Revista de Informática Teórica e Aplicada - RITA - ISSN 2175-2745
Vol. 27, Num. 04 (2020) 95-107

RESEARCH ARTICLE

Creating Digital Musical Instruments with libmosaic-sound and Mosaicode

Criando Instrumentos Musicais Digitais com libmosaic-sound e Mosaicode

Luan Luiz Gonçalves[1]*, Flávio Luiz Schiavoni[1]

Abstract: Music has been influenced by digital technology over the last few decades. With the computer and the Digital Musical Instruments, the musical composition could trespass the use of acoustic instruments demanding to musicians and composers a sort of computer programming skills for the development of musical applications. In order to simplify the development of musical applications, several tools and musical programming languages arose bringing some facilities to lay-musicians on computer programming to use the computer to make music. This work presents the development of a Visual Programming Language (VPL) to develop DMI applications in the Mosaicode programming environment, simplifying sound design and making the creation of digital instruments more accessible to digital artists. It is also presented the implementation of libmosaic-sound library, which supported the VPL development, for the specific domain of Music Computing and DMI creation.
Keywords: Mosaicode — Digital Musical Instrument — Code generation — Library development

Resumo: A música tem sido influenciada pela tecnologia ao longo das últimas décadas. Com o computador e os instrumentos musicais digitais, a composição musical pode superar a utilização de instrumentos acústicos exigindo que músicos e compositores tenham habilidades em programação para desenvolver aplicações musicais. De forma a simplificar o desenvolvimento de aplicações musicais, diversas ferramentas e linguagens de programação musical surgiram trazendo alguma facilidade para músicos leigos em programação utilizar o computador para fazer música. Este trabalho apresenta o desenvolvimento de uma linguagem de programação visual para o desenvolvimento de aplicações de instrumentos musicais no ambiente de programação Mosaicode, simplificando o projeto de novos sons e tornando a criação de instrumentos digitais mais acessível para artistas digitais. Também é apresentado a implementação da biblioteca libmosaic-sound, que auxiliou o desenvolvimento da linguagem de programação visual, para o domínio específico da computação musical e da criação de instrumento.
Palavras-Chave: Mosaicode — Instrumentos musicais digitais — Geração de Código — Desenvolvimento de biblioteca

[1] *Computer Science Department – DCOMP, Federal University of São João del-Rei – UFSJ, Brazil*
***Corresponding author**: luanlg.cco@gmail.com
DOI: http://dx.doi.org/10.22456/2175-2745.104342 • **Received:** 15/06/2020 • **Accepted:** 13/10/2020
CC BY-NC-ND 4.0 - This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

1. Introduction

The current paper expands our previous work, titled "The development of libmosaic-sound: a library for sound design and an extension for the Mosaicode Programming Environment" [1], presented in the 17th Brazilian Symposium on Computer Music realized in 2019. Previously, we had developed a library for sound design and an extension to develop applications using it into the Mosaicode Programming Environment. The present research added new features to our library, expanding it to give to programmers a better support on the creation of new Digital Musical Instruments (DMI).

In the present work, the following new features were added to libmosaic-sound and to Mosaicode a set of:

- GUI components based on Gtk lib + Cairo to create user interfaces of DMIs;
- MIDI components based on ALSA MIDI API to create MIDI controllers;
- Joystick components to use Generic external controllers based on the Linux Kernel lib;
- Basic components, like arithmetical operation and logical operations;
- OSC components, based on Lib Light OSC, to create DMIs based on OSC messages.

The development of DMIs is a kind of classic activity on computer music field and can be summarized in the definition/implementation of the instrument input, like the physical interface, the instrument output, like a synthesizer, and the mapping between them. These steps are normally be done using a programming language. The input interface normally consists in physical and logical devices, requiring the implementation of communication between the devices and the synthesizer. The instrument output is the user's feedback, normally a sonic output of a synthesizer but also including visual feedback or haptic feedback. The mapping between input and output takes place through arithmetic, logical operations and conversion of values. More about DMI will be presented in Section 1.1.

The development of a DMI can be an activity exclusive to programmers, since these three parts need to be programmed in a computational system. Fortunately to non programmers or lay-programmers, it is possible to develop a computer application, like a DMI, using non-textual programming paradigms. Visual Programming Languages (VPLs) are an example of programming paradigm that allows programmers to develop code using a two-dimensional notation and interacting with the code from a graphical representation [2]. The use of diagrams to develop applications can make the development easier and allow non-programmers or novice programmers to develop and create software. Furthermore, diagrammatic code abstraction can bring practicality in changing the code, making it suitable for rapid prototyping [3], a feature that can help even experienced programmers.

Software development and the creation of DMIs can also be done using a Domain-Specific (Programming) Language (DSL) [4]. DSLs are at a higher abstraction level than general purpose programming languages because they have the knowledge of the domain embedded in its structure. It makes the process of developing applications in a certain domain easier because DSLs require more knowledge about the specific domain than general programming knowledge [5]. Hence, the potential advantages of DSLs include reduced maintenance costs through re-use of developed resources and increased portability, reliability, optimization and testability [6].

Another form to unleash the development of specific tools, like DMIs, is to reuse code from a software library. Software libraries, APIs (Application Programming Interface) and frameworks can also assist the development of applications for specific domains. Different from the DSLs, libraries can be developed using general purpose programming languages (GPLs) and can provide resources for developing applications for specific domains. In this way, the implementation effort can be reduced and the programmer will not have to learn a DSL, focusing only in how to use the resource available in a library wrote in GPL. This reuse of code helps the creation of programs in certain domains, especially in Digital Art, if we consider that many digital artists know how to code but did not take courses that include formal knowledge in computer programming.

Digital Art applications may require high data processing, such as audio and image processing in real time. Thus, these applications need to be well optimized and with a small noticeable delay and jitter, features that are hard to reach with a naive code and that, without it, can make an application to be useless for artistic performance. When using a library, we are reusing code that has already been tested and optimized by the developers and used / tested by several users. Thus, we eliminate the concern of optimizing the parts of reused code, also allowing users to create optimized programs without having this programming skills.

Merging the readiness of VPLs, the higher abstraction of DSLs and the code reuse of libraries, we present the Mosaicode, a visual programming environment focused on the development of applications for the specific domain of Digital Art. The development of an application in the Mosaicode environment, presented in Figure 2, is accomplished by the implementation of a diagram, composed by blocks and connections between them. The schematic of a diagram is used to generate a source code in a specific programming language using a code template for it. The tool also provides resources for creating and editing components (blocks, ports, and code template) to the environment and a set of components is called an extension. Thus, by the creation of new extensions, the tool can be extended to generate code for different programming languages and specific domains – building VLPs for DSLs. For this reason, it is possible to say that Mosaicode is not restricted to generating applications only for the specific domains of Digital Art, since it allows the creation of extensions for any other specific domain.

This paper presents a set of extensions developed to Mosaicode focusing in the creation of DMIs. Our development process is presented in Section 2 covering all the steps that we did to create these extensions. We hope it can clarify the development process and help future developers to contribute to this project.

