Phausto: fast and accessible DSP programming for sound and music creation in Pharo

Domenico Cipriani[1], Nahuel Palumbo[2], Sebastian Jordan Montaño[2] and Stéphane Ducasse[2]

[1]*Pharo Association*
[2]*Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, Park Plaza, Parc scientifique de la Haute-Borne, 40 Av. Halley Bât A, 59650 Villeneuve-d'Ascq, France*

Abstract

This paper introduces Phausto, a library that generates sounds in Pharo programming language using Faust (Functional Audio Streams), a programming language designed to develop real-time digital signal processors (DSP).

In Phausto, DSP programs are created by the composition of Unit Generators written in a MUSIC-N style, like the ChucK programming language, or from a string containing a valid Faust program.

We present Phausto's API, implementation details and an overview of its syntax, and of Unit Generators and ToolKit elements. We also analyze the motivations behind the project and identify its target audiences. Finally, we present the conclusions drawn after one year of development and use, and outline the agenda for future work.

Keywords

sound synthesis, live coding, audio, DSP programming, Pharo, Faust, ChucK

1. Introduction

We present Phausto: a sound generation library for the Pharo programming language. Phausto is a library created with the primary goal of integrating state-of-the-art sound synthesis inside the Pharo ecosystem, allowed by a dynamic engine (Section 4). This engine connects a Faust [1] (Functional Audio Streams) program generated to an underlying PortAudio [2] layer that, in turn connect with the platform-specific low-level audio API. Together with these emerging sonic capabilities comes a new quest: a transparent and effective API to develop and design Digital Signal Processors (DSP) such as synthesisers and effects, both to be used in Pharo applications and for creative musical composition.

The Phausto API (Section 5) is designed to be modern and straightforward: its semantics and the subdivision of Faust's standard library functions into Phausto Unit Generators subclasses is inspired by the ChucK [3] programming language, which takes this concept from MUSIC-N style programming music languages [4]. Unit Generators (UGens) are basic building blocks for signal processing algorithms that were first developed by Max Mathews and Joan E. Miller for the Music III program [5] in 1960. UGens include processing modules such as oscillators, filters, envelopes and effects that can be connected to create synthesis instruments (sometimes referred as *patches*).

DSP can be created in Phausto from a string containing a valid Faust program, as in:

```
dsp := DSP create: 'import("stdFaust.lib");
process =  pm.marimba(400 ,3 ,7000 ,0.25 , 0.5 , ba.pulse(20000)) +  pm.marimba
    (800, 3 ,7000 ,0.25 , 0.5 , ba.pulse(10000));'
```

or by connecting Unit Generators using the Pharo basic syntax:

```
dsp := ((Marimba new freq: 800; trigger: (Pulse new period: 0.2) ) +
       (Djembe new freq: 800; trigger: (Pulse new period: 0.1) )) asDsp.
```

IWST 2024: International Workshop on Smalltalk Technologies, July 9–11, 2024, Lille, France
mspgate@gmail.com (D. Cipriani); nahuel.palumbo@inria.fr (N. Palumbo); sebastian.jordan@inria.fr (S. Jordan Montaño); stephane.ducasse@inria.fr (S. Ducasse)
0009-0005-3023-3192 (D. Cipriani)

All Unit Generators have default values for their parameters, which are themself Faust boxes, too. The main contributions of this paper are:

- A description of the Phausto library architecture and implementation details.
- Examples of sound synthesis using Phausto API.
- An overview of the Phausto Standard Library and ToolKit.

2. Motivation

We develop Phausto to generate sound directly from the Pharo environment, without needing external dependencies to play performance during live coding sessions.

