Vesa Norilo
and Alejandro Olarte
Sibelius Academy
University of the Arts Helsinki
PO Box 30, 00097 Taideyliopisto, Finland
{vno11100, alejandro.olarte}@uniarts.fi

A Visual Programming Interface for Digital Luthiery: Implementing Circuits with Veneer

Abstract: This article presents a method for programming musical signal-processing circuits visually, using expressive idioms and abstractions from functional programming. Special attention is paid to the creative workflow, framing the education in a constructionist context. Our aim is to empower musicians in signal processing: The claim was tested in a university workshop for relatively inexperienced programmers. The participants were able to study and implement signal-processing algorithms from literature and integrate them into their preexisting workflow, and appeared to gain self-confidence while doing so.

Musical signal processing has evolved in recent years and decades. The field has been active with innovation, with professional instrument makers in both academia and industry being joined by makers, musicians, and autodidacts.

Novel interfaces, abstractions, and products have been developed in the search for new expressive power. The range and diversity of digital instruments has never been greater or more accessible to practitioners. The fundamentals of signal processing have, at the same time, become obscured, abstracted away from the direct relationship between classical synthesizer controls and the topology of its sound generation machinery.

Programming music is also easier than ever. There is a wealth of free study material online, and domain-specific languages abound. Being domain languages, however, they are defined as much by what they do not try to express as that which they do. The field has an extensive history in expressing scores, orchestras, and instruments, but tackling the fundamentals of signal processing is a newer development. Many music languages are based on a set of built-in unit generators, whose inner workings are deemed out of scope.

This article explores a novel method of delving through that abstractive floor in musical signal processing, fashioned as a visual interface, Veneer, to a functional programming language, Kronos, which deals with elementary signal processing. It is capable of expressing both fundamental circuits and sophisticated abstraction, and it aims to empower musicians to build digital instruments and deploy them in a variety of ways.

Computer Music Journal, 44:4, pp. 8–25, Winter 2020
doi:10.1162/COMJ_a_00578

We discuss the pedagogical thinking upon which the software and teaching methods are based, and report findings from a university-level workshop on music-technology programming, with self-assessment by the learners before and after four hands-on sessions.

Teaching and Practicing Musical Signal Processing

The past six decades have seen a wealth of programming environments dedicated to music. These systems support the composition of musical pieces and the development of music software; and they provide instruments for live performance and scientific research (Lazzarini 2013).

Time plays a special role in music. Roger Dannenberg (2018) finds dealing with time to be a central feature of music languages. Time in music manifests itself on different scales, spanning the arch of macrostructure, rhythm, phrasing, down to the oscillation of an audible waveform. The separation of concerns in programming music is traditionally split into the analogous layers of score, instrument, and unit generator (Roads 1996, pp. 787–796).

Victor Lazzarini (2013) states that musical composition in the implicit context of the western tradition was the primary motivation and use case for early music languages. In this setting the programmer is associated with the composer, the program with the score, and the synthesizer with the orchestra of musicians. The analogy is simplistic and not intended to describe actual artistic practice, but serves as a point of departure for reflecting on digital musicianship.

Practitioners keep blurring the lines between instrument builders, composers, and performers, and creative control is desired across timescales and domains (McPherson and Tahiroğlu 2020, pp. 55–56). With increasing computational power, details of audio synthesis have become an increasingly integral aspect of music programming, expanding its focus towards the microscopic time scale of digital samples—programming of timbre and interpretation.

McPherson and Tahiroğlu (2020) find that music languages employ rhetoric that emphasizes absence of any stylistic or aesthetic bias, but that this may not be accurate in practice. Stylistic and aesthetic choices result not only from limitations, but also from what is deemed idiomatic in a language—a path of least resistance to implementation.

As such, many prominent music languages struggle to express the internal unit generators (ugens), the fundamental signal-processing algorithms that underlie our performances and repertoire. Languages in which a clear separation exists between built-in ugen library and user-defined ugens tend to focus on the construction of instruments from ugens, which can run into an impenetrable barrier when the desired algorithm cannot be expressed in terms of the built-in ugens (Brandt 2002; Nishino, Osaka, and Nakatsu 2013).

A Music Language for Signal Processing

The project under discussion addresses a specific ambition in music programming: The development of an approachable visual tool that provides the ability for music practitioners to develop production-grade signal-processing programs, suitable for integration into their artistic practice, while building and internalizing abstractions that both facilitate growth as a programmer and make creation easier. From a base of a sufficiently powerful signal-processing language, we look into developing visually oriented, pedagogically considered tools that aim to elucidate the inner workings of familiar but often opaque ugens and digital instruments.

We have described the presence or absence of certain traits related to these goals in established music languages in Table 1. In some cases we consider a specific subset or mode of a music language that is geared towards ugens and sample-level processing, such as ChucK Chugens (Salazar and Wang 2012), Csound user-defined opcodes (Lazzarini et al. 2016) and Max gen∼. The table is not intended to be exhaustive or indicative of the relative merit of various music languages, but rather their flavor of signal processing and capabilities relevant to the stated goal. Most of the included environments are open-source projects, with the exception of Max. This we included owing to its ubiquity. Reaktor, by Native Instruments, is another example of a commercial program offering programming capabilities, including sample-level signal processing.

The features noted in the columns of the table are as follows: (1) visual interface indicates that the environment presents a spatial, diagram-like programming interface rather than a sequence of instructions; (2) unit-delay feedback is the capability to specify feedback paths with one-sample delay, necessary for programming elementary ugens; (3) multirate DSP shows whether user programs can operate precise clocks that differ from the audio clock but remain synchronized with it, such as control rates or oversampled signal paths; (4) discrete events indicate the ability to respond to and schedule specific responses to one-shot events, such as MIDI notes, in contrast to regular sampled audio streams; (5) algorithmic circuits refer to any means of composing complex circuits from simple circuits (e.g., loops and control flow, programmatic instantiation, or higher-order functions); and (6) optimizing compiler is used as an umbrella term for generation of native code, including techniques like interprocedural optimization, indicating that user programs can reach a high level of computational performance.

Veneer and Kronos are purpose-built to complete this exact set of features, each of which is present in several other environments, but never all at once. In the subsequent sections, we elaborate on some of the language design trade-offs these traits engender.

Table 1. Selected DSP Features in Music Languages

Language	*Visual interface*	*Unit-delay feedback*	*Multirate DSP*	*Discrete events*	*Algorithmic circuits*	*Optimizing compiler*
ChucK (Chugen)		*		√	√	
Csound (UDO)		**			√	
Faust		√	‡		†	√
Max	√	*		√	†	
Max (with gen~)	√	√				√
Nyquist			√	√	√	
Pure Data	√	*		√		
PWGL	√	*			†	
SuperCollider		*		√	√	
Veneer/Kronos	√	√	√	††	√	√
xtlang		√		√	√	‡‡

*Available when block size is set to 1, with significant performance degradation.
**Control rate must be set to audio rate for the user-defined opcode.
‡Experimental research feature.
††Can receive events but not schedule new events.
†Special construct for parallel expansion.
‡‡Just-in-Time only, optimization limited by hot-swapping design.

Unit-Delay Feedback for Elementary DSP

The capability of expressing unit-delay feedback paths is essential for programming signal processors. Music languages typically schedule and route blocks of samples rather than individual samples for reasons of efficiency (Dannenberg and Thompson 1997). Several established environments have support for per-sample processing, however, and thus unit-delay feedback.

PWGLSynth is a real-time synthesizer for the PWGL environment (Laurson, Norilo, and Kuuskankare 2005). Built primarily for auralizing scores with physics-based instrument models, it has a per-sample scheduled ugen graph.

Csound has gained support for user-defined opcodes (Lazzarini et al. 2016) that call a user-defined routine at the control rate (krate). Because Csound allows locally specified control rates, a user-defined opcode can be made to run at the audio rate and support unit-delay feedback.

ChucK (Wang, Cook, and Salazar 2015) offers the possibility of processing audio in a "chugen," a unit generator with a custom callback for producing a sample of audio.