When connecting blocks of this extension, we create a diagram that the Mosaicode can interpret and generate code written in the C programming language, supported by the libmosaic-sound library.

The libmosaic-sound library is also an outcome of this project and it was developed with the aim of facilitating the development of Digital Art application by reducing the effort required to implement it. Thus, the user can effortlessly create audio applications in the C language using this library. The library structure provides this ease-of-use programming framework and made it easier to implement the blocks in Mosaicode. This library is based on other libraries and it is presented in Section 2.2.

1.1 Digital Musical Instruments

We can imagine DMIs as musical instruments made with / in computers, programmed musical devices that give to the user a possibility to change the way they are played, as well as their sound characteristics and feedback. Thus, the common

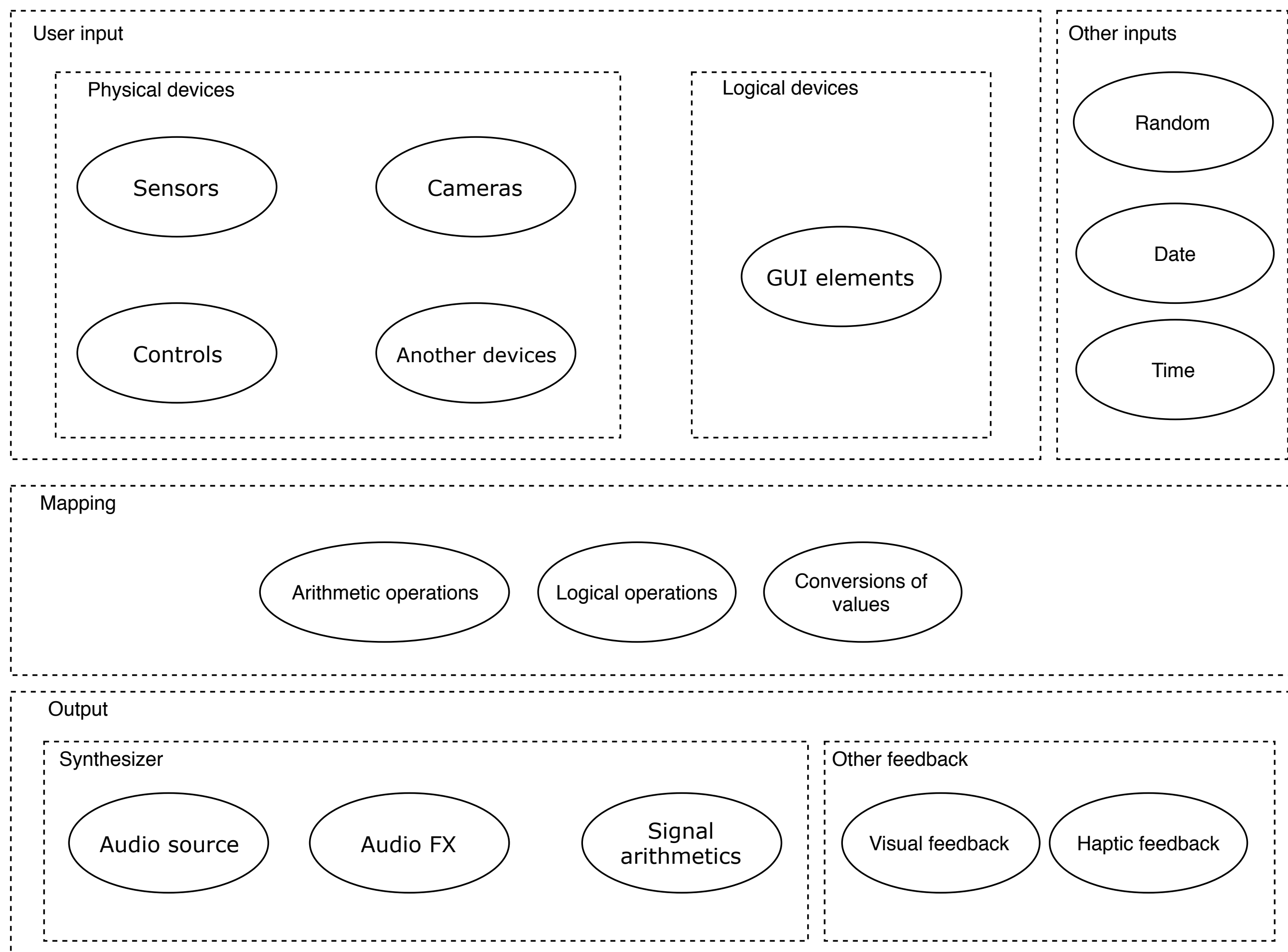

Figure 1. Basic structure of DMI [7].

schematic of a DMI splits it into three layers of abstraction: input, output and mapping. Figure 1 illustrates the basic structure of a DMI separated in layers and using components to present how it can implemented.

To control a DMI it is necessary to define an **input**, building an interaction interface between the instrument and the musician's gesture. This interface can be composed of physical and logical devices. Physical devices are any hardware that can send data messages to the computer via a data link and/or using a communication protocol. Sensors, cameras, midi controllers, joysticks, keyboard and mouse, are examples of physical devices that can be used to compose the instrument's input. Logical devices are GUI elements like buttons, sliders, and text entry boxes. Other inputs, like the computer clock provides date, time and random values, allowing the addition of stochastic parameters to the DMIs, scheduling events, delaying events and creating sequencers.

The DMI **output** defines the instrument's feedback, responding to its input values. We have the sound of the instrument as the main feedback, produced by the instrument's synthesizer but a DMI can have other outputs. Synthesizers are the voice of DMIs and can be implemented or inspired by one or more classic audio synthesis algorithms such as Amplitude Modulation, Frequency Modulation, Phase Modulation, additive synthesis, subtractive synthesis, physical modeling and others. Effects, filters and envelopes can be added to the synthesizer's sound to modify the instrument's timbre. Beyond the audible feedback, visual and haptic feedbacks can be added to the DMI's output to help the user to understand its behavior. Audio signal representations such as waveform, audio spectrum and frequency bar char; and also the activation of physical or virtual leds are examples of visual feedback. Vibration motors and other actuators can be used as haptic feedback.

There are several different forms to use input values to generate sound output. However, it is necessary to map the input values to the output and it can be an interesting task. Firstly, it can be necessary to adjust values sent by the input devices to suit an output parameters, generating values in the same range and with the same data type. A GUI component, like a slider, can control the volume of an sound output, ranging from 0.0 to 1.0, but maybe an accelerometer, ranging from -32 to 32 can be used to perform the same task needing some conversion to grant a correct range. Another example of mapping

is the use of a MIDI controller to define the frequency that the instrument will sound, if the output receives the frequency value in hertz as a parameter, the MIDI message sent by the controller must be converted to the corresponding frequency value. More about mapping strategies may be found in [8].

1.2 Related Tools

The tools presented below are widely used by digital artists and are considered to be related to this research. Some similarities are the visual programming paradigm, the code generation and the resources to implement synthesizers, analysis and audio processing, being useful for the development of DMIs.