We have identified three primary target audiences for Phausto:

1. Pharo programmers who want to include sounds or sonic interaction in their Pharo applications, including procedural audio designed Faust extensive physical modelling synthesis library;
2. Artists with little or no computer literacy who want to design and develop synthesisers and effects for their music creation fast and easily. Pharo offers a transparent syntax with a neat IDE and a browser that enables users to avoid complex library management; Phausto ensures all Faust libraries are readily accessible as well-documented classes and methods.
3. Students and beginners who want to develop their audio plug-ins, by exporting DSPs developed in Pharo to Cmajor[1] patches thanks to the Faust2CMajor export[2]. Cmajor is a new C-family programming language for writing fast and portable audio software, created by Julian Storer and Cesare Ferrari. Cmajor patches can be exported as native VST/AU/AAX plugins or loaded directly into a Digital Audio Workstation (DAW) with the Cmajor VST/AU plugin

3. Phausto Overview and Architecture

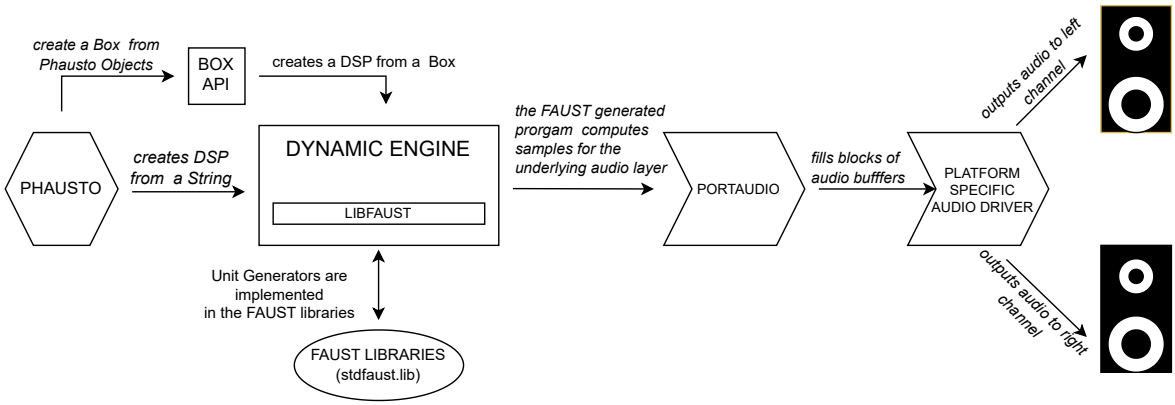

Figure 1: Overview of Phausto library and the embedded Faust dynamic engine.

Phausto is an open-source Pharo library to create synthesisers and audio effects that are converted into Faust programs by an embedded Faust compiler. Figure 1 shows an overview of its architecture.

[1] https://cmajor.dev/

[2] https://github.com/grame-cncm/Faust/tree/master-dev/architecture/cmajor#Faust2cmajor

The Dynamic Engine in the backend initialises the PortAudio library and opens a stream for audio I/O. It creates a DSP from strings of valid Faust codes or Boxes created with the Phausto API. The definitions of the Unit Generators are implemented in the Faust libraries, which are accessed by the Dynamic Engine when DSPs are created.

The DSP is a Faust program that computes floating point numbers between -1.0 and 1.0 as output. These values fill the blocks of a PortAudio stream. The platform-specific audio driver transforms the PortAudio stream into sound, using the sound interface Digital-To-Analog converter.

4. Under-the-hood: Faust programming language

Faust (Functional Audio Stream) [1] is a functional programming language tailored to sound synthesis and audio processing created at the GRAME-CNCM Research Department[3]. It provides developers with an alternative to C/C++ for designing and deploying DSPs. One of its key features is a block diagram-oriented syntax, which eases a modular approach to the creation of synthesisers and effects. A Faust program describes a signal processor, essentially mapping input signals to output signals, but it does not outline a connection to the external world[4].

