PHAUSTO: EMBEDDING THE FAUST COMPILER IN THE PHARO WORLD

Domenico Cipriani, Alessandro Anatrini, Sebastian Jordan Montaño

PHAUSTO: EMBEDDING THE FAUST COMPILER IN THE PHARO WORLD

Domenico Cipriani *

Pharo Association - Lille, France
mspgate@gmail.com

Alessandro Anatrini

Hochschule für Musik und Theater (HfMT), Hamburg - Germany
Conservatorio Statale di Musica J. Tomadini - Udine, Italia
alessandro@anatrini.com

Sebastian Jordan Montaño

Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL F-59000 Lille, France
sebastian.jordan@inria.fr

ABSTRACT

Phausto is a lightweight, open-source library for live-coding music, enabling sound generation and Digital Signal Processing (DSP) programming. Developed in the Pharo programming language, it incorporates the Faust compiler for robust audio capabilities, using Foreign Function Interface (FFI) calls for seamless integration. Phausto connects with platform-specific audio layers through PortAudio, offering a consistent API across operating systems. Designed for educational settings, it targets users interested in DSP, musicians, and sound artists with limited technical skills. Phausto addresses two main challenges: generating audio in Pharo applications and providing an accessible environment for programming digital musical instruments. It is easy to install and supports the latest Pharo versions, with instructions available on its GitHub repository.

1. INTRODUCTION

Phausto is a library for live-coding music. It enables sound generation and DSP (Digital Signal Processing) programming. Phausto is free and open source, lightweight (only 10 MB including the Faust libraries), and an accessible tool developed primarily to provide developers with a fast and easy way to integrate sound into their programs and applications. Phausto is well-suited for educational use, particularly in environments that emphasize hands-on, exploratory learning. It targets users interested in learning DSP programming, musicians, and sound artists with limited computer proficiency who want to learn about computer music.

Phausto is implemented in the Pharo programming language. It has an embedded Faust [3] compiler for producing the sound. We chose Faust because it offers incredible audio-programming capabilities [2]. The interaction between Pharo and Faust is enabled through Foreign Function Interface (FFI) calls to a dynamic library allowing seamless integration with Faust libraries and the Box-API. Phausto also manages the connections to platform-specific audio layers via PortAudio, a cross-platform audio library that provides a consistent API for audio input and output across different operating systems. From educational and artistic perspectives, Phausto aims to serve as a functional alternative to Faust and as an introductory tool for users needing more advanced DSP or custom solutions.

Phausto is easy to install. It operates on the latest stable version of Pharo, ensuring backward compatibility up to Pharo 10. Detailed installation instructions for Phausto can be found in the Phausto GitHub repository: `https://github.com/lucretiomsp/phausto`.

* This work was supported by the Pharo Association

The Phausto Library addresses two key challenges:

1. generating audio in Pharo applications;
2. providing an accessible environment for sound artists with limited computer literacy to program digital musical instruments (DMIs).

2. THE PHARO PROGRAMMING LANGUAGE

We chose Pharo as our implementation platform because it has an easy-to-read-and-learn syntax with only seven reserved words. Pharo [4, 5] is a pure object-oriented programming language that is dynamically typed. Pharo is a modern implementation of Smalltalk [6, 1] that started in 2008. It is multi-platform and has a vibrant community worldwide, welcoming coders of all experience levels.

Its simple syntax makes Pharo resemble a pidgin language[1] [7]. Pharo is also an integrated development environment (IDE) that offers a live coding environment where programmers can modify their code during execution. At the same time, GUI widgets can be opened or easily created and used in real-time development.

3. INSIDE PHAUSTO

The communication between Faust and Pharo depends on three technologies: the Faust dynamic engine, Pharo's unified Foreign Function Interface (uFFI) , and Faust Box API. Figure 1 provides an overview of the Phausto's architecture. In the following section we will discuss the implementations detail of Phausto. These implementation details require an advanced understanding of Pharo and Faust, which is only relevant within the context of this paper. The final user of Phausto does not need to know about the details of how Unit Generators are initialised and converted to `PhBox`. To combine Unit Generators and create DSP, users only need to know how to set their instance variables and how they can be patched together. All this information is contained in the class comments. This concept of *abstraction* is fundamental within the object oriented paradigm.