Processing[1] is a programming language and an Integrated Development Environment (IDE) developed by the MIT Media Lab[9]. The programming framework of Processing contains abstractions for various operations with images and drawings and allows rapid prototyping of animations in very few lines of code. The purpose of the tool is to be used for teaching programming and for graphic art development. From programs made in Processing, called sketches, the IDE generates code to other programming languages, like Java or Python, and runs the generated code.

Pure Data[2] or simply Pd is a graphical real-time programming environment for audio and video [10] application development. A program in PD is called a patch and is done, according to the author himself, through "boxes" connected by "cords". This environment is extensible through plugins, called externals, and has several libraries that allow the integration of PD with sensors, Arduino, wiimote, OSC messages, Joysticks and others. PD is an open source project and is widely used by digital artists. The environment engine was even packaged as a library, called libpd [11], which allows one to use PD as a sound engine on other systems like cellphones applications and games.

Max/MSP[3] is also a real-time graphical programming environment for audio and video [12]. Developed by Miller Puckett, the creator of Pure Data, Max is currently maintained and marketed by the Cycling 74 company. Different from the other listed related tools, Max is neither open source or free software.

EyesWeb[4] is a visual programming environment focused on real-time body motion processing and analysis [13]. According to the authors, this information from body motion processing can be used to create and control sounds, music, visual media, effects and external actuators. There is an EyesWeb version, called EyesWeb XMI – for eXtended Multimodal Interaction – intended to improve the ability to process and correlate data streams with a focus on multimodality [14]. Eyesweb is proprietary free and open source with its own license for distribution.

JythonMusic[5] is a free and open source environment based on Python for interactive musical experiences and application development that supports computer-assisted composition. It uses Jython, enabling to work with Processing, Max/MSP, PureData and other environments and languages. It also gives access to Java API and Java based libraries to be used in the code. The user can interact with external devices such as MIDI, create graphical interfaces and also manipulate images [15].

FAUST[6] is a functional programming language for sound synthesis and audio processing. A code developed in FAUST can be translated to a wide range of non-domain specific languages such as C++, C, JAVA, JavaScript, LLVM bit code, and WebAssembly[16].

1.3 Mosaicode

Mosaicode[7] is a visual programming environment that brings the advantage of Visual programming languages, like Pure Data and the flexibility of code generation, like FAUST and Processing. Mosaicode was initially developed to generate applications to the Computer Vision domain in C/C++ based on the openCV framework. Since Arts are a research field interested in image processing and computer vision, the environment took the attention of artists researching this field. Gradually, new extensions have been developed to attend the digital arts domain bringing together the areas needed to supply the demands of this domain including the processing and synthesis of audio and images, input sensors and controllers, computer vision, computer networks and others [17].

Actually, these are the current extensions being developed to Mosaicode:

(i) **mosaicode-javascript-webaudio**: implements the natives Web Audio API Nodes including the *Script Processor Node* that allows the development of new sound nodes for the Web Audio API. It also implements HTML 5 widgets that compose the generated applications GUI [18]. Further than audio processing and synthesis with Web Audio API, this extension also include other HTML 5 APIs like Web Midi, Gamepad API, Web Socket, WebRTC and others;

(ii) **mosaicode-c-opencv**: implements Computational Vision and Image Processing resources using openCV library for applications generated in the C++ language[19];

(iii) **mosaicode-c-opengl**: implements Computer Graphics resources using the openGL library based on the C programming language[19];

(iv) **mosaicode-c-sound**: with resources for analysis, manipulation and creation of sounds, making synthesis and audio processing;

[1] Available on <https://processing.org/>.
[2] Available on <http://www.puredata.info>.
[3] Available at <https://cycling74.com/products/max>
[4] Available on <http://www.infomus.org/>.
[5] Available on <http://jythonmusic.org>.
[6] Available on <https://faust.grame.fr/>.
[7] Available on https://alice.ufsj.edu.br.

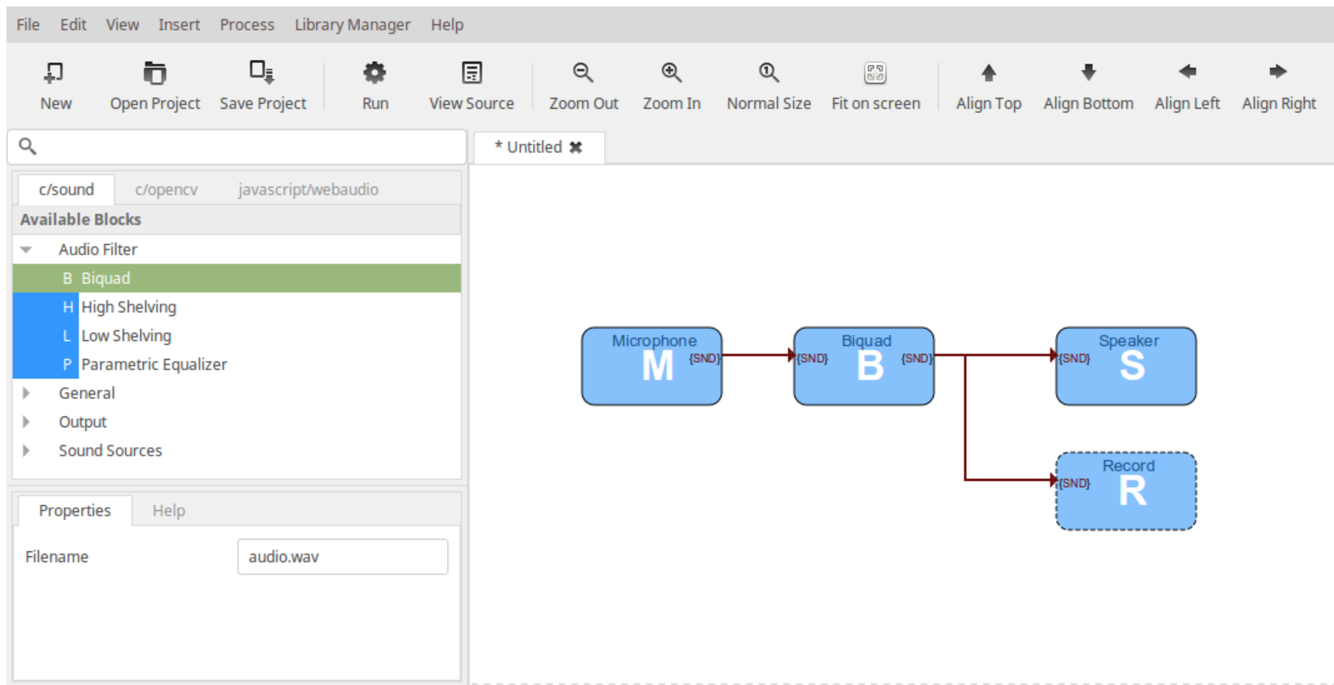

Figure 2. Mosaicode – the Visual Programming Environment presented in this paper.

(v) **mosaicode-c-gtk**: supports the development of GUI using GTK and C language;

2. The extension development

The development of the proposed extension to Mosaicode took three stages, as depicted in Figure 3, i) a Startup process, ii) the library development and iii) the extension development.

