KabelSalat: Live Coding Audio-Visual Graphs on the Web and Beyond

Felix Roos
Unaffiliated
flix91@gmail.com

Raphaël Maurice Forment
Université Jean Monnet
raphael.forment@gmail.com

ABSTRACT

This paper introduces KabelSalat, a graph-based live coding environment that targets multiple platforms and languages. It works by translating a Domain Specific Language (DSL) based on JavaScript into a signal flow graph. This graph can be compiled into a sequence of instructions optimized for real time signal processing. The compiler can either output JavaScript code to run in the browser or optimized C code to run natively. The possibility of adding other target languages is an integral part of KabelSalat's design. The browser version includes a REPL and features a range of audio DSP nodes reminiscent of modular synthesizers. Notable features include single sample feedback and multi-channel expansion inspired by the SuperCollider audio engine. The core module of KabelSalat has also been used to implement a stripped down version of the Hydra video synthesizer, thus demonstrating that the same underlying principles can be adapted both for audio and video generation. In the future, KabelSalat might become an alternative audio engine for Strudel, offering more sound design capabilities, compared to the current *SuperDough* engine, which uses the browser's built-in Web Audio graph.

1 Introduction

Graphs are often used to represent the signal flow of live coding systems, as demonstrated by Glicol (Lan and Jensenius 2021), Trane (Ash 2024), Genish.js (Roberts 2017), Hydra (Jack 2018, 2019) or Punctual (Ogborn 2018). In the broader sphere of creative software applications, the graph abstraction is a popular choice for signal flow driven applications in the audio-visual domain. Some examples are NoiseCraft (Chevalier-Boisvert 2021), cables.gl (Kombuechen 2020) and VCVRack. Graphs are sometimes perceived as more natural to the end user. They allow a direct and spatial representation of the data flow, sometimes emulating the patching of pedalboards or modular synthesizers. Many important audio programming languages from the past decades, such as Pure Data (Puckette et al. 1996) and SuperCollider (McCartney 2002) are also computing audio based on the concept of signal flow graphs. From the perspective of a software developer, graphs often allow for an optimized execution of the signal processing chain, as they can be analyzed and optimized before execution.

Audio signal processing in web browsers has historically adopted a graph based approach, as demonstrated by the Web Audio API[1]. This *dataflow paradigm* continues to gain in popularity in the context of Web Audio. The Web Audio API is based on a classic block-based processing model (Roads 1996) and relies on a set of predefined audio nodes comparable to UGens in computer music programming languages. These nodes are designed for lightweight multimedia applications rather than specialized audio and signal processing (*e.g.* basic audio filters, equalization, panning) where time and CPU usage are a critical resource. For the creative musicians, such nodes can be limiting, as they do not allow for the creation of complex audio graphs or the implementation of optimized and lightweight specialized audio algorithms. The introduction of AudioWorklets has recently opened up the possibility of performant single-sample processing in the browser (Choi 2018). They also offer a way for developers to build bespoke audio nodes. AudioWorklets are offering significant advantages over block-based processing. Single-sample processing removes the block size constraint of classic Web Audio nodes, which is often felt as both a technical and creative limitation. Computing audio in blocks can prevent or hinder the implementation of various classic algorithms: filter design, physical modeling, etc. Consequently, this can have an incidence on the sonic palette available to the musician. Despite their recent introduction, AudioWorklets have already proven their value in the implementation of audio feedback loops, granular synthesis algorithms (Roberts 2017, 2018) or physical modeling (Schaedler 2020). AudioWorklets are offering yet another advantage: they are self-contained signal processors that do not depend on web platform specifics. This enables developers to write systems that can be ported from and to the web with relative ease.

[1] The *Mozilla Developer Network* website provides a thorough introduction to the Web Audio API: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API/Using_Web_Audio_API (accessed on September 27, 2024). This API can be considered as the basic building blocks for more complex applications and libraries such as ToneJS (Mann 2015).

