FAUST IN IPLUG 2: CREATIVE CODING AUDIO PLUG-INS

Oliver Larkin
Creative Coding Lab, University of Huddersfield
Huddersfield, UK
oliver.larkin@hud.ac.uk

ABSTRACT

FAUST is a powerful domain specific programming language for audio Digital Signal Processing (DSP) with many options for quickly compiling to different "architectures" including audio plug-ins. This functionality is highly suited to rapid prototyping but can also produce robust and performant binaries. When FAUST is used in larger software, it is more often dedicated to specific elements of a DSP graph and combined with extra C++ code where additional DSP, user interface (UI) and other aspects of the application are managed. This paper discusses the author's efforts towards seamless integration of FAUST directly into iPlug - a popular C++ audio plug-in framework, that complements FAUST with a graphical interface toolkit and platform support. The integration allows developers to easily combine multiple FAUST DSP routines with additional C++ code. It also permits over-sampling the signal processing. A workflow is presented which uses the libfaust embeddable compiler for dynamic compilation during development and then the traditional architecture-based FAUST compiler to statically compile code for distribution. This is demonstrated by showing how a FAUST DSP block would be added to an existing iPlug plug-in. Finally, some examples of how this has been used are discussed.

1. INTRODUCTION

The internet and source code management platforms such as github[1] have heralded a huge change in software development practices. Open source projects are more visible, and collaboration is easier than ever before. Alongside this, improvements in compiler technology and powerful free tools such as integrated development environments (IDEs) make programming much more accessible than it has been in the past. New approaches to software development involving interactivity and real-time feedback[2] are available in mainstream IDEs[3].

There are two popular packages for "Creative Coding" in C++, namely OpenFrameworks[6] and Cinder[7]. These packages include libraries and optional extensions for a variety of audio-visual tasks. They come with tools which simplify setting up a project and they reduce the amount of time spent on non-creative tasks such as configuring the IDE. As well as reducing complexity, these packages also offer good real-time performance since they use a low-level, compiled programming language. Similarly, in the world of audio software development iPlug[11] and JUCE[12] allow developers to realise their ideas for audio plug-ins and apps across multiple plug-in formats and platforms without having to concern themselves with the low-level details of platform or plug-in APIs. Developers can simultaneously write multi-format plug-ins and focus on the DSP, user interface (UI) and user experience (UX) rather than the specifics of individual API implementations. These frameworks also have functionality for creating the kinds of UI that are often used in audio software, including widgets for buttons, dials, sliders and meters using either bitmaps or vector-based drawing. They can also simplify implementation details; for instance, in iPlug, much like in FAUST, creating a parameter can be achieved with a single line of code:

```
GetParam(0 /*id*/)->InitDouble("Gain"
/*name*/, 0. /*default*/, -70. /*min*/, 12.
/*max*/, 0.1 /*step*/, "dB"/*label*/);
```

FAUST provides similar advantages in terms of the concise expression of complex DSP algorithms, but although FAUST has the concept of "UI" this is merely an abstraction that is interpreted by an "architecture" file. It provides an interface (which may not be graphical) to the DSP [1]. If the developer wishes to provide a customised *graphical* UI (GUI) for their FAUST creation, they need to use an extra library and implement their own architecture file. In addition to GUI, there are also other situations where FAUST can be used most effectively alongside additional C++ code. For instance, in DSP requiring switchable signal paths or multi-rate capabilities a developer may use multiple FAUST **.dsp** files to produce multiple isolated processors. These can be combined with some "glue" code which handles the multi-rate processing and/or switching of code blocks. Lastly, FAUST has its own generic polyphony capabilities via `poly_dsp` [1], but these may not be flexible enough to meet the requirements of certain software synthesisers, since there are many nuances in how synthesisers handle polyphony and modulation. For example, typically virtual analogue synthesisers would have a single low frequency oscillator, which applies the same modulation across all voices, where FAUST's approach using a single **.dsp** file would limit the user to per-voice modulation.

Considering the different tools and software development trends discussed in this section, the work presented in this paper aims to address the following problems:

- How can the strengths of the FAUST DSP language be combined easily with those of C++ and iPlug to promote the creative-coding of audio plug-ins?
- How can FAUST's embeddable compiler improve the workflow and shorten the feedback loop when creating DSP for audio plug-ins?

