Veneer: Visual and Touch-based Programming for Sound

Vesa Norilo
University of the Arts
Helsinki, Finland
vno11100@uniarts.fi

ABSTRACT

This paper presents Veneer, a visual, touch-ready programming interface for the Kronos programming language. The challenges of representing high-level data flow abstractions, including higher order functions, are described. The tension between abstraction and spontaneity in programming is addressed, and gradual abstraction in live programming is proposed as a potential solution. Several novel user interactions for patching on a touch device are shown. In addition, the paper describes some of the current issues of web audio music applications and offers strategies for integrating a web-based presentation layer with a low-latency native processing backend.

Author Keywords

NIME, programming, DSP, visual, touch

CCS Concepts

•**Applied computing** → **Sound and music computing;** •**Software notations and tools** → **Visual languages;** •**Human-centered computing** → *Touch screens;*

1. INTRODUCTION

One of the transformative developments in computer programming was the introduction of Fortran, a high level programming language, in the 1950s. Fortran introduced a radical idea: scientists and engineers would no longer need the assistance of specialist computer programmers, but rather could write programs themselves. [1]

Today that is certainly the case, but the idea that musicians, music producers or composers should also write the software they use professionally may seem far-fetched. Yet it is actively pursued, with multiple music programming languages being developed both academically and commercially.

This study concerns Veneer, a visual interface developed for programming musical signal processors in the Kronos language. [21] Sources of friction, in both the user interface and the language design are considered and solutions are proposed.

2. BACKGROUND

Music seems to be a distinct programming task. One indication of this is the high number of research projects dealing with music-specific programming languages, which I will call Music Languages (MLs) in this paper. In the literature Dannenberg [5, pp. 3–4], Sorensen [28] and Wang [34] emphasize the central role of time in music. According to Kery and Myers, digital music is an example application of Exploratory Programming, which features experimentation and ideation with an open-ended specification. [11]

McLean and Wiggins discuss creativity in computer science. They reposition Klee's description of the painters' creative process into the context of computer programming: a work is a result of a continuous cycle of creation, perception, appraisal and adjustment. They note a similarity to the *bricolage programming* described by Turkle and Papert, [31] an intuitive, *ad hoc* method of programming, and emphasize the importance of perception in evaluating the progress of the work. [17]

2.1 Common Features of Music Languages

Since the early days of computer music, a three-layer abstraction has been utilized. The lowest level, that which deals directly with audio signals, consists of *ugens.*[1] [26, pp. 783-818] Several ugens are composed into systems that resemble traditional *instruments*. The musical information that drives the instruments is called the *score*. [15]

The three-layer model is not apposite to all musical tasks, and some of the more recent MLs eschew scores. SuperCollider [16] and ChucK [33] do so in favour of script-like control of the ugen graph, while Max [2] and Pure Data [24] operate without a score layer, combining aspects of the ugen and instrument layers into a all-encompassing *patch*.

Brandt [3, pp. 3-4] and Nishino [19] discuss some problems in the ugen abstraction; the provided ugens may not be suitable for the task at hand, in which case ugen systems may prove opaque dead ends. Faust [23] proposes a method of building ugen-like functionality from even simpler, more generally adaptable primitives.

2.2 Visual Languages

As early as 1964, Ivan Sutherland fashioned Sketchpad, a visually oriented programming system. [29] Fabrik was an early attempt to visualize object-oriented programming. [9] Commercial products attained a degree of success: Hypercard was widely acclaimed, and perhaps one of the earliest successful enablers of end-user developers. LabVIEW [30] is a prominent example of a commercially developed visual dataflow language. Johnston et al. provide a comprehensive overview of other visual dataflow languages. [10, pp. 21-23]

[1]"Ugens", unit generators, are primitive operators in an audio dataflow, f.e. oscillators, envelopes and filters.
[2]Commercial product by Cycling'74

NIME'19, June 3-6, 2019, Federal University of Rio Grande do Sul, Porto Alegre, Brazil.