We began exploring the new avenues introduced by AudioWorklets because of technical and creative constraints felt during the development of Strudel (Roos and McLean 2023). Strudel's audio engine, *SuperDough*[2], is implemented using the aforementioned Web Audio API. It is built in imitation to *SuperDirt*, the classic Tidal Cycles audio engine, that relies on the extensive capabilities of the SuperCollider audio server[3]. In order to provide users with a similar level of flexibility and expressiveness in audio design, we needed to find a way to implement high-performance custom audio nodes in the browser. Our long-term goal is to be able to rewrite Strudel's current audio engine, called *SuperDough*, using a flexible and portable solution inside a single AudioWorklet. For the time being, Strudel still uses the more limited set of features provided by the Web Audio API graph.

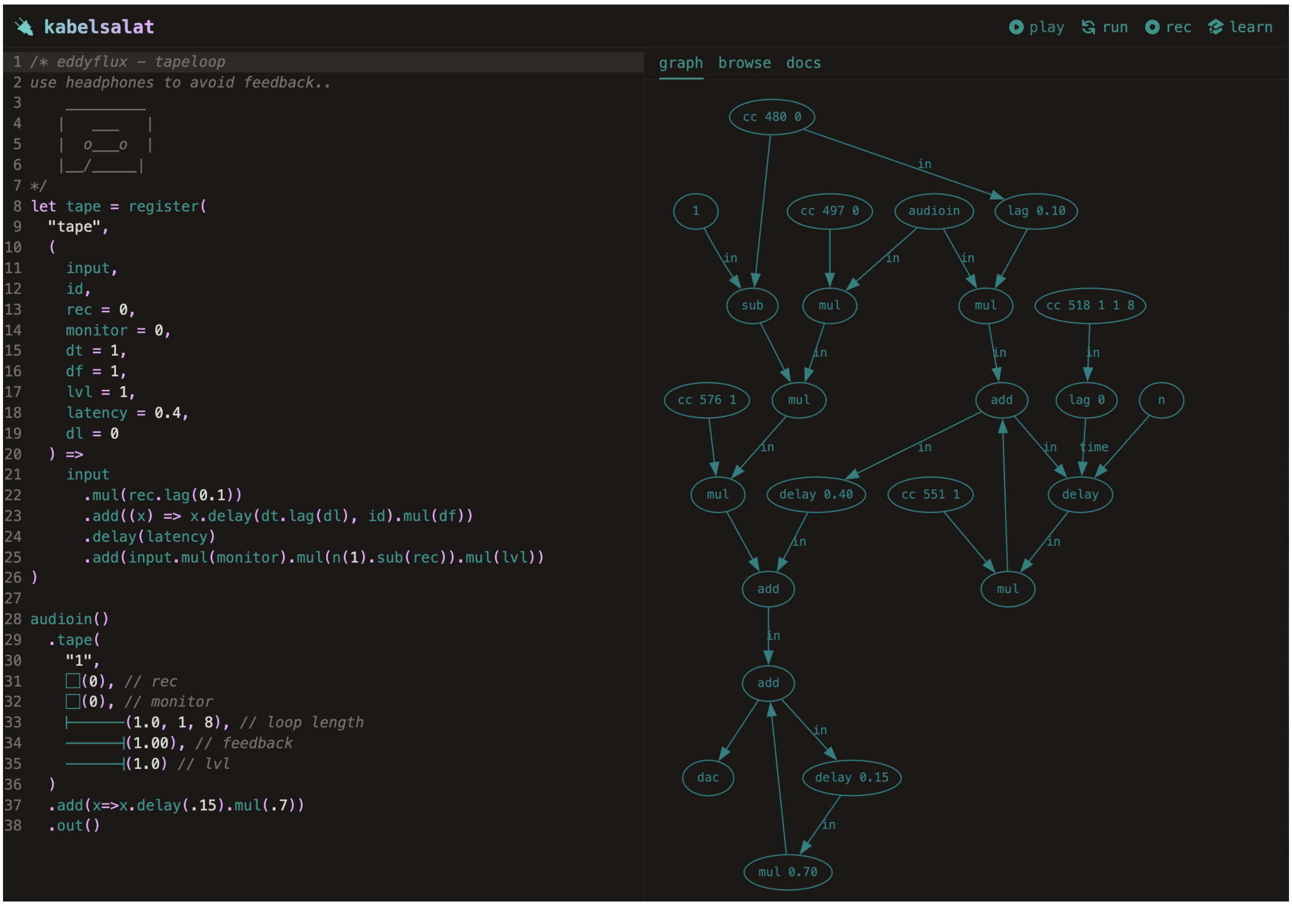

Figure 1: KabelSalat web interface (https://kabel.salat.dev, accessed on September 27, 2024) running a Karplus Strong patch. On the left pane pane: source code editor. On the right pane: audio graph visualizer.