These environments support unit-delay feedback, but pay a significant price in performance when doing so in an interpreter or a virtual machine (Norilo 2015). As such they should suffice for exploration and study of signal processing algorithms, but may run into problems if a larger number of instances is required in a musical piece.

Diversity of Scheduling and Expression

Music languages use different strategies for dealing with time. Regularly sampled signal flows are utilized for audio and control signals, which require determinism and precise synchronization. Many music languages provide a built-in set of clock domains such as block rate, control rate, and audio rate. Csound (Lazzarini et al. 2016) allows instruments to specify local control rates. Max and Pure Data (Puckette 1996) offer asynchronous message passing for flexible scheduling of low-bandwidth signals. ChucK features concurrent interacting threads with explicit control of time (Wang and Cook 2003). Xtlang, Nyquist, and Csound share the highly general scheduling mechanism of temporal recursion

(Lazzarini et al. 2016; Dannenberg 1991; Sorensen and Gardner 2010).

Many signal-processing tasks benefit from multirate processing. Some components require high-bandwidth processing, such as audio or event over-sampled audio. Control circuits, such as envelopes and low-frequency oscillators, can run at much lower bandwidth for significant efficiency gains. In the feature matrix we denote Kronos, Faust, and Nyquist as having first-class, computationally efficient support for this type of processing (Dannenberg 1997; Norilo 2015; Orlarey and Jouvelot 2016). Further, custom multirate schedulers could be implemented by the user in any language that can express imperative control flow, such as Csound or xtlang (Sorensen and Gardner 2010; Lazzarini et al. 2016).

Compiled Languages

The conventional solution to improving the performance of a program written in a high-level language is to reimplement it in a lower-level language with an optimizing compiler. This approach is supported by many music languages, including all those discussed in the previous section; new ugens can be implemented in C or C++. It is not ideal for an environment that is meant to scale from introductory to advanced use; many end users never make the transition to yet another programming language and paradigm.

Recently, an increasing number of music languages have attempted to break the efficiency barrier and support signal processing from first principles without relying on external, general-purpose systems languages. To do this, they integrate an optimizing compiler. The open code-generation framework, Low-Level Virtual Machine (LLVM cf. Lattner and Adve 2004) has made such projects feasible. Our recent work on Kronos (Norilo 2015) is one such attempt.

Faust (Orlarey, Fober, and Letz 2004) is a high-level functional signal-processing language for producing highly efficient static circuits from block-diagram algebra expressions. Faust programs are expressive but terse, and the syntax presents a challenge to many learners.

Xtlang is a new language that grew out of the Impromptu and Extempore projects, aimed at audio DSP and efficient numeric code (Sorensen and Gardner 2017). It has runtime semantics similar to C, with type inference, Lisp-like syntax, and macros. It is a general-purpose systems programming language, rather than specifically tied to music and audio, and may be as difficult to master as traditional systems languages.

Max, by Cycling '74, is likely the most popular music language. Current versions include gen~, a sublanguage for signal processing. Despite using the same visual interface, gen~ is a completely different language with synchronous evaluation, overloaded operators, and a distinct type system. As such, patch fragments, algorithms, and skills do not necessarily transfer from standard Max to gen~ or vice versa.

Visual Languages

Depicting programs visually instead of textually is a common theme in end-user programming. Brad Myers (1986) suggests that the spatial capabilities of the human brain can be better utilized in the visual domain. Ivan Sutherland (1964) developed Sketchpad, an interactive visual programming environment with elements of direct manipulation, as early as 1964. Later, Hypercard enabled many nonprogrammers to make applications (Nielsen, Frehr, and Nymand 1991).

Object-oriented visual programming was attempted by Fabrik (Ingalls et al. 1988), but especially in music, dataflow languages proved more suitable for visual representation as "patches," named after the way modular synthesizers could be "programmed" by connecting modules with cords. In addition to Max, Pure Data is a primary example. Other examples include OpenMusic (Assayag et al. 1999) and PWGL, which pioneered musical scores integrated into dataflow patches (Kuuskankare and Laurson 2009; Laurson, Kuuskankare, and Norilo 2009).

Teaching Programming

Engaging students in the learning process of a programming language involves several challenges: how to arouse curiosity to actively participate in the discovery, how to maintain the necessary motivation to persist while overcoming the obstacles related to syntax, how to integrate new knowledge in learners' personal workflow, and how to properly support the individual and group learning experience for the appropriation of a new programming language. These challenges are well known to teachers in many areas beyond computer music and demand both careful planning and a variety of approaches to the learning situation.

Programming is fundamentally about abstraction (Blackwell 2002). To learn programming, one needs to acquire and learn to deploy abstraction to solve problems. Maximal solutions are highly regarded. Instead of just focusing on the concrete problem at hand, programmers are taught to reason deductively about the problem space, seeking to solve a category or a class of similar problems that are isomorphic from a particular abstractive viewpoint. According to Brooks and Brooks (1999, p. 104), constructivist education frames the learning of tasks with closely aligned words, such as "classify," "analyze," "predict," and "create."

The natural tendency of constructivist thought is to progress from the concrete to the abstract. In the context of digital pedagogy, Seymour Papert (1993) disagrees on a fundamental level. Although computer music may seem abstract and detached to some, its ultimate form is typically the concrete, visceral experience of sound. As such we symphatize with Papert's argument that concrete thinking deserves "deeper respect."

Papert (1987) suggests that discovery learning can be guided with microworlds, playgrounds with a well-designed, limited set of tools for exploration—not unlike educationally purposed interactive patches. In the case of music programs, "tweaking" program constants and listening for changes in sound is a good, intuitive strategy, somewhat in opposition to the deductive approach and to the lionization of abstractive generalization.

Even though Papertian constructionism and discovery learning is well regarded in digital pedagogy—and this regard is shared by us—it is not universally accepted. Dave Catlin (2019, pp. 13–16) provides a recent overview of the arguments for and against.

Luthiery for the Digital Age

Our research into programming tools that could be used to teach and practice musical signal processing, using the patching metaphor with which many musician-programmers are comfortable, has led to the development of Kronos, a declarative metaprogramming language for signal processors (Norilo 2015), and Veneer, a web-based visual programming front end (Norilo 2019). They build upon the history of innovation in both signal processing and visual programming.

The Faust project (Orlarey, Fober, and Letz 2004) has demonstrated the viability of the functional programming paradigm in producing efficient static circuits. The core principle is that of zero (run-time) overhead abstraction, in which expressive programs are translated into static, deterministic, fast-running signal processors. Time nearly disappears in the realm of static circuits. The programs are fashioned as topologies instead of chronologies. Instead, time manifests itself as delay and memory operators built into the language.

Functional programming is also aligned with dataflow programming—a prominent feature of visual environments. The Faust syntax presents some challenges to visual representation, however. Its block-diagram algebra is powerful, but describes the construction of signal circuits indirectly via function composition. This may well be suitable for a visual representation, but one that is far removed from the familiar signal-flow diagrams.

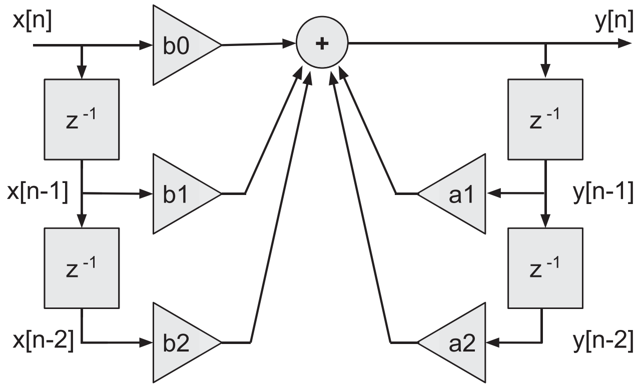
Visual Syntax

To combine familiar dataflow patches with functional signal processing, we have designed Kronos to exhibit isomorphic syntax in both text and patch

Figure 1. Textbook diagram of a biquadratic ("biquad") filter representing a second-order filter using a direct form II topology.