When it comes to music, arguably the most successful enabler of musician-programmers is Max, a commercial software package. Max is fashioned as a library of predefined program components that the user may connect with virtual patch cords.

Various academic projects have also produced visual MLs. Pure Data shares the pedigree of Max. [24] PWGL [14] is a visual programming surface built on top of Common Lisp, integrating a visual score editor. [12]

Myers suggests that the spatial capabilities of the human brain are suited for visual programming. [18] In the case of MLs, Laurson cites integration with musical scores as an additional motivation. [13] Further arguments for and against visual programming are summarized in [35].

3. PROGRAMMING AS MUSIC-MAKING

Music Languages do mean to lower the barrier to entry for non-programmers. However, it is important to consider musicians as more than just bad programmers. MLs should be designed for *domain experts*, and seek to take advantage of their unique strenghts.

In music, results are often evaluated perceptually by ear. Program "tweaking" or "tuning" is done in a rapid sequence of adjustment and listening. From the interaction perspective, this indicates that the programming tools should be as responsive as possible. Languages such as C++, where the compilation cycle from source code to executable code takes minutes, or hours, are not suitable for such workflow.

Many MLs provide real-time feedback and control. On the extreme end are the languages geared towards live coding, a form of performative programming. [27] I propose that the real-time nature of live coding is important for a musical mindset in programming, even if it is not performative at all. Interestingly, similar strategies yield benefits in digital pedagogy [8]. Perhaps real-time feedback makes it less sternuous to maintain a mental model of the system being programmed [2], facilitating both creativity and learning.

3.1 On Abstraction

Abstraction – information hiding – is a defining feature of programming, according to Blackwell. [2] For the purposes of live programming, abstraction can cut both ways: hiding *unnecessary information* allows concentrating on the essential, more can be achieved with less code, results can be had more quickly and programs can be less viscous with regard to local change.

On the other hand, the mental model of the program becomes more complicated with higher level of abstraction. The program can behave in astonishing, unexpected ways. When this happens, it is contrary to the flow of live programming, and likely to disrupt the creative flow.

3.1.1 Abstraction in Max

Max, the prominent visual music language, eschews abstraction almost entirely. This is achieved by a strict one-to-one relationship between program entities and their graphical representation on the screen. Notably, Max does contain a construct called "abstraction", which allows the programmer to encapsulate a subpatch inside a single box.

However, some of the problems cited in Max result from the very lack of proper abstraction. McCartney proposes that the static object structure is the fundamental reason for many of its limitations. Blackwell quotes abstraction as one of the defining features of programming [2], although perhaps this is what prompts McCartney to write that many users of Max "do not realize they are programming." [16, p. 61]. Petre and Green argue that even though abstraction can be a learning hurdle, in the end it could increase language comprehensibility, protect against errors and mitigate other usability problems, such as viscosity[3]. [7]

Blackwell discusses another cognitive challenge of programming: the loss of direct manipulation. The programmer does not achieve her goals directly, but rather via indirection, requiring her to maintain a mental model of the machine she is guiding. [2] Interestingly, the Max model of one-to-one mapping between graphical widgets and program objects preserves aspects of direct manipulation, by integrating control widgets within the program.

3.1.2 Adding Abstraction to the Visual Domain

If the one-to-one mapping between on-screen and program objects is abandonded, we can express powerful programs more compactly. We could encode the principle of a repetitive construct, such as a filter bank, rather than spell out each component. Perhaps we can also write programs that are less viscous: aspects of signal *type*, such as channel count or value intervals, could propagate along the data flow rather than requiring manual reconfiguration at every stage.

Both examples given in the prior paragraph are forms of abstraction. We express a concept once, and programmatically derive additional structure. This makes programs more powerful, as a small change can have far-reaching effects. Good abstraction makes for programs that are brief and to the point, while bad abstraction makes for an incomprehensible mess.