2 Introducing KabelSalat

KabelSalat implements a Domain Specific Language (DSL) to represent and compile graphs suitable for single-sample processing. It can be used both as a prototyping bench for audio algorithms or as a DSP-oriented live coding language. The project is split into several packages, allowing KabelSalat to be embedded in other applications. It can also be used as a standalone application through the online REPL (Read Evaluate Print Loop, see Figure 1). The development of KabelSalat started as an experiment, trying to use the browser-based NoiseCraft synthesizer (Chevalier-Boisvert 2021) as an audio backend for a newly developed JavaScript DSL. Instead of relying on the graphical node patching interface of NoiseCraft, the DSL was developed to be able to express similar graphs textually. Since then, NoiseCraft's code has been altered and extended in many ways. One example being its compiler, which has been reimplemented to be agnostic to its output language. With this addition, graphs can be compiled to multiple languages without changing the logic of the compiler. To test the viability of this design, we have developed a way for KabelSalat patches to be compiled to C code and played as standalone binaries. As another proof-of-concept, KabelSalat was used to compile Hydra patches to GLSL code[4], showing an application of the same concepts in another neighbouring domain.

[2] Link to *SuperDough* on the *npm* package manager: https://www.npmjs.com/package/superdough (accessed on September 27, 2024).
[3] Link to the *SuperDirt* repository on GitHub: https://github.com/musikinformatik/SuperDirt (*idem*).
[4] Link to *hydro* repository on Github: https://github.com/felixroos/hydro (*idem*).

2.1 First example: a subtractive synthesizer patch

KabelSalat is based on a terse and practical syntax that relies heavily on method chaining. It is directly inspired by the syntax of Hydra (Jack 2018) and Strudel (Roos and McLean 2023)[5]. Chaining methods can be seen as a way to emulate the patch point connections between the different modules of a synthesizer. Both functions and methods can be considered as signal generators or processors (nodes), which can be connected to other modules through chaining or reference. Arguments of these nodes can either be constant values or other nodes. The same basic principles can be used for creating patches of arbitrary depth and complexity. The following code example shows how a classic subtractive synthesizer patch can be written in KabelSalat. Figure 2 is the visual representation of the generated audio graph, which has been automatically generated in the KabelSalat REPL.

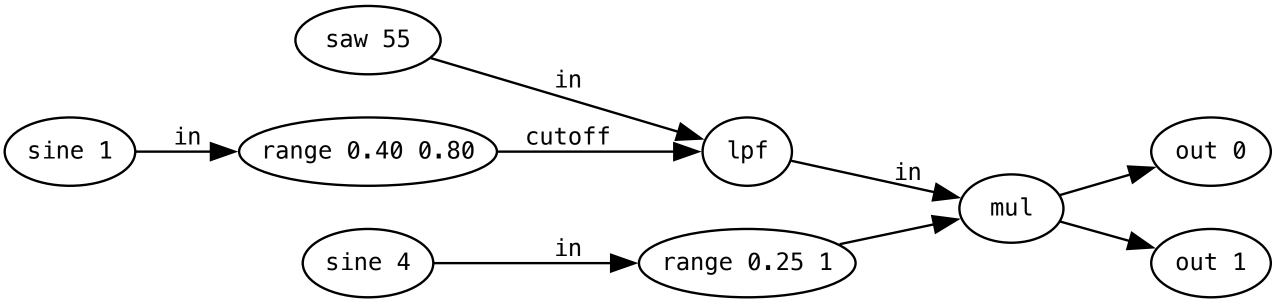

Figure 2: Visual representation of the same audio graph, generated in real-time using GraphViz.

```
// sawtooth wave at 55Hz:
saw(55)
  // modulated low-pass-filter
  .lpf(sine(1).range(0.4, 0.8))
  // modulated amplitude:
  .mul(sine(4).range(0.25, 1))
  // send to audio output:
  .out();
```