[1] In computing, a "pidgin language" refers to a programming language with a simplified syntax and minimalistic design, akin to a pidgin language in linguistics. Just as a pidgin language simplifies communication between speakers of different native languages, a pidgin programming language simplifies code writing and reading by reducing complexity and removing extraneous features. This approach aims to make the language easier to learn and use.

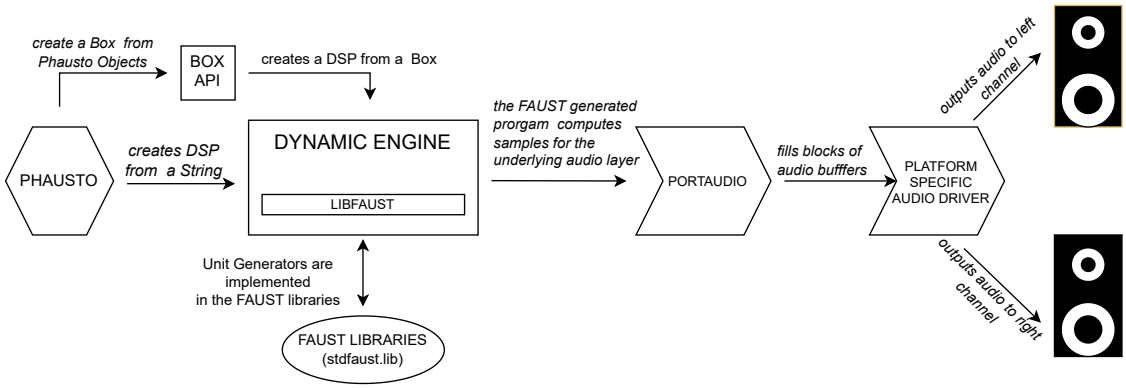

Figure 1: *A simplified diagram illustrating Phausto's framework architecture. The dynamic engine libraries use* libfaust *and the* Faust libraries *to transform into DSP programs the strings of code or the combinations of boxes written inside Pharo. The computed samples feeds the PortAudio stream that is finally rendered into sound by the platform specific audio driver.*

3.1. The Dynamic Engine

The dynamic engine is a Faust DSP architecture developed by Stéphane Letz [2]. Its C API details how to create Faust objects, initialize them with a sample rate and a buffer size and start and stop their operation. This dynamic engine can be packaged with an interpreter backend and a basic *WaveReader* for reading audio files, instead of the default LLVM compiler backend. To meet our design goals, we opted for the interpreter backend and the WaveReader due to their lack of external dependencies. Additionally, we selected PortAudio for its cross-platform compatibility. While we acknowledge that the interpreter backend is slower compared to the LLVM compiler, we do not anticipate this will impact Phausto's target audience.

New DSP objects can be instantiated from a string containing a Faust program using the following function:

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

This function is called sending the `create: aString` message to the DSP class. It can be considered the easiest way for a Faust programmer to create DSP in Phausto and it was our first choice to test the functioning of our framework.

Alternatively, we can create a DSP object from a box or a combination of boxes[3]. This approach is more flexible because it allows the user define the connections between boxes in Phausto code, taking advantage of Pharo syntax highlighting. The following C function is called when the `asDsp` message is sent to a `PhBox`[4]

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

Both methods return a pointer to the created DSP objects on success, or a null pointer on a failure; if a failure occurs, a call to:

```
{const char* getLastError();
```

returns the error.

3.2. Pharo Unified Foreign Function Interface (UFFI)

A Foreign Function Interface (FFI) is a programming mechanism that enables the use of functions and data structures written and compiled in a different language [8]. Typically these "foreign" resources are shared libraries, such as `.dll`, `.so`, and `.dylib` files on Windows, Linux, and macOS respectively.