3.2 Deductive and Inductive Programming

Let us think about programming in terms of *deduction* and *induction*. A generally accepted strategy for writing extensible, maintainable programs resembles deductive reasoning: we strive to find a maximally general abstraction that suits the task at hand. Doing so can reduce the coupling between components and enable painless extension and modification down the line.

However, I contend that deduction is a polar opposite to the proposed interactive programming model. Thinking in terms of abstractions is a valid strategy for composing music, but is far removed from the performative mindset.

The opposite model of induction is better aligned with the goals of the present study. We start from concrete values and data flows, and perform *gradual abstraction*. For example, we may realize that certain constants in our program could be exposed as *parameters*, making a subpatch reusable in different contexts. This is essentially the meaning of *abstraction* in lambda calculus; the constant that becomes a variable is the *lambda term*.

This style of abstraction is particularly well suited for the visual domain. An example implementation is described in Section 4.2.4.

4. IMPLEMENTATION

This section describes an implementation of the concepts discussed in the previous sections. It consists of the programming language Kronos, as well as Veneer, the visual interface that is the main topic of this paper.

4.1 Kronos: the Signal Processing Language

Kronos [21] is a functional language for designing signal processors as discrete reactive systems (DRS) [32]. Each DRS responds to a sequence of input events with a well-defined sequence of output events. The time–value pair is a

[3]Viscosity refers to resistance to local change. In programming it could indicate how easy it is to change the program structure after the fact, or if parts of the program can be changed independently from the rest.

good abstraction for signals in music systems, which range from the high, steady data rates of audio samples (44.1kHz or higher) to event-driven signal sources like instrument controllers or MIDI signals.

This programming model has the advantages of consistency and ubiquity, scaling from building ugens to complicated orchestras. All musical tasks are modeled similarly: by reactive data flows. Because the language only permits a deterministic signal flow, the compiler is able to generate any required stateful code, eliminating the class of errors related to variables, assignment or side effects. For a proper discussion of the language design, the reader is referred to prior work. [21]

4.1.1 *Abstract Syntax Tree and Data Flow*

Of the existing Music Languages, FAUST [23] shares many goals with Kronos. FAUST offers a strong ecosystem [6] as well as an embeddable, highly efficient compiler. On the other hand, Kronos places more emphasis on multi-rate signal processing [20] and an expressive type system [21].

Of special interest to the present study is the syntactic difference between the languages. FAUST features a concise and powerful block diagram algebra. The abstract syntax tree in FAUST describes *how functions are composed*, and the final result is a description of *how the output is computed*. On the other hand, Kronos has a more traditional syntax that describes *how data flows*, and expressions result in *signals*. From this it follows, that a simple visual representation of the abstract syntax tree of a Kronos program is a data flow diagram, apposite for a visual programming interface. The isomorphism of the language in textual and visual form is an important design criteria.

4.2 Veneer: the Patcher

Veneer is a visual frontend for the Kronos language, built on the web platform. It can connect to a native back end via a websocket, or utilize an embedded WebAssembly compiler.

4.2.1 *Interactive Programming*

Veneer is built to support the ideal of interactive programming. Program changes are compiled in real time, and the Kronos compiler enables turnaround time ranging from dozens to hundreds of milliseconds depending on program complexity.

In addition, the interface is built to enable *direct manipulation* of some aspects of the program: each numeric constant can dynamically turn into an interactive control, which can be tweaked without recompilation. Interactive widgets can also be included in the program.

Another form of rapid feedback is provided with reactive evaluators. An output visualizer can be placed at any node in the signal graph, displaying either textual, graphical or waveform results in real time.

Kronos programs are generic, and specialized to statically typed flows. The only possible program error is an ill-typed expression, such as destructuring a scalar or performing arithmetic on a string. This is arguably the most difficult aspect of the language, and Veneer tries to help by visualizing the type derivation process in order to clarify the source of the type error.