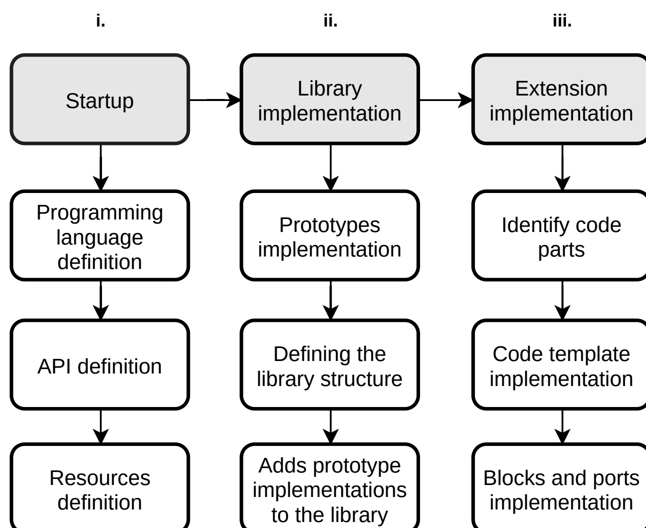

Figure 3. Flowchart of the development methodology of this work splitted into three stages (i, iii and iii).

2.1 The startup process

The first stage of this work, the startup process, was divided into three tasks: *1)* choose the programming language for the generated code; *2)* choose the audio API to aid the development and; *3)* define the resources required for a VPL/DSL that enable digital artists to develop audio applications for the Music Computing domain and to work with sound design.

i.1 - Programming language decision

There was, in our previous work, a concern to choose a suitable programming language for the proposed project as well as APIs that can simplify the development of a DMI, bringing resources already implemented and code reuse, like accessing audio, MIDI and controlling devices, offering good portability, open software license and allowing the integration with other APIs. The process of choosing the language and API was done reading papers and source code of existing tools for audio processing, looking for efficient APIs that could bring up the basic resources to develop DMI applications.

The choice of the API also influenced the choice of the programming language, and vice versa, since the compatibility between both is fundamental to simplify the development of systems. The language chosen should support an efficient audio, MIDI and controller processing, otherwise the result of the application will not be as expected [20].

Most part of the audio APIs available to audio applications development are developed using the C language [21]. In addition, C is a powerful, flexible and efficient language that has the necessary resources for the development of audio [22], so we chose this programming language for the code generated by our Mosaicode extension. Besides, using the C language

could bring interoperability with others extensions present in the environment. These choices were taken in previous work and we chose to generate code to C language.

i.2 - API definition

From several APIs available to sound development, the PortAudio API was chosen to simplify the development of the framework in the musical context. This API was chosen to be used in the audio development since it is a common choice to access audio devices. This access allows one to write to the audio buffer of the output devices and read the audio buffer of the input devices. Being a cross-platform API, PortAudio allows the implementation of audio streams using the operating system audio APIs, making it possible to write programs for Windows, Linux (OSS/ALSA) and Mac OS X [23].

Since PortAudio does not implement access to media files, the *libsoundfile* API was also used to play and record audio files. To process the audio signal from the devices, we have implemented features that allow the development of synthesizers and other audio applications.

For the development of DMIs, it was necessary to include in the library and the extension resources that enable the creation of interfaces to control the parameters of the instruments. Different kinds of feedback were also needed, in addition to audible feedback. The GTK API supported the development of GUI components, adding GUI inputs and visual output for DMIs. GTK has several graphical components already implemented and it is possible to use a canvas with Cairo to create customized GUI elements. Beyond it, GTK is available in several operating systems and it is the base of the Gnome System.

To access control devices and support for MIDI and joystick, two native lib from the Linux operating System was our choice. The MIDI devices was implemented using the ALSA API because it gives more functionalities when compared with PortMidi, like the possibility to create virtual named devices. Another possibility was to use RtMidi, another common option to access MIDI devices. However, RtMidi uses C++ instead of C. The Joystick API used joystick.h, a Linux kernel library that gives access to “uncommon” devices, like USB joysticks and accelerometers.

To enable communication via computer networks we chose to use OSC messages. OSC is a common form of communication to musical devices and it was implemented using the light OSC library (liblo). Moreover, some basic operations that would be useful to convert values, mathematical operations and more was created using ANSI C library like math.h (libm).

These additional features were implemented as a module in the libmosaic-soundlibrary, creating the modules: GUI, MIDI, Joystick, and OSC.

i.3 - Resources definition

After defining the programming language and APIs, we carried out a survey for a VPL/DSL resources that enable digital artists to develop applications to the Music Computing domain and work with sound design. A list of resources was made based on existing tools, cited in Section 1.2, and other libraries to develop system to the same domain, like the Gibberish [24] library.

Table 1. APIs chosen to be used in the development

Modules	API
Sound i/o	Port Audio
Sound files	libsoundfile
GUI	GTK + Cairo
MIDI	ALSA MIDI
OSC	liblo (light OSC)
joystick	joystick.h (kernel lib)
Base	ANSI C

Gibberish has a collection of audio processing classes classified in the following categories: Oscillators, Effects, Filters, Audio Synthesis, Mathematics and Miscellaneous [24]. We have also investigated the native objects of Pure Data and this tool has a list of objects organized in the following categories: General, Time, Mathematics, MIDI and OSC, Miscellaneous, Audio Mathematics, General Audio Manipulation, Audio Oscillators and Tables and Filters of Audio.

By meshing the categories investigated in both tools, the resources were defined to be implemented in Mosaicode in blocks form. For this work we selected some of the resources to be implemented, disregarding resources that can be implemented by combining others, such as audio synthesis. Table 2 presents the resources that have been implemented in the libmosaic-sound library and in the Mosaicode in the blocks form.

Basic GUI components were already available in Mosaicode by mosaicode-c-gtk extension. The extension had some components like a button, entry, label and scale. The spin and switch buttons were added, in addition to the mouse click, mouse move and mouse release events. The best implementation was to create GUI components specially for DMI applications such as volume, VU bar, musical keyboard, waveform and spectrogram. These components are placed in between GUI components and audio components since they have audio processing capability and may work as an output to a sound source. They were added to the libmosaic-sound library and the mosaicode-c-sound extension, reusing the library code.

2.2 The Library implementation

The second stage was the implementation of the libmosaic-sound library. The first task was to implement prototypes of the resources defined in the Startup stage. Then, the code of the prototypes were studied, to define the structure of the library. Finally, the implemented prototypes were added to the library.

ii.1 - Prototypes implementation

With the programming language, API, and resources defined (*in stage i*), the next stage was to implement these resources

Table 2. Resources implemented in libmosaic-sound and in Mosaicode, for generating audio applications. The asterisk symbol (*) indicates that the resource was implemented in the previous work.