2.2 Multichannel Expansion

KabelSalat borrows the concept of multichannel expansion from SuperCollider[6], allowing the duplication of a node or a chain of nodes to multiple channels. Large audio graphs involving parallel processing can thus be generated with relatively few characters. Multichannel expansion is based on providing function/method arguments as Arrays (*e.g.* `[1, 2, 3, 4]`).

```
// creating two channels of filtered sawtooth waves
saw([200, 300]).lpf(0.5).out([0, 1]);
```

Multichannel expansion in KabelSalat involves the use of a special `poly` node. Array arguments are converted to `poly` nodes automatically, where each Array element becomes an input. When a node receives a `poly` node with `n` inputs, `n` copies of the node are created. Each copy receives one of the values in the Array. The copied nodes are fed into a new `poly` node, which is propagated down the graph. The `poly` node will eventually end up at the bottom of the graph, where each channel is assigned to one `out` node. In cases where a node receives multiple `poly` nodes, the `poly` node with the most inputs determines the number of copies. The inputs of the other `poly` nodes wrap around. Figure 3 shows a graphical version of how the `poly` node is propagated in the above example.

2.3 Feedback

Feedback loops play an important role in digital audio synthesis, allowing the creation of comb filters, feedback delay networks (FDN), reverbs, flanger effects, among many other applications (Smith 2010; Roberts 2017). Feedback is also

[5] Strudel used the same technique applied to a different domain: the functional composition of musical patterns. Hydra uses it, similarly to KabelSalat, as a way to connect and combine video processing nodes.

[6] This feature is documented in the official SuperCollider documentation: https://doc.sccode.org/Guides/Multichannel-Expansion.html (accessed on September 28, 2024).

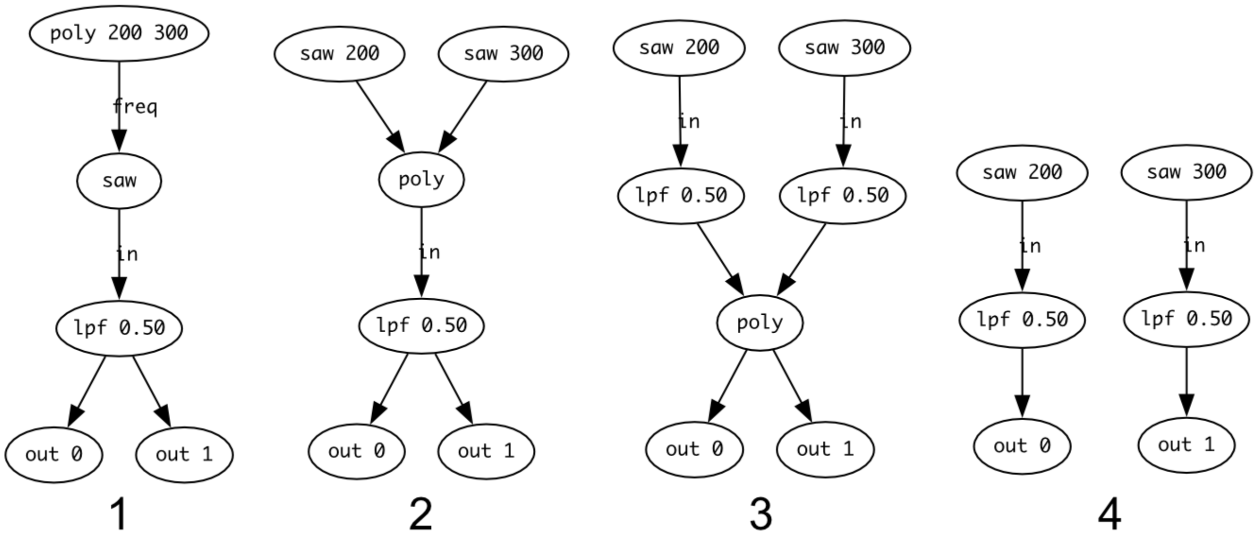

Figure 3: Multichannel Expansion Example

an important technique in the realm of video synthesis[7]. A feedback loop is created in a graph when a node uses its own output as an input. KabelSalat supports two techniques to create such loops: through anonymous functions or through a special source (`src`) node.

