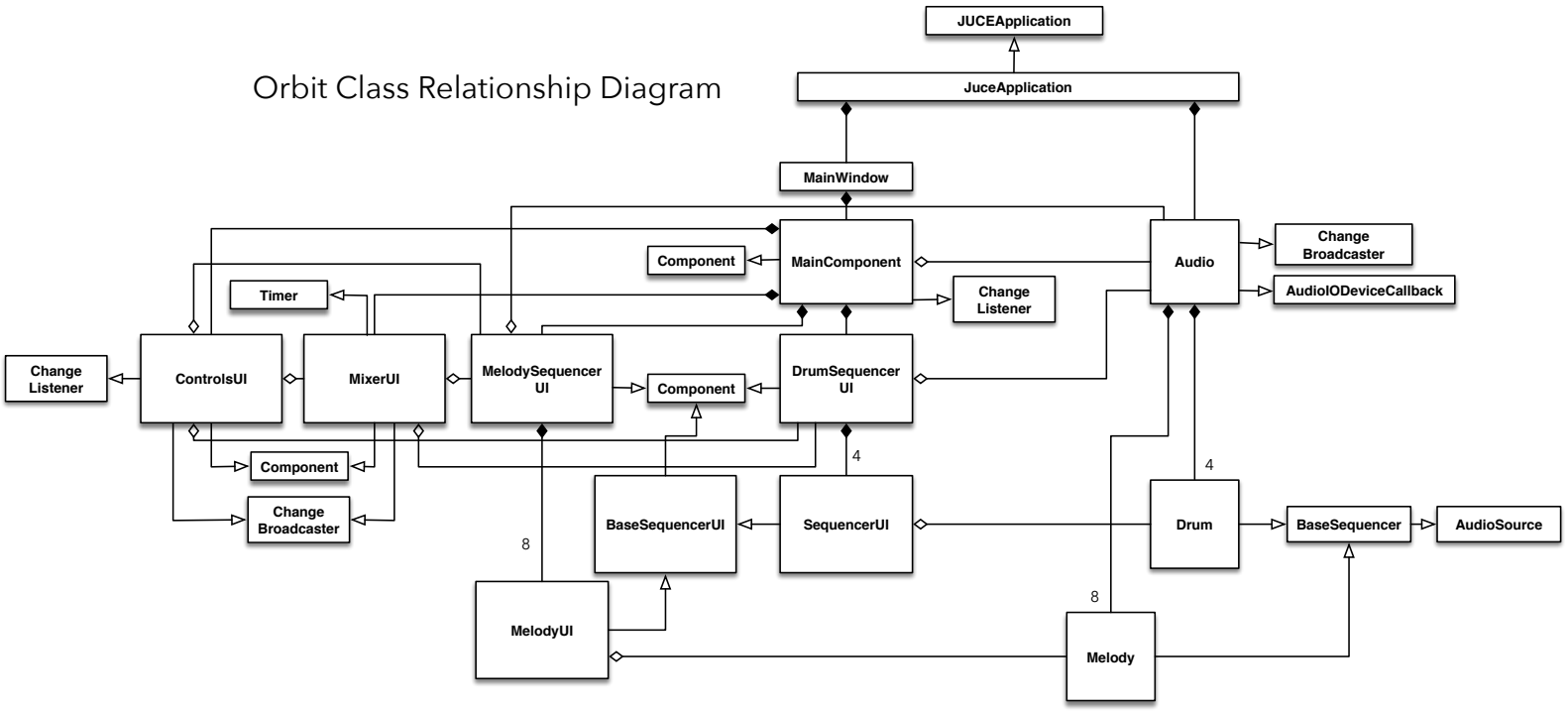

Figure 5. Orbit Class Relationship Diagram

The GUI can be separated into four components:

1. ControlsUI: Controls the playback of the sequencer as well as the recording function.
2. MixerUI: Controls levels and tempo of the sequencer and gets the directory for the files.
3. DrumSequencerUI: Contains four instances of the SequencerUI which contain a reference to a Drum object each.
4. MelodySequencerUI: Similar to DrumSequencerUI, contains eight instances of MelodyUI which contain a reference to a Melody object each.

