JSPatcher, a Visual Programming Environment for Building High-Performance Web Audio Applications

Shihong Ren, Laurent Pottier, Michel Buffa, Yang Yu

JSPatcher, a Visual Programming Environment for Building High-Performance Web Audio Applications

SHIHONG REN,[1 2] LAURENT POTTIER,[2] MICHEL BUFFA,[3] AND YANG YU[1]
shihong.ren@univ-st-etienne.fr, laurent.pottier@univ-st-etienne.fr, michel.buffa@univ-cotedazur.fr, yuyang@shcmusic.edu.cn

[1]Shanghai Conservatory of Music, SKLMA, China
[2]Université Jean Monnet, ECLLA lab., Saint-Etienne, France
[3]Université Côte d'Azur, I3S, INRIA, France

Many visual programming languages (VPLs) which include Max or PureData provide a graphic canvas for connecting between functions or data. This canvas, also called a patcher, is basically a graph meant to be interpreted as a dataflow computation by the system. Some VPLs are used for multimedia performance or content generation since the UI system is generally a significant element of the language. This paper presents a web-based VPL, JSPatcher, which allows you to build audio graphs using the Web Audio API. Users can use a web browser to graphically design and run DSP algorithms using domain specific languages (DSL) for audio processing such as FAUST or Gen and execute them in a dedicated high priority thread called AudioWorklet. The application can also be utilized to create interactive programs and shareable artworks online with other JavaScript language built-ins, Web APIs, web-based audio plugins or external JavaScript modules.

0 INTRODUCTION

A well-designed Visual Programming Language (VPL) might supplement our ability to design a multimedia human-machine communication system [1]. These VPLs can be more user-friendly to non-coders, artists, designers, or children, as they visually present the programming constructs and rules to combine them. Some audio-related VPLs use the box-line visual representation. This enables the insertion of viewing monitors at various points to easily show the data to users easily [2]. The programs developed are closer to the flowchart diagram, which often compares to the way things work in our physical world, particularly in the audio processing field. Connecting signal processors using audio cables to produce sounds and effects is a widespread practice, even though we can now bring this practice to the digital world. Max [3], PureData (Pd) [4] and Vvvv,[1] which are well-known VPLs for audio and video processing, utilizes patchers, connections with cables and boxes, to describe the program's data flow [5].

Since the birth of the Web Audio API in 2011, it is possible to implement DSPs in the browser using JavaScript for building a graph of high-level audio nodes, which includes oscillators, filters, delays, reverbs, etc. This API became a W3C recommendation (a "frozen standard") in 2021 and is now supported in the latest versions of most popular desktop and mobile browsers. The AudioWorklet node was added to the specification in 2018 [6]. It allows the implementation of low-level custom audio processors (AudioWorklet processors) that can be executed in a dedicated thread using WebAssembly or plain JavaScript. Such nodes can be arranged into the "audio graph" to create more complex audio effects or instruments. As of 2022, all major browsers support the AudioWorklet API. Therefore, popular Domain Specific Languages (DSLs) for DSP programming, such as FAUST [7] or Csound can be compiled to WebAssembly and can be run in AudioWorklet nodes at runtime [8], increasing the potential of web-based audio applications.

The graph-based design of the Web Audio API makes it easily integrable in patcher-like VPLs on the web. For example, WebPd[2] is a web-based Pd patcher interpreter that uses JavaScript as well as the Web Audio API. Cables.gl[3] is a video-oriented patcher editor on the web, which also handles WebAudio nodes. WebAudio Visual Editor,[4] WebAudioDesigner,[5] Mosaicode[6] [9] as well as Olos[7] are web-oriented VPLs for audio processing.

With many web-based VPLs, users can thus create patchers only using a limited number of box types (box

[1] https://vvvv.org/
[2] https://github.com/sebpiq/WebPd
[3] https://cables.gl/
[4] https://github.com/pckerneis/WebAudio-Visual-Editor
[5] https://github.com/g200kg/webaudiodesigner
[6] https://mosaicode.github.io/
[7] https://www.jasonsigal.cc/portfolio/olos

objects), which are high-level abstractions such as generators, audio and video processors, or UI components. It is possible to create a simple audio or video sequence with these VPLs, but they are inadequate for designing more complex web applications, that are required to deal with lower-level Web APIs.

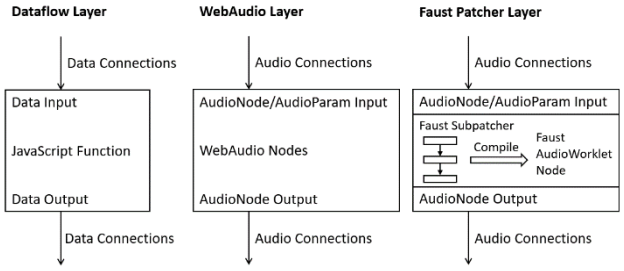

Fig. 1 The three patcher interpretation layers of JSPatcher.

The proposed patcher system JSPatcher includes three different interpretation layers. This architecture is designed to represent both dataflow and audio processing (Fig. 1). The first layer represents low-level JavaScript features, such as variables, getters, setters and functions. The language built-ins or Web APIs available under the current global scope will be imported to the system, along with features from other JavaScript modules that can be added dynamically. These imported box objects enable users to develop programs using low-level APIs just as one might code them with the JavaScript language.

The second layer, called the WebAudio layer, handles the WebAudio nodes and is used to represent a WebAudio graph where boxes are the nodes and cables are the connections between them.

These two layers are "imperative" (the code is interpreted at run time) as the patcher is intended for interactive operation through user interface components as well as for processing the data stream in real-time.

The third layer is different as it will execute "compiled" code written using the FAUST DSL [8], when custom low-level DSP is required. This can be compared to Max/Gen: Max is primarily an imperative VPL but can also include Gen [8] patchers, which are compiled to Max's DSP modules.

When JSPatcher runs a patcher that contains FAUST compiled DSPs, the corresponding WebAssembly code will be executed in an AudioWorklet node, whose processor runs in a dedicated high priority thread. The generated DSP is thus an AudioNode and can still be used as a sub-patcher of an imperative patcher.

