Proceedings of the 2nd International Faust Conference (IFC-20), Maison des Sciences de l'Homme Paris Nord, Saint-Denis, France, May 14–15, 2020

CROSS-PLATFORM UI FOR FAUST-BASED AUDIO APPLICATIONS USING FLUTTER

Andrei Ozornin

Individual
Amsterdam, Netherlands
andrey@ozorn.in

ABSTRACT

In this project, I explore possibilities of integrating Faust with Flutter, a modern UI framework capable of delivering applications for web, mobile and desktop platforms.

Also, I propose a development methodology allowing the DSP layer written entirely in Faust to be completely independent from the user-facing interface.

As a proof-of-concept, I present a working production-ready iOS application made using Faust and Flutter and a minimal reproducible open-source example. Also I describe a workflow of creating applications using this technology stack, including mobile and desktop as well as web applications.

1. INTRODUCTION

1.1. Problem statement

Faust is a great tool for DSP, but its possibilities of describing user interfaces are very limited. It's not a weakness of Faust, it is so by design, as Faust is a domain-specific language, dedicated to signal processing, whereas user interfaces are more commonly represented as state machines with a user-facing view layer.

When it comes to creation of totally custom UI components, existing UI primitives in Faust may not fit. Solution like *faust2smartkeyboard* [1] is good for prototyping and trying out ideas related to sound, but when it comes to customisation of the appearance and trying out controller-related ideas changing user experience, it lacks flexibility, because these actions would require modifying the tool itself, as would using a different scale than one of the predefined.

There are architectures generating UI using third-party libraries [2]: among others, GTK and Qt, but all of them generate UI based on metadata from Faust code. To perform changes in the UI generated by one of these tools, you would have to change code in DSP files.

These approaches don't fit the principle of single responsibility which is widely spread in object-oriented development.

1.2. Proposed technical solution

There is target *faust2api* [3], which outputs UI-independent DSP layer with platform-independent API and therefore can be used in combination with practically any tool, language and framework that is interoperable with C++. Using it, DSP developers can safely change implementation of sound engine and it won't affect UI, and UI developers can abstract themselves away from sound engine implementation details.

Moreover, this way UI can be implemented using any technology or even programming language, and changing the technology won't affect the DSP layer in any way. Also, several different user interfaces can be used with the same signal processor within the same project, as well as the DSP implementation can be easily mocked for testing purposes.

1.3. Proposed methodology

From a methodological point of view, to make it happen, the DSP code in any given project shouldn't have any references to the appearance of the controls. It only provides a set of named or numbered (depending on agreement within a team) parameters with zero information on their visual representation. Even differentiation between hslider and vslider becomes redundant, just as do metadata values like `[style:knob]` or `[unit:db]`.

Grouping parameters into semantically meaningful categories is a right thing to do but the group hierarchy should stay independent from UI implementation. Differentiation between `tgroup`, `hgroup` and `vgroup` becomes redundant as well.

Metadata in general stays allowed, as long as it describes characteristics of the sound parameter. Any metadata describing the role or appearance of the UI control changing or displaying the parameter value, is restricted.

I propose to call this style of Faust coding "UI-agnostic".

2. IMPLEMENTATION

There are plenty of technologies empowering development of cross-platform interfaces. Some of them have already been more or less integrated with Faust: most notably, GTK+, Qt and Unity.

I've chosen Flutter among them because it can produce mobile (stable), web (beta, as of August 2020), and desktop

(alpha) applications [4] and comes with rich infrastructure, including package manager, widget library and various IDE tools. It's been maintained by Google and, unlike Unity and Qt, is free to use even in proprietary commercial projects regardless of amounts of profit (see table 1). Above all, Flutter is extremely well documented and is being frequently updated.