4.1. Faust dynamic engine

The dynamic engine implementation describes a straightforward C API for Faust objects including the audio drivers responsible for rendering the samples computed by the DSP and the library used for reading audio data files. Faust DSP objects are structures that can be either instantiated from a string of valid Faust code by calling the `createDsp` function:

```
dsp* createDsp (const char* name_app, const char* dsp_content,
      int argc, const char* argv[], const char* target, int opt_level);
```

or from Boxes (Section 4.3) with the `createDspFromBoxes` function:

```
dsp* createDspFromBoxes(const char* name_app, Box box,
      int argc, const char* argv[], const char* target, int opt_level);
```

After being created, DSP needs to be initialized by calling the `initDSP` function with a sampling rate and buffer size for the audio driver. [5]. Finally, `start` function is called to open the audio stream and processing audio blocks until `stop` function is called.

```
bool initDsp(dsp* dsp, RendererType renderer, int sr, int bsize);
bool startDsp(dsp* dsp);
```

The dynamic engine also allows access to and modification of DSP parameters using the `getParamValueDsp` and `setParamValueDsp` functions:

```
FaustFLOAT getParamValueDsp (dsp* dsp_ext, int p);
FaustFLOAT setParamValueDsp (dsp* dsp_ext, int p, FaustFLOAT val);
```

The dynamic engine includes the Faust Compiler [7], *libFaust* in Figure 1. The compiler generates bytecode from the Faust Imperative Representation, which is then successively executed by a stack-based virtual machine. We chose this approach for its lightweight nature and zero dependencies, although other options with faster execution, as an LLVM [8] backend[6]. The Faust compiler meets the

[3] https://Faust.grame.fr/

[4] https://Faustdoc.grame.fr/manual/architectures/

[5] For a more detailed description of sampling rates, audio buffer size and audio drivers, refer to the first chapters of The Audio Programming Book [6]

[6] Linked with components of the LLVM compiler toolchain, the library allows the deployment of a complete dynamic compilation chain from source to executable code.

performance needs of Phausto intended goals and audience; as interactions with the Faust program occur only once it has been created and initialized, the choice between the LLLVM compiler and the backend interpreter does not affect the control on-the-fly of the DSP parameters.

4.2. The PortAudio layer

To maintain compatibility across different operating systems and keep the ToolKit as lightweight as possible, we leverage PortAudio. PortAudio is an Open Source Cross Platform C library[7] and API for audio input and output [2]. It was designed to simplify the development of real-time audio applications, and it is part of a larger initiative called PortMusic[8] that also includes MIDI[9] capabilities.

PortAudio renders the samples computed by the Faust interpreter with the PortAudio and Faust's native library[10]. This approach ensures all the dependencies (including all Faust libraries) to be less than 10 MB.

PortAudio handles the connection with the audio input and audio ports on the host platform. It internally manages audio stream buffers and requests audio processing from the client application via a callback that is associated with an opened stream. In the dynamic engine, this association is managed by the following function:

```
bool initDsp(dsp* dsp, RendererType renderer, int sr, int bsize);
```

For instance, in Phausto we initialise a DSP by calling:

```
initDsp(aDSP, 0, 44100, 512);
```

This function sets the sampling rate and the buffer size and allows one to choose a different renderer from PortAudio if needed.

4.3. The Box API

The Faust C box API[11] allows for the programmatic creation of a box expression, which is used to create a DSP object. This stage is an intermediate public entry point created in the *Semantic Phase* of Faust's compilation chain. The Semantic Phase is the first step of Faust's compilation chain; starting from the DSP source code written in Faust, the Semantic Phase produces signals that are (conceptually) infinite streams of samples. Boxes can be created by calling a specific function defined in *libFaust-box-c. e.g.,* for creating a UI button:

```
Box CboxButton(const char* label);
```

Or from a string of Faust code by calling:

```
Box CDSPToBoxes(const char* name_app, const char* dsp_content,
      int argc, const char* argv[], int* inputs, int* outputs, char* error_msg);
```

The Box API also supplies methods to convert numbers into Boxes and to connect Boxes via the five binary composition operators of the language[12]

[7] https://github.com/PortAudio/portaudio
[8] https://www.cs.cmu.edu/~music/portmusic/
[9] https://midi.org/
[10] https://github.com/grame-cncm/Faust/blob/master-dev/architecture/Faust/gui/Soundfile.h
[11] https://Faustdoc.grame.fr/tutorials/box-api/
[12] In Faust there are five binary composition operations to combine block diagrams: *sequential(:)* , *parallel(,)* , *split(<:)* *merge (:>)* , *recursive (~)* . For a detailed description of Faust binary operators see [1].