The combination of these three layers in a single system allows the coexistence of compiled and imperative patchers. This offers two advantages: first, compiled patchers are often more efficient than imperative ones as they are considered as a single functional processor at runtime. The compiled patchers can be used to design specific sub-process such as DSPs or shaders. Second, while the compiled patchers are encapsulated, they are extendable and reusable in other patchers. This saves up computing resources and developers' efforts. In addition to that, it is interesting for the system to have different optional compilers/interpreters.

JSPatcher is a VPL inspired by Max, but designed from the start to make the most of the web platform:

1. JSPatcher uses JavaScript. It is a loosely typed language. We will not have type information regarding functions' arguments or their return values. It means that it is more reasonable to only have "data" and "signal" connection types as in Max or Pd since we cannot distinguish the connection types of the imported boxes.
2. The UI is created using HTML elements, which are rich and flexible. The boxes can be designed as different interactable components like in Max or Pd.
3. In its current state, JSPatcher has all the basic building blocks to become compatible with Max, Gen or Pd, and importers for projects in the formats of these patchers are planned.
4. A recent survey of the music creation VPLs [10] shows that Max and Pd are the most notable and widespread VPLs being used by composers and musicians. They are mainly employed for interactive systems and music composition. Moreover, in the survey, more than half of the users of non-Max VPLs also know Max. If you know how to use Max or Pd, you should be able to work with JSPatcher rapidly, because it uses the established usability models of these standard programs.

Indeed, the patching paradigm and the UI of JSPatcher are close to those of Max [11] in order to facilitate the handling by users accustomed to Max-like VPLs. Even though web applications for multimedia are generally not quite as strong and robust as those on native platforms, they are also more flexible in terms of device compatibility, networking ability and interactivity. JSPatcher has been designed specifically for the web platform: for example, patchers can import any JavaScript library using their URI and it can use any of the browser APIs (i.e., webcam/microphone using the Media Device API, WebML API, WebSockets API). To sum up, JSPatcher is not a carbon copy of Max: its implementation is fundamentally different (Max is a close source project), and it is web-aware.

In this paper, we will present how the system applies Max's patching paradigm to the JavaScript language in §1. The WebAudio and the FAUST layers will be presented in §2 and §3. The implementation details, some discussions and the conclusion will be demonstrated in the last sections.

1 PATCHING JAVASCRIPT

One of the main goals of JSPatcher is to allow users to create JavaScript applications using patchers. To achieve

[8] https://docs.cycling74.com/max8/vignettes/gen_overview

this, the patcher system should in the first place provide an equivalent way to program any ECMAScript statement or expression. Then, users should be able to get, set or save values, as well as to call functions, methods, or constructors.

In some traditional VPLs, adding new functionalities to the system is cumbersome as the box objects must be developed separately to conform to the VPL's API. In JSPatcher, no additional development effort is required for an external JavaScript module to be usable in the patcher.

1.1 Operators

Most operators from the ECMAScript standard are available as box objects. Their execution sequence, as well as the argument initialization, is similar to the one in Max.

Fig. 2 shows an example of some important JavaScript operators as box objects. Here, we used `new` to construct a `Number` object; `func` to get the `Number` constructor; `?` as the ternary operator to choose between two values according to its input boolean value.

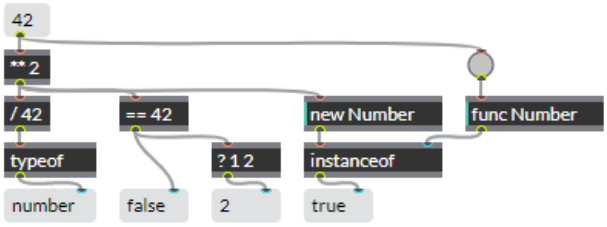

Fig. 2 Operators.

1.2 Conditions and Iterations

Patcher systems such as Max give multiple ways to handle conditions and iterations since the representation of choice branches is slightly different from literal expressions. The condition in a patcher can be an object which chooses to output the specific incoming data from many inputs or choose to output the incoming data to a specific branch. For the iterations, a loop can be created by connecting a cable from the output of a graph to its input. Moreover, we provide box objects, which will output all the iterated values with one outlet, as well as a message using another outlet when the iteration is ended, so that the rest of the program can be connected with this outlet.

For instance, in Fig. 3, conditions can be verified by utilizing the ternary operator or `gate` to block the dataflow. In Fig. 4, the graph on the left is a loop with a condition, the right one is a `for` loop with predefined borders. The message box receives a value from its second inlet to set the value without output, a *bang* (Max-style trigger event) from the first inlet will output the current value. The `sel true` will output from its first inlet a bang if the input matches `true`.

Built-in iterators such as `Array.prototype.map` can be called with a lambda function. The usage will be presented in the next subsection.

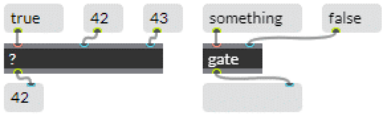

Fig. 3 Conditions

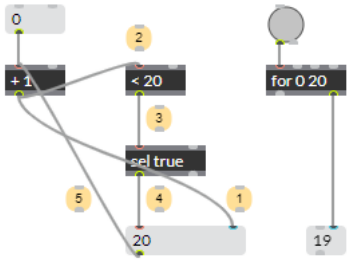

Fig. 4 Loop with a condition, the circled numbers indicate the order of the execution.

1.3 Lambda functions

In JSPatcher, a box object called `lambda` enables the creation of a JavaScript anonymous function. The function body is a graph attached to this box, taking the box's outputs as the function's arguments, then giving back to the second inlet of the box the function's return value.

The object outputs an anonymous function from its first outlet when it gets a bang from its first inlet. The function's number of arguments can be declared as the box's argument, which changes the number of outlets of the box. When the function is called, the values of the arguments are output starting from the third outlet, along with a bang from the second outlet. If the number of arguments is not declared, the arguments will be output as an array from the third outlet.

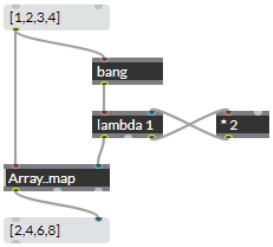

Fig. 5 Lambda function

For instance, in Fig. 5, `Array..map` represents the `Array.prototype.map` function. The first argument is the array `[1, 2, 3, 4]`, and the second is a lambda function where the function body is `* 2` which refers to multiplying each element in the array by 2.