The Pharo uFFI API framework allows us to use implementations written in `faust-dynamic-engine.h`, in `libfaust-c.h` and in `libfaust-box-c.h`. We have implemented all functions from `faust-dynamic-engine.h` as FFi calls in Pharo. Below is an example of the FFI call used to create a DSP from boxes:

[2] `https://github.com/grame-cncm/faust/blob/master-dev/architecture/faust/dsp/faust-dynamic-engine.h`

[3] A Box is an intermediate representation of a Faust primitive, or of a DSP. Boxes expression can be created and combined though a C/C++ . Boxes enable precise and granular manipulation of DSPs (Digital Signal Processing units) through our higher-level Phausto API, allowing for detailed tuning and adjustment of their operational parameters in Pharo code.

[4] The `PhBox` class is a subclass of Pharo `FFIOpaqueObject` that is essentially a pointer to a Faust box.

```
PhaustoDynamicEngine >> #createDspFromBoxes
    : aFaustBox

    ^ self ffiCall:
    #( DSP * createDspFromBoxes #( const
        char * 'MyApp', #PhBox * aFaustBox,
        int 0, void * 0, const char * ' ',
        int -1 ) )
```

3.3. The Box API

The Faust Box API serves as an intermediate public entry point in the *Semantic Phase*[5] of Faust's compilation process. It facilitates the programmatic construction of a box expression, which is subsequently used to instantiate a DSP object. Boxes can be created by invoking a specific function defined in `libfaust-box-c.h`. For example, to create a Checkbox:

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

Or from a string containing a Faust program:

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

In Phausto, we have created a subclass of `FFILibrary` named `BoxAPI` to handle all the bindings to `libfaust-box-c.h`. The `BoxAPI` provides the interface to the Faust Box API, which, in turn, enables us to define a custom API for constructing DSP objects. This design allows Pharo users to develop their own DSP without needing to understand or use Faust syntax directly.

Currently 45 functions from the Faust Box-API have been implemented as Pharo methods. This includes the five binary composition operations, most of the C-equivalent primitives, the *Wire* and the *Cut* boxes, as well as all UI primitives. The following example demonstrates the Pharo implementation of the FFI call to the `CboxHSlider` function:

```
BoxAPI >> #boxHslider: aLabel init: initBox
     min: minBox max: maxBox step: stepBox

    self createLibContext.

    self ffiCall:
    #( #PhBox * CboxHSlider #( const char *
         aLabel , #PhBox * initBox, #PhBox *
         minBox , #PhBox * maxBox , #PhBox *
         stepBox ) )
```

All functions from `libfaust-box-c.h` are implemented in Pharo as methods (FFI calls) within the `BoxAPI` class, and are available to use by instances of the PhBox class (see next subsection). Each method in the `BoxAPI` class first invokes `createLibContext` method to ensure that a global compilation context exists; if not, `createLibContext` will create one. This compilation context will be automatically destroyed when the `asDsp` message is sent to a Phausto Box.

[5]The Semantic Phase is the initial step in the Faust compilation chain and consists of multiple stages. It takes Faust code as input and produces a list of signals in Normal Form as output. This list of signals in Normal Form is then passed to the Code Generation Phase, which compiles it into imperative code (C++, LLVM IR, WebAssembly, etc.

3.3.1. Integrating Unit Generators with Phausto

The concept of Unit Generators was first introduced by Max Mathews and Johan E. Miller for the Music III program in 1960 [9]. These components serve as the foundational building blocks of signal processing algorithms. Essentially, Unit Generators are subroutines that generate an output signal and may also process an input signal. Each Unit Generator is designed to perform a specific task, such as producing sound waves, applying filters, or controlling audio parameters. They function as modular elements within a synthesis framework.