[1] https://github.com
[2] http://worrydream.com/#!/InventingOnPrinciple
[3] https://developer.apple.com/swift/blog/?id=35
[6] http://openframeworks.cc/
[7] https://libcinder.org/
[11] https://github.com/olilarkin/wdl-ol
[12] https:// juce.com

1.1. Existing workflows with FAUST

The FAUST workflow has traditionally involved processing **.dsp** files (text files describing a DSP block in the FAUST language) with a command line compiler. In recent years many more options have become available and several tools have been built using libfaust - an embeddable compiler based on LLVM, including Faustlive [2]. FAUST extensions have been created for different software packages including Max, Puredata, Chuck, Csound and Processing, some of which feature JIT (just-in-time) compilation. The author's juce_faustllvm module is an extension for the JUCE C++ framework that supports embedding libfaust in the JUCE audio processor graph [13][14]. There have also been many developments integrating FAUST with web technologies [3], facilitating a quick workflow, without requiring the user to install any software. The FAUST online compiler allows users to compile their FAUST **.dsp** file to platforms that they may not be able to compile to on their local machine. For larger projects the FAUST distribution comes with feature-rich architectures for Qt and JUCE [4], but these are designed for generic GUI's. In general, the many options FAUST provides excel at prototype outputs with a generic GUI, or small functional, interactive tools for musical composition/performance. For larger projects the most practical workflows involve writing your own architectures and/or combining the outputs from the command line compiler with existing C++ code. This process can be laborious, involving multiple pieces of software and requiring command line fluency. Attempts have been made to introduce extra constructs to simplify this process [1], but there is no one-size fits all solution in audio DSP.

1.2. iPlug 2

The iPlug C++ plug-in framework was originally developed by John Schwartz and open sourced in 2008 as part of Cockos' WDL[15], a liberally-licensed library of reusable C++ code which is used to build the DAW Reaper. Since that time, iPlug has been improved by numerous contributors, with the author's version (publicly released in 2011) being one of the most popular, well maintained and feature-complete forks. Many commercial, free and open source plug-ins have been built with this version of iPlug and it has been used by researchers, hobbyists and companies. A significant update is in progress at the time of writing (to be released in 2018) with many new features. The main motivation for this update is support for Web Audio Modules (WAMs)[16], a new format for VST-style instrument and effects plug-ins that integrate with the Web Audio API [5][6]. This new platform has its own challenges. One of those is the size of the payload since web pages must load quickly over remote connections. The lightweight nature of the iPlug framework makes it an ideal tool for producing WAMs, and there are many desktop plug-ins written using this framework already that can be ported across to the web platform.

The author has taken this opportunity to rework the framework and improve the feature set in numerous ways, whilst maintaining the simplicity of the original design. The original version of iPlug included support for developing VST2 and AudioUnit plug-ins. The author's fork extended the existing functionality and added VST3, RTAS and AAX (Pro Tools) formats, as well as stand-alone applications on Windows and macOS platforms.

[13] https://www.youtube.com/watch?v=INlqClEOhak
[14] https://github.com/iplug2
[15] http://www.cockos.com/wdl/
[16] http://www.webaudiomodules.org

Alongside FAUST support, the new features of the forthcoming iPlug 2 include:

- Modernised and simplified code base for C++11
- Extensive documentation
- Rewritten graphics interface supporting several different 2D drawing back-ends including the latest hardware accelerated technologies such as Metal on macOS/iOS
- Separation of graphics and plug-in code.
- Vector graphics support, including SVG and path-based drawing routines
- Graphic scaling and seamless support for high resolution displays
- Web Audio Module API support. A plug-in and its user interface can be compiled from the same C++ codebase and run in the web browser as well as traditional desktop hosts.
- Linux and Raspberry Pi support (standalone only)
- Reworked approach to concurrency and thread safety
- A library of "extras" featuring utility classes for common music DSP tasks such as over-sampling, polyphony, parameter smoothing, oscillators.

The following sections will introduce the ways in which the new version of iPlug integrates FAUST.

2. FAUST IN IPLUG 2

FAUST support in iPlug is implemented so that FAUST can be used for certain tasks and combined with C++ for tasks where that language is more appropriate. In this way, the developer can control the signal routing and the mixture of C++ with FAUST - they do not have to manually keep track of all the different files and extra glue code. The implementation is designed so that dynamic compilation may be used at the development/debug stage, and statically compiled code may be used for distribution/release builds. Using the dynamic compiler maintains the rapid-workflow found in FAUST's JIT-based frontends, but it means that development may be done from within the same application (the C++ IDE) that will be used to create the final distributable version.