1.4 Built-ins and Web APIs

When the JSPatcher is initialized, it scans recursively the global variable window as well as imports its content,

which involves most of the JavaScript built-ins and Web APIs. The imported variables, getters, setters and functions are then usable as different box objects.

In the example shown in Fig. 6, clicking on the button or the message is equivalent to running the JavaScript code below:

```
console.log(window);
print(escape(",\>?"));
globalThis.onclick = () => print(innerHeight);
```

Fig. 6 Imported Web APIs

As for the box objects imported from a JavaScript prototype, these identifiers omit the string `prototype`, and there will be an additional inlet and an additional outlet for passing an instance of the prototype. This facilitates the calling of the instance's methods or using its setters, getters or properties.

To build a JavaScript object from its constructor function, users can use the `new` box object, followed by the identifier of the constructor's box object and the arguments. The box will evoke the `new` operator on the constructor and will output the instance from the first outlet.

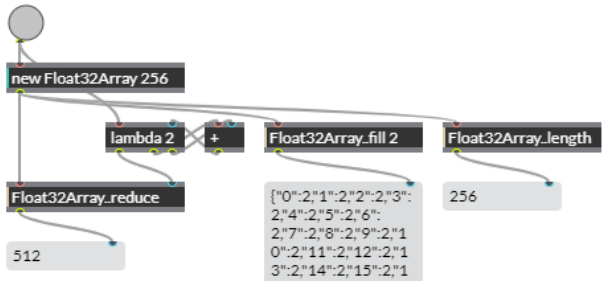

Fig. 7 Constructor and methods

Fig. 7 is an example equivalent to the following JavaScript code:

```
const array = new Float32Array(256);
array.length;
array.fill(2);
array.reduce((acc, cur) => acc + cur);
```

Getting or setting a specific JavaScript object property using the `set` and `get` box objects is possible. Moreover, a `call` box object can be used to call a specific object method by name.

Fig. 8 illustrates an example of building a WebAudio graph (oscillator-gain-destination) with JavaScript box objects. However, it needs too many boxes for connecting three audio nodes. We will present a simpler version in the next section.

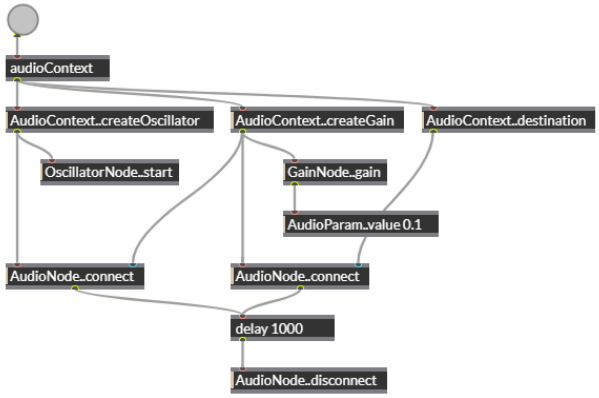

Fig. 8 Creation of a WebAudio graph with JavaScript boxes

1.5 External JavaScript Modules

JavaScript library/module authors frequently make their work available online through a Content Delivery Network (CDN) so that it may be fetched remotely. Websites like unpkg.com provide available packages on NPM, [9] a JavaScript module registry. It is practical to get these public JavaScript modules with a CDN URI and the package identifier.

The packages on NPM are mainly developed for Node.js, and handled using the CommonJS module standard. JSPatcher simulates the Node.js environment and allows the import of these packages as box objects under a given namespace. There is also a possibility to import directly ES6 modules into the system.

When JSPatcher loads a patcher, the NPM packages and ES6 modules used are automatically imported from their URIs, as well as their dependencies.

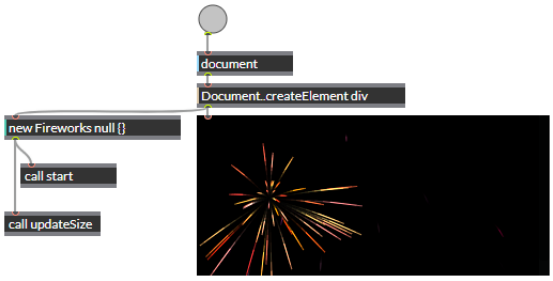

Fig. 9 Creation of graphical elements using an external module

With JavaScript in patchers, users can create and customize UI components via plain HTML/CSS/JS code. The `view` object enables the display of the created HTML element. It is also possible to make complex graphical elements using third-party modules. For instance, Fig. 9 illustrates an example for generating visual effects in a patcher using the external module fireworks-js.[10]

[9] Node Package Manager: https://www.npmjs.com/

[10] https://fireworks.js.org/

2 PATCHING WEBAUDIO

2.1 WebAudio Node Box

JSPatcher provides a dedicated layer for WebAudio nodes in addition to utilizing JavaScript box objects to construct a WebAudio graph. In this layer, each box represents one WebAudio node. Its audio connections, as well as the parameter input, become the box's inlets and outlets. If a cable is connected between an inlet and an outlet both marked as a WebAudio connection, the cable will be displayed differently, and call native WebAudio connect and disconnect methods from the AudioNode interface while manipulated.

The layer is compatible with functional box objects and data cables, and data transmitted through normal cables may still be processed. For example, inlets representing AudioParams can be connected from an AudioNode as in the WebAudio API, or be connected from box objects, which generate numbers to be set as the value of the AudioParam. Some customized WebAudio nodes can have their inlet for receiving MIDI messages as well.

One additional outlet of these WebAudio node box objects outputs the instance of the AudioNode for further possible usage via JavaScript box objects. For instance, Fig. 10 is equivalent to the example from Fig. 8.

Fig. 10 Creation of a WebAudio graph using WebAudio boxes.