2.3.1 Feedback loop using anonymous functions

Passing an anonymous function to a node can be used to create a feedback cycle. In the following example, the `add` node receives an anonymous function as its input. To close the feedback cycle, the anonymous function receives its own output as an argument. Any alteration or further processing done to the feedback line can be notated inside the function (*e.g.* amplitude modulation, introducing a delay). Figure 4 illustrates the representation of a feedback loop in the graph visualizer.

```
impulse(1)
  .add((x) => x.delay(0.1).mul(0.8))
  .out();
```

2.3.2 Feedback loop using the source (`src`) Node

Feedback can also be created using the dedicated source (`src`) node (Figure 4). This syntax is inspired by Hydra (Jack 2018), where feedback is created using the similarly named `src` and `out` nodes. This notation will not create a cycle in the graph. Instead, the `src` node references the corresponding output register, which contains the output of the previous sample.

```
impulse(1).add(src(0).delay(0.1).mul(0.8)).out();
```

[7]See for instance: https://andreijaycreativecoding.com/getting_started-with-video-feedback (accessed September 28, 2024). Feedback loops are also a popular feature of the well-known Hydra video synthesizer used by many live coders during Algoraves: https://hydra.ojack.xyz/docs/ (*idem*)

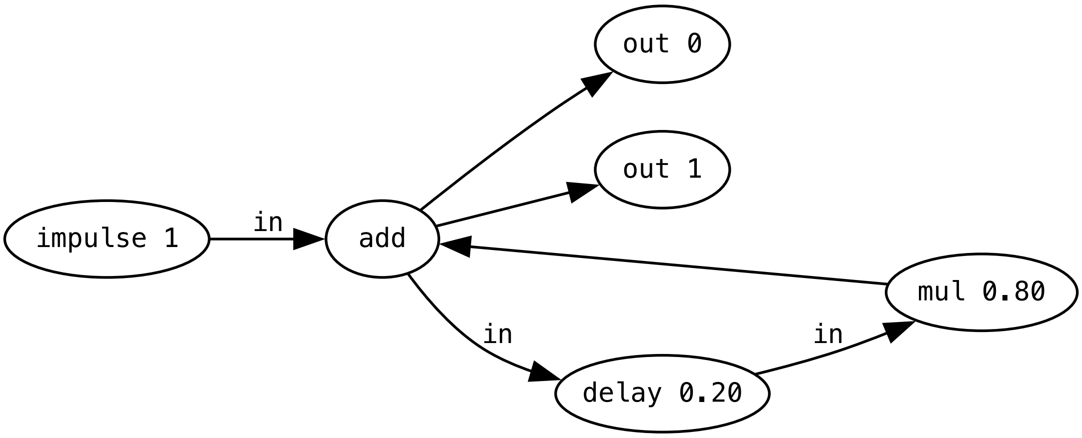

Figure 4: Representation of Figure 7 feedback loop on the graph visualizer.

3 Audio Graph Compilation

KabelSalat audio graphs are compiled into a representation optimized to run efficiently within the constraints of a real-time system. For the sake of demonstration, we are going to focus on the JavaScript output, but similar principles apply to the C or the GLSL targets.

3.1 First step: from DSL to Graph

In the KabelSalat DSL, each expression starts with a function call that represents a source node. Such a function call returns an instance of `Node`, which contains a method for each node type. Methods internally call the function of the same name, passing the `Node` they are called on as the first argument. This means that each expression using method chaining has an equivalent variant with function calls only[8]. It depends on the context and taste of the user to choose the appropriate way to write a certain expression. As a TypeScript interface, the structure of a `Node` can be described as:

```
interface Node {
  type: string;
  ins: Array<Node | number>;
}
```

Each `Node` has a type and an Array of inputs called `ins`. Elements inside `ins` are either other instances of `Node` or constant numeric values. A typical KabelSalat patch will create one or more deeply nested nodes of type `out`. Each `out` node represents the whole signal graph that is connected to one channel of audio. For example, the `Node` instance representing a filtered sawtooth wave, might look as follows:

```
{
  "type": "out",
  "ins": [{ "type": "lpf", "ins": [{ "type": "saw", "ins": [200] }, 0.5] }, 0]
}
```

Note that the above data is represented as JSON only for the purpose of readability. The actual implementation uses JavaScript Objects, where each `Node` is only referenced, meaning reused `Node` instances will not be copied. For cyclical graphs, a JSON representation does not exist, because it would create an infinite loop.