5. Phausto programming by example: Fast and easy sound synthesis

Programming DSP with Phausto is fast and easy because it combines the modular synthesiser [9] creative and flexible mindset with the Pharo Object Oriented Programming paradigm. Phausto takes advantage of Pharo's concise and iconic syntax, the reflectiveness of the environment, and the ease of developing reusable code. With Phausto, Pharo becomes a fully-fledged IDE for audio synthesis and algorithmic composition without needing additional extensions (in contrast to other IDEs and code editors in mainstream programming languages). Moreover, manipulating the DSP parameters using Pharo syntax provides musicians with a tool offering unnoticeable latency[13] for their live performances and compositions.

5.1. Start your engine

```
pulse := PulseOsc new. "Instantiate an Unit Generator"
dsp := pulse stereo asDsp. "Create the DSP in stereo mode"
dsp init. "Initialize the DSP"
dsp start. "Start the sound"
dsp stop. "Stop the sound"
dsp destroy. "Destroy the DSP"
```

Listing 1: Simple example using Phausto

Listing 1 shows a simple example using Phausto playing a Pulse Oscillator. First, we instantiate the UGen to play, in this case, a Pulse Oscillator. Then we create a stereo DSP by sending the messages `asDSP` to our Pulse Oscillator. (To change the mono output of a synth to stereo, we send the `stereo` message to the Box that is the result of the multiplication).

Once our DSP is created, we initialize and start it to hear the sound it generates. We can stop the DSP whenever we want. If we are finished with our DSP programming, it would be a good practice to destroy it.

5.2. Taking Control of Configuration and Parameters

Each UGen has different configuration parameters with default values, each of them documented in the class comments. For example, to control the parameters of a Pulse Oscillator we need to specify the *freq* - for frequency in Hertz, and *duty* - to change the oscillator duty cycle (between 0 and 1).

```
pulse := PulseOsc new freq: 220; duty: (LFOTriPos new freq: 4).
dsp := pulse stereo asDsp.
dsp init.
dsp start.
```

Listing 2: Configuring UGen parameters

[13] The latency time refers to the duration required to execute the message `setValue:parameter:` (which is is an FFI call), to execute. If we run a simple benchmark:

```
[1000 timesRepeat: [dsp setValue: (Random new nextIntegerBetween: 90 and: 900) parameter:
'SquareOscFreq' ]] timeToRun.
```

The results show that a single call takes approximately 0.007 milliseconds. At a 192 kHz sample rate, this latency is slightly more than 1 sample, making it effectively instantaneous compared to the rate at which the computer can generate sound. However, it's important to consider that the latency of calling the function `void setParamValueDsp(dsp* dsp_ext, int p, FaustFLOAT val);` through the dynamic engine primarily depends on the buffer size and sample rate parameters used in the Phausto audio layer. By default, Phausto initialize its DSPs with a 44100 sample rate and a buffer size of 512 samples. This corresponds to a latency of approximately 11.6 ms, which is below the 20 ms threshold generally considered to be imperceptible or minimally disruptive for human perception, as demonstrated by the Haas effect.

The code in Listing 2 sets the frequency of the Pulse Oscillator to 220 Hz (default frequency is 440 Hz) and assigns the value of the duty cycle to a Low-Frequency Oscillator (LFO) with a Triangular Positive Waveform (*LFOTriPos* oscillates between 0 and 1).

5.3. Live programming

To modify a parameter of a UnitGenerator in real time, while the DSP is playing, we create a PhHSlider by sending the message `init: initialValue` to a `ByteSymbol`. This creates a `PhHSlider` with the `ByteSymbol` as a label and the required initial value. Then, we control the parameter *on live* by sending the message `setValue: parameter:` to our DSP.

```
lfoFreq := #lfoFreq init: 2.
pulse := PulseOsc new freq: 220; duty: (LFOTriPos new freq: lfoFreq ).
dsp := pulse asDsp.
dsp init.
dsp start.

"Change the value of a parameter while the sound is playing"
dsp setValue: 8 parameter: 'lfoFreq'.

... "Continue using the DSP until the end"
```

Listing 3: Changing UGen parameters while playing

Listing 3 shows an example of changing the frequency of the LFO from 2 to 8 Hz after the DSP is started.

