Il BELAtore

CAPR group:

Claudio, Eutizi (ID: 10812073)
Andrés, Bertazzi (ID: 10488849)
Pierluigi, Tartabini(ID: 10845797)
Riccardo, Martinelli (ID: 10456202)

Computer Music System Homework 3
Computer Music: Languages & Systems
Politecnico di Milano

June 2, 2022

a

Contents

1 Introduction **1**

2 The BELA Board **1**

3 GUIs **2**
3.1 BELAbirinto 2
3.2 Events of the game 3
3.3 BELAtore 3

4 Interaction between the BELAtore and the BELAbirinto **3**
4.1 OSC Messages 3

5 The BELAtore plugin **5**
5.1 AudioProcessorGraph structure 5
5.2 Class structure 5
5.2.1 Processors, Groups and OSC-controlled Noise 5
5.2.2 Editor, Components and OSC messages 7

1 Introduction

The **Belatore** is an effect plugin that permits to use, manage, and mix two audio effects in a funny and interactive way using a joystick and a touch sensor connected to a `BELA` board. The BELAtore can be used either as a standalone application or as a VST in a DAW, such as `Ableton`, `Pro Tools` etc. and it consists in two main parts:

- The **BELAtore**: the audio plugin.
- The **BELAbirinto**: an interactive GUI that communicate with the Belatore using `OSC` messages.

  It has been implemented using `Processing`, `JUCE 6`, `BELA IDE` and `Visual Studio`.

2 The BELA Board

In order to play the BELAbirinto maze game, the user has to use a **BELA board** connected to the computer, in turn connected with a joystick and a touch sensor. In this project, the devices that have been used are **DFROBOT Joystick V2** and **DFROBOT Touch Sensor V2**, connected to a **BELA** microcontroller through a solderless breadboard.
The X and Y potentiometers of the Joystick are connected to analog inputs 0 and 1, while the Z button and the touch sensor work with digital inputs.
In the Figure 1 there is shown how the connections of the project were made.
The colors of them are the same of the real structure: black for the GND, red for the VCC and green for the output.
Digital inputs are connected into the PIN 0 and PIN 1 on the right of the micro-controller with blue and white cables.

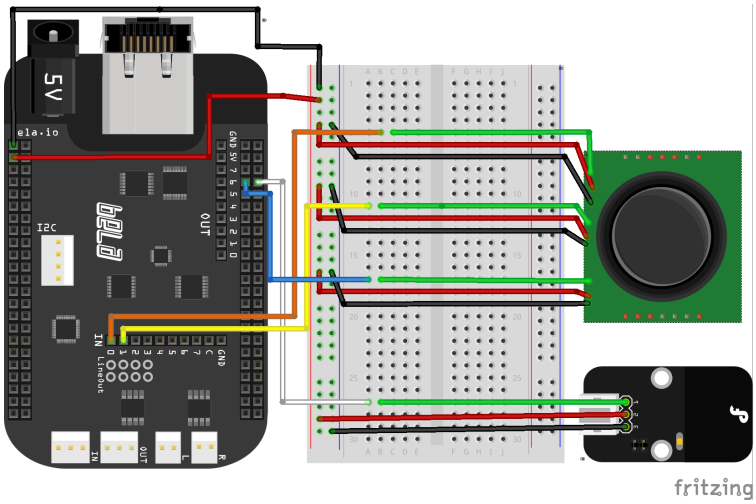


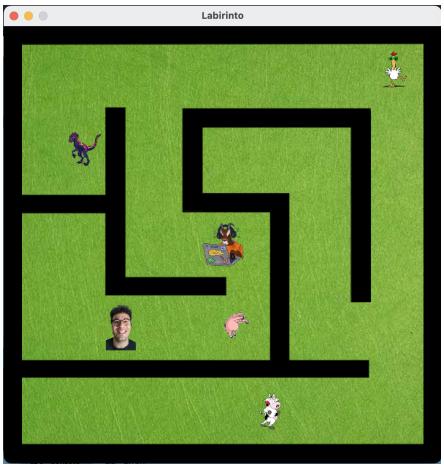
Figure 1: BELA structure.