Table 1: *Comparison of UI frameworks.*

Framework	Win	Mac	Linux	Android	iOS	Web	Open-Source	Free for commercial use
GTK	Yes	Yes	Yes	No	No	No	Yes	Yes
Qt	Yes	Yes	Yes	Yes	Yes	No	Yes	Limited [5]
Unity	Yes	Yes	Yes	Yes	Yes	Yes	No	Limited [6]
Flutter	Yes*	Yes*	Yes*	Yes	Yes	Yes	Yes	Yes

* available in alpha channel

2.1. User interface

Basic Flutter app can be created in just one CLI command, given that all necessary dependencies are installed:

```
flutter create app-name
```

The created app can then be started in debug mode either on real device or in simulation with another command

```
flutter run
```

The interface itself can be built of blocks implementing Material design or using low-level API. All changes made in code immediately become visible in the running application, thanks to "Hot reloading" built in Flutter.

Flutter applications are written in Dart: it's a high-level object-oriented language with a strong type system. Dart since its origin has been designed with UI development as its main purpose, so in certain sense Dart can be called a domain-specific language dedicated to user interfaces.

Further description as well as multiple tutorials can be found on the official site of Flutter: *flutter.dev*.

For purposes of proof-of-concept, we add a simple gate button in `main.dart` file:

```
Widget build(BuildContext context) {
  return Center(
    child: const RaisedButton(
      onPressed: null,
      child: Text(
        'Start beeping',
        style: TextStyle(fontSize: 20)
      ),
    )
  )
}
```

For now this button does nothing, but that will change at the end of chapter 2.3.

2.2. DSP

For demonstration purposes we create an oscillator:

```
import("stdfaust.lib");
gate = button("gate") : si.smoo;
process = os.sawtooth(440) * gate;
```

From this Faust code then we generate platform-specific C++ files for all necessary platforms. For instance, to generate DSP layer for iOS, we run this command:

```
faust2api -ios -nozip -target
../ios/Runner/DSP main.dsp
```

The idea is to put the generated C++ files to the folder where the code can be included and then used in application. XCode projects can't include files from outside the project directory, so we create a folder called DSP inside the iOS project and put the generated C++ files together with the generated README file there.

2.3. DSP / UI interoperability

To call API methods of the DSP layer, the UI layer has to communicate with platform-level code written in C++. Dart has this capability, but requires some tweaking. Out of the box it allows communication with platform-specific code in languages native for the platform (Objective C and Swift on iOS or Java and Kotlin on Android) [7].

Such communication happens via platform channels: platform-independent Dart code sends a message to a channel, the underlying layer written in native language receives it and sends it further to platform-specific C++, which in its turn, performs required calls to platform API (for example, to CoreAudio).

Sample Dart method calling `setParamValue` method on platform channel:

```
import 'package:flutter/services.dart';

class DspApi {
  static const _platform = const
MethodChannel('channel_name');
  static Future<void> setParamValue(int id,
double value) {
       return
_platform.invokeMethod('setParamValue',
<String, dynamic>{
       'id': id,
       'value': value,
       });
  }
}
```

Sample Objective C method receiving the call above and calling corresponding method of the platform-specific DSP C++ implementation (must be placed inside `AppDelegate` class implementation of the XCode project):

```
[synthControlChannel
setMethodCallHandler:^(FlutterMethodCall*
call, FlutterResult result) {
  if ([@"setParamValue"
isEqualToString:call.method]) {
      NSNumber* idArg =
call.arguments[@"id"];
      NSNumber* valueArg =
call.arguments[@"value"];
      [weakSelf setParamById:idArg.intValue
Value:valueArg.floatValue];
      result(nil);
  }
}
```

to call this method, we also add `onPressed` action handler to the button we've created in p. 2.1:

```
onPressed: () => DspApi.setParamValue(0, 1)
```

2.4. Implementation summary

A generic Faust-Flutter project has the following structure (see *Figure 1*):

UI-agnostic DSP code written entirely in Faust is used to generate platform-specific C++ files which are then included in the project build.

Cross-platform UI code written entirely in Dart calls native methods via `MethodChannel` technology coming with Flutter.

Native code receives calls from `MethodChannel` and performs corresponding calls to the C++ code generated from Faust. Then it sends the response back to the UI layer.

