Build WebAudio and JavaScript Web Applications using JSPatcher: A Web-based Visual Programming Editor

Shihong Ren, Laurent Pottier, Michel Buffa

To cite this version:

Shihong Ren, Laurent Pottier, Michel Buffa. Build WebAudio and JavaScript Web Applications using JSPatcher: A Web-based Visual Programming Editor. Web Audio Conference 2021, Jun 2021, Barcelone, Spain. hal-03519504

HAL Id: hal-03519504
https://hal.inria.fr/hal-03519504

Submitted on 10 Jan 2022

Build WebAudio and JavaScript Web Applications using JSPatcher: A Web-based Visual Programming Editor

Shihong Ren
Université Jean Monnet
Saint-Etienne, France
shihong.ren@univ-st-etienne.com

Laurent Pottier
Université Jean Monnet
Saint-Etienne, France
laurent.pottier@univ-st-etienne.fr

Michel Buffa
Université Côte d'Azur, CNRS, INRIA
Sophia Antipolis, France
michel.buffa@univ-cotedazur.fr

ABSTRACT
Many visual programming languages (VPLs) such as Max [1] or PureData [2] provide a graphic canvas to allow developers to connect functions or data between them. This canvas, also known as a patcher [3], is basically a graph meant to be interpreted as dataflow computation by the system. Some VPLs are used for multimedia performance or content generation as the UI system is often an important part of the language. This paper presents a web-based VPL, JSPatcher, which allows not only to build audio graphs using the WebAudio API, but also to design graphically AudioWorklet DSPs with FAUST toolchain, [4] [5] or to create interactive programs with other language built-ins, Web APIs or any JavaScript modules.

1. INTRODUCTION
Visual programming languages are likely to be more user-friendly to non-coders, artists, designers or children as these programs seem closer to the flowchart diagram, which often corresponds to the way things work in our physical world, especially in the audio processing field. Connecting signal processors using audio cables to produce sounds and effects is a common practice even though we can now bring this practice to the digital world. Max, PureData and Vvvv,[1] which are well-known VPLs for audio and video processing, use patchers, connections with cables and boxes, to describe the dataflow of the program.

Patcher-like VPLs are massively developed especially on the web. WebPd [2] is a web-based PureData patcher interpreter using JavaScript and the WebAudio API. Cables.gl[3] is a video-oriented patcher editor on the web that also handles WebAudio nodes. WebAudio Visual Editor,[4] WebAudioDesigner,[5] Mosaicode[6] [6] and Olos[7] are web-oriented VPLs for audio processing.

However, with many web-based VPL, users can create patchers only from a limited number of different types of boxes (box objects), which are high-level abstractions like generators, audio and video processors, or UI components. It is possible to create simple audio or video sequences, but insufficient to implement more complex web applications that need to deal with lower-level Web APIs.

The patcher system we designed aims to be able to create a patcher from boxes that represent JavaScript usages, 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 usages from other JavaScript modules that can be included dynamically. These imported box objects allow users to create programs from lower-level APIs just like code with JavaScript.

On top of these lower-level box objects, we implemented two additional layers of patcher interpretation. The first is a representation of the WebAudio graph which contains connections between WebAudio Nodes. The graph is similar and fit to a patcher system in which boxes are the Nodes and cables are the connections between them. Another layer is designed to carry subpatchers (patchers in patcher) that can be in different modes: imperative or compiled.

A patcher can be imperative, interactive with UI components and process dataflow in real-time; or compiled, to generate a program to execute at runtime. For example, Max is mainly an imperative VPL but can include Gen[8] patchers, which will be compiled to Max's DSP modules after edit.

The mixed system like Max and its integrated Gen, allowing the coexistence of compiled and imperative patchers in a single environment, provides two advantages. First, compiled modules are often more efficient compared to 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, which economizes computing resources and developer's efforts.

[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.

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

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

Besides, it would be interesting for the system to have different compilers as options to interpret these patchers.

Using this approach, JSPatcher offers possibilities to design AudioWorklet DSPs with compiled patchers thanks to FAUST WebAssembly compiler [7] and to interact with them in real-time from an imperative patcher.

The UI of JSPatcher is inspired by Max and meant to be close to Max to facilitate the comprehension and the usage of Max-like VPL developers. Yet, JSPatcher is designed for different purposes compared to Max, as web applications for multimedia will not perform as well as on native platforms in terms of efficiency and reliability, but more flexible on device-compatibility, networking and interactivity.