5.4. Music composition with Pharo Processes

Once we have started our DSP we can create music compositions using Pharo Processes, Blocks and Delays.

```
synth := (SquareOsc new freq: #SquareNote) * (AREnv new trigger: #SquareGate).
dsp := (synth => SatRev new) stereo asDsp.
dsp init.
dsp start.
note := 72.
time := 2.

[
    24 timesRepeat: [
        dsp playNote: note prefix: 'Square' dur: 0.1.
        time wait.
        note := note + 1 .
        time := time * 0.85 ]
] fork.

dsp stop.
dsp destroy.
```

Listing 4: Algorithmic music composition

For instance, code in Listing 4 plays a sequence of 24 chromatic notes starting from C5 (MIDI note number 72) playing at exponentially shorter intervals of time.

In this example, we have created a Square Oscillator which has its frequency connected to a PhHSlider (to be controlled), multiplied for an Attack-Release Envelope controlled by other PhHSlider.

The `playNote:prefix:dur:` method modifies two parameters:

- it converts a MIDI note number to a frequency in Hertz and sends this value to the `SquareNote` parameter;
- it sends the value 1 and after 0.1 seconds a value of 0 to the parameter `SquareGate`.

Essentially, the `playNote:prefix:dur:` the method simulates the action of playing a key on a MIDI keyboard. For this operation, our DSP needs a parameter with the *gate* suffix to trigger its envelope and a parameter named *Note* for controlling the frequency. Both parameters must share the same prefix, which is *'Square'* in this example.

6. Core Implementation

Phausto allows Pharo users to use functions and data structures from the Faust programming language via Foreign Function Interface (FFI) [10]. The library contains the API to instantiate, access and modify DSP and create Faust Boxes as described in Section 4.

To use Phausto, the *librariesBundle* folder must be downloaded from the GitHub repository and placed next to the current Pharo image[14]. The libraries bundle includes hundreds of functions for synthesis and processing audio written in Faust (see Section 7).

6.1. DSP creation from Phausto Boxes

We create UnitGenerators objects in Phausto using the Faust Box API (described in Section 4.3). It enables writing DSP in Pharo with a dedicated API that hides Faust syntax from the user.

The Box API class provides the connection to over 50 functions that we have considered fundamental for developing the Phausto API. These functions include those to create Faust primitives, composition operators and mathematical expressions. The methods that create or combine boxes return a new Box. If the operation fails, a null pointer is returned.

6.2. The lifecycle of a DSP

A DSP object is originally initialised (*i.e.,* connected to the PortAudio layer) and subsequently started, stopped, and at the end of its use destroyed to avoid dangling pointers. All these methods call to the corresponding function defined in the *dynamic engine*.

Once a DSP is initialised we can check its parameters on a Transcript (traceAllParams) and modify the value of its parameters sending the message `setValue: aParameter` with a Number or a Box as the first argument and with a String as the second argument.

6.3. Managing the global compilation context

To use the Faust Box API we create a singleton global compilation context before we create a Box, and destroy this context after the Box has been used to create a DSP. For this reason, the Box class has a shared variable called `libContext` not initialised when the Pharo image is opened. Every call to the BoxAPI class in Phausto first ensures the creation of the context by calling `createLibContextFFI` if not exist. At the same time, every time a DSP is created, the compilation context is destroyed by calling BoxAPI unique instance `destroyLibContext`.

7. Phausto Standard Library and the ToolKit

The Phausto Standard Library includes the bindings to functions in the standard Faust library *stdFaust.lib* [15]. They are implemented as subclasses of UnitGenerator and organised in package tags, while the *ma.lib* (which includes mathematical operations) is ported as methods of the PhBox superclass.