4.2.2 *Some Gestures for Patching Interfaces*

One purpose of the present study is to explore the design space of patcher interfaces. This section gives a brief overview of the interactions in Veneer.

The patch display is implemented with scalable vector graphics, and continuous pan and zoom are available with second button mouse drags or swipe and pinch gestures.

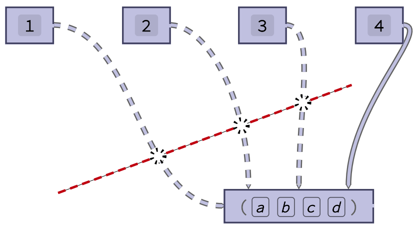

Figure 1: Cutting cables

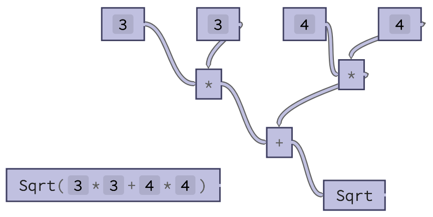

Figure 2: Applying *explode* three times on a complex expression

Nodes can be connected by clicking and dragging, with cables snapping to nearby compatible sockets. Disconnection is accomplished by a cut cesture, dragging over a set of cables with a modifier key pressed. The cutter widget is illustrated in Figure 1.

Alternatively, a node may be disconnected from all cables by shaking vigorously; this idea is adapted from the eponymous visual compositing software Shake.

The user interface has an automatic patching facility, which tries to connect all unconnected slots of selected nodes into available slots, so that signal flows left-to-right and top-to-bottom. Automatic layout functions are available for aligning nodes along the top, bottom, left or right edges, as well as justifying horizontally or vertically.

In addition to the basic functions outlined above, Veneer offers tools to refactor expressions. Because nodes can hold arbitrary Kronos expressions, subpatches can be collapsed into a single node. Likewise, nodes containing more complicated expressions can be exploded into subpatches. An example is shown in Figure 2.

A common subset of *collapse* is gathering nodes into a *list* or a *tuple*. There are direct commands for these tasks; a wrapping tuple or list is created automatically and all selected nodes are collapsed into it.

4.2.3 *Visual Abstraction in Veneer*

One of the main features of the Kronos language is the support for anonymous functions (lambda abstractions) in signal processing without efficiency tradeoffs. This allows for functional programming staples, such as *map* and *fold* to be used for constructing filter banks and cascades, fulfilling the role loops have in imperative languages.

Veneer represents *function application* by patch cord connections. An unconnected node is therefore a non-applied function: its output is itself a *function of the missing input*, a verb rather than a noun. Partially connected nodes are shown in a grayed-out color and a rounded corner.

Such a verb can be provided as a parameter to a higher order function, which can use it to construct algorithmic routings. For example, see Figure 3, in which *map* is used to

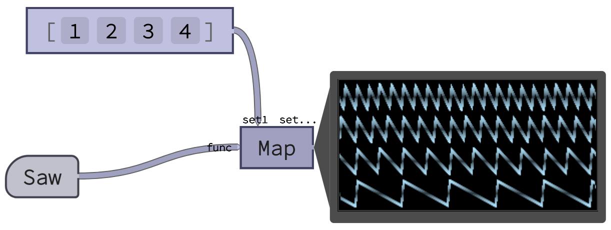

Figure 3: Additive synthesis via higher order function

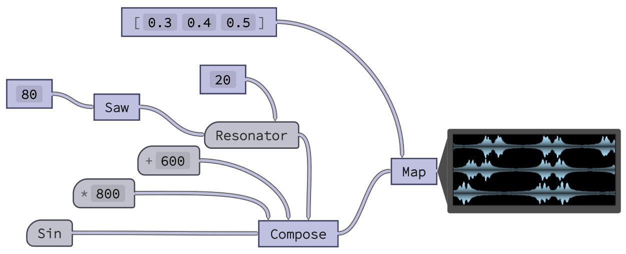

Figure 4: Partial application and function composition

produce an additive synthesizer by *applying* an oscillator to each element of the list. Even though this is an audio-rate process, the example uses very low frequencies in order to provide discernible waveform plots.