There are two pure virtual classes implemented in the code, BaseSequencer and BaseSequencerUI, these classes are responsible for triggering the sounds if the user has prompted the application to do so.

3.2 Doxygen generated documentation

Class Documentation

Audio Class Reference

Class containing all audio processes.
`#include <Audio.h>`
Inheritance diagram for Audio:

Public Member Functions

Audio ()
Constructor.
~Audio ()
Destructor.
AudioDeviceManager & **getAudioDeviceManager** ()
Returns the audio device manager, don't keep a copy of it!
Drum & **getSequencer** (int index)
MelodySequencer & **getMelodySeq** (int index)
void **timerCalled** ()
AudioSampleBuffer & **getBufferToSave** ()
returns an audioBuffer, it is sent to the mainComponent to create the file if the user has requested to save
void **recordButtonPressed** (bool newRecordButtonState)
call this function whenever the recordButton is pressed outside of this class
bool **recordingState** ()
returns the recordState
void **stop** ()
void **audioDeviceIOCallback** (const float **inputChannelData, int numInputChannels, float **outputChannelData, int numOutputChannels, int numSamples) override
void **audioDeviceAboutToStart** (AudioIODevice *device) override
void **audioDeviceStopped** () override

Detailed Description

Class containing all audio processes.

Parameters:

outputBuffer
this is the buffer that will be filled when the sequencer is recording

Member Function Documentation

AudioSampleBuffer& Audio::getBufferToSave () `[inline]`

returns an audioBuffer, it is sent to the mainComponent to create the file if the user has requested to save

See also:
MainComponent::record()

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/audio/Audio.h

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/audio/Audio.cpp

BaseSequencer Class Reference

Base Class that contains the main function of the sequencers, enables to load files and assign files as the pure virtual functions.

`#include <BaseSequencer.hpp>`

Inheritance diagram for BaseSequencer:

Public Member Functions

BaseSequencer ()
constructor

~BaseSequencer ()
destructor

void **setPlaying** (bool newState)
starts playing if newState = true

bool **isPlaying** () const
returns true if its playing

void **loadFile** (const File &newFile)
loads a file depending on its ID

virtual void **assignID** (int newSequencerID)=0
assigns the ID and corresponding file for the sequencer

virtual void **setFile** (File *file)=0
*in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs*

void **mute** ()
mutes the sequencer

void **unmute** ()
unmutes the sequencer

bool **isMuted** ()
returns true if sequencer is muted

void **mutedBecauseOfSolo** ()
if another sequencer is solo it is important to use this function so that this component is muted without affecting the overall structure of this sequencer

void **unmuteSolo** ()
*if this sequencer was **mutedBecauseOfSolo()** it is important to unmute using this function*

void **changeGain** (float newGain)
sets a new gain for the sequencer

void **prepareToPlay** (int samplesPerBlockExpected, double sampleRate) override
void **releaseResources** () override
void **getNextAudioBlock** (const AudioSourceChannelInfo &bufferToFill) override

Detailed Description

Base Class that contains the main function of the sequencers, enables to load files and assign files as the pure virtual functions.

Inherits from audio source and takes care of all those processes as well as allowing the sequencers to be solo or muted.

See also:

Sequencer, **MelodySequencer**

Member Function Documentation

virtual void BaseSequencer::setFile (File * *file*) [pure virtual]

in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs

See also:

loadFile()

Implemented in **MelodySequencer** (*p.13*), and **Drum** (*p.9*).

The documentation for this class was generated from the following files:

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/BaseSequencer/BaseSequencer.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/BaseSequencer/BaseSequencer.cpp

BaseSequencerUI Class Reference

Base class for the UI element of the rows in the sequencer, takes care of most UI functions and loads different scenes when required.