Figure 2. Biquad filter in xtlang.

Figure 3. Biquad filter in Faust.



```
(bind-func static biquad
  (lambda ()
    (let* ((y1 0.0)
           (y2 0.0)
           (x1 0.0)
           (x2 0.0)
      (lambda (x b0 b1 b2 a1 a2)
        (let ((y (+ (* b0 x)
                    (* b1 x1)
                    (* b2 x2)
                    (* a1 y1)
                    (* a2 y2))))
          (set! y2 y1)
          (set! y1 y)
          (set! x2 x1)
          (set! x1 x)
          y))))))
```

Figure 2.

```
biquad(a1,a2,b0,b1,b2) = + ~ conv2(a1,a2) :
  conv3(b0,b1,b2)
with {
  conv3(k0,k1,k2,x) = k0*x + k1*x' + k2*x" ;
  conv2(k0,k1,x) = k0*x + k1*x' ;
};
```

Figure 3.

form (Norilo 2015, 2019). The nodes in the Veneer interface are not tied to particular "built-ins" or ugens, but are rather defined as arbitrary expressions in the Kronos language. This affords power and flexibility for advanced users, while the user interface steers beginners towards the familiar model of one node per operation.

Consider the learner who looks to implement a biquadratic filter, more commonly known by the informal term "biquad." A typical textbook diagram is shown in Figure 1. The diagram consists of feedforward and feedback unit delays, additions, and multiplications, and its working principle is relatively easy to grasp.

An implementation in xtlang (Sorensen and Gardner 2010) is shown in Figure 2. The listing shows a constructor function that returns a closure. The closure is object-like, and captures variables from the constructor that are used as state for the delay memories. In a mainstream object-oriented language, the implementation would be very similar. The captured variables would be instance variables, and the programmer would similarly think about ordering of side-effects while plumbing samples through delay memory and lifetimes of the various program entities.

Faust (Orlarey, Fober, and Letz 2004) is more restricted to the signal-processing domain, and thus affords a higher level of abstraction without sacrificing efficiency. The program shown in Figure 3 shows the recursive composition of two short convolution kernels. There is certainly elegance in this approach, but the gap to bridge between textbooks and this idiom remains daunting. Nor does it lend itself to interactive visualization—although the Faust compiler generates dataflow diagrams from source code, it is not clear if actually editing Faust programs as diagrams is feasible or even desirable.

Figures 4 and 5 show the biquad filter in Max 8 and Veneer, respectively. Both examples can be thought of as executable diagrams in the sense that they very closely resemble the textbook diagram in Figure 1—indeed, translating simple flow diagrams to programs is an ideal application for a visual language. Gen~ makes use of implicit summation, and chooses to call unit delay "history." Veneer uses DSP vernacular, "z^{-1}", but other than that, the closer resemblance between the Veneer patch and the textbook diagram is largely due to the fact

Figure 4. Biquad filter in Max gen~.

Figure 5. Biquad filter in Veneer.

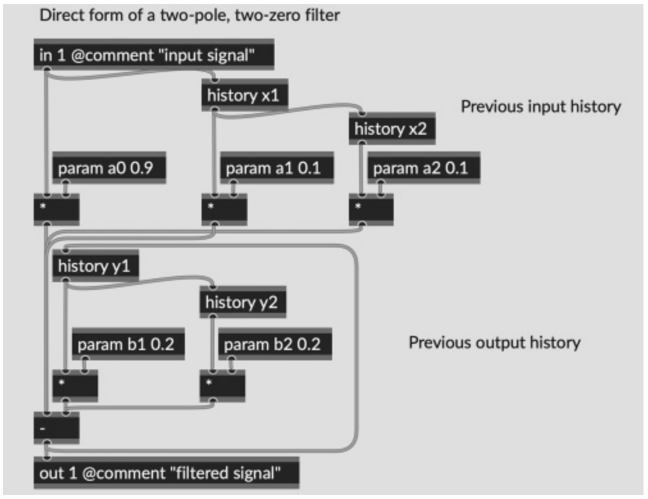


Figure 4.

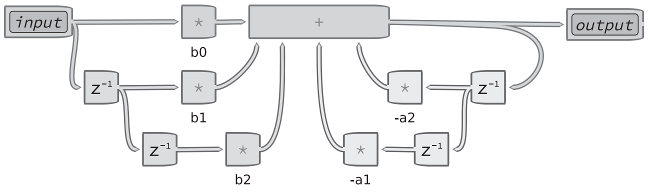


Figure 5.

that it is a straightforward transcription. Please note that the Max implementation is the only one in this section that does not invert the feedback coefficients.

Functional Abstraction in the Visual Domain

Veneer explores the design space of visual programming, with the goal of studying higher-level abstraction in the context of signal processing and its impact on learning. The representation of straightforward dataflow diagrams is well established, as is evident in gen~, but there is more ambiguity in visualizing more-abstract programs that contain conditionals or loops.

Much of this problem stems from diverging data flow and control flow. In a pure dataflow program, the order of computation is solely determined by data dependencies. Imperative loops, on the other hand, define program order explicitly.

How to present explicit program order and control flow visually? Max supports a form of control flow in the form of active inlets and bang messages, but both are notably divergent and missing from both the old-style buffered audio graph and the newer sample-level gen~. Patch cords in Max serve two different purposes: to relay data, and to transfer program control. This polysemy—a coupled data-and-control flow—is a common source of confusion and makes the programming model harder to understand.

Kronos and Veneer opt for a pure dataflow model with no explicit control flow. In the absence of loops and conditionals, program constructs that involve repetition must be different. Suitable solutions are abundant in the functional programming tradition (Hudak 1989), based on *higher-order functions*.

A higher-order function is a function that receives another function as a parameter. One use for this arrangement is to decouple traversal of a data structure from a transformation applied to its elements. Some well-known higher-order functions that implement traversals are incorporated into Kronos, including (1) *Map*, which applies a single-argument parameter function to all members of a parameter set, and (2) *Reduce*, which threads members of a parameter set through a two-argument parameter function from left to right.

Applied to signal processing, Map generates parallel routing, such as a filter bank, and Reduce encodes serial routing, such as summing or cascade circuits. More generally, higher-order functions in signal processing are circuit generators that construct algorithmic compositions of simpler circuits (Norilo 2015).

Additive Synthesis with Higher-Order Functions

We will now present the approximation of a downward sawtooth waveform via additive synthesis as an example of using higher-order functions to compose signal-processing circuits. The

Figure 6. Example of higher-order functions in Veneer. Count produces a list of harmonic numbers, and Map applies the Sin function to members of this list. The higher-order function Zip-With applies a binary function (division) to pairs of elements from each list, and Reduce sums the harmonics generated by Zip-With.

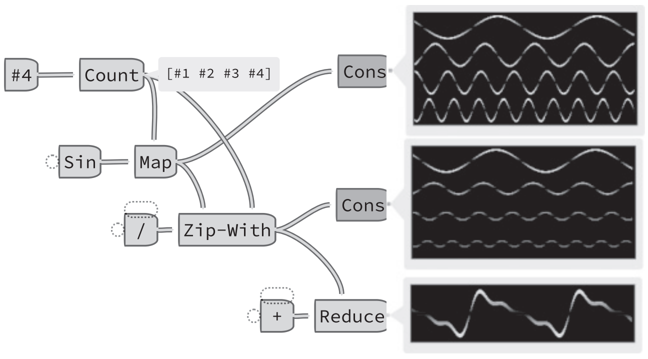
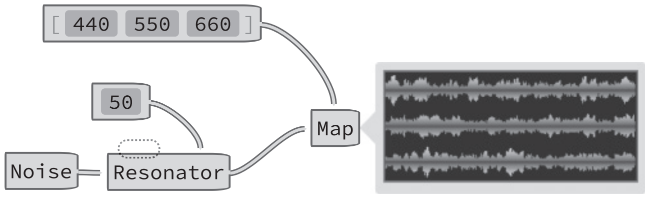


Figure 7. Partial application of the Resonator function by a signal source (Noise) and bandwidth (50), resulting in a unary function of frequency. Map applies this function to a list of frequencies to generate multiple channels of resonant noise.



approximation with N partials is described by the following equation:

$$f_{saw}(\theta) = \sum_{k=1}^{N} \frac{cos(k\theta)}{k}. \qquad (1)$$

The corresponding Veneer patch for $N = 4$ is shown in Figure 6. Count is used to produce a list of harmonic numbers from 1 to 4. Map applies Gen:Sin to members of this list, producing one sinusoidal partial for each.