[8]For example, `saw(200).lpf(.5).out()` is equivalent to `out(lpf(saw(200), .5))`.

3.2 Second step: from Graph to Output Language

To generate efficient runtime code, the graph is converted into a sequence of steps. The processing of each `Node` corresponds to one step of the generated code. Before compilation, `Nodes` are sorted topologically[9], making sure each `Node`'s inputs are computed first. In an AudioWorklet, the resulting imperative code will run once for each sample at the given sample rate, typically at 44.100kHz or 48kHz. The generated code expects some variable to be defined beforehand by the compiler: - `nodes`: instances of stateful nodes (ADD DETAILS, WHAT DOES IT MEAN) - `r`: node value registers (SAME) - `o`: output channel registers (SAME)

As a demonstration, the compiler output for the graph of FIGURE ????? is as follows:

```
r[1] = nodes[0].update(200); /* saw */
r[3] = nodes[1].update(r[1], 0.5, 0); /* lpf */
o[0] = r[3]; /* out 0 */
```

3.2.1 Stateful Nodes

The `nodes` Array contains instances of stateful signal processors, which are expected to be provided to the compiled function. Stateful nodes are essential for many audio DSP techniques, for example to keep track of the phase of an oscillator while its frequency is being modulated. Each audio processor, defined as a class, needs to implement an `update` method to compute the next sample based on its input arguments. Figure XX demonstrates the implementation of the same sawtooth oscillator written in JavaScript and C.

```
class SawOsc {
  constructor() {
    this.phase = 0;
  }
  update(freq) {
    this.phase += SAMPLE_TIME * freq;
    return (this.phase % 1) * 2 - 1;
  }
}
```

In the C language, a similar pattern can be implemented with an update function operating on a struct. In GLSL, nodes are stateless due to the parallel nature of graphics rendering.

3.2.2 Value Registers

The `r` Array contains the latest output values of each `Node`. When a graph contains cycles, the node that receives the feedback depends on a node that has not been calculated yet. By saving each `Node`'s result into the `r` Array, those nodes will automatically receive the value from the previous iteration. To illustrate this point, here is the compiled output of Figure 3:

```
r[1] = nodes[0].update(1); /* impulse */
r[3] = nodes[1].update(r[6], 0.2); /* delay */
r[5] = r[3] * 0.8; /* mul */
r[6] = r[1] + r[5]; /* add */
o[1] = r[6]; /* out 1 */
o[0] = r[6]; /* out 0 */
```

In Line 2, `r[6]` references the value of the previous iteration, making feedback possible.

3.2.3 Output Registers

The `o` Array keeps track of each output channel. After each iteration of the compiled sequence, `o[0]` and `o[1]` can be passed to the sound card for playback. The `out` function of the DSL takes a channel as its only argument, which falls back to `[0,1]`. This ensures both stereo channels receive a value by default.

[9] The topological sort is implemented using a simple Depth First Search algorithm (DFS).

3.3 Node Compilation

To encapsulate the compiler logic from the output language, each node definition contains a `compile` function that is expected to output its target language. The compiler's sole responsibility is to pass the correct register names and constant values to the compile function. An `impulse` node could be defined to output C code as:

```
let saw = registerNode("impulse", {
  ugen: "ImpulseOsc",
  compile: ({ vars: [freq = 0], name, node, ugen }) =>
    `${name} = ${ugen}_update(${node},${freq}); /* ${ugen} */`,
});
```

In comparison to the JavaScript version, the C version of Figure 3 is:

```
r[1] = ImpulseOsc_update(nodes[0],1); /* ImpulseOsc */
r[3] = Delay_update(nodes[1],r[6],0.2); /* Delay */
r[5] = r[3] * 0.8;
r[6] = r[1] + r[5];
o[0] = r[6]; /* out 0 */
o[1] = r[6]; /* out 1 */
```