FAUST support in iPlug 2 comes in the form of a C++ interface class and two different implementation classes that inherit it. Firstly, there is a traditional FAUST architecture file. Even without using libfaust, a developer can utilise this architecture file on the command line, to produce a single C++ file – a module - ready to integrate with an iPlug plug-in. The iPlug FAUST C++ interface includes functionality for binding iPlug parameters to FAUST parameters and over-sampling the DSP. The other class that inherits from the interface uses the libfaust dynamic compiler, and its interface file contains compile-time pre-processor macros to facilitate switching between using the dynamically compiled version of a FAUST **.dsp** file and the static version. It is envisaged that when building debug builds of a project, the JIT compiler will be used. When building release versions, the static compiled code will be used. Every time the **.dsp** file is dynamically compiled, it is also statically compiled. Just by switching build configurations in the C++ IDE or makefile, a completely different code path is chosen, although the C++ class interface is identical. When using the JIT compiler, a file watcher keeps track of any modifications to the FAUST **.dsp** file on disk, and recompiles both the JIT and the static code if the source code changes. In this way development can take place whilst a plug-in is loaded in a Digital Audio Workstation (DAW), providing a

significantly faster workflow to the traditional C++ approach where the binary must be recompiled every time the code is modified.

2.1. Adding FAUST to an iPlug 2 project

Assuming the developer has built and installed libfaust in the standard locations, it is trivial to add FAUST elements to an iPlug C++ project. iPlug provides a well-defined folder structure and global configuration files to set build settings that apply to multiple plug-ins. Several of these default build settings relate to FAUST and specify the exact include paths for header files and libraries. In an iPlug 2 project it is simply necessary to add two build settings and all the dependencies will be found. The header file `IPlugFaustGen.h` provides the necessary pre-processor macros to deal with switching between JIT and static compiled DSP. For each unique FAUST processor, the macro `FAUST_BLOCK` needs to be declared with an identifier/name and a C++ variable name passed as an argument, along with an absolute path to a **.dsp** file on disk, a voice count (in the case of a polyphonic DSP) and over-sampling factor (see line 16 in Figure 1). This will declare a C++ variable linked to the block, which would typically be a member variable of the iPlug plug-in class. Calling the `Init()` method will trigger an initial JIT compilation. The method `FaustGen::CompileCPP()`can be called to invoke the command line FAUST compiler and produce a single C++ file containing implementations for every unique `FAUST_BLOCK` declared. This statically compiled C++ file is automatically included by the pre-processor when the macro `FAUST_COMPILED` is defined and all dynamic compilation related code is excluded. Since the interface is the same for both static and dynamic modes, no modifications need to be made to the code. When `FAUST_COMPILED` is not defined, calling `FaustGen::EnableTimer(true)` will mean that dynamic compilation and static compilation will be triggered automatically if the specified **.dsp** file for a block is modified.

The following code listings with comments demonstrate how a Faust **.dsp** file "Fuzz.dsp" can be integrated into an iPlug project. The project does not have a user interface to keep the code as simple as possible.

```
declare name "Fuzz";
import("maths.lib");

g = vslider("Gain", 0, 0., 1, 0.1)

process = (*(g) : tanh), (*(g) : tanh);
```

```
1  #pragma once
2
3  #include "IPlug_include_in_plug_hdr.h"
4  #include "IPlugFaustGen.h"
5
6  class IPlugEffect : public IPlug
7  {
8  public:
9    IPlugEffect(IPlugInstanceInfo instanceInfo);
10   void OnReset() override;
11   void OnParamChange(int paramIdx) override;
12   void ProcessBlock(sample** inputs, sample** outputs, int nFrames) override;
13 private:
14   // This member variable will either hold dynamic or static compiled
15   // code depending on whether FAUST_COMPILED is defined
16   FAUST_BLOCK(Fuzz, mFuzz, "~/IPlug2/Examples/IPlugEffect/Fuzz.dsp", 1, 1);
17 };
```