To achieve appropriate harmonic weights to approximate the sawtooth wave, each partial n must have an amplitude of $1/n$. The higher-order function Zip-With applies a binary function pairwise to each element of the two lists, and is used here to divide each partial waveform by its corresponding harmonic number. Finally, summation is performed with Reduce, which recursively combines the first two elements of a list until just one element remains. Notably, this patch generalizes to any constant value of N, the input to Count, and produces the appropriate number of elementary circuits to match.

Each of the nodes that represent a parameter function that is passed to a higher-order function has disconnected inputs, empty slots indicated by dotted circles. Veneer considers disconnected nodes as unapplied functions. Unapplied function nodes output a stream of function-typed values—verbs—rather than the usual concretely typed stream of nouns. Function-typed streams provide the parameter functions to a higher-order function.

Visual Representation of Partial Application

To make higher-order functions ergonomical, functional languages often make it easy to specify anonymous ad hoc functions. Veneer allows arbitrary textual expressions in any node, but the visually oriented mechanism for creating ad hoc functions is *partial application*. As disconnected nodes produce unapplied functions, partially connected nodes produce partially applied functions.

Figure 7 demonstrates a fan-out circuit built with partial application and Map. The parameter function here is Resonator, which has three inputs for signal, frequency, and bandwidth, respectively. We have connected the signal and bandwidth inputs, resulting in a partially applied resonator with the frequency input disconnected. From the perspective of Map, this looks like a single-argument function, which—applied by Map to the list of frequencies 440, 550, and 660—produces three resonant noise signals.

For further details, please refer to our prior work (Norilo 2019).

Higher-Order Functions and Beginners

It is reasonable to question whether the proposed use of higher-order functions is really compatible with teaching signal processing to nonprogrammers, as the programs quickly become much more abstract. Higher-order functions are not commonly used in the context of signal processing. Although the dataflow remains visually consistent, the increased complexity of the data types themselves presents a challenge.

The consistency of the dataflow model is a pedagogical boon, however. Understanding data types of increasing complexity can be developed incrementally, without need of special rules for evaluation and execution of special forms, or for keywords in the language. Someone comfortable in the imperative programming idiom easily forgets how baffling mutable state and control flow can be. Although functional abstraction is certainly not trivial, it can avoid much of the imperative baggage—we hypothesize that it is a ladder of abstraction that affords a more logical progression in the context of visual dataflow programs.

Partially applied nodes are a gentle nudge towards discovering the concept of lambda abstraction—conversion of a concrete program fragment into a parametrized, more generally useful function (Norilo 2019).

Students often independently discover the desire and need to replicate parts of their program in an algorithmic manner, and are thus primed to internalize the rationale for higher-order functions. Further, the knowledge and concepts are highly transferable to other languages and domains besides computer music.

Exploratory Programming and Rapid Development

Veneer aims for exploratory programming and rapid development. We believe that these workflows support the learner in generating frictionless mental models of a program (Blackwell 2002), and that in addition to learning, such mental models are conductive to fostering creativity and spontaneous expression.

Instant feedback and the ability to see and manipulate program flow in real time are key to developing this mental model. The read-evaluate-print loop (REPL) is a well-known example: It is an interactive prompt into which the user may type expressions to be evaluated and see the results printed.

Debuggers and REPLs are some of the interactive tools of inspection and interrogation of programs and their state. More-recent advances include deeper integration between the programming tool and the inspection tool (Hancock 2003). Such an approach is a natural fit to visual languages: Max and Pure Data programs are routinely interspersed with numeric and graphical displays of program state. Similarly, the integration of control widgets, such as sliders or dials, into the source of the program is a valuable aid in helping programmers explore the parameter space of their creation.

Veneer aims to make both the display of data and the introduction of configurable parameters as frictionless as possible. A real-time output display bubble can be popped out at any point of the patch with a single keystroke, and any constant numbers in the program code can be turned into tweakable parameters by simply dragging them with the mouse. Additionally, sample-level programming is greatly assisted by Veneer's ability to stop or slow down the audio clock by a factor of 256—allowing the programmer to observe samples one by one as they flow through the dataflow graph.

Deployment

Part of the barrier to learning programming is the friction of installing and setting up the environment. By deploying Veneer as a web application, we are able to avoid the requirement for installing a program, similar to several earlier projects (Michon and Orlarey 2012; Roberts and Kuchera-Morin 2012; Lazzarini et al. 2014). The web platform is not without trade-offs, however, especially in the case of an audio-intensive application. The various strategies for targeting audio applications to the web have recently been discussed by Yi and Letz (2020).

Veneer makes use of two recent developments, WebAssembly and AudioWorklets, to approach near-native audio performance in the browser. The Kronos compiler has been compiled to WebAssembly, and also enhanced to generate executable DSP units as WebAssembly "blobs." This allows the real-time audio processing to sidestep Javascript garbage collection and to use a compact bytecode format.

AudioWorklet allows custom audio-processing code on the web platform to run in the real audio thread, no longer contesting the application main thread. Before AudioWorklets, audio processing had to be cooperatively scheduled in the application's main user-interface thread, which meant that robust low-latency operation was technically unattainable. Mozilla has recently produced an implementation of AudioWorklet in Firefox, thus the technology is no longer exclusive to Google's Chrome browser. As of this writing, Apple's Safari browser also supports AudioWorklets in its technology preview release, boding well for the adoption of the standard.

The web platform is thus gradually reaching towards parity with native applications, but a wide gap remains to be closed. Thread priorities and CPU numeric modes are constrained by browser security and semantics, and WebAssembly performance is still below that of native code. Use cases such as multichannel input and output work sporadically. For these reasons, Veneer ships with a WebAssembly backend for maximum accessibility, but can also connect to a locally running native compute server for best performance.

Interfacing with the World

The role of tools is essential in defining digital musicianship and creative identity (Partti 2014, pp. 12–13). As such it is important for music software, especially programming languages, to be designed so that they can be readily assembled with other components and interoperate in various ways to accommodate even unanticipated workflows and creative strategies.

An interesting effect of the compiler-based strategy is that the executable artifacts that result from user programs can be made independent of the host language. This is in contrast to environments that rely on a large interpreter or a runtime component. Kronos generates artifacts that interface natively via the C calling convention and require only that the target platform be supported by the LLVM framework. The dependency-free executables are relatively easy to port to new platforms, such as WebAssembly—essentially a virtual bare-metal platform—or as extensions to other environments, similarly to how Faust produces artifacts for a wide range of audio software (Fober, Orlarey, and Letz 2011).

Veneer Workshop

We undertook a hands-on workshop with undergraduate-level music technology students to evaluate the software and method described in this article. The workshop was conducted at the University of the Arts Helsinki from February to March 2020. It consisted of four three-hour sessions with students who had registered for a course on SuperCollider (McCartney 2002).