[14] https://github.com/lucretiomsp/phausto
[15] https://faustlibraries.grame.fr/

The design of the Phausto API exposes all original arguments as instance variables of the UGen object and default values are set during initialisation[16]. The setters convert their arguments into a Box, allowing the user to use both, fixed numbers or other boxes (*e.g.,* other UnitGenerators, or a UIPrimitive to control the variable in real-time).

One specific UnitGenerator is created by sending the `asBox` message. This method first creates an *intermediateBox* from a Faust code string without parameters and returns a *finalBox* from the connection of the instance variables to the *intermediateBox*. Listing 5 presents the binding between the `djembe` function in Faust and the `Djembe` class in Phausto.

```
"djembe function in physmodels.lib"
pm.djembe(freq, strikePosition, strikeSharpness, gain, trigger) : _

"Djembe class Phausto"
Djembe >>> initialize
    super initialize.
    processExpression := 'process = pm.djembe;'.
    self freq: 440.
    self strikePosition: 0.5.
    self strikeSharpness: 0.8.
    self gain: 0.5.
    self trigger: 0.0

Djembe >>> asBox
    | intermediateBox finalBox |
    intermediateBox := super asBox.
    finalBox := (freq , strikePosition , strikeSharpness , gain , trigger )
connectTo: intermediateBox.
    ^ finalBox
```

Listing 5: Pharo Djembe binding

The ToolKit[17] tag of the Phausto package includes a set of classes that do not have a corresponding function in the standard Faust library. These classes encompass units for sound generations that combine one or more UnitGenerators.

The ToolKit includes a combination of effects and modulators (*e.g.,* FIlterEnvelope or FilterLFO), effects and synthesizers with a custom design (*e.g.,* DelayFeedBack or SamplePlayer) or 'utilities' as the *Incrementer* (that increments its output by an increment value at every step).

8. Related work

One of the primary sources of inspiration for Phausto is Kyma [12], an object-oriented environment for music composition written in Smalltalk-80, created by Carla Scaletti in the mid of 1980. The first version of Kyma ran on a Macintosh 512k computer and could not manipulate sound in real time. This is why Scaletti and Kurt J. Hebel collaborated to integrate a separate multiprocessor to compute and process audio [13]. Although there is no direct connection between Phausto and Kyma, we share Scaletti's desire to make music programming more accessible to composers without a strong background in computer science. Our goal is also to make music creation more accessible to programmers without a strong background in music and sound synthesis.

Pharo programmers could play sound samples in Pharo since Pharo 9.0 using the Sound package[18].

[16]The default values have been an arbitrary choice, influenced by other programming languages such as ChucK, PureData, SuperCollider, as well as our direct experience and by empirical audio tests during development.

[17]The ToolKit name is a homage to the Synthesis ToolKit [11] in C++(STK)by Perry R. Cook and Gary Scavone, as its primary goal is to facilitate faster development of synthesisers and effects.

[18]https://github.com/pharo-contributions/Sound

Pharo Sound plays samples and sounds by FFI calls to the SDL2 library[19]. This package is poorly documented and offers very few audio generators and effects based on a Pharo implementation of low-level C techniques for computing samples. Consequently, its API is difficult to access and complex to expand, it does not use terminology consistent with modern audio programming practices.

FaustPy[20] is a Python wrapper for Faust implemented using CFFI that works with Python 2.7 and 3.2+. It is the only wrapper for a high-level general-purpose programming language supporting object-oriented programming.

9. Future Work

In its first year of development, Phausto has become a cross-platform stable library and we have been able to provide bindings to more of the 30% of the Faust Box API. Additionally, 25% of the functions of the Faust standard library has been implemented as classes and tested. In the following months, our focus will be on porting all the functionalities of the Faust programming language. This includes the ability to visualize the DSP block diagram as a .svg file and the possibility to save DSP written in Pharo as a file, which can be exported to different target languages, such as CMajor, C++ and WAST[21].

