Hiding What from Whom? A Critical Review of the History of Programming languages for Music

Tomoya Matsuura
Tokyo University of the Arts
me@matsuuratomoya.com

ABSTRACT

This paper critically reviews the history of programming languages for music, distinct from computer music as a genre, by drawing on discussions from sound studies. The paper focuses on the universalist assumptions around pulse-code modulation and the Unit Generator concept established by the MUSIC-N family, which established a lineage of role between composers and scientists which tends to turn composers into consumers. The paper concludes that programming languages for music developed after the 2000s present alternatives to the often-invisible technological infrastructures surrounding music, such as formats and protocols, rather than solely aiming to create novel musical styles.

1. INTRODUCTION

Programming languages and environments for music such as Max, Pure Data, Csound, and SuperCollider, have been referred to as "computer music language" [1, 2, 3], "language for computer music" [4], and "computer music programming systems" [5], though there is no clear consensus on the use of these terms. However, as the shared term "computer music" implies, these programming languages are deeply intertwined with the history of technology-driven music, which developed under the premise that "almost any sound can be produced" [6, p557] through the use of computers.

In the early days, when computers existed only in research laboratories and neither displays nor mice existed, creating sound or music with computers was inevitably equivalent to programming. Today, however, programming as a means to produce sound on a computer—rather than employing digital audio workstation (DAW) software, such as Pro Tools is not popular. In other words, programming languages for music developed after the proliferation of personal computers are the software tools that intentionally chose programming (whether textual or graphical) as their frontend for making sound.

Since the 1990s, the theoretical development of programming languages and the various constraints required for real-time audio processing have significantly increased the specialized knowledge necessary for developing programming languages for music today. Furthermore, some languages developed after the 2000s are not necessarily aimed at pursuing new forms of musical expression, and there is still no unified perspective on how their values should be evaluated.

This paper is a critical historical review that draws on discussions from sound studies and existing surveys to examine programming languages for music as distinct from computer music as the specific genre.

1.1 Use of the Term "Computer Music"

Since the 1990s, the term "computer music," despite its literal and potentially broad meaning, has been noted for being used within a narrowly defined framework tied to specific styles or communities, as explored in Ostertag's *Why Computer Music Sucks* [7].

As Lyon observed nearly two decades ago, it is now nearly impossible to imagine a situation in which computers are not involved at any stage from the production to experience of music [8, p1]. The necessity of using the term "computer music" in academic contexts has consequently diminished.

Holbrook and Rudi extended Lyon's discussion by proposing the use of frameworks such as post-acoutmatic [9] to redefine computer music. Their approach situates the tradition of pre-computer experimental/electronic music as part of the broader continuum of technology-based or technology-driven music [10].

Although the strict definition of post-acousmatic music is deliberately left open, one of its key aspects is the expansion of music production from institutional settings to individuals and as well as the diversification of technological usage [9, p113]. However, despite integrating the historical fact that declining computer costs and increasing accessibility beyond laboratories have enabled diverse musical expressions, the post-acousmatic discourse still marginalizes much of the music that is "just using computers" and fails to provide insights into this divided landscape.

Lyon argues that the term "computer music" is a style-agnostic definition, almost like "piano music," implying that it ignores the style and form of music produced by the instrument. However, one of the defining characteristics of computers as a medium lies in their ability to treat musical styles themselves as subjects of meta-manipulation through simulation and modeling. When creating instruments with computers or using such instruments, sound production involves programming—manipulating symbols embedded in a particular musical culture. This recursive embedding of language and recognition, which construct that musical culture, into the resulting music is a process

that exceeds what is possible with acoustic instruments or analog instruments. Magnusson refers to this characteristic of digital instruments as "epistemic tools" and points out that the computer serves to "create a snapshot of musical theory, freezing musical culture in time" [11, p.173] through formalization.

Today, many people use computers for music production not because they consciously leverage the uniqueness of the meta-medium, but simply because there are no quicker or more convenient alternatives available. Even so, within a musical culture where computers are used out of necessity rather than preference, musicians are inevitably influenced by the underlying infrastructures such as software, protocols, and formats. As long as the history of programming languages for music remains intertwined with the history of computer music as it relates to specific genres or communities, it will be difficult to analyze music created with computers as merely a passive means.