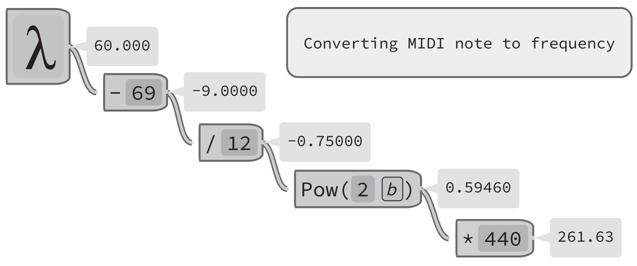
Methodology

Music technology programs tend to have a small student body. The biannual intake at the University of the Arts is between six and ten, which means that the active student population is under 30 for the five-and-a-half-year program. We secured twelve registrants to the workshop, which is probably near the practical upper bound at our institution. Regardless, as a sample size for statistical methods it is small. Kölling and McKay (2016, pp. 2–4) describe this and other challenges in a formal study of programming environments. Considering these factors, we decided against a control group. Direct evidence of superiority or inferiority of a programming environment may be unobtainable in any case.

In teaching art, the end goals are open-ended. There are few skill requirements set in stone; rather, studies should support and nurture the creative expression of the individual. This should be reflected in how we evaluate systems built to support such learning, including the present project. We designed a battery of questions meant to measure the levels of confidence and motivation learners felt in a range of specific aspects of audio programming. These questions are a crude measure of empowerment felt by the participants, and increased empowerment is our ideal outcome.

The participants returned two electronic surveys, one before and one after the hands-on sessions. The multiple-choice "empowerment" question battery was identical in both surveys. In addition, the prior survey contained an interest profile, and the posterior survey solicited feedback on the software

Figure 8. Live interrogation of patch dataflow, including real-time display of intermediate results from the computation pipeline.



and workshop with both multiple-choice and free-form questions.

Workshop Content

To start with, the workshop deals with the basic usage of the Veneer patcher. Novel features were introduced early, such as the use of higher-order functions in DSP, as in Figure 6, and real-time interrogation of dataflows, as in Figure 8.

The rest of the workshop was devoted to exercises and case studies. The culmination was a simplistic modeling of the well-known Minimoog synthesizer. We built a band-limited sawtooth oscillator using the differentiated parabolic-wave algorithm (Valimaki et al. 2009), as well as a model of one stage of the ladder filter. This stage includes a first-order filter and soft saturation. A synthesis voice, in which these components are included as abstracted functions, is shown in Figure 9. We then studied a simple step sequencer connected to a polyphonic instantation of the synthesizer. Further, we studied Schroeder reverberation (2002), feedback-delay networks (Rocchesso 1997), and granular synthesis.

A significant motivator for the learners was the mechanism to export Veneer patches as native extensions for SuperCollider. This capability was provided to the web platform by a server-side compilation service, similar to the Faust online compiler by Michon and Orlarey (2012).

In our efforts to apply Papert's paradigm (1993) to abstraction in musical programming, we have worked with the concepts of inductive and gradual abstraction (Norilo 2019). We believe musical programming tasks should focus on objects that make a sound and can be manipulated. Direct manipulation is one of the first casualties of abstraction (Blackwell 2002).

By starting with concrete values for things like frequencies, amplitudes, and bandwidths, patches can be more easily understood. Abstraction can subsequently be introduced gradually: By replacing constants with lambda terms, programs are parametrized; they are abstracted inductively. Veneer supports this programming style by allowing automatic extraction of a subpatch as a new reusable function, and the frictionless transformation of constants into interactive controls.

In practice, the workshop was taught in three broad categories or modes, which we discuss in the subsequent subsections.

Here Is the Patch

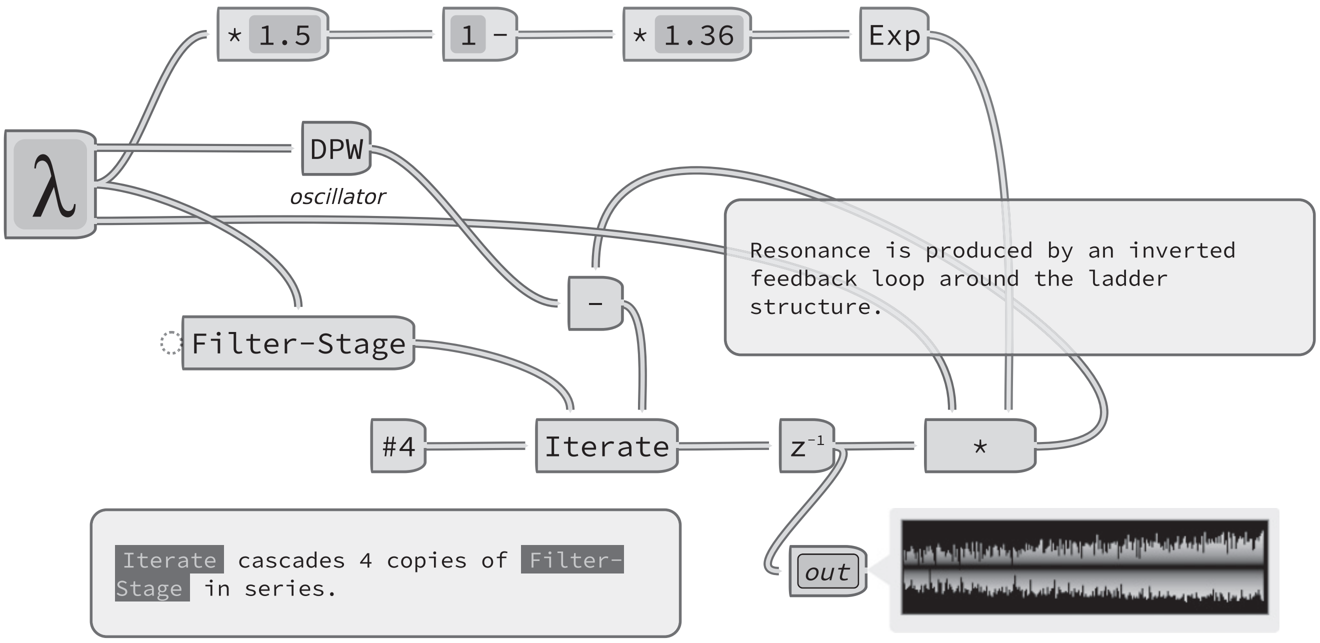
Dismantling a working circuit or a well-coded example aligns with deconstructionist learning theory. Learners are exposed to the essential components and relationships in the signal processing circuit while reviewing the terminology, syntax, and conceptual framework of the patch (Griffin 2019). A ready-made educational patch is a good example of a microworld (Papert 1987), a delimited space that allows learners to explore. This was the mode of presentation for the Minimoog example discussed previously.

Watch Me Do, Hear Me Think

By observing an experienced programmer—the teacher—stating a goal and implementing it, explicitly communicating the thinking process, learners are exposed to to a well-developed workflow, but also to the general strategy of problem solving. This approach allows the teacher to underline what is novel in the language in question and how it relates to other languages the learners may already be familiar with.

For example, while building reverberators from primitive delay lines, we simultaneously exhibit the theory of digital reverberation as well as its expression with higher-order functions. Verbalizing the process helps learners to identify patterns and categories, and to engage in building

Figure 9. Minimoog-inspired virtual analog synthesiser. A differentiated parabolic wave oscillator is fed into the ladder filter, which is generated by repeating the Filter-Stage function four times with the Iterate function. Resonance is provided by a unit-delay feedback path around the entire ladder circuit.



Resonance is produced by an inverted feedback loop around the ladder structure.

Iterate cascades 4 copies of Filter-Stage in series.

more-complex programs and abstraction—the "procedural thinking" necessary to accomplish tasks (Selby 2015).

Recreate and Transcribe

While independently recreating a patch or an transcribing an implementation from literature, the focus is on the learner's own thinking and acting. There is a lot of ground for discovery, experimentation, knowledge manipulation, and application. Errors may be common, but are secondary to design principles and strategies. They may also lead to "free experimentation," "creatively failing," and "serendipitous" discoveries.

Constructing the knowledge of the environment is done by manipulating and directly interacting with the concepts and objects: not only the software environment, but also the block diagrams, literature, and sketches that "surround and embellish the code" (Noss and Clayson 2015). For example, as students independently implemented synthesis with feedback amplitude modulation (Kleimola et al. 2011), a series of creative answers emerged from building, reflecting, and debugging as a process of "knowledge appropriation" (Papert 1980).