`#include <BaseSequencerUI.hpp>`

Inheritance diagram for BaseSequencerUI:

Public Member Functions

BaseSequencerUI ()
constructor

virtual **~BaseSequencerUI** ()
destructor

bool **isSolo** ()
returns true if the row is solo

void **clearAll** ()
clears all the active buttons

virtual void **assignID** (int newID)=0
assigns an ID to the sequencer so it knows what file to load override in order to assign your own files and IDs

virtual void **somethingElseIsSolo** (bool anotherSequencerIsSolo)=0
returns true if something else is solo override to fit the number of sequencers

virtual void **mute** ()=0
mutes the row, override to fit the number of sequencers

virtual void **unmute** ()=0
unmutes the row, override to fit the number of sequencers

virtual void **changeGain** (float newGain)=0
changes the gain of the sequencer, override to fit the number of sequencers

bool **shouldPlay** (int sequencerPos)
returns true if step is active at the sequencer pos

bool **shouldMute** ()
returns true if its been muted

void **loadScene** (int numOfStep, bool shouldBeActive)
loads a scene with the information passed

void **resized** () override
void **buttonClicked** (Button *button) override
void **paint** (Graphics &g) override

Detailed Description

Base class for the UI element of the rows in the sequencer, takes care of most UI functions and loads different scenes when required.

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/BaseSequencerUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/BaseSequencerUI.cpp

ControlsUI Class Reference

Class that includes play and pause button as well as name of the application and save and load scenes feature.

`#include <ControlsUI.hpp>`

Inheritance diagram for ControlsUI:

Public Member Functions

ControlsUI (**MelodySequencerUI** &ms, **DrumSequencerUI** &ds, **MixerUI** &mui)
*construcor, needs a reference to melodySequencerUI and **DrumSequencerUI***

~ControlsUI ()
destructor

void **changeScene** ()
saves current data and loads data for the respective scene

bool **sceneHasData** (int sceneNum)
returns true if the scene has at least one button active

bool **playButtonIsPressed** ()
returns toggle state of the playButton

bool **recordButtonIsPressed** ()
returns toggle state of the recordButton

void **changeListenerCallback** (ChangeBroadcaster *source) override
void **changeToggleStates** (bool pauseButtonShouldBePressed, bool playButtonShouldBePressed)
changes the toggle states of the pause and play button

void **resized** () override
void **paint** (Graphics &g) override
void **buttonClicked** (Button *button) override
void **timerCallback** () override

Detailed Description

Class that includes play and pause button as well as name of the application and save and load scenes feature.

Communicates with **MelodySequencerUI** and **DrumSequencerUI** and **MixerUI**

See also:
MelodySequencerUI, **DrumSequencerUI**, **MixerUI**

Member Function Documentation

bool ControlsUI::recordButtonIsPressed () `[inline]`

returns toggle state of the recordButton

See also:
MainComponent::record(), ::getBufferToSave()

Parameters:
timerLoops
int that stores the amount of times the record Button has flashed

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/ControlsUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/ControlsUI.cpp

Drum Class Reference

Sequencer class for the drum sequencer, inherits from **BaseSequencer** and loads the sample for the drumSequencer.
`#include <Sequencer.hpp>`
Inheritance diagram for Drum:

Public Member Functions

Drum ()
constructor

~Drum ()
destructor

void **setFile** (File *file) override
*in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs*

void **assignID** (int newSequencerID) override
assigns an ID to the sequencer so it knows what file to load

Detailed Description

Sequencer class for the drum sequencer, inherits from **BaseSequencer** and loads the sample for the drumSequencer.

See also:
BaseSequencer, **SequencerUI**, **DrumSequencerUI**

Member Function Documentation

void Drum::setFile (File * *file*) `[override], [virtual]`

in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs

See also:
loadFile()

Implements **BaseSequencer** (*p.5*).

The documentation for this class was generated from the following files:

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/Sequencer.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/Sequencer.cpp

DrumSequencerUI Class Reference

DrumSequencerUI contains 4 sequencerUI objects and acts as a bridge between **MixerUI** and **ControlsUI** and **SequencerUI**.

`#include <DrumSequencerUI.hpp>`

Inheritance diagram for DrumSequencerUI:

Public Member Functions

DrumSequencerUI (**Audio** &a)
constructor, needs a reference to the audio object

~DrumSequencerUI ()
destructor

void **play** ()
starts the audio

void **checkForSolos** ()
checks if one of the sequencers are solo

void **checkForMutes** ()

int **getSequencerPos** ()
returns sequencerPos

bool **isStepActive** (int numOfSequencer, int numOfStep)
returns true if the current step is active

void **loadScene** (int numOfSequencer, int numOfStep, bool shouldBeActive)
retrieves the information and loads a scene

void **timerCalled** ()
updates the state of the sequencer

void **setFile** (File *fileDirectory)
sets the file to be loaded

void **buttonClicked** (Button *button) override

void **changeGain** (float newGain)

void **resized** () override

void **paint** (Graphics &g) override

Detailed Description

DrumSequencerUI contains 4 sequencerUI objects and acts as a bridge between **MixerUI** and **ControlsUI** and **SequencerUI**.