In this paper, the history of programming languages for music is reexamined with an approach that, unlike Lyon's, adopts a radically style-agnostic perspective. Rather than focusing on what has been created with these tools, the emphasis is placed on how these tools themselves have been constructed. The paper centers on the following two topics: 1. A critique of the universality of sound representation using pulse-code modulation (PCM)—the foundational concept underlying most of today's sound programming, by referencing early attempts at sound generation using electronic computers. 2. An examination of the MUSIC-N family, the origin of PCM-based sound programming, to highlight that its design varies significantly across systems from the perspective of today's programming language design and that it has evolved over time into a black box, eliminating the need for users to understand its internal workings.

Ultimately, the paper concludes that programming languages for music developed since the 2000s are not solely aimed at creating new music but also serve as alternatives to the often-invisible technological infrastructures surrounding music such as formats and protocols. Thus, the paper proposes new perspectives for the historical study of music created with computers.

2. PCM AND EARLY COMPUTER MUSIC

The MUSIC I (1957) in Bell Labs [12] and succeeding MUSIC-N family are highlighted as the earliest examples of computer music research. However, attempts to create music with computers in the UK and Australia prior to MUSIC have also been documented [13].

The earliest experiments with sound generation on computers in the 1950s involved controlling the intervals of one-bit pulses to control pitch. This was partly because the operational clock frequencies of early computers fell within the audible range, making the sonification of electrical signals a practical and cost-effective debugging method compared to visualizing them on displays or oscilloscopes.

For instance, Louis Wilson, who was an engineer of the BINAC in the UK, noticed that an AM radio placed near the computer could pick up weak electromagnetic waves generated during the switching of vacuum tubes, producing sounds. He leveraged this phenomenon by connecting a speaker and a power amplifier to the computer's circuit to assist with debugging. Frances Elizabeth Holberton took this a step further by programming the computer to generate pulses at desired intervals, creating melodies in 1949 [14].

Further, some computers at this time, such as the CSIR Mark I (CSIRAC) in Australia often had primitive "hoot" instructions that emitted a single pulse to a speaker. Early sound generation using computers, including the BINAC and CSIR Mark I, primarily involved playing melodies of existing music.

However, not all sound generation at this time was merely the reproduction of existing music. Doornbusch highlights experiments on the Pilot ACE (the Prototype for Automatic Computing Engine) in the UK, which utilized acoustic delay line memory to produce unique sounds [13, pp.303-304]. Acoustic delay line memory, used as the main memory in early computers, such as the BINAC and the CSIR Mark I, employed the feedback of pulses traveling through mercury via a speaker and microphone setup to retain data. Donald Davis, an engineer on the ACE project, described the sounds it produced as follows [15, pp.19-20]:

> The Ace Pilot Model and its successor, the Ace proper, were both capable of composing their own music and playing it on a little speaker built into the control desk. I say composing because no human had any intentional part in choosing the notes. The music was very interesting, though atonal, and began by playing rising arpeggios: these gradually became more complex and faster, like a developing fugue. They dissolved into colored noise as the complexity went beyond human understanding.

This music arose unintentionally during program optimization and was made possible by the "misuse" of switches installed for debugging delay line memory. Media scholar, Miyazaki, described the practice of listening to sounds generated by algorithms and their bit patterns, integrated into programming, as "Algo-*rhythmic* Listening" [16].

Doornbusch warns against ignoring these early computer music practices simply because they did not directly influence subsequent research [13, p.305]. Indeed, the sounds produced by the Pilot ACE challenge the post-acousmatic historical narrative, which suggests that computer music transitioned from being democratized in closed electro-acoustic music laboratories to being embraced by individual musicians.

This is because the sounds generated by the Pilot ACE were not created by musical experts, nor were they solely intended for debugging purposes. Instead, they were programmed with the goal of producing interesting sounds. Moreover, these sounds were tied to the hardware of the acoustic delay line memory—a feature that is likely difficult to replicate, even in today's sound programming environments.

Similarly, in the 1960s at the Massachusetts Institute of Technology (MIT), Peter Samson exploited the debugging speaker on the TX-0, a machine that had become outdated and was freely available for students to use. He conducted

experiments in which he played melodies, such as Bach fugues, using "hoot" instruction [17].

Building on this, Samson developed a program called the Harmony Compiler for the DEC PDP-1, which was derived from the TX-0. This program gained significant popularity among MIT students. Around 1972, Samson began surveying various digital synthesizers that were under development at the time and went on to create a system specialized for computer music. The resulting Samson Box was used at Stanford University's CCRMA (Center for Computer Research in Music and Acoustics) for over a decade until the early 1990s and became a tool for many composers to create their works [18]. Considering his example, it is not appropriate to separate the early experiments in sound generation by computers from the history of computer music solely because their initial purpose was debugging.