Modules	Categories	Resources/Blocks
Sound	Audio Filter	Biquad filter* (All-pass, Bandpass, High-pass, Low-pass), High Shelving*, Low Shelving* and Parametric Equalizer*
Sound	Audio Math	Addition*, Subtraction*, Division* and Multiplication*
MIDI	MIDI	MIDI In and MIDI Out
MIDI	Conversion	Float to MIDI, MIDI to Float, Frequency to MIDI Note and MIDI Note to Frequency
Sound	Conversion	RMS
Sound	Envelope	ADSR
Sound	FX	Delay
Sound	General	Audio Devices* and Channel Shooter Splitter*
Sound	Event	Metronome
GUI	Event	Mouse click, Mouse move and Mouse Release
GUI	GUI	Main window, Grid, Led, Waveform, Spectrogram, Volume and VU bar
GUI	Form	Button, Entry, Label, Scale, Spin and Switch
Sound	Input Device	Microphone
Joystick	Input Device	Joystick
Sound	Output	Record to audio files* and Speaker*
Sound	Sound Sources	Oscillators*, White Noise* and Playback audio files*
OSC	OSC	OSC In and OSC Out

by developing a library to work with sound design, focusing on the development of DMIs. The idea at this stage was to implement an example of each feature separately, instead of starting the library implementation. Thus, it was possible to study these prototypes and think of an appropriate structure for the implementation of the library. It was also better to deal with the difficulties of each implementation separately.

At first, only audio processing resources were implemented, allowing the development of audio applications such as audio synthesis. An audio processing software requires an implementation concerned on the high data processing. An audio application with a sample rate setting of 44100 implies the processing of 44100 audio samples per second. If, at that moment, the processing of the samples has a high computational cost, the result may not be as expected specially if the computational power of the machine is not enough to perform these tasks. With this amount of data to be processed, concern with the time and space complexity of the algorithms was essential.

In addition to audio processing, we implemented examples of GUI components (GTK API), communication with MIDI devices (ALSA API) and Joystick (Joystick API) and communication on computer networks (OSC protocol). The integration of audio processing with GUI components also required performance care. At that moment, we started to have the cost of audio processing added to the cost of drawing the competent GUI and updating it according to the processed audio.

GUI components, such as VU bar and Waveform, graphically represent information about the audio signal. It is infeasible to update these visual feedback for each audio sample. Thus, we allow the configuration of the update rate of the components passing the parameter value in milliseconds. In the implementation, to control the time, the amount of sample that corresponds to the time was used. To control the feedback behavior of the graphic components a IIR filter – Infinite Impulse Response was used.

In the implementation to support MIDI devices, the Abstract Data Type (ADT) called MIDI was created, adding the resources to send and receive MIDI messages. It is possible to send any type of MIDI messages using the *send_event* function, setting the *snd_seq_event_t*[8] type variable that must be passed as a function parameter. In order to simplify the sending of notes and controls, the *send_note* and *send_control* functions were also implemented, receiving by parameter primitive values that the message must contain.

To receive MIDI messages sent by physical or logical devices, the user needs to implement a callback function and pass it as a parameter in the *mscsound_create_midi* initialization function of ADT MIDI. This callback will be called by the ALSA API whenever the message event sent to the associated MIDI device occurs, passing the *send_event* type variable as a parameter. The MIDI message will be contained within the event message and the user can define the destiny of the message implementing the callback function.

Other features available by ADT MIDI are the functions of converting the value of the MIDI note to the corresponding frequency value and the reverse conversion. For converting the MIDI note to the frequency, we created an array with 128 positions (indexes from 0 to 127), initializing the first position with the frequency of 8.18 hz and using the Pythagorean tuning ratio to generate the remaining values. Certainly, other tuning methods should be implemented in the future.

An ADT to the joystick and one to the OSC messages was also added. Following the MIDI implementation, we created a running thread to wait for messages and a callback message

[8] Available on <https://www.alsa-project.org/>.

to receive incoming events. Thus, the user can define how to use the messages just implementing the callback functions.

ii.2 - Defining the library structure

We developed a library to make these resources available and easy to use, also thinking about a structure that is adequate for the development of DMI applications, making it easier to develop it through code reuse.

The prototypes implemented in the previous step were important to define the structure of the library. Studying the source code of each implemented resource and thinking about how the programming in Mosaicode works (creating a diagram composed of blocks and connections), we identified a structure for the library that allows programming similar to the visual programming of Mosaicode, but in a textual form. For each resource, listed in Table 2, an ADT was implemented following the same pattern, as shown below:

- **input**: input data to be processed. ADTs can have more than one input;
- **output**: processed data. ADTs can have more than one output;
- **framesPerBuffer**: buffer size to be processed in each interaction;
- **process**: function that processes the input data and stores it at the output if the ADT has output;
- **create**: function to create/initialize the ADT.
- **others**: each resource has its properties and values to be stored for processing, so there are variables to store these values.

The implementation also included a *namespace* definition using the *mscsound_* prefix added in library functions, types and definitions to ensure that there were no conflicts with reserved words from other libraries. Another detail of implementation is the audio processing without memory copy, using pointers to reference the same memory address to all processing ADTs. If one needs to process two outputs differently, it is possible to use the ADT called *Channel Shooter Splitter*, which creates a copy of the output in another memory space. That way, there will only be memory copy only spending when it is necessary and clearly defined. The details of how to compile, install, and run the code are described on *README.md* file, available on library's repository at *GitHub*[9].

Source Code at Listing 1 shows the ADT that provides the implementation of data capture from a microphone (input device):

Listing 1. ADT Definition *mscsound_mic_t, la* abstracting the microphone implementation.

```
#ifndef MSCSOUND_MIC_H
#define MSCSOUND_MIC_H

typedef struct{
  float **output0;
  int framesPerBuffer;
  void (*process)(void *self, float *);
}mscsound_mic_t;

mscsound_mic_t* mscsound_create_mic(
  int framesPerBuffer);
void mscsound_mic_process();
#endif /* mic.h */
```

[9]Available on https://github.com/Alice-ArtsLab/libmosaic-sound/blob/master/README.md.

ii.3 - Adds prototype implementations to the library

The implementation of an application using the libmosaic-sound library depends on some functions that must be defined by the developer and functions that must be called. These functions are described below:

- **mscsound_callback**: a function called to process the input values for every block. This function overrides the PortAudio callback thread copying data read from application's ring buffer to the Portaudio audio output buffer [25]. User must override this library function;
- **mscsound_finished**: function called by the library when the callback function is done. User must also override this function;
- **mscsound_initialize**: function that user must call to initialize the audio application;
- **mscsound_terminate**: function that the user must call to end up the audio application. This function finish the library cleaning up memory allocation.

As an example of the libmosaic-sound library usage, Figure 4 presents the running flow of a code that capture the audio with a microphone, store the audio in an audio file and send the audio to the computer speaker.

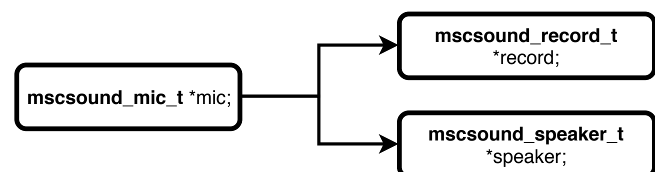

Figure 4. The running flow of a simple audio application developed with the ADTs of the libmosaic-sound library.