See also:

MixerUI, **ControlsUI**, **SequencerUI**

The documentation for this class was generated from the following files:

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/DrumSequencerUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/DrumSequencerUI.cpp

MainComponent Class Reference

Inheritance diagram for MainComponent:

Public Member Functions

MainComponent (**Audio** &a)
Constructor.

~MainComponent ()
Destructor.

void **changeListenerCallback** (ChangeBroadcaster *source) override
void **resized** () override
void **paint** (Graphics &g) override
void **record** (AudioSampleBuffer *bufferToSave)
takes an AudioSampleBuffer pointer, prompts the user to create a file that is stored into the desktop

Member Function Documentation

void MainComponent::record (AudioSampleBuffer * *bufferToSave*)

takes an AudioSampleBuffer pointer, prompts the user to create a file that is stored into the desktop

See also:
audio::getBufferToSave(), MixerUI::recordButtonIsPressed()

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/MainComponent.h
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/MainComponent.cpp

MelodySequencer Class Reference

MelodySequencer class, loads samples into the steps generated in the melody UI.
`#include <Melody.hpp>`
Inheritance diagram for MelodySequencer:

Public Member Functions

MelodySequencer ()
Constructor.

~MelodySequencer ()
destructor

void **assignID** (int newSequencerID) override
assigns the ID and corresponding file for the sequencer

void **setFile** (File *file) override
*in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs*

Detailed Description

MelodySequencer class, loads samples into the steps generated in the melody UI.

See also:

BaseSequencer, **MelodyUI**, **MelodySequencerUI**

Member Function Documentation

void MelodySequencer::setFile (File * *file*) [override], [virtual]

in this function retrieve the directory sent from **MixerUI** and assign to each sequencer depending on its ID the file that it needs

See also:

loadFile()

Implements **BaseSequencer** (*p.5*).

The documentation for this class was generated from the following files:

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/Melody.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/Melody.cpp

MelodySequencerUI Class Reference

MelodySequencerUI class, contains 8 **MelodyUI** objects and acts as the bridge between **MixerUI** and **ControlsUI** and **MelodyUI**.

`#include <MelodySequencerUI.hpp>`

Inheritance diagram for MelodySequencerUI:

Public Member Functions

MelodySequencerUI (**Audio** &a)
constructor, needs a reference to the audio object

~MelodySequencerUI ()
destructor

void **play** ()
starts the audio

void **checkForSolos** ()
checks if one of the sequencers are solo

void **checkForMutes** ()
checks if another component is mute

void **changeGain** (float newGain)
changes the gain of all the sequencers

void **timerCalled** ()
updates the state of the sequencer

void **setFile** (File *file)
sets the file to be loaded

void **loadScene** (int numOfSequencer, int numOfStep, bool shouldBeActive)
retrieves the information and loads a scene

bool **isStepActive** (int numOfSequencer, int numOfStep)
returns true if the current step is active

void **buttonClicked** (Button *button) override
void **resized** () override
void **paint** (Graphics &g) override

Detailed Description

MelodySequencerUI class, contains 8 **MelodyUI** objects and acts as the bridge between **MixerUI** and **ControlsUI** and **MelodyUI**.

See also:

MelodyUI, **MelodySequencer**

The documentation for this class was generated from the following files:

/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/MelodySequencerUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/MelodySequencerUI.cpp

MelodyUI Class Reference

MelodyUI class, generates one row of 8 steps and communicates with **MelodySequencerUI** and **MelodySequencer**.