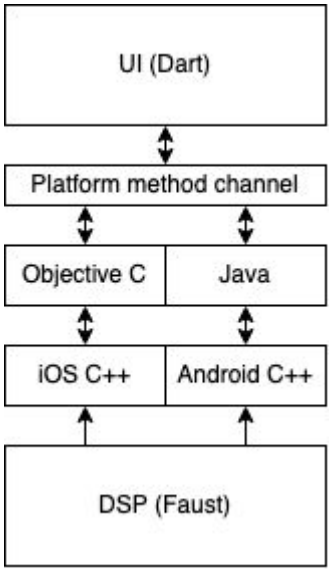

Figure 1: *Structure of a generic Faust-Flutter project*

Developed this way, a cross-platform user interface can be implemented using the technology that is best applicable for UI production, and stay independent from both platform code and high-level DSP code.

On the other hand, DSP chains can be developed with the tool that is created specifically for signal processing staying completely independent from UI implementation.

2.5. Working examples

Minimal working Flutter-Faust open-source project practically identical to the code in this article can be found in *faust_flutter* repository on GitHub [8].

More sophisticated sound application called *Synt* is also created using both Faust DSP and Flutter UI, but is not open-source so far. However, it can be tried out and used in AppStore [9] and act as a demonstration of possibilities of this technology stack.

2.6. Other platforms

Android, MacOS, Windows and Linux [4] applications can be created by analogy: *faust2api* generates platform-specific DSP modules from the same Faust codebase and Flutter generates a user interface for that platform from the same Dart codebase. Dart code communicates with native code via `MethodChannel` the same way it does on iOS.

Web application is going to be a bit trickier, but still the same approach applies. Instead of *faust2api*, another Faust target has to be used: *faust2webaudio* or *faust2webaudiowasm*. And the Dart code should communicate with it the same way it did on other platforms.

3. CONCLUSIONS

Faust is a powerful tool for DSP, but often lacks flexibility when it comes to UI. There already were several attempts to fix it or to have a workaround: some of them even have become part of the language, while others evolved as architecture files. One can get very interesting results using them, but these implementations tend to look hacky because description of UI is not the main purpose Faust was designed for.

Thanks to the existence of *faust2api* [3], the user interface implementation can be completely delegated to other technologies, leaving DSP code *UI-agnostic*.

User interface can be implemented using any technology best suitable for the job. In this article, I've shown an example with *Flutter*, which is Google's UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.

This approach allows splitting areas of responsibility between developers, effectively separating DSP implementation from UI, giving more freedom in implementation of both.

Also, it enables building sophisticated cross-platform user interfaces, expanding boundaries of what was possible using existing Faust architectures.

4. REFERENCES

[1] Romain Michon, Julius O. Smith, Chris Chafe, Ge Wang, and Matthew Wright, *faust2smartkeyb: a tool to make*

mobile instruments focusing on skills transfer in the Faust programming language, in *Proceedings of the 1st International Faust Conference (IFC-18), Mainz, Germany, July 17–18, 2018*.

[2] D. Fober, Y. Orlarey and S. Letz, *Faust Architectures Design and OSC Support*, chapter *4. GUI Architecture files* in *Proceedings of the 11th Int. Conference on Digital Audio Effects (DAFx-11)*, Paris, France.

[3] R. Michon, J. Smith, S. Letz, C. Chafe and Y. Orlarey, "faust2api: a Comprehensive API Generator for Android and iOS" in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, 2017.

[4] *Desktop support for Flutter*, https://flutter.dev/desktop, August 30, 2020.

[5] *Legal | Licencing - Qt*, https://www.qt.io/licensing/, August 31, 2020.

[6] *Compare Unity Plans: Pro vs Plus vs Free,* https://store.unity.com/compare-plans, August 31, 2020.

[7] *Writing custom platform-specific code*, https://flutter.dev/docs/development/platform-integration/platform-channels, August 31, 2020.

[8] *oshibka404/faust_flutter: Minimal audio application with Flutter UI and Faust DSP*, https://github.com/oshibka404/faust_flutter, August 31, 2020

[9] *Synt | An instrument you can play even if you think you can't*, https://synt.ozorn.in/, August 31, 2020