The ADTs also have some code patterns in the library definition. Some code parts called *declaration*, *execution*, *setup* and *connections* have been defined so one can use these code parts to define the implementation of the audio application, being that:

- **declaration**: code part to declare the ADTs used in the code.

- **execution**: code part to define the call order of the *process* functions of each ADT declared in the code part *declaration*. This part must be included within the Mosaicode callback function;
- **setup**: code part to initialize ADTs variables, defining their values and calling their respective *create* functions;
- **connections**: part of the code to define the connections between the ADTs. These connections defines the audio processing chain associating the output of a ADT to the input of another ADT.

For the identification of the code parts, several examples have been created using all the library ADTs. Looking at these examples, it was possible to identify the characteristics of each code part listed above. The implementation code of the libmosaic-sound library and the examples are available on GitHub[10], including new modules in addition to Sound: GUI, MIDI, Joystick and OSC.

2.3 Extension implementation

The last stage (*stage iii*) consisted in the implementation of the extension to develop a DMI within the Mosaicode programming environment and using the library previously developed to complete this task.

iii.1 - Identify code parts

To create the extension its needed to implement a code template. The code template informs the code generator of the Mosaicode how to generate source code. By setting the code template, the Mosaicode generator can interpret the diagram and generate the desired source code. Thus, the first step of this stage was to observe in the library and examples developed in stage *ii* the code parts that are common in every example, independently of the implementation, and the unusual parts that are different in every code example.

The code parts that are generated from the diagram are those cited in the development of the libmosaic-sound library in Section 2.2 – *declaration*, *execution*, *setup* and *connections*. The remaining code will always be the same in all implementations, so it is fixed in the code template.

iii.2 - Code template implementation

The second step was to create a code template. This process is summarized in the definition of the following attributes:

- **name** (string): code template name;
- **language** (string): code template language;
- **description** (string): code template description;
- **command** (string): command to compile the source code;
- **code_parts** (string array): name of code parts;
- **files** (dictionary): generated files – file name is the index and the item is the file content.

[10] Available on https://github.com/Alice-ArtsLab/libmosaic-sound.

In the code template files, wildcards were fixed indicating to the generator the location of each part to replace it with code, for example: *$code[declaration]$* are replaced by code. The implementation of the new modules in the library includes new API dependencies, making it necessary to change the code template, adding the necessary lines to be able to use the APIs. In addition to including the APIs and initializing them, it was added to the compile command the flags for linking.

iii.3 - Blocks and ports implementation

The third step was to create the input/output port types for the blocks connections. Three types of ports have been implemented for the extension: SOUND, MIDI and OSC. In the implementation, the connection code is defined, establishing how an output block port must be connected to an input block port. The *var_name* attribute is also defined, which informs the code generator a pattern to define the variable that stores the port value. The Mosaicode automatically generates these connections by interpreting the block diagram.

There are two forms to create a connection between ports. The first one is simply assigning an output value, normally stored into a variable, to an input value, also stored into a variable. This form, that we call it "passive", sends the value from one block to the other. However, since the value is only a parameter passing, it does not allow to take a decision and perform some function when receiving a value from a block.

The other form, that we call it "active", is based in the Observer design pattern [26]. In this implementation, the output ports are arrays and the input ports are functions. When we connect the output port to an input port, the function corresponding to the input port is added to the output port array. In this way, the propagation of values is performed by going through the array of the output port calling the functions contained in each position. The functions of the input ports, in addition to using the value received by parameter, passes this value to every output port registered in the array. It is also possible to take decisions in this function and perform some actions when receiving a value.

The connection of the SOUND port is passive, where the output port variable receives the memory address that stores the value of the input port. The MIDI, OSC and primitive ports (integer, float, char and string – implemented in the mosaicode-c-base extension) implement an active port.

With the code template and the port created, the last step was the implementation of the blocks for the Mosaicode. Each developed block contains the code abstraction of a resource defined in stage *i*. This strategy allows a reuse of code by using the library developed in stage *ii*.

The GUI module of the library contains the grid component that allows organizing the other GUI components. The grid (GtkWidget) is one of the component organization options, offered by the GTK API. For the extension, we chose to use the configuration of a fixed layout, where the user informs

the x and y position that the component must be inserted in the main window. As it is simpler to configure, we believe it is more suited to the Mosaicode proposal to make implementations simpler. This fixed layout was recently adopted and it was necessary to adapt existing extensions to that implementation. The disadvantages of this fixed layout is that the components can present in an unorganized way, overlapping other components and not adapting to the program window's resizing.

The implementation code of the mosaicode-c-sound extension – blocks, ports and code template – are available on GitHub[11]. This extension uses ports implemented in the mosaicode-c-base extensions, creating a dependency. In addition, the base extension implements common features in programming languages, complementing the other extensions. All Mosaicode extensions are available on GitHub[12].

3. Results

The developed extension and library offer resources for working with audio processing, enabling audio applications such as DMIs. In addition to sound processing, as presented in Section 1.1, the development of a DMI also needs resources to provide user control (input) and other feedback (output). To supply this need for inputs and outputs, GUI components were implemented using the GTK API and features for communicating with MIDI devices, using the ALSA MIDI API. Support for the joystick device has also been added, implementing with the Joystick API, and the OSC protocol has also been added for network communication between devices and applications. In this way, now it is possible to control audio processing through graphical interfaces and physical devices such as MIDI controllers and joysticks, including communication by computer network. This expansion of resources allows a fully development of DMIs, easily allowing the exchange of resources used in each layer of the instrument: Input, mapping and output. The development can be done through textual programming, using only the library, or visual programming, using the extension to Mosaicode.

This work resulted in a library for DMI development packed as an extension to the Mosaicode programming environment defining a Visual Programming Language to musical applications development. With this VPL, we simplified application development for Computer Music domain, allowing to generate audio applications and work with sound design by dragging and connecting blocks. We hope it can increasing the facility of digital artists to work with audio applications development.

The developed VPL brings all the resources offered by the libmosaic-sound library, including simple waveform sound sources, enabling the implementation of audio synthesis, sound effects and envelopes to the generation of more complex sounds. It is possible to implement classic synthesizing examples like AM, FM, additive and subtractive synthesizers and implement other techniques of Computer Music, without worrying about code syntax and commands, just dragging and connecting blocks. The user also has the option to obtain the source code of the application defined by the diagram, having complete freedom to modify, study, distribute and use this code.

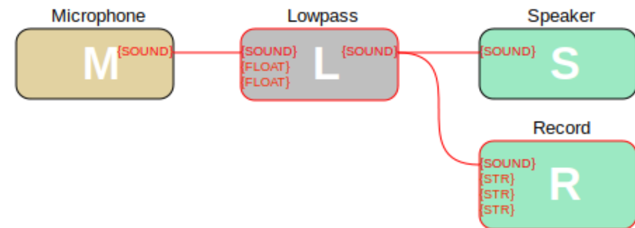

Figure 5. Example of a Mosaicode diagram using the mosaicode-c-sound extension.