4 Runtime

The purpose of the runtime is to create an environment where the compiled code can be executed. This runtime should also handle code update and hot-reloading, which is an essential feature for a live coding system. The Web Audio runtime of KabelSalat is located in an AudioWorklet that communicates with the rest of the application via a MessagePort (Roberts 2018). After a graph is compiled, its code, along with some metadata is sent to the worklet. Inside the worklet, a `Unit` is spawned, which contains a unit generator for each stateful node. The main processing loop of the AudioWorklet sums all spawned `Unit`'s to calculate the final mix. When the code is updated, a new `Unit` is spawned and a crossfade between the old and new `Unit` is performed. This avoids cracks in the audio due to sudden amplitude jumps. Similar to the JITLIB library in the SuperCollider ecosystem (Rohrhuber and De Campo 2011), KabelSalat allows to adjust the fade time of the crossfade.

In the GLSL version, fades are not necessary. The worst case in the visual domain is a flash from a light to a dark color. Instead, a new shader program is created and swapped with the old one when the code is updated. At the time of this writing, the runtime of the C version only supports running a single graph without the ability to update, which is not yet enough for live coding.

4.1 Real Time Input

The Web Audio version of KabelSalat supports both Audio and MIDI Input (through the Web MIDI API[10]). These iniputs allow direct integration with the code through a microphone, through MIDI Controllers and/or in-source UI elements. As a result, KabelSalat can be used as a synthesis-oriented companion tool for various live coding setups, allowing the live coding of synthesizers and audio treatments on-the-fly.

5 Future Outlook

In the future, further steps will be taken in the direction of becoming an event based audio engine. So far, it is not possible to schedule sample accurate events from the outside, which is a requirement of Strudel's audio engine. Additionally, Tidal patterns (McLean and Wiggins 2010; McLean 2014) might be combined with an audio graph in a different way, by using Patterns as inputs for individual nodes, rather than composing entire expression to a single pattern. In the current version, nodes are not reused from Unit to Unit, meaning the node state will reset on each update. This leads to sequences being reset as well, which is often undesirable. Finding nodes that can be kept across evaluations would be possible by employing a diffing algorithm between the old and the new graph. Furthermore, the handling of Unit's could be extended to allow evaluating graphs in a block based fashion, where multiple Unit's can coexist in parallel. On the web, potential performance gains could be achieved by compiling to WebAssembly instead of JavaScript (Robert 2022).

10

6 REPL

KabelSalat's website[11] hosts the latest version of the Web Audio compiler of KabelSalat. It can be used as a way to experiment, share patches and live code without any audio interruption. It consists of a code editor (1), a graph visualizer (2), example patches (3) and an interactive documentation (4). Similar to the Strudel REPL (Roos and McLean 2023), the code editor supports in-source UI elements, such as buttons and sliders. The URL always reflects the latest code change, allowing patches to be shared as a hyperlink.

7 License

All code is open source under the AGPL-3.0 License ("GNU Affero General Public License - GNU Project - Free Software Foundation — Gnu.org"). Contributions are welcome and can be made through the GitHub repository[12].

8 Acknowledgements

Thanks to the Strudel and wider Tidal, live coding, WebAudio and free/open source software communities for inspiration and support. Special thanks to Maxime Chevalier-Boisvert for creating NoiseCraft, which was the starting point of the project. Special thanks to pulu for being an early adopter, creating mesmerizing patches and providing valuable feedback and contributions. Special thanks to Raphaël Maurice Forment for being a good conversation partner during the journey of the implementation of KabelSalat, eventually joining this paper as a second author, helping with proof-reading, phrasing and further research.

[11] Website link: https://kabel.salat.dev/ (accessed on September 27, 2024).
[12] GitHub repository link: https://github.com/felixroos/KabelSalat (*idem*).

References

Ash, George. 2024. "Trane: Musical Janet on the Web." In *Proceedings of the 12th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, 30–35. FARM 2024. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3677996.3678285.