Figure 1: IPlugEffect.h C++ interface with FAUST DSP

```
1  #include "IPlugEffect.h"
2  #include "IPlug_include_in_plug_src.h"
3  #include "config.h"
4
5  IPlugEffect::IPlugEffect(IPlugInstanceInfo instanceInfo)
6  : IPLUG_CTOR(1 /*NParams*/, 1 /*NPresets*/, instanceInfo)
7  {
8    mFuzz.Init(); // initialise (JITs)
9    mFuzz.CreateIPlugParameters(*this); // helper method for plug-in parameters
10   mFuzz.SetParameterValue("Gain", 1.); // parameters can be set with strings or indexes
11
12   // calling CompileCPP() will compile all used .dsp files to a single C++
13   // source code file in the same folder as the .dsp which will be automatically
14   // included in the release build. This will also be called if the .dsp changes
15   FaustGen::CompileCPP();
16   FaustGen::EnableTimer(true); // enables automatic recompile
17 }
18
19 void IPlugEffect::ProcessBlock(sample** inputs, sample** outputs, int nFrames)
20 {
21   // run FAUST processor code
22   mFuzz.ProcessBlock(inputs, outputs, nFrames);
23 }
24
25 void IPlugEffect::OnReset()
26 {
27   // update FAUST sample rate
28   mFuzz.SetSampleRate(GetSampleRate());
29 }
30
31 void IPlugEffect::OnParamChange(int paramIdx)
32 {
33   // update FAUST parameters
34   mFuzz.SetParameterValue(paramIdx, GetParam(paramIdx)->Value());
35 }
```

Figure 2: IPlugEffect.cpp C++ implementation

2.2. Over-sampling a FAUST DSP processor

Over-sampling is a common technique in signal processing that is used to perform certain calculations at a higher sample rate than others. In audio DSP this is often needed when working with nonlinear processes such as wave-shaping which can introduce spectral components that alias. In its current version, FAUST does not include functionality for multi-rate processing, so the developer is required to implement this externally in C++. Over-sampling is nontrivial to implement, requiring precise filtering to remove spectral components that might cause aliasing at each stage of the up/down sampling process. iPlug 2 comes with helper classes for up to 16x over-sampling based on the HIIR library[17] using IIR half band poly-phase filters. These classes are integrated into iPlug's FAUST interface code, and when enabled no additional boilerplate code is required to over-sample a FAUST processor.

3. EXAMPLES

This section introduces some examples of how the iPlug 2 FAUST integration has been used.

3.1. H.A.C.K Decoder Tester

The Ambisonic Decoder Toolkit (ADT) [7] is a set of Matlab/GNU Octave scripts that generate ambisonic decoders in a variety of formats including FAUST **.dsp** files. The FAUST output is the most feature complete of all the options provided. Alongside the code that is generated, a collection of visualisations help the user understand the decoder's performance. The author required a tool for rapidly auditioning different decoders generated using the ADT. By combining iPlug and FAUST a VST plug-in was developed that displays the visualisations, whilst JIT compiling and running the associated **.dsp** file. This plug-in is part of the forthcoming Huddersfield Ambisonic Creation Kit (H.A.C.K), which is a collection of utility plug-ins for working with ambisonics in the Huddersfield multi-channel studios.

[17] http://ldesoras.free.fr/prod.html

3.2. Bell Field

As a proof of concept, the JIT compilation was used to produce an experimental musical instrument plug-in that generates 3D spatial audio using HOA. Using the physical models of bells presented in [8], which are available as FAUST **.dsp** files, the author created a polyphonic synthesiser that operates in HOA. Tuned bells are spatially positioned around the soundfield and may be struck by playing MIDI notes. The bell model may be changed on the fly by JIT compiling a different **.dsp** file from the collection and all voices are updated. The tools developed here allowed a very quick turnaround time from idea to realisation. Working in HOA in real time can be demanding even on modern CPUs; due to the high channel counts required and coupled with the polyphonic nature of the synthesiser, it was challenging to produce efficient code. For this reason, it was beneficial to be able to mix between C++ and FAUST code, in order to have fine-grained control of the rate at which expensive operations are performed.

3.3. Faustgen plug-in

As well as using the libfaust embeddable compiler as a tool in the development chain, iPlug's FAUST support also allows it to be used in a release build plug-in, if desired. A JIT compiling audio plug-in has been developed mirroring the functionality of GRAME's faustgen~ Max object. This plug-in includes an integrated code editor, based on the WDL ncurses[18] emulator, and is designed for prototyping and for end users who would like access to FAUST's libraries directly inside their DAW.