2.1 Acousmatic Listening, the premise of the Universality of PCM

One of the reasons why MUSIC led to subsequent advancements in research was not simply that it was developed early, but because it was the first to implement sound representation on a computer based on PCM, which theoretically can generate "almost any sound".

PCM, the foundational digital sound representation today, involves sampling audio waveforms at discrete intervals and quantizing the sound pressure at each interval as discrete numerical values.

The problem with the universalism of PCM in the history of computer music is inherent in the concept of acousmatic listening, which serves as a premise for post-acousmatic. Acousmatic listening, introduced by Piegnot as a listening style for tape music, such as musique concrète, and later theorized by Schaeffer [9, p106], refers to a mode of listening in which the listener refrains from imagining a specific sound source. This concept has been widely applied in theories of listening to recorded sound, including Michel Chion's analysis of sound design in film.

However, as sound studies scholar, Jonathan Sterne, has observed, discourses surrounding acousmatic listening often work to delineate pre-recording auditory experiences as "natural" by contrast [1]. This implies that prior to the advent of sound reproduction technologies, listening was unmediated and holistic—a narrative that obscures the constructed nature of these assumptions.

> For instance, the claim that sound reproduction has "alienated" the voice from the human body implies that the voice and the body existed in some prior holistic, unalienated, and self present relation. [20, p20-21]

The claim that PCM-based sound synthesis can produce "almost any sound" is underpinned by an ideology associated with sound reproduction technologies. This ideology assumes that recorded sound contains an "original" source and that listeners can distinguish distortions or noise from it. Sampling theory builds on this premise through Shannon's information theory by statistically modeling human auditory characteristics: it assumes that humans cannot discern volume differences below certain thresholds or perceive vibrations outside specific frequency ranges. By limiting representation to the reconizable range, sampling theory ensures that all audible sounds can be effectively encoded.

Incidentally, the actual implementation of PCM in MUSIC I only allowed for monophonic triangle waves with controllable volume, pitch, and timing [12]. Would anyone today describe such a system as capable of producing "almost any sound"?

Even when considering more contemporary applications, processes like ring modulation and amplitude modulation, or distortion often cause aliasing artifacts unless proper oversampling is applied. These artifacts occur because PCM, while universally suitable for reproducing recorded sound, is not inherently versatile as a medium for generating new sounds. As Puckette argues, alternative representations, for instance, representation by a sequence of linear segments or physical modeling synthesis, offer other possibilities [21]. Therefore, PCM is not a completely universal tool for creating sound.

3. WHAT DOES THE UNIT GENERATOR HIDE?

Beginning with Version III, MUSIC took the form of a block diagram compiler that processes two input sources: a score language, which represents a list of time-varying parameters, and an orchestra language, which describes the connections between **unit generator (UGen)** such as oscillators and filters. In this paper, the term "UGen" refers to a signal processing module whose implementation is either not open or written in a language different from the one used by the user.

The MUSIC family, in the context of computer music research, achieved success for performing sound synthesis based on PCM but this success came with the establishment of a division of labor between professional musicians and computer engineers through the development of domain-specific languages. Mathews explained that he developed a compiler for MUSIC III in response to requests from many composers for additional features in MUSIC II, such as envelopes and vibrato, while also ensuring that the program would not be restricted to a specialized form of musical expression [22, 13:10-17:50]. He repeatedly stated that his role was that of a scientist rather than a musician:

> When we first made these music programs the original users were not composers; they were the psychologist Guttman, John Pierce, and myself, who are fundamentally scientists. We wanted to have musicians try the system to see if they could learn the language and express themselves with it. So we looked for adventurous musicians and composers who were willing to experiment. [12, p17]

[1] Sterne later critiques the phenomenological basis of acousmatic listening, which presupposes an idealized, intact body as the listening subject. He proposes a methodology of political phenomenology centered on impairment, challenging these normative assumptions [19]. Discussions of universality in computer music should also address ableism, particularly in relation to recording technologies and auditory disabilities.

This clear delineation of roles between musicians and scientists became one of the defining characteristics of post-MUSIC computer music research. Paradoxically, although computer music research aimed to create sounds never heard before, it also paved the way for further research by allowing musicians to focus on composition without having to understand the cumbersome work of programming.

3.1 Example: Hiding Internal State Variables in Signal Processing

Although the MUSIC N series shares a common workflow of using a score language and an orchestra language, the actual implementation of each programming language varies significantly, even within the series.