2.2 AudioWorklet

JSPatcher comes with a collection of box objects that helps to code, register and utilize an AudioWorklet node in the patcher system. Firstly, users can add a code box to write an AudioWorklet processor with plain JavaScript code. The box object `audioWorklet` then allows users to register the processor from the code, using internally `createObjectURL`. The box will emit a bang when the processor has been registered, which may be used to create the AudioWorklet AudioNode with the processor's identifier. The `node~` box object can bring any AudioNode into the WebAudio connection layer so that the constructed AudioWorklet node can be connected to other AudioNode boxes.

In Fig. 11, the AudioWorkletProcessor is written in a code box, registered by the `audioWorklet` box. Then it is created through the AudioWorkletNode constructor, and transformed using `node~` into an AudioNode box.

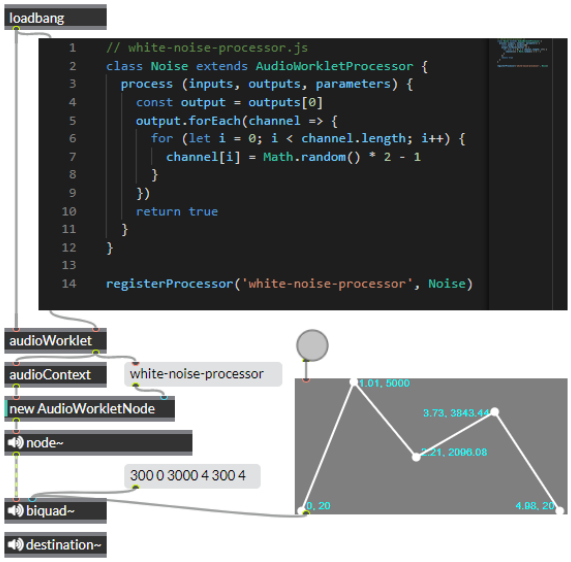

Fig. 11 AudioWorklet example

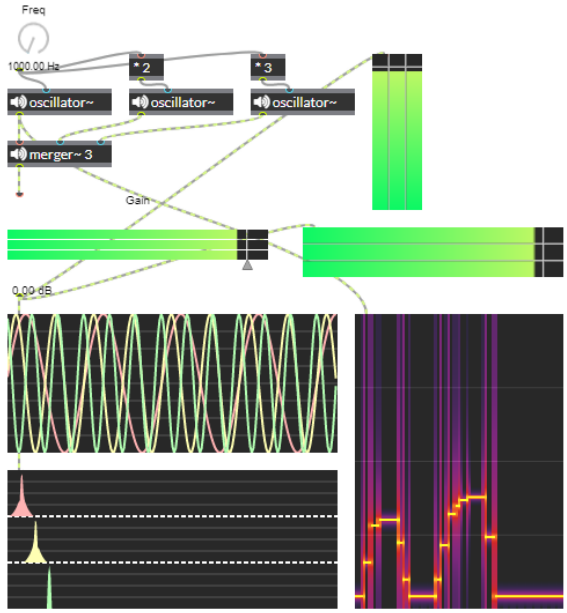

Fig. 12 Visualizations

2.3 UI with WebAudio Node

Visualizations are key features for developing audio applications. These can be achieved using an AudioNode which receives and analyzes the real-time signal, and HTML elements to display the result of the analysis. In JSPatcher, an analyzer node with visualization can be packed into one WebAudio node box.

For instance, a level meter can be a box object which displays instant RMS (root mean square) values graphically, with one inlet as a connection to an analyzer AudioNode.

Fig. 12 is an example of different visualizations of three sine-wave oscillators.

2.4 AudioNode generated by FAUST[11]

FAUST is a functional, synchronous, domain-specific programming language designed for real-time audio signal processing and synthesis.

Several developments have been done to use the language on the Web platform. Thanks to the Emscripten transpiler and the WebAssembly format, the FAUST compiler is available as a JavaScript module `faust2webaudio` [12] which can compile FAUST code to a fully functional WebAudio AudioWorklet node.

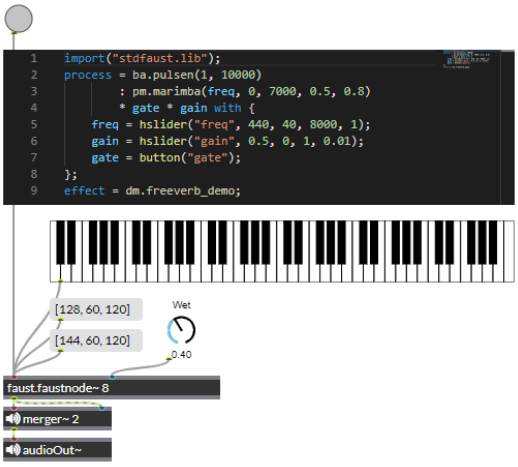

Fig. 13 FAUST node

The language also allows us to describe MIDI-controllable parameters of the DSP or polyphonic MIDI instruments. The parameters will be interpreted as AudioParams, and the node has APIs to handle MIDI messages.

The compiler is available with the `faustnode~` box object. When receiving the FAUST code, it will attempt to compile the code and turn itself into a WebAudio node box. Like the AudioWorklet box, its AudioNode and AudioParams are connectable with other WebAudio node boxes, moreover, it handles incoming MIDI messages from its first inlet.

Fig. 13 illustrates the compilation of an eight-voice polyphonic instrument from FAUST. The instrument is handling MIDI messages from its first inlet.

3 PATCHING FAUST LANGUAGE

The compiler of the FAUST programming language transforms the code into a patcher-like graph called block-diagram algebra (BDA) [13, 14] which can be optimized and transformed into a high-performance low-level code. The BDA serves as a bridge between user-written code and the compiled internal code. Using the BDA, FAUST compiler can generate a block diagram that shows the processing structure of the compiling DSP.