As a functional language, Faust does not use any hierarchical organisation of Unit Generators, which is possible in object-oriented languages like Pharo, ChucK or SuperCollider through inheritance and abstraction. At the same time, Faust provides hundreds of DSP functions for synthesis and audio processing within the Faust Libraries, which yield the same output of our Unit Generators. The `UnitGenerator` class is a subclass of the `PhBox` class. Instances of the PhBox class are FFIOpaqueObjects, which correspond to pointers to Faust Boxes. UnitGenerators exists only in the Pharo environment and become PhBoxes when they receive the `asBox` message. To understand this mechanism, consider a simplified implementation of the `LFOTriPos` class, corresponding to Faust's `os.lf_trianglepos`. First, let's look at its initialise method:

```
LFOTriPos >> #initialize

faustCode := 'import("stdfaust.lib");
    process = os.lf_squarewavepos;'

freq := PhHSlider new
    label: self label , 'Freq'
    init: 440
    min: 1
    max: 2000
    step: 0.001.

amount := PhHSlider new
    label: self label , 'Amount'
    init: 1
    min: 0
    max: 28000
    step: 1.

offset := PhHSlider new
    label: self label , 'Offset'
    init: 0
    min: 0
    max: 800
    step: 0.01.
```

Next, the `asBox` method converts the UnitGenerator into an instance of its superclass, PhBox. :

```
LFOTriPos >> #asBox

    | intermediateBox finalBox |

    BoxAPI uniqueInstance createLibContext.
    intermediateBox := BoxAPI
        uniqueInstance
        boxFromString: self faustCode
        inputs: self inputs
        outputs: self outputs
        buffer: self errorBuffer.
    finalBox := freq asBox connectTo:
        intermediateBox.
    ^ finalBox * self amount asBox + self
        offset asBox
```

This final steps is fundamental as it enables the creation of a DSP object from a combination of boxes. Instances of the `PhBox` class serve as our entry point into the Faust compilation chain of the backend interpreter.

The Phausto API converts the majority of Faust libraries into Unit Generators. This design choice aligns with the object-oriented principle of inheritance while preventing the bloat of a single class with hundreds of methods.[6] The organisation of these units into subclasses and their interconnection capabilities are heavily inspired by the design principles of the ChucK programming language [10]. Indeed, in ChucK, there is no distinction between Unit Generators that operate at audio rate and those operating at control rate [11].

All Phausto Unit Generators are provided with initialised instance variables for the parameters specified in the corresponding Faust function. When possible (i.e. the argument is neither a constant value nor another function) they are initialised to a Faust `hslider` or `button` primitive, enabling both the parameter control and the on-the-fly creation of UI elements for the given parameter. If the Unit Generator's label has not been changed via the `label:` message, all UI elements use the class name as prefix. For example the `PulseOsc` has two controls: `'PulseOscFreq '` for its frequency and `'PulseOscDuty'` for its duty cycle.

4. SYNTAX IN A NUTSHELL

To create a DSP in Phausto, simply send the `asDsp` message to a Unit Generator or a combination of them. The `stereo` message converts it into a stereo DSP. Next, the DSP must be initialised and started to produce sound. A slider can be opened in the Pharo window to control a parameter of the DSP, and finally, the sound can be stopped.

```
sqr := SquareOsc new.
dsp := sqr stereo asDsp.
dsp init.
dsp start.
dsp openSliderFor: 'SquareOscFreq'.
dsp stop.
```

[6] Integrating over 200 methods directly into the DSP class would have been impractical. Instead, we organised these methods into Unit Generators, which enhances modularity and readability. Similarly, the Faust programming language efficiently manages large numbers of functions by structuring them within environments.