An extension of the analogy is *partial application*, which is, in Veneer, a partially connected node. By connecting some inputs, but not others, we can leave blanks for a higher order function to fill. This is useful in the case of a filter bank, where a common signal fanout is implemented via partial application.

A number of visual abstractions are shown in Figure 4. The example is somewhat contrived for demonstration purposes. A bank of resonator sweeps is built, again via *map*. This time, the mapping function is *composed* as a serial connection of four non-applied functions. First, the sweep frequency becomes a signal via *Sin*, a sinusoid oscillator. We follow up with partially applied, one-sided binary operators to rescale the sine output to the interval of $[200, 1400]$. The final stage is a partially applied *Resonator*, where *signal* and *bandwidth* slots are connected, but *frequency* is lunconnected, thus becoming the parameter to the newly unary function. The final mapping function provided to *map* is a serial composition of all four stages. Even in an abstract patch like this one, all the numeric values can be tweaked in real time.

Map corresponds to a parallel routing. For serial routing, *Reduce* can be used. *Reduce* works by combining the first two elements in a list, using a function passed in as a parameter. A new list is constructed from the result of the function and the tail of the original list, and subsequently reduced again until the list has a single element. A simple frequency modulation cascade is shown in Figure 5: the reducer function is an ad-hoc specified binary function, combining a modulator signal on the left hand side with an operator frequency parameter on the right hand side.

4.2.4 Gradual Abstraction

Veneer patches are organized in workspaces of tabs, where each tab can be a *drafting tab*, resembling a REPL, or a

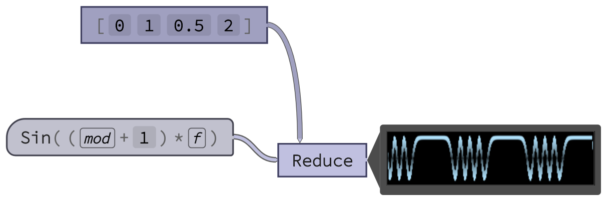

Figure 5: Frequency Modulation synthesis with *Reduce*

named *function*. To support gradual abstraction, fragments of a patch can be extracted to new functions. The fragment is replaced with a single node, and a tab containing the fragment is added to the workspace.

This simple refactoring tool is enough to enable visual designation of lambda terms, as alluded to in Section 3.2. The user may supply metadata for the function, which is subsequently available in all the menus as well as autocompletion. User functions are first-class in terms of application with *Map*, *Reduce* or other higher order functions.

4.3 Multi-touch Patching

Recently, multi-touch capable mobile devices have grown in computational power, making them increasingly attractive to signal processing tasks. Many mobile platforms have restricted JiT compilers, such as Kronos, in native applications. However, the adoption of WebAssembly in modern mobile browsers has dramatically improved the outlook.

The scale-variable graphics in Veneer are a good fit for conventional mobile gestures, such as finger panning and pinch-zooming. Nodes can be moved with a finger, and the shake gesture to disconnect is particularly satisfying.

I found touch also attractive for making connections: the user can simply touch the endpoints simultaneously. Because touch targets can be hard to hit precisely, Veneer prioritizes the nearest unconnected input slot when touch-patching. This makes the act of replacing a connection in a partially connected node less convenient (the cable has to be cut first), but the common case is significantly easier.

Because touch devices often have no physical keyboards, and thus no convenient keyboard shorcuts, some shortcuts are represented in touch mode as an additional floating toolbar. The toolbar affects touch modality: each icon can be tapped, in which case it affects the mode of the subsequent touch gesture. Alternatively, the mode is kept active while the user holds the touch on the icon.