[11] A video demonstration is available on: https://youtu.be/vYgqjakKYwo

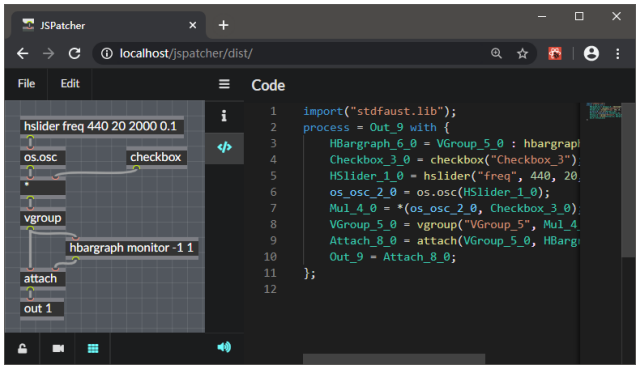

Fig. 14 The generated code can be previewed in the right panel (synchronized to the patcher)

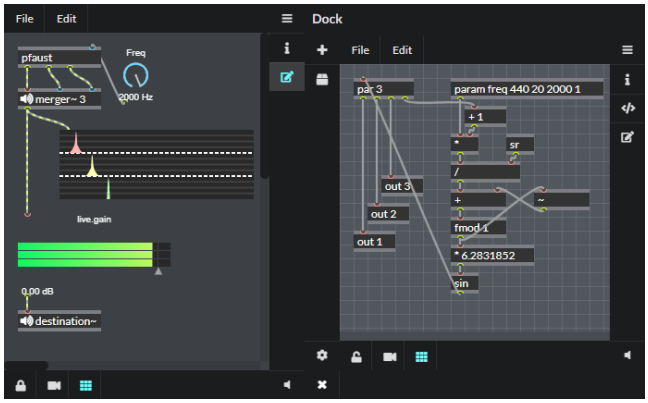

Fig. 15 A FAUST patcher can be compiled to a WebAudio node box (the patcher on the right is the FAUST patcher)

Thus, a FAUST code is always represented by a graph, which leaves the possibility of generating code from an equivalent graph. In JSPatcher, we designed a specific mode for building a FAUST-compatible graph in the patcher, that will be firstly interpreted to an equivalent FAUST code which can be used in other FAUST tools, then be compiled to a WebAudio node using the WebAssembly version of the FAUST compiler in real-time. While patching in this mode, users have a panel that shows the interpreted code of the actual patcher in real-time.

The implementation of this layer has been inspired by Gen, which is also a graph-to-code system that can be compiled into a high-performance DSP in Max.

Fig. 14 demonstrates the editor of a FAUST subpatcher with the code display. Fig. 15 illustrates a running FAUST subpatcher as a WebAudio node.

3.1 Inlets, Outlets, Inputs and Outputs

When a FAUST patcher is interpreted, the boxes and cables in the patcher are analyzed to generate the FAUST code. The boxes with different text on them represent different FAUST functions. They carry a certain number of inlets and outlets as their function parameters as well as

results (inputs and outputs). The text on the box can be followed by arguments to override its parameters and suppress the inlets (Fig. 16). Unconnected inlets will be replaced by a default value (usually 0) while compiled.

Fig. 16 The "+" function has two inlets by default, one will be suppressed by adding an argument

At least one output is required for a patcher to be valid for the interpreter. Outputs are the starting points of the analysis. Inputs are involved in the code generation only if a path from input to output can be found. Inputs and outputs are represented as boxes named `in` and `out`. For instance, in Fig. 17, `out 1` outputs the addition of `in 1` and `in 2`. `out 3` outputs 0, the graph above `out 3` will not be interpreted.

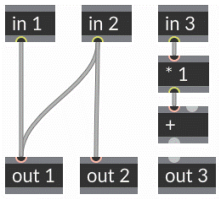

Fig. 17 Inputs and outputs of a FAUST patcher

3.2 Functions

The system imports not just the FAUST language's built-in functions, but also the FAUST standard library `stdfaust.lib` [15]. These functions are available as box objects under a patcher in FAUST mode.

When using these box objects, their arguments and properties provided by the user are analyzed with several rules to generate the corresponding FAUST code. For example, the "_" as an argument represents a placeholder, which can be used to preserve an inlet port among predefined arguments. The ins and outs properties force the number of inputs and outputs of the box, in cases like pattern-matching functions.

3.3 Loops

Dealing with audio signal loops is a common and critical issue in DSP language design, particularly for its usage as feedback in the simulation of echo, reverb and amplifier effects. In FAUST language, it is possible to create a loop with the recursive operator "~," which introduces a one-sample delay between its input and output. To create a loop in the patcher, users can use the "~" box object and connect looped cables around it (Fig. 18).

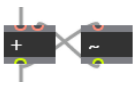

Fig. 18 Loop in FAUST patcher

3.4 Subprocess

A code in a box and subpatcher can be used in a FAUST patcher for creating reusable subprocesses. FAUST has four iterators: `par` to duplicate a signal in parallel; `sum` to calculate the sum of duplicated signals; `prod` to calculate the product of duplicated signals; `seq` to calculate in series the duplicated signals. To iterate, these boxes must be connected to a subprocess.

A code block can carry an independent FAUST code to be treated as a subprocess. While the code is being changed, the system will evaluate it to get the number of its IOs, then create corresponded inlets and outlets (Fig. 19).

Fig. 19 code block in a FAUST patcher

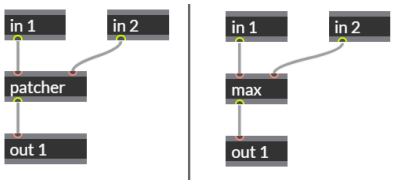

Fig. 20 subpatcher (right) in a FAUST patcher (left)

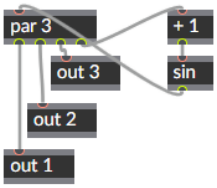

Fig. 21 `par` iteration graph

A subprocess can also be created as a FAUST subpatcher. The subpatcher's number of IOs depends on the `in` and `out` boxes in it. For instance, Fig. 20 is equivalent to Fig. 19.

The iterators' boxes behave like the `lambda` box where the last outlet represents a 0-based incremental variable local to the attached subgraph, the subgraph needs to have its output attached to the inlet of the box so that the graph would be considered as a subprocess to iterate. Fig. 21, for example, generates three parallel signals: `sin(1), sin(2), sin(3).`

3.5 Parameters

FAUST proposes some primitives for UI description (checkbox, button, sliders, etc.), the function names utilized for describing UI components are available as box objects, which also generates the corresponding

AudioParams at compilation time. For instance, in Fig. 22, both the `hslider` and the `checkbox` are user-controllable parameters, that will become AudioParams and have a dedicated inlet on the compiled FAUST subpatcher box.