Chevalier-Boisvert, Maxime. 2021. "NoiseCraft." https://github.com/maximecb/noisecraft.

Choi, Hongchan. 2018. "Audioworklet: The Future of Web Audio." In *International Conference on Mathematics and Computing*. https://api.semanticscholar.org/CorpusID:69755393.

"GNU Affero General Public License - GNU Project - Free Software Foundation — Gnu.org." https://www.gnu.org/licenses/agpl-3.0.html.

Jack, Olivia. 2018. "Hydra." https://github.com/ojack/hydra.

———. 2019. "Hydra: Live Coding Networked Visuals." In *Proceedings of the Fourth International Conference on Live Coding*, 353–54. https://github.com/iclc/iclc.github.io/tree/master/2019/proceedingsICLC2019.pdf.zip.

Kombuechen, Thomas. 2020. "Cables.gl." https://github.com/maximecb/noisecraft.

Lan, Qichao, and Alexander Refsum Jensenius. 2021. "Glicol: A Graph-Oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet." In *Proceedings of the International Web Audio Conference*, edited by Luis Joglar-Ongay, Xavier Serra, Frederic Font, Philip Tovstogan, Ariane Stolfi, Albin A. Correya, Antonio Ramires, Dmitry Bogdanov, Angel Faraldo, and Xavier Favory. WAC '21. Barcelona, Spain: UPF.

Mann, Yotam. 2015. "Interactive Music with Tone.js." In *Proceedings of the International Web Audio Conference*, edited by Samuel Goldszmidt, Norbert Schnell, Victor Saiz, and Benjamin Matuszewski. WAC '15. Paris, France: IRCAM.

McCartney, James. 2002. "Rethinking the Computer Music Language: SuperCollider." *Computer Music Journal* 26 (4): 61–68. https://doi.org/10.1162/014892602320991383.

McLean, Alex. 2014. "Making Programming Languages to Dance to: Live Coding with Tidal." In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Functional Art, Music, Modeling & Design*, 63–70. FARM '14. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/2633638.2633647.

McLean, Alex, and Geraint Wiggins. 2010. "Tidal–Pattern Language for the Live Coding of Music." In *Proceedings of the 7th Sound and Music Computing Conference*, 331–34. https://doi.org/10.5281/zenodo.849841.

Ogborn, David. 2018. "Punctual." https://github.com/dktr0/Punctual.

Puckette, Miller et al. 1996. "Pure Data: Another Integrated Computer Music Environment." *Proceedings of the Second Intercollege Computer Music Concerts*, 37–41.

Roads, Curtis. 1996. *The Computer Music Tutorial*. The MIT Press. London, England: MIT Press.

Robert, Charles. 2022. "Dynamic Per-Sample Processing with WebAssembly." In *Proceedings of the International Web Audio Conference*. Cannes, France: Université côte d'Azur. https://doi.org/10.5281/zenodo.6767578.

Roberts, Charles. 2017. "Strategies for Per-Sample Processing of Audio Graphs in the Browser." In *Proceedings of the International Web Audio Conference*, edited by Florian Thalmann and Sebastian Ewert. WAC '17. London, United Kingdom: Queen Mary University of London.

———. 2018. "Metaprogramming Strategies for AudioWorklets." In *Proceedings of the International Web Audio Conference*, edited by Jan Monschke, Christoph Guttandin, Norbert Schnell, Thomas Jenkinson, and Jack Schaedler. WAC '18. Berlin, Germany: TU Berlin.

Rohrhuber, Julian, and Alberto De Campo. 2011. "Just-in-Time Programming." In *The SuperCollider Book*, edited by Scott Wilson, David Cottle, and Nick Collins, 207–37. London, England: The MIT Press.

Roos, Felix, and Alex McLean. 2023. *Strudel: Live Coding Patterns on the Web*. Zenodo. https://doi.org/10.5281/zenodo.7842142.

Schaedler, Jack. 2020. "Karplus-Stress-Tester." https://github.com/jackschaedler/karplus-stress-tester.

Smith, Julius O. 2010. *Physical Audio Signal Processing : For Virtual Musical Instruments and Audio Effects*. https://ccrma.stanford.edu/ jos/pasp/.