One notable but often overlooked example is MUSIGOL, a derivative of MUSIC IV [23]. In MUSIGOL, the system, the score and orchestra defined by user were written entirely as ALGOL 60 language. Similar to today's Processing or Arduino, MUSIGOL is one of the earliest internal domain-specific languages (DSL) for music; thus, it is implemented as an library [2]. (According to the definition in this paper, MUSIGOL does not qualify as a language that uses UGen.)

The level of abstraction deemed intuitive for musicians varied across different iterations of the MUSIC N series. This can be illustrated by examining the description of a second-order band-pass filter. The filter mixes the current input signal S_n, the output signal from t time steps prior O_{n-t}, and an arbitrary amplitude parameter I_1, as shown in the following equation:

$$O_n = I_1 \cdot S_n + I_2 \cdot O_{n-1} - I_3 \cdot O_{n-2}$$

In MUSIC V, this band-pass filter can be used as shown in Listing 1 [25, p.78]. Here, `I1` represents the input bus, and `O` is the output bus. The parameters `I2` and `I3` correspond to the normalized values of the coefficients I_2 and I_3, divided by I_1 (as a result, the overall gain of the filter can be greater or less than 1). The parameters `Pi` and `Pj` are normally used to receive parameters from the score, specifically among the available `P0` to `P30`. In this case, however, these parameters are repurposed as general-purpose memory to temporarily store feedback signals. Similarly, other UGens, such as oscillators, reuse note parameters to handle operations like phase accumulation. As a result, users needed to manually calculate feedback gains based on the desired frequency characteristics [3], and they also had to considder at least two sample memory spaces.

On the other hand, in newer MUSIC 11, and its successor, Csound, by Barry Vercoe, the band-pass filter is defined as a UGen named `reson`. This UGen takes four parameters: the input signal, center cutoff frequency, bandwidth, and Q factor [26, p248]. Unlike previous implementations, users no longer need to calculate coefficients manually, nor do they need to be aware of the two-sample memory space. However, in MUSIC 11 and Csound, it is possible to implement this band-pass filter from scratch as a user-defined opcode (UDO) as shown in Listing 2. Vercoe emphasized that while signal processing primitives should allow for low-level operations, such as single-sample feedback, and eliminate black boxes, it is equally important to provide high-level modules that avoid unnecessary complexity ("avoid the clutter") when users do not need to understand the internal details [26, p.247].

```
FLT I1 O I2 I3 Pi Pj;
```

Listing 1. Example of the use of FLT UGen in MUSIC V.

```
      instr  1
la1   init   0
la2   init   0
i3    = exp(-6.28 * p6 / 10000)
i2    = 4*i3*cos(6.283185 * p5/10000) / (1+
    i3)
i1    = (1-i3) * sqrt(1-1 - i2*i2/(4*i3))
a1    rand   p4
la3   =      la2
la2   =      la1
la1   =      i1*a1 + i2 * la2 - i3 * la3
      out    la1
      endin

      instr 2
a1    rand   p4
a1    reson  a1,p5,p6,1
      endin
```

Listing 2. Example of scratch implementation and built-in operation of RESON UGen respectively, in MUSIC11. Retrieved from the original paper. (Comments are omitted owing to space restriction.)

On the other hand, in succeeding environments that inherit the UGen paradigm, such as Pure Data [27], Max (whose signal processing functionalities were ported from Pure Data as MSP), SuperCollider [28], and ChucK [29], primitive UGens are implemented in general-purpose languages such as C or C++ [4]. If users wish to define low-level UGens (called external objects in Max and Pd), they need to set up a development environment for C or C++.

When UGens are implemented in low-level languages like C, even if the implementation is open-source, the division of knowledge effectively forces users (composers) to treat UGens as black boxes. This reliance on UGens as black boxes reflects and deepens the division of labor between musicians and scientists that was established in MUSIC, though it can be interpreted as both a cause and a result.

For example, Puckette, the developer of Max and Pure Data, notes that the division of labor at IRCAM between researchers, musical assistants (realizers), and composers has parallels in the current Max ecosystem, where roles are divided among Max developers themselves, developers of external objects, and Max users [31]. As described in the ethnography of 1980s IRCAM by anthropologist Georgina Born, the division of labor between fundamental research scientists and composers at IRCAM was extremely clear.

[2] While MUS10, used at Stanford University, was not an internal DSL, it was created by modifying an existing ALGOL parser [24, p.248].

[3] It is said that a preprocessing feature called `CONVT` could be used to transform frequency characteristics into coefficients [25, p77].

[4] ChucK later introduced ChuGen, which is similar extension to Csound's UDO, allowing users to define UGens within the ChucK language itself [30]. However, not all existing UGens are replaced by UDOs by default both in Csound and ChucK, which remain supplemental features possibly because the runtime performance of UDO is inferior to natively implemented UGens.