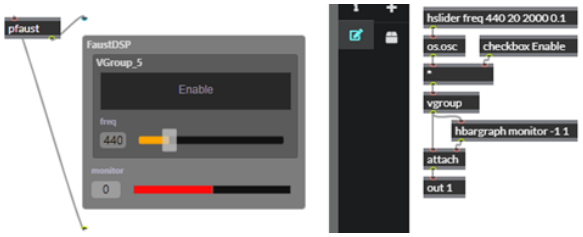

Fig. 22 FAUST patcher with UI descriptors (right) and its result (left)

4 AUDIO PLUGIN SUPPORT

4.1 WebAudioModule Plugin Standard

In the music production industry, most hardware devices have been substituted by software solutions over the past decades. Using DAWs with third-party audio plugins that act as synthesizers or audio effects is a common workflow for modern audio projects. VST (Virtual Studio Technology), introduced by Steinberg in 1996, is one of the most used cross-platform native plug-in formats based on C++. The VST standard comes with an SDK and API documents to allow plugin vendors and host developers to implement them properly.

Owing to the limitation of the web platform, native audio plugins such as VSTs cannot be utilized in a web application without installing additional native software. Therefore, for web-based DAWs, the need arose for a new standard of WebAudio plugins, which offers similar functionality to their native counterparts.

A new version of the WebAudioModule (WAM) standard [16, 17], which had been first released in 2015, was redesigned for reflecting recent technological developments. A URI can be used to fetch a WAM plugin from the web, and the current WebAudio context can be used to initialize it. WAM plugins have a standardized API. A host can use this API to display the WAM plugin UI, get/set the plugin parameter values, schedule events such as parameter automation or MIDI messages, and connect the plugin with other WAMs plugins or with native WebAudio nodes.

4.2 WAM in Patchers

It is critical to be able to use WAM plugins (WAMs) in JSPatcher to construct interactive audio programs. Using the `plugin~` object, users can load WAMs via a URI into a patcher.

When the `plugin~` receives a text string (as URI), it will remotely download the code from the URI and initialize it as a WAM. Moreover, it will automatically wrap the WAM and load its UI. Its inlets and outlets will be connectable with other WebAudio node objects. Additional inlets will be created to accept real-time parameter change messages.

Since WAMs accept MIDI messages as input, the first inlet of the object takes numbered arrays as MIDI event messages to the WAM within. Moreover, as WAMs can transmit non-audio events between each other, the patcher connection between two WAMs will be treated as a special one and make the necessary connection.

Fig. 23 is an example of a WAM loaded in a patcher with an audio player.

Fig. 23 WAM with an audio player in a patcher

5 IMPLEMENTATION

5.1 Tool-chain

We chose TypeScript[12] as our primary development language since it has a higher level of maintainability than JavaScript, thanks to its typing system. This language also supports multiple tools that we used, such as React,[13] a framework we utilized primarily for the UI layer; Babel,[14] a set of JavaScript compilation utilities that help keep our code runnable on different browser versions; Webpack,[15] a production tool that manages code dependencies, etc.

For the UI, we use the React version of the Semantic UI[16] component library, Monaco[17] source code editor, SCSS[18] for page layout design, etc.

5.2 Event System

An event management system is the core of the whole JSPatcher. The boxes are event emitters in an imperative patcher, whereas the cables are event listeners. While a box attempts to output some data, the cables attached to the related outlet handle the event output, then call the method of the box attached at the other end.

When a box receives data from any inlet, or when its arguments and properties are updated, it emits

[12] https://www.typescriptlang.org/
[13] https://reactjs.org/
[14] https://babeljs.io/
[15] https://webpack.js.org/
[16] https://react.semantic-ui.com/
[17] https://microsoft.github.io/monaco-editor/
[18] https://sass-lang.com/

corresponding events. The box's behavior responding to these events is defined as event handlers. This makes it easy to extend a box object and make it interact with other boxes.

The UI system is loose-bounded owing to the event system. This is interesting as it means that the patcher can also be run without a user interface or only partially mounted, making it easily embeddable into any web application.

6 DISCUSSION

6.1 Evaluation

6.1.1 User Test

We have set up tests with professional users who have a background in computer music, to collect feedback and opinions about JSPatcher. To do so, we asked them to create several audio programs and to answer a series of questions.[19]

The survey was published on February 26, 2022. Attendees were invited to open JSPatcher in the Google Chrome browser, using a desktop computer. Then they had to follow a tutorial and complete a 3-step task in JSPatcher. Finally, they had to fill out a survey and give their opinions. The 3 steps are:

1. Generating a random pitch,
2. Attaching a metronome to the random pitch generator,
3. Connecting the generator to a synthesizer and make some sounds.

These steps are guided by patcher examples and proposed objects. A possible solution can be revealed optionally.

The survey questionnaire asked participants whether they found the application worked well, if it was easy to use, if the requested tasks were simple to perform, and more generally, what they liked about the proposed application, or on the contrary, what they disliked. Other broader questions were aimed at understanding what aspects of JSPatcher interested them the most.

According to the collected feedback, the main reason for users to realize projects in JSPatcher is its accessibility on the web, and its similarity with Max and Pd. Having a sort of "Max on the Web" also facilitates the prototyping of audio programs, and in the FAUST patcher layer, the generated code can also be compiled for other platforms. In addition, it appeared that JSPatcher is well suited for pedagogical use, as students can quickly realize simple algorithms on various devices just by using a web browser.

The integration of the Web API and the support for external JavaScript modules have highly been appreciated as it enables developers to design interactive web pages with less coding effort.

6.1.2 Artwork

These two aspects make the system suitable for hosting online artworks. As a proof of concept, we created *Urban Sound Tales*, [20] an online interactive multimedia installation made with JSPatcher [18].

This work has been developed by composer Tak-Cheung Hui, who also designed the music and the sound part of the piece. Photographers (Sean Wang and Chon Ip) have been engaged to contribute a set of images and videos that are life scenes from two Asian cities.