2. PRINCIPLES OF PATCHING[9]

2.1 Cables and Boxes

A patcher in JSPatcher, following Max's convention, usually contains cables and boxes. A box represents a function with or without UI, can take data from its input ports (inlets) and send processed data to its output ports (outlets). A cable represents a connection between one inlet and one outlet, meaning that data is flowing from the outlet to the inlet. One-to-many or many-to-one port connections are possible.

The inlets of a box are on its top, the outlets are on its bottom, aligned horizontally. A box will normally send out data to its outlets from right to left. A box can be positioned anywhere in a 2D space. The position will influence the priority while receiving data from one outlet: When data coming from one outlet should be delivered to multiple destination inlets, the position of these inlets will be used to compare the priority. The inlet at the right side will have a higher priority and receive firstly the data, if aligned vertically, the one at the bottom will have a higher priority.

For example, in Figure 1, `print B` and `print A` are connected to a message. In the two cases, `print B` will receive the message earlier than `print A`.

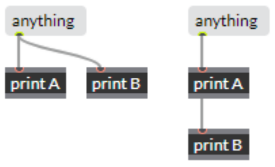


Figure 1 Messaging priority

2.2 Box's Class, Arguments and Properties

For any box with the default UI, its function is dynamic and changeable by editing its text. When the box's function is changed, it's number of IOs will also be actualized, the cables no longer being used will be removed.

The behavior of a box is mainly determined by its text. The text will be parsed to three parts: class, arguments and properties, each element is separated by a white-space and be considered as a JSON string. The first element is the class identifier to a registry in the system that contains all the available classes. Elements after the identifier are the arguments, as the parameters of a function. Then, if an element string starts with the character "@", it will be considered as an identifier of the box's property, elements after the property identifier are its value.

The arguments and the properties indicated in the box's text only determine its initial state, they can be changed any time with any operations without changing the text.

For example, in Figure 2, the box `+ 2 @textAlign right` initialized the box's class as `+`, with one argument `2`, and the value `right` of the property `textAlign`.

2.3 States of a Patcher

A patcher in the presented system is editable while the patcher is unlocked. In this state, users can add or remove boxes and cables, move or resize the boxes, change the endpoints of the lines, or change the boxes' text, arguments or properties. If a patcher is locked, the user can interact with the boxes if they provide UIs.

A patcher can also be in the presentation state, in which boxes can be displayed or not, presented with another position and size without affecting the program. As cables and non-UI-related boxes will be hidden in the presentation state, this is an interesting feature for the design of a user-friendly application with the system.

2.4 Bang Object

Similar to Max, JSPatcher uses a specific object "Bang" as an event to tell any box object to proceed with its task. The Bang contains no additional information, its only purpose is to trigger immediately anything, which is likely to output the previous result or stored value.

For example, in Figure 3, when the user clicks the button, it will output a Bang that triggers the message to output the string "Do something", then `alert` will display the string in a dialog.

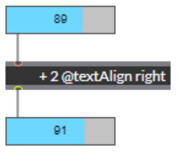


Figure 2 Box's text

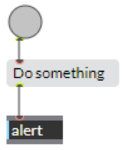


Figure 3 Bang

3. PATCHING JAVASCRIPT

One of our main goals in JSPatcher is to offer a possibility to create JavaScript programs with patchers. To achieve that, the patcher system should in the first place have an equivalent way to program for any ECMAScript statement or expression if necessary. Then, get, set or store values, calling functions, methods or constructors should be possible with any language built-ins, Web APIs or external JavaScript modules.

3.1 Operators[10]

Most of the operators in the ECMAScript standard are available as box objects. For binary operators, one argument can be provided to initialize the second component which can also be changed from the second inlet. The first component will be determined by the first inlet, then, the operation will be executed and output immediately

[9] Interactive examples can be found at following URL: https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=01.%20basics.jspat

[10] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=02.%20ops.jspat

while receiving. For ternary operators, we are using the same implementation unless the number of arguments is three.

