The development of libmosaic-sound: a library for sound design and an extension for the Mosaicode Programming Environment

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

[1] Arts Lab in Interfaces, Computers, and Everything Else - ALICE
Computer Science Department – DCOMP
Federal University of São João del-Rei – UFSJ

luanlg.cco@gmail.com, fls@ufsj.edu.br

Abstract. *Music has been influenced by digital technology over the last few decades. With the computer, 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) for audio applications in the Mosaicode programming environment, simplifying sound design and making the synthesis and manipulation of audio more accessible to digital artists. It is also presented the implementation of libmosaic-sound library for the specific domain of Music Computing, which supported the VPL development.*

1 Introduction

Music has been influenced by technology for decades, especially after technological advances, bringing the idea of music and technology together, providing new electronic instruments and new ways of making music. With the computer, musical composition goes beyond the limitations of the artist's body and its acoustic instruments and it started requiring knowledge of computer programming for the development of audio applications and compositions. Since the skills to create a music piece can be totally different from the ability to develop a software, digital artists can find it difficult to start their research and work with digital art due to non-computer programming knowledge.

Fortunately, it is possible to program a computer application using non-textual programming paradigms. Visual Programming Languages (VPLs) allow programmers to develop code using a two-dimensional notation and interacting with the code from a graphical representation [1]. The usage 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 [2], a feature that can help even experienced programmers. Textual programming paradigms require the use of one-dimensional stream of characters code, demanding the memorization of commands and textual syntax while visual programming languages are more about data flow and abstraction of software functionalities.

Another possibility to further simplify the software development is to use a Domain-Specific (Programming) Language (DSL) [3]. 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 within your domain easier and more efficient because DSLs require more knowledge about the domain than programming knowledge [4]. Hence, the potential advantages of DSLs include reduced maintenance costs through re-use of developed resources and increased portability, reliability, optimization and testability [5].

Merging the readiness of VPLs and the higher abstraction of DSLs, 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 1, 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. Hence, 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 domains.

Initially Mosaicode was developed to generate applications to the Computer Vision domain in C/C++ based on the openCV framework. Gradually, new extensions have being 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 [6]. Figure 2 shows the areas involved in the generation of applications for digital art by Mosaicode and highlights the area of Music Computing as the area approached in this work [6].

Apart from the extension to Computer Vision in C/C++ language, Mosaicode also has extensions to generate applications in Javascript/HTML5. These are the current Mosaicode extensions to support the development of applications for specific domains of digital art:

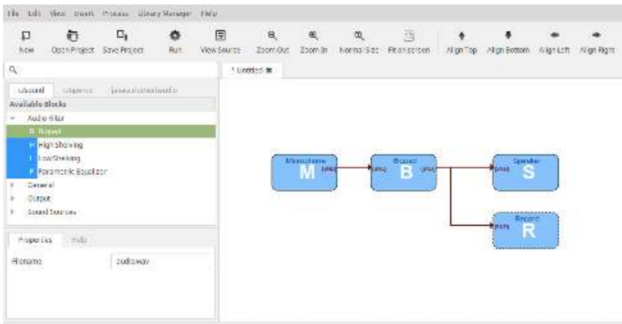

Figure 1: Mosaicode screenshot – the Visual Programming Environment presented in this paper.

(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 [7]. 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;

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

(iv) **mosaicode-c-joystick**: based on C language, allows to control applications using joysticks as interface;

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

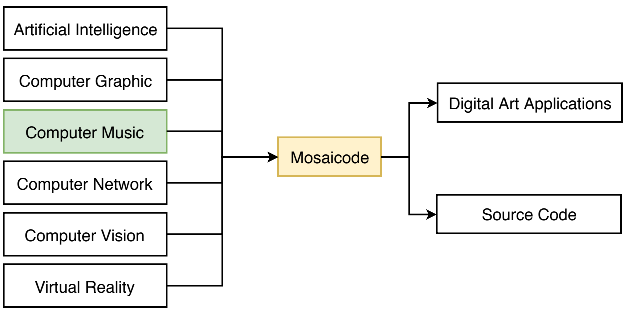

Figure 2: The scheme shows the Computer Science areas involved in the generation of applications for digital art in the Mosaicode.

In order to unleash the development of audio applications in C language, this work presents an extension for the Mosaicode environment focused on the Computer Music domain. This extension in Mosaicode is intended to simplify sound design (manipulation and creation of sounds), making audio synthesis and sound processing more accessible to digital artists. For the language of the generated code was chosen the programming language C, used in the libmosaic-sound library. This library is also the result of this project, developed to assist in the development of the extension, with the aim of facilitating this development by reducing the effort required to implement it.