3 GUIs

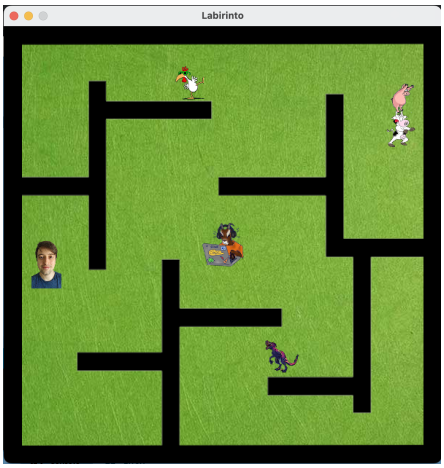
The GUIs of the **BELAtore** project are 2, one for the maze game and the other for the JUCE application.
The first one, called **BELAbirinto** has been made using `Processing`, a specialised software to code within the context of the visual arts, while the second one, the **BELAtore** GUI, has been implemented with `JUCE`.

3.1 BELAbirinto

The game window (600x600 pixels) represents a green lawn with 4 possible maze's sets. The labyrinth's fence is black and the elements are set in casual initial conditions, as seen in the next images.



(a) Maze 1

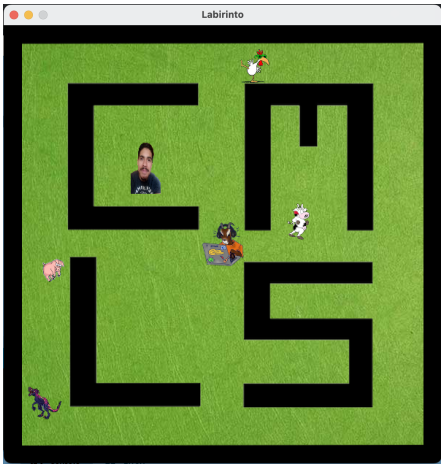


(b) Maze 2

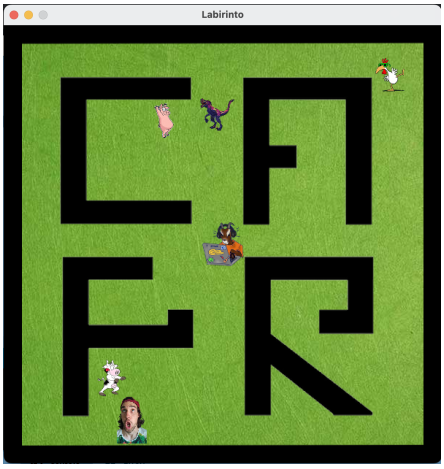
Figure 2: Typical mazes

In this game the user can play a maze game where, controlling a goat, they has to elude other animals and the scary faces. In fact the elements presented in the green lawn are:

- **CAPR The GOAT**: he is the main character that the user can move through the labyrinth. he starts the game from the center of the window and can move only in the green lawn, obviously not in the black fence.
- **Animals**: there are 4 different animated animals that block the passage to CAPR: a dancing cow, a techno chicken, a sporty pig and a screaming velociraptor. They are always present in the lawn.
- **Faces**: also scary faces block the passage to CAPR. They represent the close-ups of the CAPR group components. Only one face is displayed at a time.
- **Black fence**: the walls where CAPR can not go.



(a) "CMLS" maze



(b) "CAPR" maze

Figure 3: Particular mazes

The characters are implemented as GIF files: they are computed with about 40 frames and they change state 24 times per second. Also the `draw` method implemented in `Processing` compute 24 times per second.

3.2 Events of the game

The user can move **CAPR The GOAT** through the labyrinth using the **Joystick**. The goal of the game is to explore the maze perceiving the sound effects changes by moving in all directions (in particular in x and y ways).
The user can use 2 different methods to clear the way for the goat:

- Touch the **Touch Sensor** to remove the animals.
- Push the **Z Joystick Button** to remove the scary faces.

3.3 BELAtore

The BELAtore GUI is developed from the top to the bottom in a series of rectangles. The two effects' rectangles change dynamically with the choice the user does via the combo box. The user can not choose the same type of effect for both slots. Using the `CustomLookAndFeel` JUCE class were imported custom knobs that have been created for this project using the **JKnobMan** application.

4 Interaction between the BELAtore and the BELAbirinto

4.1 OSC Messages

BELAtore and BELAbirinto applications communicate using OSC (Open Sound Control) Protocol. It is based on messages and/or bundles that can be sent and received through **UDP Sockets <IPAddress, LogicPort>**.

Figure 4: Belatore Juce GUI

Figure 5: Interaction scheme of the BELAtore project

In the particular case of the BELAtore, the path that the messages follow is described in Figure 5: BELA sends messages to the BELAbirinto application that contain the values it receives from the joystick and the sensor via the analog and digital inputs. This permits CAPR The GOAT's movement and interation within the maze. Then, to the JUCE plugin are sent as OSC messages the instantaneous position (x, y) of CAPR and flag signals if this last hits or booms an animal or a scary face, in order to manage the channel strip parameters.