3.2 Conditions and Iterations[11]

Patcher systems like Max provide several ways to handle conditions and iterations because the representation of decision branches is slightly different from literal expressions. A choice from many inputs to one output, or output to a chosen branch from one input, is likely to be easier to understand as a condition testing object in a patcher system. For the iterations, a loop can be created by connecting a cable from the output of a graph to its input. Also, we provide box objects that will output all the iterated value with one outlet, and a message using another outlet while the iteration is ended, so that the rest of the program can be connected with this outlet.

For example, in Figure 4, conditions can be verified using the ternary operator or `gate` to block the dataflow. in Figure 5, 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 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`.

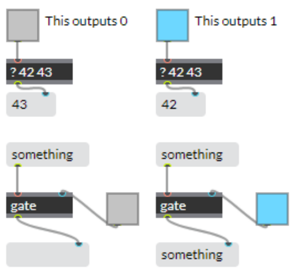


Figure 4 Conditions

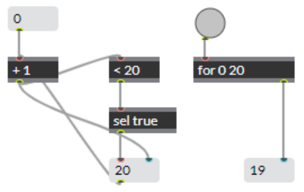


Figure 5 Loop with condition

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

3.3 Lambda functions[12]

In JSPatcher, a box object called `lambda` allows to create a JavaScript anonymous function. The function body will be a graph attached to this box, taking the box's outputs as the function's arguments, then give back to the second inlet of the box the function's return value.

When the object receives a Bang from its first inlet, it will output an anonymous function from its first outlet. 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 arguments will be 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.

For example, in Figure 6, `Array..map` represents the `Array.prototype.map` function. the first argument is the array `[1,2,3,4]`, the second is a lambda function where the function body is `* 2` which means to multiply each element in the array by 2.

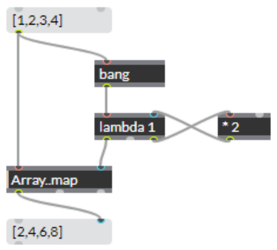


Figure 6 Lambda function

3.4 Built-ins and Web APIs[13]

When the JSPatcher is initialized, it scans recursively the global variable window and imports its content which includes most of the JavaScript built-ins and Web APIs. The imported variables, getters, setters and functions are then usable as different box objects.

Box objects with imported variables have two inlets and one outlet. A Bang from its first inlet will output the current value from its first outlet, the second inlet can be used to set the value. For example:

Box objects with property getters have one inlet for a Bang to trigger the getter and output its value from the first outlet.

Box object with property setters has one inlet that receives value to be set.

If the property has both setter and getter, the box object will behave like a variable box, have two inlets where the first serves as the getter and the second serves as the setter.

In the example (Figure 7), click on the button or the message is equivalent to execute the following JavaScript code:

```
console.log(window);
escape(",\>?");
Number.MAX_SAFE_INTEGER;
```

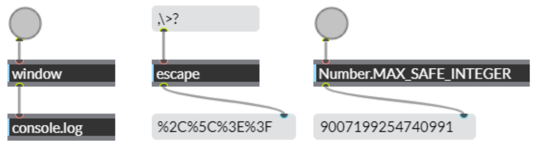


Figure 7 Imported Web APIs

For box objects with imported functions, the default number of inlets corresponds to the number of arguments of the function, in case the number is variable, users can also set the number by the `args` property. The function's argument values can be initialized from the box's arguments, and be set from the inlets. While the box receives a Bang or an argument from its first inlet, the function will be called with the arguments stored, then output the return value from its first outlet, along with the arguments after calling the function from the rest of the outlets.

[11] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=03.%20cond-loop.jspat

[12] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=04.%20lambda.jspat

[13] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=05.%20imported.jspat

https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=06.%20graph.jspat

For the box objects which are 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 construct an object from its constructor function, users can use the `new` box object, following by the identifier of the constructor's box object and the arguments. The box will evoke the `new` operator on the constructor and output the instance from the first outlet.

To get of set a specific property by name from an object is possible using `set` and `get` box object. Plus, `call` object can be used to call a specific method by name from an object.