The libmosaic-sound library was based on the PortAudio API to access the audio input and output system and provide resources that this API does not have implemented on it. To read and write audio files, the libsoundfile API was also used and integrated into our library. So, the user can effortlessly generate applications for the Computer Music domain in C and integrate it with another APIs available for this programming language. The library structure provides this ease of use programming framework and made it easier to implement the blocks in Mosaicode, resulting in the VPL for the Music Computing domain.

2 Related tools

The tools presented below are widely used by digital artists and are considered to be related to this research.

Processing[1] is a programming language and an Integrated Development Environment (IDE) developed by the MIT Media Lab[8]. 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 Java code and runs the generated code.

Pure Data[2] or simply PD is a graphical real-time programming environment for audio and video [9] 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 [10], 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 [11]. 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 [12]. According to the authors, this information from body motion processing can be used to create and control

[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/`

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 [13]. 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. The user can interact with external devices such as MIDI, create graphical interfaces and also manipulate images [14].

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[15].

The present project brings the advantage of Visual programming languages, like Max/MSP and Pure Data and the flexibility of code generation, like FAUST and Processing. All together, this project can be an alternative to these programming languages and programming environments.

3 The extension development

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

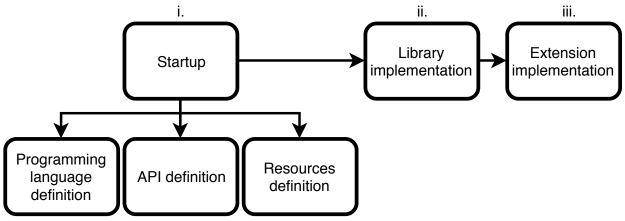

Figure 3: Flowchart of the development methodology of this work splitted into three stages (i, iii and iii).

3.1 The start up process

The first stage of this work, The start up process, was divided into three parts: *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.

There was a concern to choose a suitable language for the proposed project as well as an API that can simplify the development, bringing resources already implemented, like the access of the audio input and output device, and offering good portability, free software license and allowing the integration with other APIs, like MIDI, OSC and sensors in the future. The process of choosing the language and API was done reading papers and source code of existing tools for audio processing, looking for an efficient API that could bring up the basic resources to develop audio applications.

The choice of the API also influenced the choice of the programming language since the compatibility between both is fundamental to simplify the development of systems. Another concern for implementing audio applications is the efficiency of the programming language. The language chosen should support an efficient audio processing, otherwise the result of the application will not be as expected [16].

Most part of the audio APIs available to audio applications development are developed using the C language [17]. In addition, C is a powerful, flexible and efficient language that has the necessary resources for the development of audio [18],so we chose this programming language for the code generated by Mosaicode. Besides, using the C language could bring interoperability with others extensions present in the environment.

From several APIs available to sound development, the PortAudio API was chosen to simplify the development of the framework in the musical context. 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. PortAudio uses the MIT license and can be integrated with PortMidi, a library to work with the MIDI standard [19]. Since PortAudio does not implement access to media files, the *libsoundfile* API was also used to play and record audio files.

After defining the programming language and the audio API, 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 2, and other libraries to develop system to the same domain, like the Gibberish [20] library.

Gibberish has a collection of audio processing classes classified in the following categories: Oscillators, Effects, Filters, Audio Synthesis, Mathematics and Miscellaneous [20]. 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

[5] Available on `http://jythonmusic.org`
[6] Available on `https://faust.grame.fr/`

that can be implemented by combining others, such as FM synthesis and envelopes. Table 1 presents the resources that have been implemented in the libmosaic-sound library and in the Mosaicode in the blocks form.

Table 1: Resources implemented in libmosaic-sound and in Mosaicode, for generating audio applications.

Categories	Resources/Blocks
Audio Filter	Biquad filter (All-pass, Bandpass, High -pass, Low-pass), High Shelving, Low Shelving and Parametric Equalizer.
Audio Math	Addition, Subtraction, Division and Multiplication.
General	Audio Devices and Channel Shooter Splitter.
Output	Record to audio files e Speaker.
Sound Sources	Oscillators, White Noise, Microphone and Playback audio files.

4 LIBMOSAIC-SOUND Library

With the programming language, API, and resources defined (*in stage i*), the next stage was to implement these resources by developing a library to work with sound design. This library, called libmosaic-sound, had to implement the listed resources looking for an easy way to implement audio applications and requiring less programming effort to complete this task. Existing literature, like the book *DAFX – Digital Audio Effects* [21], aided the implementation of these resources.