At need, we can combine UnitGenerators by assign them to instance variables, and we can change the value of a parameter with the keyword message `setValue: parameter:`:

```
pulse := PulseOsc new duty: LFOTriPos new.
dsp := pulse stereo asDsp.
dsp init.
dsp start.
dsp setValue: (Random new
    nextIntegerBetween: 50 and: 800)
    parameter: 'PulseOscFreq'
dsp stop.
```

The binary operator `=>` from the ChucK programming language was adopted to simplify the connection between UnitGenerators. This approach abstracts the connections while adhering to the principles of modular synthesis patching.For example a simple synth could look like this:

```
synth := SawOsc new => ADSREnv new =>
    ResonLp new => SatRev new;
```

Due to the double dispatch mechanism in the Phausto implementation of the `=>` message, its meaning depends on the argument provided. If the argument is an envelope, it performs signal multiplication; if it is a filter or a reverb, it connects the input(s) of a Unit Generator or a combination of them.

4.1. Dynamic Control of DSPs with Pharo Processes

Once our DSPs have been created, initialised and started, Pharo's syntax enables us to create algorithmic compositions. This is achieved by defining a process that repeatedly executes a `BlockClosure` for a number of times. Within this `BlockClosure`, time advancement is managed by sending the `wait` message to an instance of the `Delay` class. The process is then initiated by sending the `fork` message. This approach allows forked processes to run concurrently.

```
djembe := Djembe new.
dsp :=  djembe stereo asDsp.
dsp init.
dsp start.

pos := 0.
[128 timesRepeat:
    [ dsp setValue: (Random new
        nextIntegerBetween: 200 and: 900)
        parameter: 'DjembeFreq'.
     dsp setValue: (pos  \% 1) parameter: '
         DjembeStrikePos'.
     dsp trig: 'DjembeGate'.
     pos := pos + 0.1.
     (Delay forSeconds: 0.2) wait ] ] fork.
```

5. THE TOOLKIT AND TURBOPHAUSTO

In Phausto, we have implemented two sets of classes to simplify DSP programming and to provide musicians an ensemble of instruments effects for programming music on-the-fly without the need to use external audio generators.

5.1. The ToolKit

The ToolKit is a collection of synthesisers, effects and utilities included in the Phausto package. The name *ToolKit* pays tribute to Perry Cook's and Gary Scavone's *Synthesis ToolKit* [12]. Within the ToolKit, one can find utilities as an *incrementer*, an LFO that outputs a pseudo random signal, a *reader* and a *resetter* for reading sound files, and a basic *SamplePlayer* for playing back .wav files in Pharo applications.

5.2. TurboPhausto

TurboPhausto is a collection of synthesisers and effects designed to be the counterpart of SuperCollider's *SuperDirt* engine, intended for use with Coypu, the package that has been developed over the past three years for programming music on-the-fly with Pharo. Currently, 4 instruments and 2 effects are ready in *TurboPhausto*:

- *TpSampler*- a monophonic multisample player, that looks for all the sample in a specified folder (maximum 256 files);
- *Fm2Op* - a monophonic FM synth with 2 operators;
- *PsgPlus* - a monophonic chirpy synth inspired by Sega Master System Programmable Sound Generator (PSG);
- *Chordy* - a pseudo polyphonic virtual analog synth, in which different chords can be selected with the `mode:` message;
- *DelayMonoFB* - a smoothed mono delay with resonant feedback and dry/wet control;
- *GreyHoleDW* - a mono version of Faust's GreyHole reverb with dry/wet control.

All *TurboPhausto* synthesisers come with an AR/ADSR envelope and optionally with filters and effects on their output. All the effects are provided with dry/wet control. Here is a brief example of an extract of a live performance using Coypu and *TurboPhausto*:

```
"create, initialize the DSP"
dsp := (TpSampler new + PsgPlus new + Fm2Op
    new) stereo asDsp.
dsp init.
dsp start.

"initialize the Performance and assign the
   DSP"
p := PerformanceRecorder uniqueInstance .
p performer: PerformerPhaust new.
p freq: 143 bpm.
p activeDSP: dsp.

"assign TurboPhausto instruments to
   Performance sequencers"
16 downbeats index: '1' to: #TpSample.
16 quavers notes: '38 41 45 50' to: #
   PsgPlus .
16 rumba to: #Fm2Op.
p playFor: 32 bars.
```