Survey Results

The workshop results were assessed with web surveys the participants returned prior to the first session and after the last session. Twelve participants answered the first survey (A), and nine participants answered the second survey (B). The answers were collected with Google Forms.

Quantitative

Survey A contained a section on the interests of the participants, shown in Figure 10. The bars indicate the mean of the responses, with standard error of the mean shown as a range. The participants reported interest in a wide range of topics and activities broadly related to DSP. They generally reported most interest in artistic activities, followed by academic topics. Some ambivalence was shown towards engineering and design.

Figure 10. Learner interest profile. The right-hand ends of the thick bars indicate the mean values, with the standard error indicated by the range at the end.

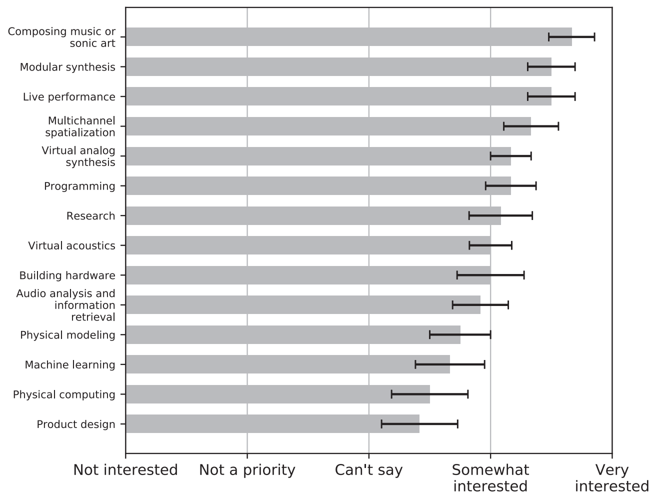
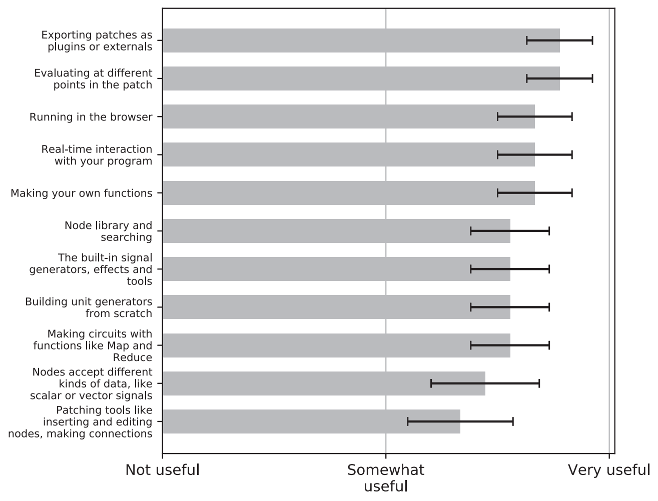


Figure 11. Usefulness of Veneer features. The right-hand ends of the thick bars indicate the mean values, with the standard error indicated by the range at the end.



We solicited feedback on Veneer in two sections of Survey B. The participants were asked to assess the usefulness of various features, and to compare Veneer to other software environments with which they had prior familiarity.

Usefulness

Generally, the participants viewed Veneer favorably. Given the amount of personal interaction between us during the workshop, and the fact that they knew they were responding to the author of the software, empathy and a certain level of courtesy may have created a positive bias. Further, we do not know the reactions of those who only attended the first session and did not return survey B. Perhaps they only wanted a preview, or perhaps they reacted negatively.

Given these caveats, we believe the results can at best indicate which features of Veneer and Kronos the respondents found the most and the least impressive.

Responses to the questions on the usefulness of various features are shown in Figure 11, sorted in descending order. The bars indicate the mean of responses, with standard error of the mean shown as a range. The respondents could indicate the usefulness of each feature, or that they did not understand what it was about. In quantifying the answers, we categorized the latter together with the response "not useful." The rationale for this is that people who feel they are not qualified to comment on the usefulness of a feature might gravitate to the middle of the scale, even though a feature they do not understand is quite literally not useful to them.

Respondents were most convinced of the usefulness of exporting Veneer patches as native extensions, which is probably influenced by the fact that they were studying SuperCollider. The interactivity features, such as patch interrogation and real-time interaction were also highly regarded.

The users were least impressed with the basic patching tools, which may indicate usability problems. Generics also ranked low; the topic may be too complicated for an introductory workshop. Notably, the deviation was high for these questions. Bugs specific to the browser version might contribute to usability problems, and prior programming background may be a factor in appreciating generics.

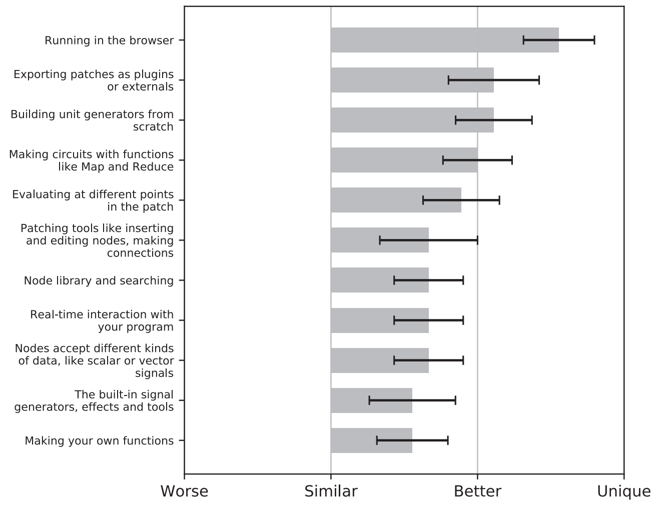
Comparison to Other Environments

We also asked the respondents to compare the features listed in the preceding section with those of other environments with which they had prior experience. The mean of the responses, and standard error, are shown in Figure 12.

Figure 12. Comparison with other environments. The right-hand ends of the thick bars indicate the mean values, with the standard error indicated by the range at the end.

Figure 13. Learners' self-confidence and motivation before and after the workshop.

Figure 14. Effect sizes for prior versus posterior survey.



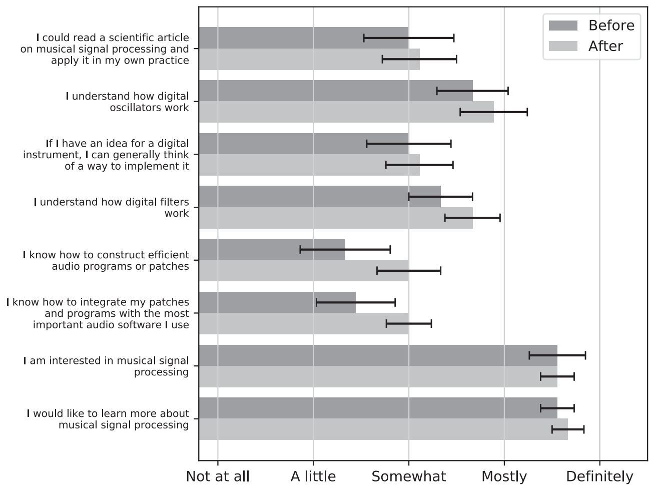


Figure 13.

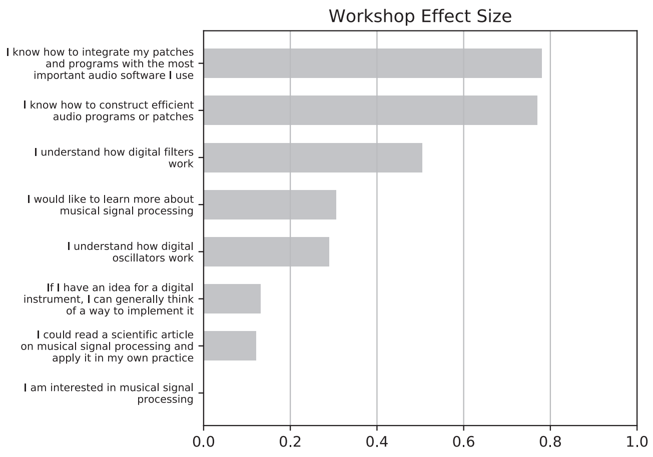


Figure 14.