The chosen sockets are the following:

- **BELA**: sends on **<192.168.6.1 ; 7563>**;
- **Processing**: receives from BELA's same socket and in turn sends on **<127.0.0.1 ; 9000>** the position of CAPR in the maze and on **<127.0.0.1 ; 9001>** the "boom" or "hit" events.
- **JUCE**: receives from Processing's same sockets.

5 The BELAtore plugin

5.1 AudioProcessorGraph structure

The main idea of the project was to build a multi-FX channel strip plugin that would be able to create some unique combinations of sounds. Due to the fact that the effects order is not fixed, the implementation choice fell on designing a **JUCE AudioProcessorGraph**. It is a special type of **AudioProcessor** that allows us to connect several AudioProcessor objects together as `JUCE::Node` objects in a graph and play back the result of the combined processing.

In order to wire-up graph nodes together, it is required to add an order of connections between channels of nodes to process the audio signal. The structure of the graph is described in Figure 6.

Figure 6: BELAtore Audio Processor Graph structure

The just described architecture is created in the **PluginProcessor.cpp** with the `initializeGraph()` function and it is maintained up to date calling the `updateGraph()` function in the `processBlock` function, whenever the user changes the slots' effects. This last function recreates from scratch the graph, plugging in it the nodes that contain the instances of the selected Processors created using the `make_unique` template function.

5.2 Class structure

The JUCE application has been efficiently organized in three main **modulus**: **Processors**, **Groups** and **Components**.

- **Processors**: different processors classes that can be use within the channel strip plugin in order to alter the incoming audio signal;
- **Groups**: structs that contain the `AudioProcessorValueTreeState` parameters for each processor;
- **Components**: classes that group all the needed GUI component (sliders, labels, buttons and combo boxes) for each group of parameter defined in the Groups module.

In Figure 7 is shown a pseudo-UML graph that clearly describes the internal structure of the plugin.

5.2.1 Processors, Groups and OSC-controlled Noise

Starting from the Processors, each class of this group implements an AudioProcessor base class called `ProcessorBase` that has three main virtual functions: `prepareToPlay`, `processBlock`, `reset` and `updateOscParameter`.

All the processors have been implemented using the useful functionalities provided by the JUCE **DSP module**. In order to avoid useless repetitions, all the gain in the processors are controlled exploiting the `Gain` dsp class, all the filters are `FirstOrderTPTFilter` filters and the dry/wet