Figure 5 shows a diagram as an example of using the extension developed in this work. In this example we apply the lowpass filter (Biquad) to an audio signal captured by a microphone. The filter output is directed to the speaker and recorded in an audio file. Another examples is available in the extension repository, already available in this document in the Section 2.3.

Comparing the generated code with implementations using PortAudio API, we can notice that the audio structure was maintained. The difference is that function calls are made instead of implementing the abstracted code in these functions.

The example in Figure 6 creates a DMI, adding and ADSR envelope to the oscillator output. To control the instrument, communication with a MIDI controller was added, which triggers the envelope event when playing a note on the device. Volume control (GUI), audible feedback and visual feedback VU Bar and Waveform have also been added. Some blocks were used during the mapping, the block to strip the MIDI message, to convert the MIDI note value to the frequency value, to convert the float value to integer and the block to multiply the audio signal with the float value.

Figure 7 shows the GUI generated by the diagram in figure 6. The VU Bar and Waveform components draw their representations of the audio signal the moment a key was pressed on the MIDI devices. There is also the volume control, showing the current value.

4. Conclusion

This work proposes the development of an extension for audio application development within the Mosaicode visual programming environment. This development allows the generation of source code from diagrams composed of blocks and connections, making the sound design more accessible to digital artists.

In the first stage of this project, a study has been done to define a programming language and APIs to develop our library. It was also necessary to define resources for a DSL/VPL

[11] Available on https://github.com/Alice-ArtsLab/mosaicode-c-sound .
[12] Available on https://github.com/Alice-ArtsLab.

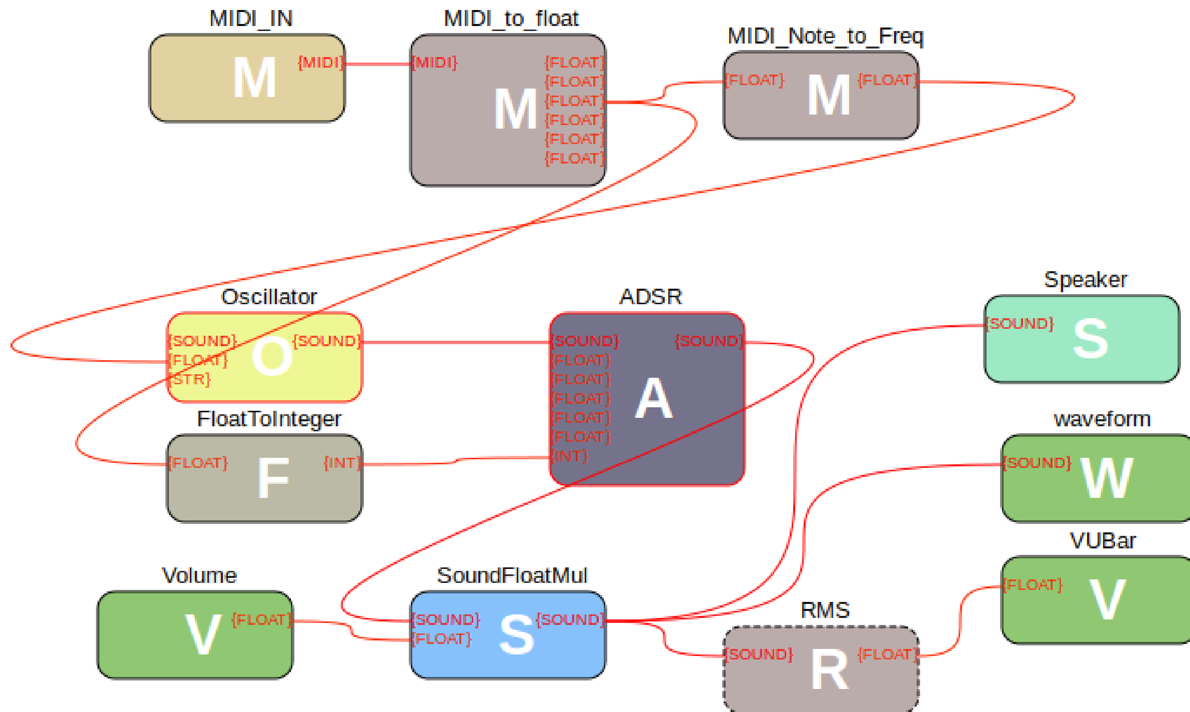

Figure 6. Example using SOUND, MIDI and GUI modules.

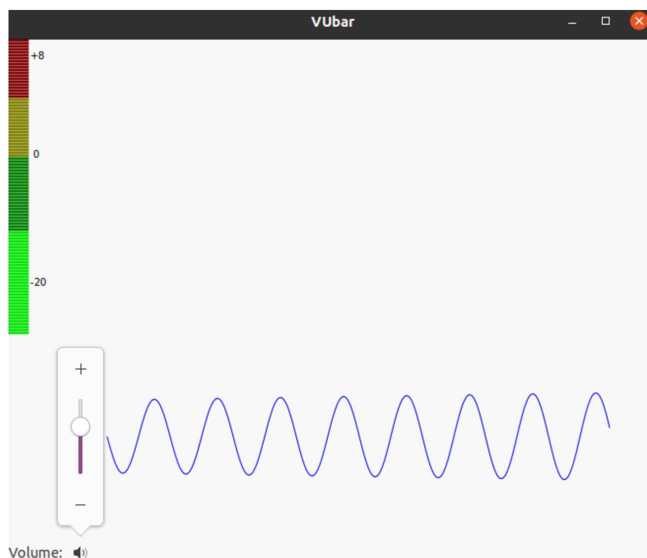

Figure 7. GUI generated from diagram of the example in the Figure 7.

that would supply the needs of digital artists in the development of applications for the Computer Music domain. In addition, research of the Related tools, like Pure Data, and the Gibberish library helped to define these resources.

In the second stage we discussed the development of the libmosaic-sound library, which supported the implementation of the mosaicode-c-sound extension for the Mosaicode and allows the development of audio applications in an easier way. The library structure is analogue the manipulation of Mosaicode blocks and connections, as if each ADT is a block and each assignment between output and input was a connection. This structure has drastically reduced the number of lines necessary to develop an audio application compared to the direct use of the PortAudio API. It happens mainly because this API provides only the manipulation of input and output interfaces, requiring the user to implement the processing of the data read/written by the interfaces to generate the applications.

In the third stage we discussed the development of the mosaicode-c-sound extension to work with sound design in the Mosaicode. This extension was based on the libmosaic-sound library, in which each block uses resources developed and present on the library. In this way, only library function calls are made, making it easier to implement the blocks and generating a smaller source code. This implementation resulted in a VPL for the Computer Music domain and, because it was developed in Mosaicode , allows the generation of the source code that can be studied and modified. In addition, it contributes to Mosaicode with one more extension.

For implementation of the code template in Mosaicode, first, we created several examples of codes using the libmosaic-

sound library. These examples have been studied in order to understand each code part and define which parts are fixed in the code template and which parts are generated by the extension blocks.