"Running in the browser" was clearly the most significant feature in this section. The participants may be unaware of prior work (Michon and Orlarey 2012; Roberts and Kuchera-Morin 2012; Lazzarini et al. 2014). Other highly rated features included building ugens from scratch and exporting them to other environments, as well as algorithmic circuits and patch interrogation.

The built-in ugen library was among the lowest-rated features. The Kronos built-in library is less extensive than most other environments because of the emphasis on composability and ground-up construction. The fact that the feature was still rated better than "similar to other environments" may result from a positive bias in the responses. The deviation for all questions in this section indicates a diversity of experience and opinion in this small sample.

Impact of the Workshop on Learner Self-Confidence

The primary goal of the Kronos project is to empower and enable musicians and music technologists in the development of tools for their specific artistic needs and purposes. To assess whether the experimental workshop helped the learners to take steps in that direction, we asked questions about learners' self-confidence and motivation in both surveys. The mean values to the responses, and their standard errors, both before and after the workshop, are shown in Figure 13.

All self-assessment means were stable or improved during the workshop. Effect sizes according to Cohen's *d*-formula are shown in Figure 14. Because of the small and self-selected sample, we do not make any claims of statistical significance. With this caveat, we observe a large effect in "integrating patches with the most important audio software I use" and "construction of efficient audio programs." A medium effect is shown in "understanding how digital filters work."

It is somewhat surprising that self-confidence in constructing efficient programs was among the largest effects. Optimization was not a major theme in the workshop. Several participants expressed surprise at how little CPU time the exported patches consumed, however. Also, the questions with the largest effect size also had the lowest initial baseline confidence. Apparently the learners had little experience of constructing efficient DSP units and integrating them, and they found that it was easier than expected with the presented method.

For many learners, the workshop was the first time they had constructed elementary digital filters from unit delays, accumulators, and gain coefficients. Most of them were able to implement filters by directly translating textbooks diagrams to Veneer patches, which may have made the topic appear more accessible.

The course seemed to have the greatest impact on those with the lowest baseline confidence. The Pearson correlation coefficient of initial confidence versus confidence improvement is −0.8, indicating a strong inverse correlation. This is an encouraging, if not unexpected, result for teaching a topic many beginners find daunting.

Responses to Free-Form Questions

Finally, respondents were invited to give free-form feedback in three parts; positive, negative, and "other." Six, four, and seven answers, respectively, were submitted.

Low-level, per-sample processing and exporting binaries to other environments were mentioned as good or interesting in most responses. Participants were most critical towards the perceived lack of documentation. Several of them felt that the workshop was too short to really develop an interest in the software or in the topic in general, and that they were not always able to follow the mathematical reasoning of the patches we built.

Future Work

The proposed programming method with Veneer and Kronos is based on the idea of applying higher-order functions and functional abstraction to increase the expressive power of a signal processing language. Although there is an undeniable compatibility between visual dataflow and functional programming, we cannot at this time make any strong claims about whether that translates to improved learning outcomes. It appears to us that the set of abstractions most beneficial in a visual-first programming environment may be different from those in a textual functional language, but this too requires further study.

Music languages have expanded in scope to cover growing subsets of musical programming tasks. Traditional score- and instrument-oriented languages gain capabilities to express signal processing tasks; our efforts with Kronos focus on expanding upwards from elementary DSP by means of increased abstractive power of functions and types. Yet it is unclear whether a "theory of everything" in musical programming is attainable or even desirable.

Workshops and Tools

Our survey results demonstrate that instead of trying to build a single tool for everything, focusing on interoperability may be a more viable strategy. Building extensions for another music language was a successful strategy for motivating students to learn DSP. Regarding the development of patching tools, the interactive features and adhoc interrogation of patches resonated with our workshop participants, suggesting that the ideas are worth pursuing further.

The critical feedback indicates avenues of further improvement in our user interface and documentation. The discoverability of features and documentation seems to be an issue. More attention should be paid to familiarizing new users with the system and providing relevant online help within the patcher application.

Several users mentioned problems with mathematical or theoretical prerequisites, even though we designed the material to be as accessible as possible.

In a short-form practical workshop, some theory is necessarily glossed over. A longer-term course with technology-assisted, musician-focused teaching of the fundamentals of signal theory is an intriguing possibility for further work.

Technology-assisted teaching of mathematical fundamentals motivates additional development of the interactive and visual affordances in Veneer, such as improving its capabilities for interactive mathematical plots. Some useful ideas can be found in our prior work (Norilo 2012), as well as projects like Jupyter (Perez and Granger 2015). Given the positive reception of the web application, we should further pursue the advantages of the web platform, like network and cloud features, or embedding of Veneer patches into pedagogical online hypermedia.

User Research

Further user research is necessary to gain a more-complete picture of how well Veneer and Kronos work for practitioners. The workshop presented in this article could be extended in time, given to a more diverse demographic, and otherwise enhanced in scope. Multiple iterations would allow for refinement and experimentation on the content, presentation, and survey.

As a web application, Veneer would also be suitable for a large-scale online user study. Detailed data could be gathered by telemetry from consenting participants, enabling a more detailed view into how learners interact with it.

Outside of the pedagogical context, user research on any music language should also encompass creative and artistic practices and processes.

Conclusions

This article discussed our approach to teaching and democratizing the fundamentals of signal processing that underlie our musical repertoire. The goal of empowering musicians to study and master their tools has informed the development of Veneer, a visual patching environment built for Kronos, an expressive language for signal processing. It extends the prior art on visual dataflow programming with powerful abstractions from functional programming, while maintaining focus on interactivity, gradual abstraction, and the performative and creative musical mindset. The entire programming environment is freely available online as a web application. It was found to run well, demonstrating the capability of the web platform to adequately support a complex, high-performance audio application.

We evaluated our approach in a hands-on workshop and a survey. The results indicate that there is an empty niche for an online signal-processing language that is easily approachable and integrates smoothly with other tools. Relatively inexperienced learners were able to transcribe implementations from literature and seemed to gain self-confidence while doing so. The responses also highlighted the weaker parts of our method, including discoverability of patcher program features and code written by others.

The open-source software and material described in this article is freely available at https://kronoslang.io. As always, we welcome contributions, discussion, and feedback from all interested parties.

References

Assayag, G., et al. 1999. "Computer-Assisted Composition at IRCAM: From PatchWork to OpenMusic." *Computer Music Journal* 23(3):59–72. 10.1162/014892699559896

Blackwell, A. F. 2002. "First Steps in Programming: A Rationale for Attention Investment Models." In *Proceedings of the IEEE Symposia on Human Centric Computing Languages and Environments*, pp. 2–10.

Brandt, E. 2002. "Temporal Type Constructors for Computer Music Programming." PhD dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Brooks, J. G., and M. G. Brooks. 1999. *In Search of Understanding: The Case for Constructivist Classrooms*. Alexandria, Virginia: ASCD.

Catlin, D. 2019. "Beyond Coding: Back to the Future with Education Robots." In *Smart Learning with Educational Robotics*. Berlin: Springer, pp. 1–41.

Dannenberg, R. 1991. "Expressing Temporal Behavior Declaratively." In R. F. Rashid, ed. *CMU Computer*

Science: A 25th Anniversary Commemorative. New York City: ACM Press, pp. 47–68.