6. CONCLUSIONS AND FUTURE WORK

After a year of development, Phausto provides a comprehensive solution for Pharo programmers to integrate sound synthesis into their applications. It includes sample players, basic oscillators, envelopes, various physical models, resonant filters, reverbs, and delays. The extensive array of Unit Generators features a streamlined API for parameter manipulation, which has been well-received by Pharo programmers. This was highlighted at the ESUG 2024 conference[7] in Lille, where Phausto earned 3rd place in the *Innovation Technology Awards*. Additionally, TurboPhausto's synthesizers and effects were showcased in a 30-minute live performance titled *Riding the MoofLod*.

In the coming months, the primary goal will be to port all functions from the Faust libraries and the Box-API. Subsequently, the focus will shift to MIDI and polyphonic support, which may necessitate the implementation of a MIDI handler using PortMIDI, an open-source library for which FFI bindings are already available in the Pharo-Sound package. Additionally, classes and methods will be provided to export DSPs into different architectures directly from Phausto and to display their signal flow along with a default GUI. Finally, an exhaustive and robust ensemble of instruments and effects for TurboPhausto will be designed and implemented.

7. ACKNOWLEDGMENTS

Thanks to Stéphane Letz for the never ending advice on Faust and its ecosystem, and to Yann Orlarey for being enthusiast about Phausto. Thanks Stéphane Ducasse, Guillermo Polito, Esteban Lorenzano, Nahuel Palumbo, for their invaluable support, push and assistance throughout the development of Phausto. Thanks to the Pharo Association for the support.

8. REFERENCES

[1] Alan C. Kay, "The early history of smalltalk," *ACM Sigplan Notices*, vol. 28, no. 3, pp. 69–95, 1993.

[2] Giorgio C. Buttazzo, *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications*, Springer, Berlin, Germany, 2011.

[3] Stéphane Letz, Yann Orlarey, and Dominique Fober, "An overview of the faust developer ecosystem," in *Proceedings of International Faust Conference (IFC18)*, Mainz, Germany, July 2018.

[4] Stéphane Ducasse, Gordana Rakic, Sebastijan Kaplar, Quentin Ducasse Originally written by A. Black, S. Ducasse, O. Nierstrasz, D. Pollet with D. Cassou, and M. Denker, *Pharo 9 by Example*, Book on Demand – Keepers of the lighthouse, 2022.

[5] Stéphane Ducasse, *Pharo with Style*, Creative Commons, 2022, `http://books.pharo.org/booklet-WithStyle/pdf/WithStyle.pdf`.

[6] Adele Goldberg and David Robson, *Smalltalk-80: the language and its implementation*, Addison-Wesley Longman Publishing Co., Inc., 1983.

[7] Trudcill P. Dell Hymes, "Pidginization and creolization of languages," *Journal of Linguistics*, vol. 9, no. 1, pp. 193–195, 1971.

[7] ESUG stands for European Smalltalk User Group.

[8] Guillermo Polito, Stéphane Ducasse, Pablo Tesone, and Ted Brunzie, *Unified FFI - Calling Foreign Functions from Pharo*, Creative Commons, 2020, `http://books.pharo.org/booklet-uffi/pdf/2020-02-12-uFFI-V1.0.1`.

[9] Julius O. Smith III, "Viewpoints on the history of digital synthesis," in *Proceedings of the International Computer Music Conference (ICMC91)*, Montréal, Canada, October 1991.

[10] Ge Wang and Perry R. Cook, "Chuck: A concurrent, on-the-fly, audio programming language," in *Proceedings of the International Computer Music Conference (ICMC03)*, Singapore, September 2003.

[11] Ge Wang, *The ChucK Audio Programming Language. "A Strongly-timed and On-the-fly Environ/mentality"*, Ph.D. thesis, Princeton University, 2008.

[12] Perry R. Cook and Gary P. Scavone, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC99)*, Beijing, China, October 1999.