Here (Figure 8) are two examples to build a WebAudio graph (oscillator-gain-destination) with JavaScript box objects, they are equivalent:

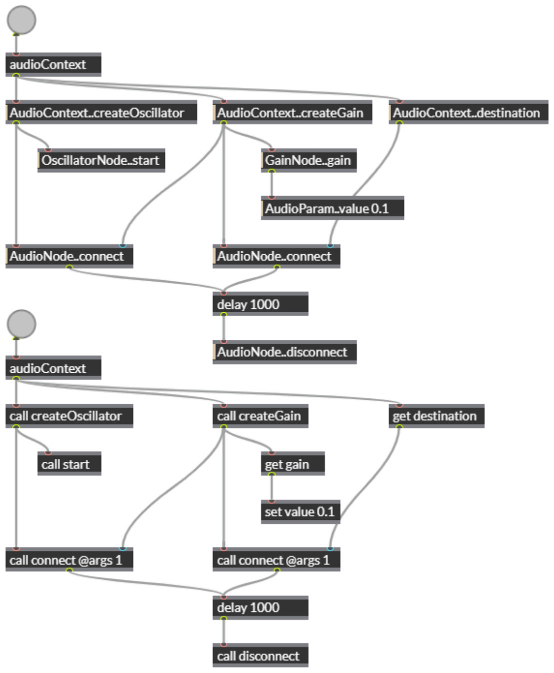


Figure 8 WebAudio graph with JavaScript boxes

3.5 External JavaScript Modules

It is common that JavaScript module creators make their work available in a CDN and can be fetched remotely. Websites like unpkg.com provide available packages on NPM, a JavaScript module registry. It is practical to get these public JavaScript modules with a CDN URL and the package identifier.

The packages on NPM are designed for Node.js, using the CommonJS module standard for import and export. The system will simulate the Node.js's environment to import these packages as box objects under a given namespace. It is also possible to import ES6 modules into the system.

A patcher can add packages with their URL and namespace as its dependencies. When JSPatcher loads a patcher, it will automatically import these packages from these URLs.

[14] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=07.%20audioworklet.jspat

4. PATCHING WEBAUDIO

4.1 WebAudio Node Box

Apart from using JavaScript box objects to build a WebAudio graph, JSPatcher provides a dedicated layer for WebAudio nodes. In this layer, each box is a representation of one WebAudio node that has node connections and its AudioParams becoming 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 while manipulated.

The layer is compatible with normal box objects and cable, the data passed through the normal cables can still be treated. For example, inlets representing AudioParams can be connected from an AudioNode as in the WebAudio API, or be connected from box objects that generate numbers to be set as the value of the AudioParam. Some customized WebAudio nodes can have their inlet for receiving MIDI messages at the same time.

One additional outlet of these WebAudio node box object outputs the instance of the AudioNode for further possible usage via JavaScript box object. For example, Figure 9 is equivalent to two examples from Figure 8.

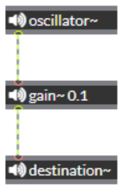
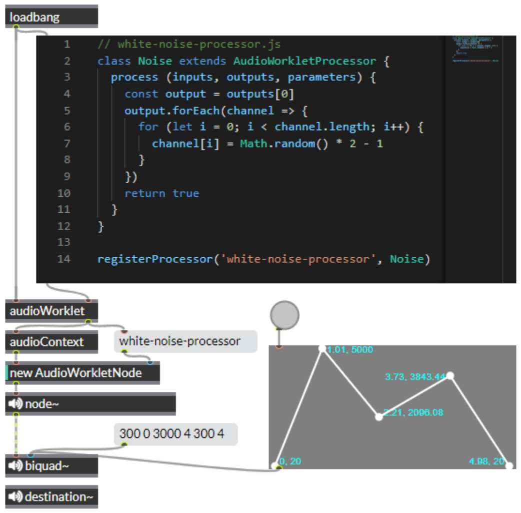


Figure 9 WebAudio graph



```
// white-noise-processor.js
class Noise extends AudioWorkletProcessor {
  process (inputs, outputs, parameters) {
    const output = outputs[0]
    output.forEach(channel => {
      for (let i = 0; i < channel.length; i++) {
        channel[i] = Math.random() * 2 - 1
      }
    })
    return true
  }
}

registerProcessor('white-noise-processor', Noise)
```

Figure 10 AudioWorklet example[14]

4.2 AudioWorklet

JSPatcher includes a set of box object that helps to code, register and use an AudioWorklet node in the patcher system. Firstly, users can add a code box to write an AudioWorklet processor with plain JavaScript code. Then the box object `audioWorklet` allows users to register the processor from the code, using internally createObjectURL. After being registered, the box will output a Bang that can be used to construct 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 Figure 10, the AudioWorkletProcessor is written in a code box, registered by the audioWorklet box. Then it's created using the AudioWorkletNode constructor, and transformed using node~ into an AudioNode box.