Dannenberg, R. 1997. "The Implementation of Nyquist, a Sound Synthesis Language." *Computer Music Journal* 21(3):71–82. 10.2307/3681015

Dannenberg, R., and N. Thompson. 1997. "Real-Time Software Synthesis on Superscalar Architectures." *Computer Music Journal* 21(3):83–94. 10.2307/3681016

Dannenberg, R. B. 2018. "Languages for Computer Music." *Frontiers in Digital Humanities* 5:Art. 26. 10.3389/fdigh.2018.00026

Fober, D., Y. Orlarey, and S. Letz. 2011. "Faust Architectures Design and OSC Support." In *Proceedings of the International Conference on Digital Audio Effects*, pp. 213–216.

Griffin, J. M. 2019. "Constructionism and De-Constructionism: Opposite Yet Complementary Pedagogies." *Constructivist Foundations* 14(3):234–243.

Hancock, C. M. 2003. "Real-Time Programming and the Big Ideas of Computational Literacy." PhD dissertation, Massachusetts Institute of Technology, School of Architecture and Planning, Cambridge, Massachusetts.

Hudak, P. 1989. "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys* 21(3):359–411. 10.1145/72551.72554

Ingalls, D., et al. 1988. "Fabrik: A Visual Programming Environment." *ACM SIGPLAN Notices* 23(11):176–190. 10.1145/62084.62100

Kleimola, J., et al. 2011. "Feedback Amplitude Modulation Synthesis." *EURASIP Journal on Advances in Signal Processing* 2011:Art. 434378. 10.1155/2011/434378, PubMed: 24348546

Kölling, M., and F. McKay. 2016. "Heuristic Evaluation for Novice Programming Systems." *ACM Transactions on Computing Education* 16(3):Art. 12.

Kuuskankare, M., and M. Laurson. 2009. "ENP: A System for Contemporary Music Notation." *Contemporary Music Review* 28(2):221–235. 10.1080/07494460903322505

Lattner, C., and V. Adve. 2004. "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation." In *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 75–86.

Laurson, M., M. Kuuskankare, and V. Norilo. 2009. "An Overview of PWGL, a Visual Programming Environment for Music." *Computer Music Journal* 33(1):19–31. 10.1162/comj.2009.33.1.19

Laurson, M., V. Norilo, and M. Kuuskankare. 2005. "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control." *Computer Music Journal* 29(3):29–41. 10.1162/0148926054798223

Lazzarini, V. 2013. "The Development of Computer Music Programming Systems." *Journal of New Music Research* 42(1):97–110. 10.1080/09298215.2013.778890

Lazzarini, V., et al. 2014. "Csound on the Web." In *Proceedings of the Linux Audio Conference*, pp. 77–84.

Lazzarini, V., et al. 2016. "User-Defined Opcodes." In *Csound*. Berlin: Springer, pp. 139–151.

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

McPherson, A., and K. Tahiroğlu., 2020. "Idiomatic Patterns and Aesthetic Influence in Computer Music Languages." *Organised Sound* 25(1):53–63. 10.1017/S1355771819000463

Michon, R., and Y. Orlarey. 2012. "The Faust Online Compiler: A Web-based IDE for the Faust Programming Language." In *Proceedings of the Linux Audio Conference*, pp. 111–116.

Myers, B. 1986. "Visual Programming, Programming by Example, and Program Visualization: A Taxonomy." *ACM SIGCHI Bulletin* 17(4):59–66. 10.1145/22339.22349

Nielsen, J., I. Frehr, and H. O. Nymand. 1991. "The Learnability of HyperCard as an Object-Oriented Programming System." *Behaviour and Information Technology* 10(2):111–120. 10.1080/01449299108924276

Nishino, H., N. Osaka, and R. Nakatsu. 2013. "Unit-Generators Considered Harmful (for Microsound Synthesis): A Novel Programming Model for Microsound Synthesis in LCSynth." In *Proceedings of the International Computer Music Conference*, pp. 148–155.

Norilo, V. 2012. "Visualization of Signals and Algorithms in Kronos." In *Proceedings of the International Conference on Digital Audio Effects*, pp. 15–18.

Norilo, V. 2015. "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing." *Computer Music Journal* 39(4):30–48. 10.1162/COMJ_a_00330

Norilo, V. 2019. "Veneer: Visual and Touch-based Programming for Audio." In M. Queiroz and A. X. Sedó, eds. *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 319–324.

Noss, R., and J. Clayson. 2015. "Reconstructing Constructionism." *Constructivist Foundations* 10(3):285–288.

Orlarey, Y., D. Fober, and S. Letz 2004. "Syntactical and Semantical Aspects of Faust." *Soft Computing* 8(9):623–632. 10.1007/s00500-004-0388-1

Orlarey, Y., and P. Jouvelot. 2016. "Signal Rate Inference for Multidimensional Faust." In *Proceedings of the Symposium on the Implementation and Application of Functional Programming Languages*, Art. 1.

Papert, S. 1980. *Mindstorms: Children, Computers, and Powerful Ideas*. New York: Basic Books.

Papert, S. 1987. "Microworlds: Transforming Education." In *Artificial Intelligence and Education*, pp. 79–94.

Papert, S. 1993. *The Children's Machine: Rethinking School in the Age of the Computer*. New York: Basic Books.

Partti, H. 2014. "Cosmopolitan Musicianship under Construction: Digital Musicians Illuminating Emerging Values in Music Education." *International Journal of Music Education* 32(1):3–18. 10.1177/0255761411433727

Perez, F., and B. E. Granger. 2015. "Project Jupyter: Computational Narratives as the Engine of Collaborative Data Science." Available online at archive.ipython.org/JupyterGrantNarrative-2015.pdf. Accessed 3 June 2020.

Puckette, M. 1996. "Pure Data: Another Integrated Computer Music Environment." In *Proceedings of the International Computer Music Conference*, pp. 269–272.

Roads, C. 1996. *The Computer Music Tutorial*. Cambridge, Massachusetts: MIT Press.

Roberts, C., and J. Kuchera-Morin. 2012. "Gibber: Live Coding Audio in the Browser." In *Proceedings of the International Computer Music Conference*, pp. 64–69.

Rocchesso, D. 1997. "Maximally Diffusive Yet Efficient Feedback Delay Networks for Artificial Reverberation." *IEEE Signal Processing Letters* 4(9):252–255. 10.1109/97.623041

Salazar, S., and G. Wang. 2012. "Chugens, Chubgraphs, ChuGins: Three Tiers for Extending ChucK." In *Proceedings of the International Computer Music Conference*, pp. 60–63.

Schröder, B. 2002. *Ordered Sets: An Introduction*. Boston, Massachusetts: Birkhäuser.

Selby, C. C. 2015. "Relationships: Computational Thinking, Pedagogy of Programming, and Bloom's Taxonomy." In *Proceedings of the Workshop in Primary and Secondary Computing Education*, pp. 80–87.

Sorensen, A., and H. Gardner. 2010. "Programming with Time Cyber: Physical Programming with Impromptu." *ACM SIGPLAN Notices* 45(10):822–834. 10.1145/1932682.1869526

Sorensen, A., and H. Gardner. 2017. "Systems Level Liveness with Extempore." In *Proceedings of the ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pp. 214–228.

Sutherland, I. 1964. "Sketchpad: A Man–Machine Graphical Communication System." *Transactions of the Society for Computer Simulation* 2(5):3–20.

Valimaki, V., et al. 2009. "Alias-Suppressed Oscillators Based on Differentiated Polynomial Waveforms." *IEEE Transactions on Audio, Speech, and Language Processing* 18(4):786–798. 10.1109/TASL.2009.2026507

Wang, G., P. Cook, and S. Salazar. 2015. "ChucK: A Strongly Timed Computer Music Language." *Computer Music Journal* 39(4):10–29. 10.1162/COMJ_a_00324

Wang, G., and P. R. Cook 2003. "ChucK: A Concurrent, On-the-Fly, Audio Programming Language." In *Proceedings of the International Computer Music Conference*, pp. 219–226.

Yi, S., and S. Letz. 2020. "The Browser as a Platform for Ubiquitous Music." In V. Lazzarini, D. Keller, N. Otero, and Turchet, eds. *Ubiquitous Music Ecologies*. New York: Routledge, pp. 170–189.