Spectators are invited to open four shared links to run prepared patchers in JSPatcher, rendered in the presentation mode of the tool (without menu and sidebars displayed) (Fig. 24). They can click on different parts of the images, or interact with UI components, to hear the "sonic past" of the two cities. The sounds are processed or synthesized in real-time by WebAudio native nodes and FAUST DSPs. Spectators can activate multiple soundtracks for creating their own audio mixes.

Fig. 24 A patcher of *Urban Sound Tales* in the presentation mode

The realization of this work has highlighted some strengths of JSPatcher, but also some limitations. The composer started the work by prototyping ideas on Max patchers. We, JSPatcher developers, audited these patchers to evaluate the possibility to port them to the web platform. In this process, some large media files have been reduced in size, or replaced by real-time DSPs to speed up the file transfer time on the web. Then, these files have been uploaded on a server and added to JSPatcher as HTML elements, allowing displaying, positioning of the play head, and changing the replay speed through the JavaScript API. The DSPs have been ported using FAUST subpatchers and optimized - reducing the number of synthesizers and parallelly running voices in polyphonic instruments - for low performance machines. Then the composer retouched the work this time on JSPatcher, taking ported patchers as examples.

The workflow showed that technical work on the optimization was needed for running as web-based artwork. Even though, for artists who are familiar with VPLs, JSPatcher provides a way to bring their ideas on the web with little effort. They can understand the system and quickly start working if examples are provided.

We are regularly updating JSPatcher's documentation and examples, in particular to illustrate the usage of each box objects. Some limitations are also related to the current

[19] https://forms.gle/dhA8zDypLdYqDjQm9

[20] https://urbansoundtale.com/

Web Audio specification and implementations such as the lack of a performance profiling tool or the support of some features in the Safari browser.

6.2 Future Work

We are first planning to work on adding collaborative work features to share real-time projects and enable groups of users to create music pieces together at the same time. The networking features like WebSocket or WebRTC can be used to synchronize music between different devices.

We will organize more user evaluations, especially with computer music students, since the system will be used in pedagogical scenarios. Their use cases are important for us to improve the system ergonomically, and align it with the newest web technologies.

We also plan to add the following features:

6.2.1 Timeline and Musical Notation

Patcher-like VPLs are good choices to build musical applications or event music generators, as a timeline or a musical score can be displayed in real-time in a patcher. For instance, OpenMusic [19] is a VPL based on the LISP language for computer-assisted composition (CAC). It has a graphical user interface for calculating and representing musical data. It is interesting for JSPatcher to add related features, including the display of the score and the timeline, replay of MIDI files, abstractions of musical concepts, etc.

Moreover, since JavaScript supports the functional programming paradigm, users who are familiar with the LISP language could run their CAC project in JSPatcher with less migration effort.

6.2.2 AudioWorklet Generators

We are planning to extend the choice of languages for writing AudioWorklet processors running custom DSP code. In addition to Faust, we started to look at a Csound [20] integration. Csound is a popular language that can be compiled into an AudioWorklet node using the WebAssembly version of its compiler [21]. Moreover, it would be interesting to design an AudioWorklet processor with JavaScript boxes in a subpatcher by having a dedicated patcher system in the audio thread. It will then be possible to create imperative patchers for processing audio buffers or FFT data of events, directly with JavaScript.

6.2.3 SDK

The box objects in the JSPatcher are extendable and intended to be fully accessible for further developments from community contributors. We are working on a software development kit (SDK) based on these built-in box objects. With this SDK, developers will be able to create their own box object packages, distribute them and make them available to the JSPatcher user community. JSPatcher can import such extensions just using URIs.

7 CONCLUSIONS

The advantages and disadvantages of dataflow VPLs have been discussed for decades. In comparison to textual languages, VPLs have been more accessible and illustrative in some fields such as multimedia processing but lack clearance and performance with some complex algorithms [22]. For designing WebAudio applications, JSPatcher is similar to some other platforms which enable users to manipulate an audio graph and control the parameters graphically. Our original intentions, however, were to give JSPatcher greater flexibility and potential, such as the ability to create low-level AudioWorklet DSPs directly from the patcher system and to take control of other JavaScript-based web functionality.

Developers can also write code in boxes to implement complex algorithms and then connect user interface components to them. JSPatcher is a hybrid system where compiled and imperative code and patching coexist. With this approach, we tried to overcome the drawbacks and limitations of current VPLs.

This project started with the desire to primarily do audio programming, but when we implemented it, it became apparent that there were more use cases. For example, we used the JSPatcher platform to program applications with a 3D graphics layer, with OpenGL rendering via the three.js[21] library, we developed patchers with data visualization using the d3.js library, [22] or exploited Machine Learning models with neural networks based on the Tensorflow.js web framework.[23] None of these applications required any modifications to JSPatcher, which is able to naturally integrate JavaScript libraries available on the web. Its first-class support of the web platform makes JSPatcher an original tool, which distinguishes itself from its big brothers: desktop applications such as Max or Pd. JSPatcher, along with many patcher examples, is available on a GitHub repository.[24]

8 ACKNOWLEDGMENTS

This research project has been commissioned by Shanghai Key Laboratory for Music Acoustics (SKLMA-2022-01).

Our thanks to Association Francophone d'Informatique Musicale for the continuous support.

Our thanks to GRAME-CNCM (Lyon, France) for the support and ideas for the design of the FAUST-based patcher.

9 REFERENCES

[1] S.-K. Chang, "Visual languages: A tutorial and survey," in *Visualization in Programming*, pp. 1-23 (Springer Berlin Heidelberg, 1987), https://doi.org/10.1007/3-540-18507-0_1.

[2] D. D. Hils, "Visual languages and computing survey: Data flow visual programming languages," *Journal of*

[21] https://threejs.org/

[22] https://d3js.org/

[23] https://www.tensorflow.org/js

[24] https://github.com/fr0stbyter/jspatcher

Visual Languages & Computing, vol. 3, pp. 69-101 (1992 Mar.).

[3] M. Puckette and D. e. a. Zicarelli, "Max/msp," *Cycling* (1990).

[4] M. Puckette, "Pure Data," in *Proceedings of the International Computer Music Conference*, pp. 224–227 (Thessaloniki, Hellas) (1997 Sep.).