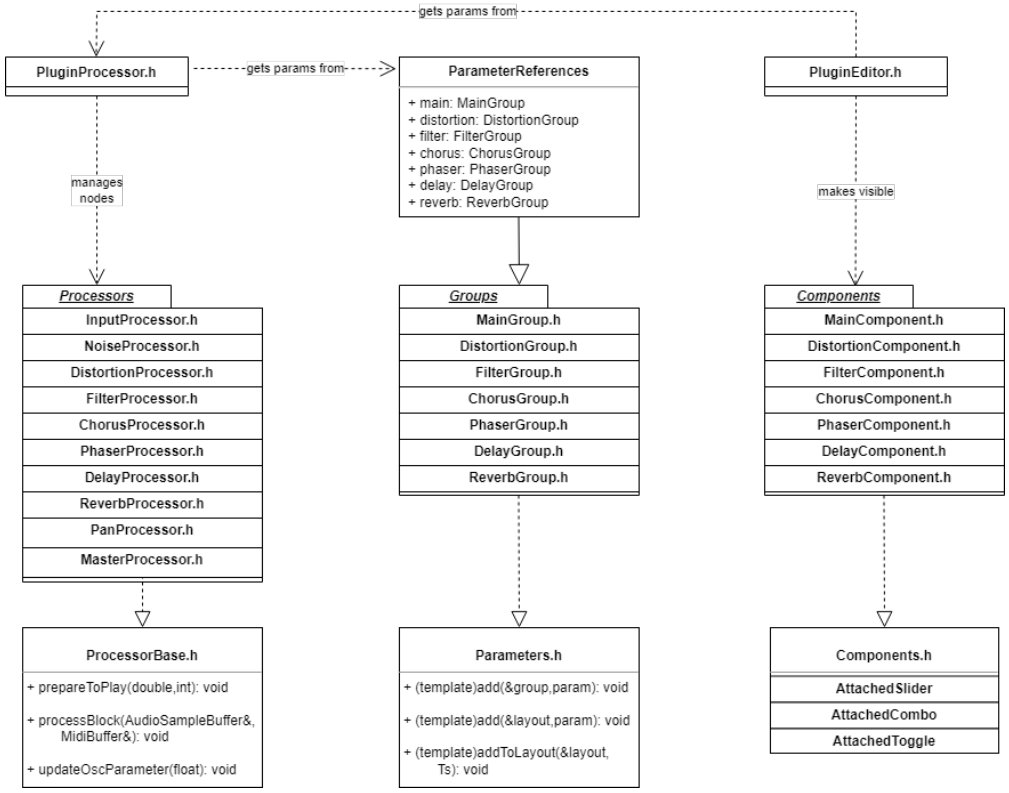


Figure 7: BELAtore pseudo-UML graph

mixers are managed using the `DryWetMixer` juce::dsp class or with specific internal parameters of the DSP classes. Here is a more detailed description of the BELAtore processors:

- **InputProcessor**: it controls the **input gain**;
- **NoiseProcessor**: it manages the gain and the dry/wet mix of a **white noise** that will occur if CAPR hits an enemy in the maze. This noise is generated using the `JUCE::Random` class, and its gain and its mix levels are randomly generated. The more enemy are hit, the higher these levels. The boundaries of such levels are 0 dB for the noise gain and 1 (completely wet) for the dry/wet mixer. The only way to remove the noise is to **"boom"** an enemy as described in the previous section. "hit" and "boom" messages are received from the NoiseProcessor class listening for OSC messages from the BELAbirinto application to the port `9001`;
- **DistortionProcessor**: A distortion effect designed as a `juce::dsp::ProcessorChain`, i.e a series of DSP processors in series. The chosen clipping function is the *tanh* or hyperbolic tangent. In addition to the distortion effect controls, there is also a combo box where to choose a **Cabinet** whose impulse response will be convolved with the signal, in order to simulate the sound of some of the best guitar amplifiers of all time;
- **FilterProcessor**: A **2-band filter** that divides the spectrum in low and high frequencies. Each subband has a gain control that can emphasize or attenuate the correspondent frequencies.

- **ChorusProcessor**: A chorus effect that exploits the `Chorus` class of the juce DSP module and all of its parameters.
- **PhaserProcessor**: A phaser effect that exploits the `Phaser` class of the juce DSP module and all of its parameters.
- **DelayProcessor**: A linear-interpolated delay effect implemented using the `DelayLine` juce::dsp class. It has a maximum of 2 seconds of delay length and a feedback control in order to make the repetitions' decay quicker or slower.
- **ReverbProcessor**: A reverb effect created using the `juce::dsp::Reverb` and `juce::dsp::Reverb::Parameters` classes. This last class contains all the parameters that the Reverb class needs to set its values. Downstream of the Reverb, a highpass filter and a lowpass filter can eliminate the annoying low and high frequencies that the reverb tails can contain.
- **PanProcessor**: it controls the stereo position of the audio signal.
- **MasterProcessor**: it controls the output gain.

The `PluginProcessor` class imports all these processors and an instance of the `ParameterReferences` class, which in turn contains instances of every group struct the Groups module. With these references, the `PluginProcessor` class implements an `update` function that provides the data it takes from the `ParameterReferences` class via the `AudioProcessorValueTreeState` class instance to the `updateParameters` functions contained in each Processor class.

5.2.2 Editor, Components and OSC messages

Regarding the `PluginEditor` class, it imports all the components of the Components module and make them visible in the window using the `paint()` and `resized()` functions.

Moreover, based on which effects, and then which components, has been chosen in the GUI, the `PluginEditor` class adds and removes as **OSC listeners** these components so as to connect properly the **OSC parameters** of each Component to the desired OSC message. In particular, the OSC-controlled components are the **Mix sliders** of every component, except the **FilterComponent** and **PanComponent**, in which OSC messages control the cutoff frequency and the panning respectively.

Those messages come from the port 9000 and they have **"\coordinates\xPosition"** and **"\coordinates\yPosition"** address patterns. The editor will set as listeners the components that correspond to the chosen effects: the first one's OSC-controlled slider will be modified by the x position of CAPR in the maze, while the second one's by its y position.

The sliders are set with rearranged values that come from the BELAbirinto maze game, according to the boundaries of the parameters they have to control. Then, the attachment with the `AudioProcessorValueTreeState` parameters layout ensures that the effects will be set with the correspondent values of these sliders.

The OSC-controlled sliders rotate **clockwise** when CAPR moves from **left to right and from the bottom to the top**, counter clockwise vice versa. Therefore, the user will be able to achieve a completely dry signal when CAPR is at the bottom-left of the maze and a completely wet signal when he is at the top-right corner.