`#include <MelodyUI.hpp>`

Inheritance diagram for MelodyUI:

Public Member Functions

MelodyUI (**MelodySequencer** &melodySeq_)
constructor, needs a reference to the sequencer

~**MelodyUI** ()
destructor

void **assignID** (int newID) override
assigns an ID to the sequencer so it knows what file to load

void **somethingElseIsSolo** (bool anotherSequencerIsSolo) override
asks if another component is solo

void **mute** () override
mutes the row

void **unmute** () override
unmutes the row

void **changeGain** (float newGain) override
changes the Gain of the sequencer

void **setFile** (File *file)
sends the file directory to the melody sequencer

void **play** (int sequencerPos)

Detailed Description

MelodyUI class, generates one row of 8 steps and communicates with **MelodySequencerUI** and **MelodySequencer**.

See also:

MelodySequencer, **MelodySequencerUI**

Member Function Documentation

void MelodyUI::setFile (File * *file*) `[inline]`

sends the file directory to the melody sequencer

See also:
BaseSequencer::setFile()

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/MelodyUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/MelodySequencer/MelodyUI.cpp

MixerUI Class Reference

Class that includes a mixer which allows to change gain and sync tempo.
`#include <MixerUI.hpp>`
Inheritance diagram for MixerUI:

Public Member Functions

MixerUI (**MelodySequencerUI** &ms, **DrumSequencerUI** &ds)
constructor, needs a reference to the melody sequencer and to the drum sequencer

~MixerUI ()
destructor

void **changeGain** (int index, float newGain)
changes the gain with the sliders of the individual components

void **play** ()
starts the sequencer

void **pause** ()
pauses the sequencer

void **chooseFiles** ()
called when the selectFileButton is pressed, opens up a dialogue box for the user to select the folder where the samples are

void **resized** () override
void **paint** (Graphics &g) override
void **buttonClicked** (Button *button) override
void **sliderValueChanged** (Slider *slider) override
void **timerCallback** () override

Detailed Description

Class that includes a mixer which allows to change gain and sync tempo.

Communicates with **MelodySequencerUI** and **DrumSequencerUI**.

See also:
MelodySequencerUI, **DrumSequencerUI**

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/MixerUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/ui/MixerUI.cpp

SequencerUI Class Reference

This class is the UI for a single row of the sequencer, it is composed of buttons and works together with Sequencer and **SequencerUI** to control the sequencer, by using the assign ID it tells the sequencer what file it should load.

`#include <SequencerUI.hpp>`

Inheritance diagram for SequencerUI:

Public Member Functions

SequencerUI (**Drum** &sequencer_)
constructor, needs a reference to a sequencer

~SequencerUI ()
destructor

void **assignID** (int newID) override
assigns an ID to the sequencer so it knows what file to load

void **somethingElseIsSolo** (bool anotherSequencerIsSolo) override
returns true if another row is solo

void **mute** () override
mutes the row

void **unmute** () override
unmutes the row

void **changeGain** (float newGain) override
changes the gain of the sequencer, override to fit the number of sequencers

void **play** (int sequencerPos)
starts producing sound

void **setFile** (File *file)
sends the file directory to the melody sequencer

Detailed Description

This class is the UI for a single row of the sequencer, it is composed of buttons and works together with Sequencer and **SequencerUI** to control the sequencer, by using the assign ID it tells the sequencer what file it should load.

See also:
Sequencer, **DrumSequencerUI**

Member Function Documentation

void SequencerUI::setFile (File * *file*) `[inline]`

sends the file directory to the melody sequencer

See also:
BaseSequencer::setFile()

The documentation for this class was generated from the following files:
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/SequencerUI.hpp
/Users/david/Documents/UWE/Year3/SDA/AssignmentTake2/Source/DrumSequencer/SequencerUI.cpp

4. Conclusion

The development of Orbit has allowed me to have a greater understanding of the life cycle of a software development project. I have extensively used OOP techniques in order to develop the classes and design principles to create the structure of the program. Developing Orbit also challenged me to understand the process of working with a third-person library such as JUCE, by doing this I explored the API and acquired knowledge on a well structured documentation.

5. Further Development

Orbit would benefit by having a save and load feature that could allow users to save their projects in XML format in order to return later to them and keep developing their sequence. This feature however, is currently in development. Future plans for Orbit also include adding DSP controls so the user can, at the very least, apply filters and EQ to the drums or the melody thus creating unique sequences whilst still using the sounds provided in the ExampleSounds folder.