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 audio applications, making it easier to develop it through code reuse. The library was also designed to not depend on the PortAudio API beyond the access to the audio devices. The PortAudio was used only to list the input and output audio interfaces and to set up the resources of those interfaces. The *libsoundfile* API was used to read a media file and to record audio signals to file.

For each resource, listed in Table 1, an Abstract Data Type (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*[7].

Source Code at Listing 1 shows the ADT that abstracts 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 */
```

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 [22]. 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 the computer speaker.

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

[7] Available at `https://github.com/Mosaicode/libmosaic-sound/blob/master/README.md`

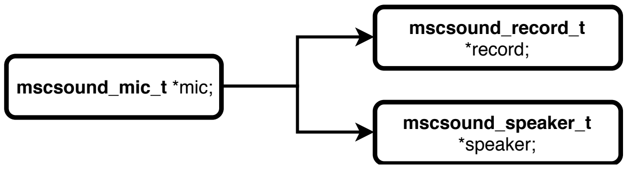

Figure 4: The running flow of a simple audio application developed with the ADTs of the libmosaic-sound library.

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[8].

5 MOSAICODE-C-SOUND Extension

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

To create the extension, properties such as name, programming language, description, command to compile, code parts and code template implementation have been defined. 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 4 – *declaration*, *execution*, *setup* e *connections*. The remaining code will always be the same in all implementations, so it is fixed in the code template.

[8]Available at `https://github.com/Mosaicode/libmosaic-sound`

The second step was to create the input/output port types for the blocks connections, which in this case was just one type, the sound-type port. The connection code has also been defined, establishing how an output block port must be connected to an input block port. The Mosaicode automatically generates these connections by interpreting the block diagram. 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 Mosaicode blocks can have dynamic and static properties. Dynamic properties can be changed at run-time using the block input ports. Static properties can be changed at programming time, before generating the source code.

The implementation code of the mosaicode-c-sound extension – blocks, ports and code template – are available on GitHub[9].

6 Results

This work resulted in a library for audio application 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.

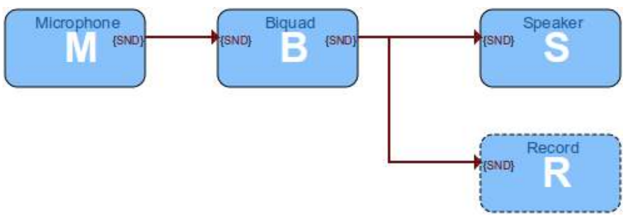

Figure 5: Example of a Mosaicode diagram using the mosaicode-c-sound extension.

Figure 5 shows a diagram as an example of using the extension developed in this work. In this example we

[9]Available at `https://github.com/Mosaicode/mosaicode-c-sound`

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. The code generated from the diagram of the Figure 5 is shown next. Another examples is available in the extension repository, already available in this document in the Section 5.

Listing 2: Code generated from the diagram in Figure 5.

```
#include <mosaic-sound.h>
#include <portaudio.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define NUM_SECONDS 12
#define SAMPLE_RATE 44100
#define FRAMES_PER_BUFFER 256

/* Declaration part */
mscsound_mic_t *block_1;
mscsound_biquad_t *block_2;
mscsound_record_t *block_3;
mscsound_speaker_t *block_4;

static int mscsound_callback(
  const void *inputBuffer,
  void *outputBuffer,
  unsigned long framesPerBuffer,
  const PaStreamCallbackTimeInfo
    *timeInfo,
  PaStreamCallbackFlags statusFlags,
  void *userData) {

    float *in = (float *)
      inputBuffer;
    float *out = (float *)
      outputBuffer;

    (void)timeInfo; /* Prevent unused
      variable warnings. */
    (void)statusFlags;
    (void)userData;

    /* Execution and Connection
parts */
    block_1->process(block_1, in);

    block_2->input = block_1->output;
    block_2->process(block_2);

    block_3->input = block_2->output;
    block_3->process(block_3);

    block_4->input = block_2->output;
    block_4->process(block_4, out);

    return paContinue;
}

static voimscsound_finished(
  void *data) {
    printf("Stream Completed!\n"); }

int main(int argc, char *argv[]) {

  /* Setup part */
  block_1 = mscsound_create_mic(
    FRAMES_PER_BUFFER);
  block_2 = mscsound_create_biquad(
    1, 2, FRAMES_PER_BUFFER);
  block_2->sampleRate = SAMPLE_RATE;
  block_2->cutOff = 3000.0;
  block_2->slope = 0.1;
  block_3 = mscsound_create_record(
    "examples/record_mic_lowpass.wav",
    FRAMES_PER_BUFFER, 44100);
  block_4 = mscsound_create_speaker(
    FRAMES_PER_BUFFER);