4.3 WebAudio Plugin Box

We provide in this layer a box object plugin~ to bring any WebAudio Plugin [8] [9] into the patcher with its UI. The box behaves like a WebAudio node box, creating automatically corresponded inlets and outlets. According to the WebAudio Plugin standard, an URL is needed to fetch from a remote server a JavaScript file that loaded its dependencies and returns an HTML element as its UI, and an AudioNode to be connected and output from the box's last outlet.

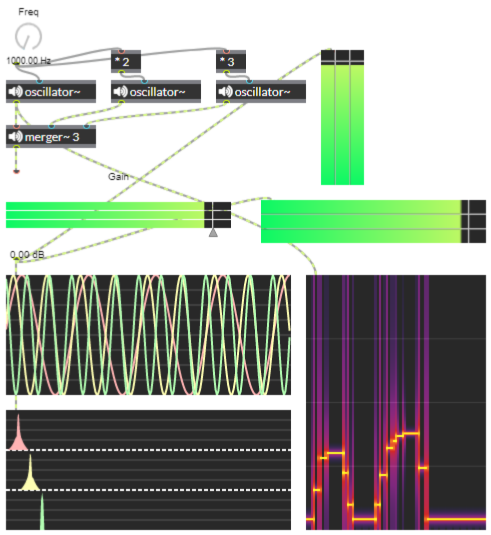


Figure 11 Visualizations

4.4 UI with WebAudio Node

Audio signal analyzers with UI visualizations are commonly used to display the features of audio streams. This can be achieved by an AudioNode which receives and analyzes the real-time signal, and HTML elements to display the result of the analysis. In JSPatcher, like the WebAudio Plugins, an analyzer with visualization can be packed in one WebAudio node box.

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

Figure 11 is an example[15] of different visualizations of three sine-wave oscillators.

4.5 AudioNode generated by FAUST[16]

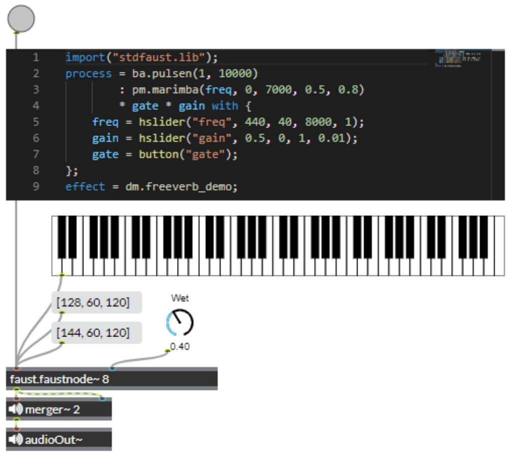
FAUST is a functional, synchronous, domain-specific programming language designed for real-time audio signal processing and synthesis.

Multiple 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 [10] which can compile FAUST code to a fully functional WebAudio AudioWorklet 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 try to compile the code and transform itself into a WebAudio node box. Like the AudioWorklet box, its AudioNode and AudioParams are connectable with other WebAudio node boxes, in addition, it handles incoming MIDI messages from its first inlet.

Figure 12 is an example to compile an eight-voice polyphonic instrument from FAUST. The instrument is handling MIDI messages from its first inlet.



```
import("stdfaust.lib");
process = ba.pulsen(1, 10000)
        : pm.marimba(freq, 0, 7000, 0.5, 0.8)
        * gate * gain with {
    freq = hslider("freq", 440, 40, 8000, 1);
    gain = hslider("gain", 0.5, 0, 1, 0.01);
    gate = button("gate");
};
effect = dm.freeverb_demo;
```

Figure 12 Faust node

5. FAUST SUBPATCHER

The design of the FAUST programming language represents its code with a patcher-like graph called block-diagram algebra (BDA) [11] [12] that can be optimized and transformed into a high-performance low-level code. The BDA acts as a middleware between the user-written code and its internal code. Using the BDA, FAUST compiler can generate a block diagram that shows the processing structure of the compiling DSP.