As of this writing, the mode icons include *evaluate*, which activates audio output or a display on the nodes the user touches, *cut*, which changes finger swipes on the canvas to cut gestures (see Section 4.2.2) and causes taps to cut nodes, and *select*, which allows the user to select additional nodes without clearing the existing selection. In addition, undo and redo are provided on the toolbar when applicable. The less common functionality is available in the application menu.

4.4 Audio on the Web Platform

For a long time, user-generated audio was an afterthought on the web platform. In the recent years this has been changing with the introduction of more latency-friendly technologies in all major browsers.

4.4.1 WebAssembly

WebAssembly is a new low-level, efficient bytecode format designed for the web and JiT-compilers. It offers performance that approaches native code. As WebAssembly features linear memory, latency issues due to garbage collection are not inherent to the design. For embedding into Veneer, Kronos was compiled to WebAssembly, as well as enhanced with a WebAssembly backend to dynamically generate new signal processors within the browser.

4.4.2 ScriptProcessorNode and AudioWorklet

ScriptProcessorNode was the initial mechanism for writing low-latency audio processors for the web audio API. It remains the only one that is widely implemented. The main problem in the design is the fact that audio processors must share the single javascript thread and event loop with the entire application. Any functionality in the application can block audio computation and result in a glitch.

AudioWorklet is an improved design for custom audio processors. AudioWorklets run in a separate javascript context, and do not implicitly share resources with the UI thread. AudioWorklets can theoretically reach latency characteristics similar to native browser audio functions. In practice, browser implementations are not yet up to the standard of native audio applications.

Browsers fall short particularly in input latency, as well as good support for multi-channel pro audio interfaces. With regard to the developer perspective, low latency multithreading (atomics and shared memory) is nascent, especially due to setbacks related to security. [4] Audioworklets are supported in Chrome only, and the implementation fails to set the correct floating point mode, leading to denormal issues. Hopefully these issues will be worked out in the near future.

4.4.3 Native Backend Integration

For best performance, lowest latency and widest I/O support, native applications are still superior to web applications. As an alternative to running the embedded WebAssembly compiler within the browser, Veneer can also act as the front-end to a native compile server.

As Veneer is still a web application, it is limited to the TCP protocol for communication. In addition to code, control data must be sent to the backend: communication latency is still important, even though audio processing happens outside of the browser.

Initial tests with HTTP were successful up to a point, but the problem with HTTP is a lack of pipelining and true full duplex communication. The client must poll the server, and no further requests can be sent before the full round trip has been completed for the previous one. Experimental support for HTTP pipelining can help in this regard, but fortunately there is a better solution: *WebSockets* enable full-duplex streaming over TCP and are well-supported in browsers.

The Kronos package includes *krpcsrv*, a websocket server capable of acting as a compilation server and DSP engine for Veneer.

5. DISCUSSION

At present, Veneer is more like a hypothesis than a research result. It is not hard to find examples where increased visual abstraction can become counterproductive. For example, to understand, or better yet, create, the patch in Figure 4, one has to internalize several abstract concepts. How does that go along with the stated goal of spontaneous, real-time programming?

Obtaining solid data on how well a programming tool suits its purpose is notoriously difficult. [35, p. 2] It is particularly challenging to assemble a sufficient number of musician-programmers willing to partake in user tests. If testing the methodology requires users to learn new concepts, the ask is even greater.

Clearly this work needs to move towards such user tests in order to validate or falsify usability claims. In order to reach an adequate mass of users, it is essential to lower the barrier to entry, and build compelling example implementations to motivate musicians to try the technology and provide feedback.

[4]"Spectre" and "Meltdown" were widespread security exploits that caused Chrome to temporarily disable thread shared memory for Javascript altogether, and likely delayed other implementations.