  void *stream = mscsound_initialize(
    SAMPLE_RATE, FRAMES_PER_BUFFER);

  printf("Recording until
    the Enter key is pressed.\n");
  getchar();

  mscsound_terminate(stream);
  return 0;
}
```

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.

7 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 the appropriate programming language and audio APIs to complete the proposed task. There was also a need to define the necessary resources for a DSL/VPL 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 Mosaicodeand 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 also drastically reduced the number of lines a user needs to write developing 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 a 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.

This project also contributed to the development of Mosaicode, which has undergone code refactoring in order to improve its structure and simplify its maintenance and extension.

7.1 Future works

We intended to review the list of resources in order to expand the library and the extension for audio application. 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.

8 Acknowledgments

Authors would like to thanks to all ALICE members that made this research and development possible. The authors would like also to thank the support of the funding agencies CNPq and FAPEMIG.

References

[1] Paul E. Haeberli. Conman: A visual programming language for interactive graphics. *SIGGRAPH Comput. Graph.*, 1988.

[2] Daniel D Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 1992.

[3] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley, 2009.

[4] Marjan Mernik, Jan Heering, and Anthony M Sloane. When and how to develop domain-specific languages. *ACM computing surveys (CSUR)*, 2005.

[5] Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *CIT. Journal of computing and information technology*, 2002.

[6] Flávio Luiz Schiavoni and Luan Luiz Gonçalves. From virtual reality to digital arts with mosaicode. In *2017 19th Symposium on Virtual and Augmented Reality (SVR)*, pages 200–206, Curitiba - PR - Brazil, Nov 2017.

[7] Flávio Luiz Schiavoni, Luan Luiz Gonçalves, and André Lucas Nascimento Gomes. Web audio application development with mosaicode. In *Proceedings of the 16th Brazilian Symposium on Computer Music*, pages 107–114, São Paulo - SP - Brazil, 2017.

[8] Casey Reas and Ben Fry. *Processing: a programming handbook for visual designers and artists*. Mit Press, 2007.

[9] Miller S Puckette et al. Pure data. In *ICMC*, 1997.

[10] Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. Embedding pure data with libpd. In *Proceedings of the Pure Data Convention*. Citeseer, 2011.

[11] Matthew Wright, Richard Dudas, Sami Khoury, Raymond Wang, and David Zicarelli. Supporting the sound description interchange format in the max/msp environment. In *ICMC*, 1999.

[12] Antonio Camurri, Shuji Hashimoto, Matteo Ricchetti, Andrea Ricci, Kenji Suzuki, Riccardo Trocca, and Gualtiero Volpe. Eyesweb: Toward gesture and affect recognition in interactive dance and music systems. *Computer Music Journal*, 2000.

[13] Antonio Camurri, Paolo Coletta, Giovanna Varni, and Simone Ghisio. Developing multimodal interactive systems with eyesweb xmi. In *Proceedings of the 7th International Conference on New Interfaces for Musical Expression*, NIME '07, pages 305–308, New York, NY, USA, 2007. ACM.

[14] Bill Manaris, Blake Stevens, and Andrew R Brown. Jythonmusic: An environment for teaching algorithmic music composition, dynamic coding and musical performativity. *Journal of Music, Technology & Education*, 9(1):33–56, 2016.

[15] Yann Orlarey, Dominique Fober, and Stéphane Letz. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 290:14, 2009.

[16] Z. Jiang, R.W. Allred, and J.R. Hochschild. Multi-rate digital filter for audio sample-rate conversion, 2002.

[17] Flávio Luiz Schiavoni, Antonio José Homsi Goulart, and Marcelo Queiroz. Apis para o desenvolvimento de aplicações de áudio. *Seminário Música Ciência Tecnologia*, 2012.

[18] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[19] Mark Nelson and Belinda Thom. A survey of real-time midi performance. In *Proceedings of the 2004 conference on New interfaces for musical expression*, pages 35–38. National University of Singapore, 2004.

[20] Charles Roberts, Graham Wakefield, and Matthew Wright. The web browser as synthesizer and interface. In *NIME*. Citeseer, 2013.

[21] Udo Zolzer. *DAFX: Digital Audio Effects*. Wiley Publishing, 2nd edition, 2011.

[22] Marc H Sosnick and William T Hsu. Implementing a finite difference-based real-time sound synthesizer using gpus. In *NIME*, pages 264–267, 2011.