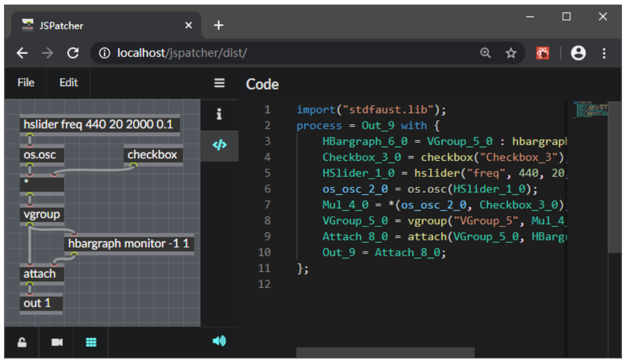
A FAUST code is therefore always represented by a graph that leaves the possibility to generate code from an equivalent graph. In JSPatcher, we designed a specific mode of patcher to build a FAUST-compatible graph, that will be firstly interpreted to an equivalent FAUST code which can be used in other FAUST tools, then be compiled to a WebAudio node using faust2webaudio. While

[15] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=08.%20meter.jspat

[16] Video demonstration on https://youtu.be/vYgqjakKYwo

patching in this mode, users have a panel that shows the interpreted code of the actual patcher in real-time. [13]

The implementation of this mode of patcher is inspired by Gen, which is also a graph-to-code system that can be compiled into a high-performance DSP.



```
import("stdfaust.lib");
process = Out_9 with {
    HBargraph_6_0 = VGroup_5_0 : hbargrapl
    Checkbox_3_0 = checkbox("Checkbox_3")
    HSlider_1_0 = hslider("freq", 440, 20
    os_osc_2_0 = os.osc(HSlider_1_0);
    Mul_4_0 = *(os_osc_2_0, Checkbox_3_0)
    VGroup_5_0 = vgroup("VGroup_5", Mul_4_
    Attach_8_0 = attach(VGroup_5_0, HBarg
    Out_9 = Attach_8_0;
};
```

Figure 13 The generated code can be previewed in on the right panel (synchronized to the patcher)

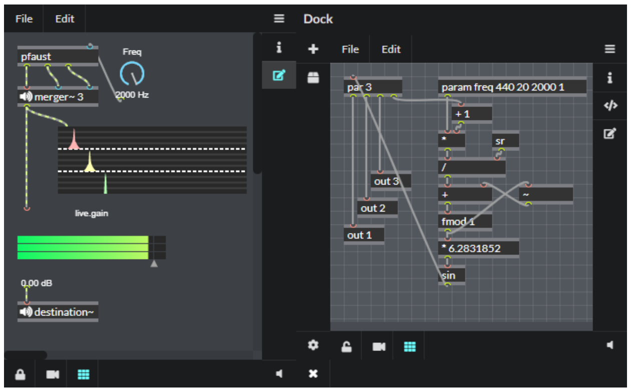


Figure 14 A FAUST patcher can be compiled to a WebAudio node box (the patcher on the right is the FAUST patcher)[17]

6. FUTURE WORK

6.1 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. As an example, OpenMusic [14] is a VPL to design computer-aided composition which provides different musical representation including score and timeline. For the timeline, a JavaScript audio library Tone.js implemented an AudioWorklet-based timing and scheduling system, which is an interesting and more accurate method for musical timing. These features could be included in the JSPatcher.

6.2 AudioWorklet Generators

Aside from FAUST, Csound [15] is also a language that can be compiled into an AudioWorklet node thanks to the WebAssembly version of the Csound compiler. [16] Using the compiler, it is possible to generate a WebAudio node box from Csound code.

Besides, it will be interesting to design an AudioWorklet's processor with JavaScript boxes in a subpatcher by having a dedicated patcher system in the audio thread. It enables the possibility to design imperative patchers to process audio buffers or event FFT data.

6.3 File System

We are working on a virtual file system in the JSPatcher so that dependencies like audio files or subpatchers can be loaded and saved to the file system. This feature can change the design of a project under JSPatcher, as it can be separated into several patchers for different roles in the project. For example, an interactive performance can have a host that sends messages to clients using WebRTC standard, in this case, the project will have a dedicated patcher for host and another for clients.

Some DSPs or synthesizers, like a sampler or a granular synthesizer, need to load audio files in advance. They can load files remotely using an URL, but it will be more efficient to get them directly from the virtual file system.