4.1 Limitations and Discussions

Until now, the library and the extension have gone through some stages of development. We first implemented the prototypes of resources for working with sound, which were studied to define the structure of the library and were reused in the implementation of the library. Then, we implemented the Mosaicode extension using the library. With the need for resources besides sound, for the creation of DMIs, we separated the resources of the library by modules. To add the new features, we followed the same procedure from creating examples. Then we implemented the resources in their respective library modules, and then add them to the extension.

When inspecting the generated code, it is possible to notice that, despite using only blocks of the sound module, the generated code contains codes referring to the GUI module. The same happens if we do not use the sound module, some lines of unnecessary codes will be inserted, referring to the sound module. To clean the generated code by adding only the used modules, we will have to add a new code generation feature in Mosaicode that includes code generation in the code template. In this way, codes for specific modules will only be added when the diagram contains blocks that depend on these code snippets.

4.2 Future works

We intended to review the list of resources in order to expand the library and the extension for audio application and improve the library usability. An example of this would be the use of macros to allow calling functions from ADTs without having to pass the variable by parameter, so the user will just type *<variable name>.<function name>()* instead of *<variable name>.<function name>(<variable name>)*.

To create the library documentation, the Doxygen[13] tool will be used to generate the documentation from the source code. The possibility of generating the documentation for the Mosaicode extensions will be studied, also from the source code.

It is also intended to link this project to other projects of the Mosaicode development team. There are several works in progress implementing extensions to Digital Image Processing, Computer Vision, Artificial Intelligence, Computer Networking and Virtual Reality domains. The intention is to connect all these extensions in the environment, offering resources to generate more complex applications for the specific domains of digital art.

Acknowledgements

Authors would like to thanks to all ALICE members that made this research and development possible. Also would like to thank the support of the funding agencies CNPq, (Grant Number 151975/2019-1) and FAPEMIG.

[13] Available on <https://www.doxygen.nl/>.

Author contributions

Luan Luiz Gonçalves developed the library and the extension during his undergrad in Computer Science and now is working on it in his Master degree, both supervised by Flávio Luiz Schiavoni. Both authors use to code, write and play together.

References

[1] GONÇALVES, L.; SCHIAVONI, F. The development of libmosaic-sound: a library for sound design and an extension for the mosaicode programming environment. In: SCHIAVONI, F. et al. (Ed.). *Proceedings of the 17th Brazilian Symposium on Computer Music*. São João del-Rei - MG - Brazil: Sociedade Brasileira de Computação, 2019. p. 99–105.

[2] HAEBERLI, P. E. Conman: A visual programming language for interactive graphics. *SIGGRAPH Comput. Graph.*, ACM, 1988.

[3] HILS, D. D. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, Elsevier, 1992.

[4] GRONBACK, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. [S.l.]: Addison-Wesley, 2009.

[5] MERNIK, M.; HEERING, J.; SLOANE, A. M. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, ACM, 2005.

[6] DEURSEN, A. V.; KLINT, P. Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, SRCE-Sveučilišni računski centar, 2002.

[7] GOMES, A. L. N. et al. Prototyping web instruments with mosaicode. In: *Proceedings of the 17th Brazilian Symposium on Computer Music*. São Jão del-Rei - MG - Brazil: [s.n.], 2019. p. 114–120.

[8] HUNT, A.; WANDERLEY, M. M.; KIRK, R. Towards a model for instrumental mapping in expert musical interaction. In: CITESEER. *ICMC*. [S.l.], 2000.

[9] REAS, C.; FRY, B. *Processing: a programming handbook for visual designers and artists*. [S.l.]: Mit Press, 2007.

[10] PUCKETTE, M. S. et al. Pure data. In: *ICMC*. [S.l.: s.n.], 1997.

[11] BRINKMANN, P. et al. Embedding pure data with libpd. In: CITESEER. *Proceedings of the Pure Data Convention*. [S.l.], 2011.

[12] WRIGHT, M. et al. Supporting the sound description interchange format in the max/msp environment. In: *ICMC*. [S.l.: s.n.], 1999.

[13] CAMURRI, A. et al. Eyesweb: Toward gesture and affect recognition in interactive dance and music systems. *Computer Music Journal*, MIT Press, 2000.

[14] CAMURRI, A. et al. Developing multimodal interactive systems with eyesweb xmi. In: *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*. New York, NY, USA: ACM, 2007. (NIME '07), p. 305–308. Disponível em: <http://doi.acm.org/10.1145/1279740.1279806>.

[15] MANARIS, B.; STEVENS, B.; BROWN, A. R. Jythonmusic: An environment for teaching algorithmic music composition, dynamic coding and musical performativity. *Journal of Music, Technology & Education*, Intellect, v. 9, n. 1, p. 33–56, 2016.

[16] ORLAREY, Y.; FOBER, D.; LETZ, S. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, Editions Delatour, Paris, France, v. 290, p. 14, 2009.

[17] SCHIAVONI, F. L.; GONÇALVES, L. L. From virtual reality to digital arts with mosaicode. In: *2017 19th Symposium on Virtual and Augmented Reality (SVR)*. Curitiba - PR - Brazil: [s.n.], 2017. p. 200–206.

[18] SCHIAVONI, F. L.; GONÇALVES, L. L.; GOMES, A. L. N. Web audio application development with mosaicode. In: *Proceedings of the 16th Brazilian Symposium on Computer Music*. São Paulo - SP - Brazil: [s.n.], 2017. p. 107–114.

[19] GOMES, A. L. N.; RESENDE, F. R.; SCHIAVONI, F. L. Desenvolvimento de extensões de processamento e síntese de imagens para a ferramenta mosaicode. In: *Proceedings of the CONFERENCE ON GRAPHICS, PATTERNS AND IMAGES, 31 (SIBGRAPI)*. Foz do Iguaçu - PR - Brazil: [s.n.], 2018. p. 1–4.

[20] JIANG, Z.; ALLRED, R.; HOCHSCHILD, J. *Multi-rate digital filter for audio sample-rate conversion*. [S.l.]: Google Patents, 2002.

[21] SCHIAVONI, F. L.; GOULART, A. J. H.; QUEIROZ, M. Apis para o desenvolvimento de aplicações de áudio. *Seminário Música Ciência Tecnologia*, 2012.

[22] ANDERSEN, L. O. *Program analysis and specialization for the C programming language*. Tese (Doutorado) — University of Cophenhagen, 1994.

[23] NELSON, M.; THOM, B. A survey of real-time midi performance. In: NATIONAL UNIVERSITY OF SINGAPORE. *Proceedings of the 2004 conference on New interfaces for musical expression*. [S.l.], 2004. p. 35–38.

[24] ROBERTS, C.; WAKEFIELD, G.; WRIGHT, M. The web browser as synthesizer and interface. In: CITESEER. *NIME*. [S.l.], 2013.

[25] SOSNICK, M. H.; HSU, W. T. Implementing a finite difference-based real-time sound synthesizer using gpus. In: *NIME*. [S.l.: s.n.], 2011. p. 264–267.

[26] NAUMOVICH, G. Using the observer design pattern for implementation of data flow analyses. *ACM SIGSOFT Software Engineering Notes*, ACM New York, NY, USA, v. 28, n. 1, p. 61–68, 2002.