Along with the implementation of all the UnitGenerators, the new ToolKit classes will be designed and implemented to meet the needs of musician and sonic artists. As Phausto expands, more effort will be required to prepare valid tests for all the UnitGenerators, which include an inquiry into how to write valid tests for audio generators.

In addition, we have planned to develop a new package called TurboPhausto is specifically designed to program music on the fly with Coypu. TurboPhausto is inspired by the SuperDirt engine for SuperCollider. It provides a set of ready-to-use synthesizers and effects and an API to easily play a collection of audio samples located in subfolders of a specific folder. We will include a substantial amount of high-quality samples for instant gratification and the option to expand the collection.

Finally, in collaboration with the Evref team, we are working on a live programming development environment. It encompasses a custom Playground and a set of custom UI elements (faders, rotary sliders, buttons, and more) for Phausto made with Toplo[22].

10. Conclusion

After ten months of development, Phausto is a comprehensive solution for enabling sound synthesis into Pharo applications. Phausto offers a diverse ToolKit that includes sample players, basic oscillators, envelopes, various physical models, resonant filters, reverbs, and delays. The extensive list of Unit Generators features a streamlined API for parameter manipulation.

Additionally, TurboPhausto's synthesizers and effects were showcased in a 30-minute live performance titled "Riding the MoofLod" during the ESUG2024 conference at Lille.

Acknowledgments

We would like to thank Stephane Sletz, Yann Orlarey, Guillermo Polito, Esteban Lorenzano and Pablo Tesone, for their invaluable support, push and assistance throughout the development of Phausto.

We express our gratitude to the organisation of the ESUG for the inspiration and the knowledge we gained during the conferences. In particular, the meetings with the members of the Pharo team and GRAME in Lyon 2023. It was instrumental in the birth of Phausto.

[19] SimpleDirectMedia is a cross-platform development library designed to provide low-level access to audio, keyboard, mouse and graphics hardware.
[20] https://github.com/marcecj/Faust_python
[21] https://webassembly.js.org/docs/contrib-wat-vs-wast.html
[22] https://github.com/pharo-graphics/Toplo

Special thanks to Carla Scaletti and Kurt J. Hebel at Symbolic Sound Corporation fo their lifelong effort to spread love for Smalltalk and computer music, and to Cristian Vogel for pushing Kyma into the world of electronic dance music.

This work has been supported by the Pharo Consortium.

References

[1] E. Gaudrain, Y. Orlarey, A FAUST Tutorial, 2003. URL: https://hal.science/hal-02158895, manual.

[2] R. Bencina, P. Burk, Portaudio-an open source cross platform audio api, in: ICMC, 2001.

[3] G. Wang, P. Cook, Chuck: A concurrent, on-the-fly audio programming language, 2003.

[4] G. Wang, The chuck audio programming language. "a strongly-timed and on-the-fly environ/mentality", Ph.D. thesis, USA, 2008. AAI3323202.

[5] C. Roads, The Computer Music Tutorial, MIT Press, Cambridge, MA, USA, 1996.

[6] R. Boulanger, V. Lazzarini, The Audio Programming Book, The MIT Press, 2010.

[7] S. Letz, Y. Orlarey, D. Fober, An Overview of the FAUST Developer Ecosystem, in: International Faust Conference, Mainz, Germany, 2018. URL: https://hal.science/hal-02158929.

[8] C. Lattner, V. Adve, Llvm: a compilation framework for lifelong program analysis and transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.

[9] M. Vail, The synthesizer: a comprehensive guide to understanding, programming, playing, and recording the ultimate electronic music instrument, Oxford University Press, 2014.

[10] G. Polito, S. Ducasse, P. Tesone, T. Brunzie, Unified ffi - calling foreign functions from pharo, 2020. URL: http://books.pharo.org/booklet-uffi/.

[11] P. R. Cook, G. P. Scavone, The synthesis toolkit (stk), 1999.

[12] C. Scaletti, Composing sound objects in kyma, Perspectives of New Music (1989) 42–69.

[13] M. Heying, A Complex and Interactive Network: Carla Scaletti, the Kyma System, and the Kyma User Community, University of California, Santa Cruz, 2019.