6.4 SDK

The box objects in the JSPatcher are extendable and meant to be fully accessible for developments from community contributors. We should offer a software development kit (SDK) based on these built-in box objects. With the SDK, developers can create their box object packages which can be imported from an URL into the JSPatcher.

7. CONCLUSIONS

The pros and cons of dataflow VPLs have been discussed for decades. Compared to textual languages, VPLs are more accessible and illustrative in some fields like multimedia processing, but lack clearance and performance in some complex algorithms. [17] In the design of WebAudio applications, JSPatcher is similar to some other platforms, allowing users to manipulate an audio graph and control the parameters. But we try to provide more flexibilities and potentials to JSPatcher, to design an AudioWorklet, and to gain control of other JavaScript-based web features. Developers can also write code in boxes to implement complex algorithms, then connect UI components with them. With this hybrid system where compiled and imperative patcher and code coexist, we try to overcome the disadvantages of VPLs.

Indeed, the project starts from an aspect of audio programming, but its actual implementation seems to have more use cases to us. We have experimented on the platform to program with three.js[18] OpenGL rendering, d3.js[19] data visualization, or Tensorflow.js[20] web-based neural networking as proofs of concept. Hopefully, the platform could facilitate the design of interactive multimedia projects in the future.

[17] https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=09.%20pfaust.jspat

[18] https://threejs.org/, example at https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=10.%20gl.jspat

[19] https://d3js.org/, example at https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=11.%20d3.jspat

[20] https://www.tensorflow.org/js, example at https://fr0stbyter.github.io/jspatcher/dist/?projectZip=../examples/wac.zip&file=12.%20prnn.jspat

8. ACKNOWLEDGMENTS

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

9. REFERENCES

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

[2] M. Puckette, "Pure Data," in *Proceedings of the International Computer Music Conference*, Thessaloniki, 1997.

[3] M. Puckette, "The patcher," in *Proceedings of the International Computer Music Conference*, San Francisco, United States, 1986.

[4] Y. Orlarey, D. Fober and S. Letz, "FAUST : an Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, E. D. France, Ed., 2009, p. 65–96.

[5] H. Choi, AudioWorklet: The future of web audio, Ann Arbor, MI: Michigan Publishing, University of Michigan Library, 2018.

[6] 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*, São Paulo, Brazil, 2017.

[7] S. Letz, S. Denoux, Y. Orlarey and D. Fober, "Faust audio DSP language in the Web," in *Proceedings of the Linux Audio Conference*, Mainz, 2015.

[8] M. Buffa, J. Lebrun, J. Kleimola, O. Larkin and S. Letz, "Towards an open Web Audio plugin standard," in *Companion Proceedings of the The Web Conference 2018*, 2018.

[9] M. Buffa, J. Lebrun, S. Ren, S. Letz, Y. Orlarey, R. Michon and D. Fober, "Emerging W3C APIs opened up commercial opportunities for computer music applications," in *The Web Conference 2020 DevTrack*, 2020.

[10] S. Ren, S. Letz, Y. Orlarey, R. Michon, D. Fober, M. Buffa, E. Ammari and J. Lebrun, "FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins," in *Proceedings of the Web Audio Conference*, Trondheim, 2019.

[11] Y. Orlarey, D. Fober and S. Letz, "An Algebra for Block Diagram Languages," in *Proceedings of the International Computer Music Conference*, Gothenburg, 2002.

[12] Y. Orlarey, D. Fober and S. Letz, "Syntactical and Semantical Aspects of Faust," *Soft Computing*, 2004.

[13] S. Ren, L. Pottier and M. Buffa, "From Diagram to Code: a Web-based Interactive Graph Editor for Faust DSP Design and Code Generation," in *Proceedings of the 2nd International Faust Conference*, Saint-Denis, 2020.

[14] J. Bresson, C. Agon and G. Assayag, "OpenMusic: visual programming environment for music composition, analysis and research," in *Proceedings of the 19th ACM international conference on Multimedia*, 2011.

[15] V. Lazzarini, S. Yi, J. Heintz, Ø. Brandtsegg, I. McCurdy and others, Csound: a sound and music computing system, Springer, 2016.

[16] S. Yi, V. Lazzarini and E. Costello, "WebAssembly AudioWorklet Csound," in *Proceedings of the Web Audio Conference*, Berlin, 2018.

[17] R. Stephens, "A survey of stream processing," *Acta Informatica*, vol. 34, p. 491–541, 1997.