5.1 Future Work

The web platform is a central driver in making the technology more accessible without the hurdles installation or configuration. It opens up the possibility of building an online course around the concepts. This could a viable strategy for gathering user tests and approaching a quantitatively credible appraisal of the proposed methodology.

5.1.1 New Interactions

The design space of visual, spatial programming interfaces remains interesting. New interactions specific to *time* are intriguing. In this paper, subsonic frequencies were used in examples in order to make waveforms discernible. It could be useful and educational to be able to manipulate time on the global scale, and zoom into the behavior of a signal graph on a sample-per-sample level. Further, a rewindable timeline, where the programmer could inspect the state of the signal graph at any given point, could be useful.

5.1.2 Language Support

Veneer has a relatively loose coupling with the Kronos language. It could be used as a patching frontend to other languages as well. Due to consistent syntax, Lisp-family languages could be particularly suitable. One possibility for a front-end integration is PWGL, a Lisp-based visual programming environment sharing the pedigree with Kronos. [14]

5.1.3 Timelines

Veneer and Kronos handle time via *memory* and *state*. As of now, there are is no Veneer representation of the more recent temporal mechanisms added to Kronos, such as metasequencing and temporal recursion. [22]

The integration of timelines into the patching metaphor, or augmenting the patcher with sequencer-like capabilities is an intriguing avenue of further research.

5.2 Availability

Kronos, along with a minimal but growing set of learning material, as well as the source code, is freely available at `https://kronoslang.io`. The web-based patcher can be tried at `https://kronoslang.io/live`. The example patches from this paper are also available online in interactive form: `https://kronoslang.io/nime2019.html`.

6. CONCLUSIONS

This paper discussed aspects of visual programming of musical systems, including background on both Music Languages and visual languages. Hypotheses on a good interface for musical programming were proposed, and an implementation to test them out was described. The study extends to touch-based programming with the patching metaphor, and proposes some gestures and design concepts for a good user experience.

The technology backing the research in this study features an advanced functional dataflow language and compiler, Kronos, as well as Veneer, an exploratory programming surface built on the web platform. The recent WebAssembly integration of Kronos enables the entire system to function in a modern browser. The program code is freely available as open source.

This paper is intended as an opening for exploration and discussion of the patching metaphor and novel interfaces and interactions. The hypotheses presented can not be properly tested without extensive community interaction. Any interested parties are thus more than welcome to engage with the technology, and I remain grateful for all the valuable input the community may wish to provide.

7. ACKNOWLEDGMENTS

Vesa Norilo's work was supported by the Academy of Finland, award number SA311535.

8. REFERENCES

[1] J. Backus. The History of FORTRAN I, II, and III. *Annals of the History of Computing*, 1(1):21–37, 1979.

[2] A. Blackwell. First steps in programming: A rationale for attention investment models. In *Human Centric Computing Languages and Environments, 2002. Proceedings. IEEE 2002 Symposia on*, pages 2–10. IEEE, 2002.

[3] E. Brandt. *Temporal type constructors for computer music programming.* PhD thesis, Carnegie Mellon University, 2002.

[4] Cycling'74. Max 6, 2011.

[5] R. Dannenberg. Languages for computer music. *Frontiers in Digital Humanities*, 5:26, 11 2018.

[6] D. Fober, Y. Orlarey, and S. Letz. FAUST Architectures Design and OSC Support. In *Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 213–216, 2011.

[7] T. Green and M. Petre. Usability analysis of visual programming environments: a "cognitive dimensions" framework. *Journal of Visual Languages and Computing*, 7(2):131–174, 1996.

[8] C. Hancock. *Real-time programming and the big ideas of computational literacy.* PhD thesis, Massachusetts Institute of Technology, 2003.

[9] D. Ingalls, S. Wallace, Y. Chow, F. Ludolph, and K. Doyle. Fabrik: A visual programming environment. *SIGPLAN Not.*, 23(11):176–190, 1988.