[5] M. Puckette, "The patcher," in *Proceedings of the 1986 International Computer Music Conference*, pp. 420–429 (San Francisco, United States, 1988).

[6] H. Choi, "AudioWorklet: The future of web audio," in *Proceedings of the International Computer Music Conference*, pp. 110–116 (Daegu, South Korea) (1988 Aug.).
https://doi.org/10.1016/1045-926X(92)90034-J.

[7] Y. Orlarey, D. Fober, and S. Letz, "FAUST: an Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, pp. 65–96 (Editions Delatour France, France, 2009).

[8] S. Letz, S. Denoux, Y. Orlarey, and D. Fober, "Faust audio DSP language in the Web," in *Proceedings of the Linux Audio Conference*, pp. 29–36 (Mainz, Germany) (2015 Apr.).

[9] F. L. Schiavoni, L. L. Gonçalves, and A. L. N. Gomes, "Web Audio application development with Mosaicode," in *Proceedings of the 16th Brazilian Symposium on Computer Music*, pp. 107–114 (São Paulo, Brazil, 2017).

[10] A. Pošćić, G. Kreković, and A. Butković, "Desirable Aspects of Visual Programming Languages for Different Applications in Music Creation," *in Proceedings of the Sound and Music Computing Conference*, pp. 329-336 (Ireland) (2015 Jul.), https://doi.org/10.5281/zenodo.1400776.

[11] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, pp. 31–43 (2002).

[12] S. Ren et al., "FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins," in *Proceedings of the International Web Audio Conference*, pp. 71–76 (Trondheim, Norway) (2019 Dec.).

[13] Y. Orlarey, D. Fober, and S. Letz, "An Algebra for Block Diagram Languages," in *Proceedings of the International Computer Music Conference*, pp. 542–547 (Gothenburg, Sweden) (2002 Sep.).

[14] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and Semantical Aspects of Faust," *Soft Computing*, vol. 8, pp. 623–632 (2004 Sep.), https://doi.org/10.1007/s00500-004-0388-1.

[15] R. Michon, J. Smith, and Y. Orlarey, "New Signal Processing Libraries for Faust," in *Proceedings of the Linux Audio Conference*, pp. 83–87 (Saint-Etienne, France) (2017 Jun.).

[16] M. Buffa, J. Lebrun, J. Kleimola, O. Larkin, and S. Letz, "Towards an open Web Audio plugin standard," in *Companion Proceedings of the Web Conference 2018*, pp. 759–766 (Lyon, France) (2018 Apr.), http://doi.org/10.1145/3184558.3188737.

[17] M. Buffa et al., "Emerging W3C APIs opened up commercial opportunities for computer music applications," in *The Web Conference 2020 DevTrack* (Taipei) (2020 Apr.).

[18] T.-C. Hui and S. Ren, "Urban Sound Tales: The Invisible Landscapes – the Sonic Past of the Two Cities," in *Proceedings of the International Web Audio Conference* (Cannes, France) (2022 Jul.).
https://doi.org/10.5281/zenodo.6769891

[19] J. Bresson, C. Agon, and G. Assayag, "OpenMusic – Visual Programming Environment for Music Composition, Analysis and Research," in *Proceedings of the 2011 ACM Multimedia Conference on Multimedia*, pp. 743–746 (Scottsdale, Arizona, United States) (2011 Nov.), https://doi.org/10.1145/2072298.2072434.

[20] V. Lazzarini, S. Yi, J. Heintz, Ø. Brandtsegg, I. McCurdy, and others, *Csound: a sound and music computing system*. (Springer Publishing Company, Incorporated, 2016).

[21] S. Yi, V. Lazzarini, and E. Costello, "WebAssembly AudioWorklet Csound," in *Proceedings of the International Web Audio Conference* (Berlin, Germany) (2018 Sep.).

[22] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, no. 7, pp. 491–541 (1997 Jul.), https://doi.org/10.1007/s002360050095.

THE AUTHORS


Shihong Ren


Laurent Pottier


Michel Buffa


Yang Yu

Shihong Ren is a composer/researcher in computer music, currently at Shanghai Conservatory of Music, Jean Monnet University, a member of the Shanghai Key Laboratory for Music Acoustics, a member of the ECLLA research laboratory, a member of WIMMICS research group, common to INRIA and to the I3S Laboratory (CNRS). He entered the Conservatoire national supérieur musique et danse de Lyon in 2011 in the electroacoustic composition class, and graduated in 2016 as the youngest DNSPM and master's degree owner in the composition major. He got the Artist Diploma in 2018, and followed the cursus of composition in IRCAM in the same year. He attended an internship at GRAME-CNCM (Lyon, France) in 2019.

•

Laurent Pottier is a professor/researcher at Jean Monnet University, a member of the ECLLA research laboratory. His activities are related to music using electronic and digital technologies. He teaches in the music department where he created the professional master's degree RIM in 2011 (Director in Computer Music). He taught at IRCAM (1992-1996) then directed the research sector at GMEM in Marseille (1997-2005). He has worked with many composers including J.-B. Barrière, T. De Mey, A. Liberovicci, C. Maïda, A. Markeas, F. Martin, T. Murail, J.-C. Risset, F. Romitelli, KT Toeplitz.…

•

Michel Buffa is a professor/researcher at University Côte d'Azur, a member of the WIMMICS research group, common to INRIA and to the I3S Laboratory (CNRS). He contributed to the development of the WebAudio research field, since he participated in all WebAudio Conferences, being part of each program committee between 2015 and 2019. He actively works with the W3C WebAudio working group. With other researchers and developers, he co-created the WebAudio Plugin standard.

•

Yang Yu is a composer/professor, doctoral advisor, currently director of the department of music engineering at Shanghai Conservatory of Music, senior researcher at the He Luting Advanced Research Institute for Chinese Music, director of the Shanghai Key Laboratory for Music Acoustic Art, deputy director of the Academic Committee of the Key Laboratory of Digital Music Intelligent Processing Technology (Chinese Ministry of Culture and Tourism), deputy director of Art and Artificial Intelligence Committee of CAAI, executive vice president of Film Sound Art Committee of China Film Association, deputy director of Computational Art Branch of CCF, deputy director of Music and Sound Committee of China College Film and Television Association.