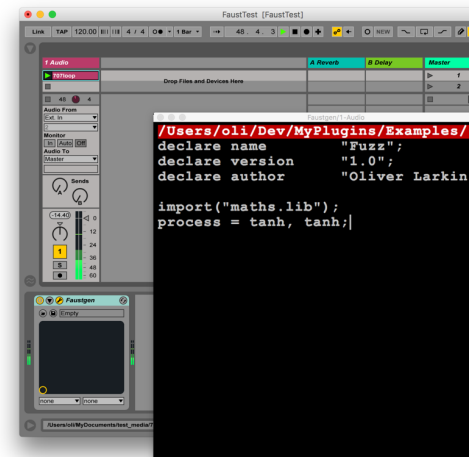

Figure 3: *Faustgen VST in Ableton Live*

4. FUTURE WORK

Development of iPlug 2 is ongoing. The primary focus of the efforts regarding FAUST integration discussed in this paper involve the DSP, with the assumption that most professional plug-ins (or at least most widely distributed plug-ins) will require a custom user interface that extends beyond the remit of FAUST's UI layout metadata. However, it would be useful to be able to create widgets automatically based on FAUST UI specifications (checkbox, hslider, vgroup etc.), much in the same way is it possible using FAUST's Qt and JUCE [4] architecture files. Support for control metadata should be added to allow MIDI control of parameters as well as sample accurate timing via `timed_dsp` as described in [1]. The author plans to improve the reliability of the JIT compilation in iPlug's FAUST support, by investigating compiling out of process and synchronising updated DSP safely in a click free manor in real-time. Work also needs to be done on the safe management of parameter count changes.

5. CONCLUSIONS

This paper presented the integration of two open-source projects to provide a complete workflow for the development of professional quality audio software targeting multiple platforms. The power of FAUST's embeddable compiler, concise functional syntax and extensive DSP libraries combined with iPlug's simplicity, graphical user interface and platform support makes a complementary package. By abstracting the complex elements of plug-in APIs, development becomes arguably more enjoyable, and more time is spent on the elements that can be called "creative", such as DSP, UI and UX, hence the paper's title "creative coding audio plug-ins". The author hopes that this integration will help promote the use of the FAUST language in both professional and hobbyist audio software development.

6. ACKNOWLEDGMENTS

The author would like to thank Alex Harker and Hyunkook Lee, Jari Kleimola, Justin Frankel, John Schwartz, the FAUST team and The Creative Coding Lab at the University of Huddersfield.

7. REFERENCES

[1] Stéphane Letz, Yann Orlarey, Dominique Fober, Romain Michon. "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files" *Linux Audio Conference (LAC-17)*. Saint-Etienne, France, 2017

[2] Sarah. Denoux, Stéphane. Letz, Yann. Orlarey, and Dominique. Fober, "Faustlive: Just-in-time faust compiler... and much more," in *Proceedings of the Linux Audio Conference (LAC-12)*, Karlsruhe, Germany, April 2014.

[3] Stéphane. Letz, Sarah. Denoux, Yann. Orlarey and Dominique. Fober, "Faust audio DSP language in the Web", *In Proceedings of the Linux Audio Conference (LAC-15),* Mainz, Germany, 2015

[4] Adrien Albouy and Stéphane Letz. Faust audio DSP language for JUCE *Linux Audio Conference (LAC-17).* Saint-Etienne, France, 2017

[5] Jari Kleimola and Oliver Larkin, "Web Audio Modules", in *Proceedings of the 12th Sound and Music Computing Conference (SMC-2015)* Maynooth, Ireland, 2015

[6] Michel Buffa, Jérôme Lebrun, Jari Kleimola, Oliver Larkin, Stéphane Letz. "Towards an open Web Audio plug-in standard.", in *Proceedings of the International World Wide Web Conference (WWW '18),* Lyon, France, 2018

[7] Heller, Aaron J., Eric Benjamin, and Richard Lee. "A toolkit for the design of ambisonic decoders." *In Proceedings of the Linux Audio Conference (LAC-12),* 2012

[8] Romain Michon, Sara R Martín. "Faust Foundry: A Software Kit to Make Bell Physical Models for Musical Applications". *Resonance and Remembrance: An Interdisciplinary Bell Studies Symposium*, Michigan, 2017

[18] https://en.wikipedia.org/wiki/Curses_(programming library)