[10] W. Johnston, J. Hanna, and R. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[11] M. Kery and B. Myers. Exploring exploratory programming. In *Visual Languages and Human-Centric Computing*, Raleigh, USA, 2017.

[12] M. Kuuskankare and M. Laurson. ENP: A system for contemporary music notation. *Contemporary Music Review*, 28(2):221–235, 2009.

[13] M. Laurson. *Patchwork: A Visual Programming Language and some Musical Applications.* PhD thesis, Sibelius Academy Helsinki, 1996.

[14] M. Laurson, M. Kuuskankare, and Vesa. Norilo. An Overview of PWGL, a Visual Programming Environment for Music. *Computer Music Journal*, 33(1):19–31, 2009.

[15] V. Lazzarini. The Development of Computer Music Programming Systems. *Journal of New Music Research*, 42(1):97–110, 2013.

[16] J. McCartney. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68, 2002.

[17] A. McLean and G. Wiggins. Bricolage programming in the creative arts. *22nd Annual Psychology of Programming Interest Group*, 2010.

[18] B. Myers. Visual programming, programming by example, and program visualization: a taxonomy. *ACM SIGCHI Bulletin*, 17(4):59–66, 1986.

[19] H. Nishino, N. Osaka, and R. Nakatsu. Unit-generators considered harmful (for microsound synthesis): A novel programming model for microsound synthesis in lcsynth. In *ICMC*, 2013.

[20] Vesa. Norilo. Recent Developments in the Kronos Programming Language. In *Proceedings of the International Computer Music Conference*, Perth, 2013.

[21] Vesa. Norilo. Kronos: A declarative metaprogramming language for digital signal processing. *Computer Music Journal*, 39(4):30–48, 2015.

[22] Vesa. Norilo. Kronos meta-sequencer–from ugens to orchestra, score and beyond. In *Proceedings of the International Computer Music Conference*, 2016.

[23] Y. Orlarey, D. Fober, and S. Letz. FAUST: An Efficient Functional Approach to DSP Programming. In G. Assayag and A. Gerszo, editors, *New Computational Paradigms for Music*, pages 65–97. Delatour France, IRCAM, Paris, 2009.

[24] M. Puckette. Pure data: another integrated computer music environment. In *Proceedings of the 1996 International Computer Music Conference*, pages 269–272, 1996.

[25] C. Roads. *the Computer Music Tutorial.* MIT Press, Cambridge, 1996.

[26] A. Sorensen and A. Brown. aa-cell in practice: An approach to musical live coding. In *Proceedings of the International Computer Music Conference*, pages 292–299. International Computer Music Association (ICMA) Copenhagen, 2007.

[27] A. Sorensen and H. Gardner. Programming With Time Cyber-physical programming with Impromptu. *Time*, 45:822–834, 2010.

[28] I. Sutherland. Sketchpad a man-machine graphical communication system. *Transactions of the Society for Computer Simulation*, 2(5):R–3, 1964.

[29] J. Travis and J. Kring. *LabVIEW for Everyone: Graphical Programming Made Easy and Fun (National Instruments Virtual Instrumentation Series).* Prentice Hall PTR, 2006.

[30] S. Turkle and S. Papert. Epistemological Pluralism and the Revaluation of the Concrete. *Journal of Mathematical Behavior*, 11(1):3–33, 1992.

[31] P. Van Roy. Programming Paradigms for Dummies: What Every Programmer Should Know. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Music*, pages 9–49. Delatour France, IRCAM, Paris, 2009.

[32] G. Wang and P. Cook. ChucK : A Concurrent , On-the-fly , Audio Programming Language. In *International Computer Music Conference*, pages 1–8, 2003.

[33] G. Wang, P. Cook, and S. Salazar. ChucK: A Strongly Timed Computer Music Language. *Computer Music Journal2*, 39(4):10–29, 2015.

[34] K. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, 8(1):109 – 142, 1997.