This structure was also tied to the exclusion of popular music and its associated technologies from IRCAM's research focus [32].

However, such divisions are not necessarily the result of differences in values along the axes analyzed by Born, such as modernist/postmodernist/populist or low-tech/high-tech distinctions [5] . This is because the black-boxing of technology through the division of knowledge occurs in popular music as well. Paul Théberge pointed out that the "democratization" of synthesizers in the 1980s was achieved through the concealment of technology, which transformed musicians as creators into consumers.

> Lacking adequate knowledge of the technical system, musicians increasingly found themselves drawn to prefabricated programs as a source of new sound material. (. . .)it also suggests a reconceptualization on the part of the industry of the musician as a particular type of consumer. [34, p.89]

This argument can be extended beyond electronic music to encompass computer-based music in general. For example, media researcher Lori Emerson noted that although the proliferation of personal computers began with the vision of a "metamedium"—tools that users could modify themselves, as exemplified by Xerox PARC's Dynabook—the vision was ultimately realized in an incomplete form through devices such as the Macintosh and iPad, which distanced users from programming by black-boxing functionality [35]. In fact, Alan Kay, the architect behind the Dynabook concept, remarked that while the iPad's appearance may resemble the ideal he originally envisioned, its lack of extensibility through programming rendered it merely a device for media consumption [36].

Musicians have attempted to resist the consumeristic use of those tools through appropriation and exploitation [37]. However, just as circuit bending has been narrowed down to its potential by a literal black box - one big closed IC of aggregated functions [38, p225], and glitching has shifted from methodology to a superficial auditory style [**?**], capitalism-based technology expands in a direction that prevents users from misusing it. Under these circumstances, designing a new programming language does not merely provide musicians with the means to create new music, but is itself contextualized as a musicking practice following hacking, an active reconstruction of the technological infrastructure that is allowed to be hacked.

4. CONTEXT OF PROGRAMMING LANGUAGES FOR MUSIC AFTER 2000

Under this premise, music programming languages developed after the 2000s can be categorized into two distinct directions: those that narrow the scope of the language's role by introducing alternative abstractions at a higher-level, distinct from the UGen paradigm, and those that expand the general-purpose capabilities of the language, reducing black-boxing.

[5] David Wessel revealed that the individual referred to as RIG in Born's ethnography was himself and commented that Born oversimplified her portrayal of Pierre Boulez, then director of IRCAM, as a modernist. [33]

Languages that pursued alternative higher-level abstractions have evolved alongside the culture of live coding, where performances are conducted by rewriting code in real time. The activities of the live coding community, including groups, such as TOPLAP, since the 2000s, were not only about turning coding itself into a performance but also served as a resistance against laptop performances that relied on black-boxed music software. This is evident in the community's manifesto, which states, "Obscurantism is dangerous" [39].

Languages implemented as clients for SuperCollider, such as **IXI** (on Ruby) [40], **Sonic Pi** (on Ruby), **Overtone** (on Clojure) [41], **TidalCycles** (on Haskell) [42], and **FoxDot** (on Python) [43], leverage the expressive power of more general-purpose programming languages. While embracing the UGen paradigm, they enable high-level abstractions for previously difficult-to-express elements like note values and rhythm. For example, the abstraction of patterns in TidalCycles is not limited to music but can also be applied to visual patterns and other outputs, meaning it is not inherently tied to PCM-based waveform output as the final result.

On the other hand, owing to their high-level design, these languages often rely on ad-hoc implementations for tasks like sound manipulation and low-level signal processing, such as effects. McCartney, the developer of SuperCollider, stated that if general-purpose programming languages were sufficiently expressive, there would be no need to create specialized languages [1], which appears reasonable when considering examples like MUSIGOL. However, in practice, scripting languages that excel in dynamic program modification face challenges in modern preemptive operating system (OS) environments. For instance, dynamic memory management techniques such as garbage collection can hinder deterministic execution timing required for real-time processing [44].

Historically, programming languages, such as FORTRAN or C, served as a portable way for implementing programs across different architectures. However, with the proliferation of higher-level languages, programming in C or C++ has become relatively more difficult, akin to assembly language in earlier times. Furthermore, considering the challenges of portability not only across different CPUs but also diverse host environments such as OSs and the Web, these languages are no longer as portable as they once were. Consequently, internal DSL for music, including signal processing, have become exceedingly rare, with only a few examples such as LuaAV [45].

Instead, an approach has emerged to create general-purpose languages specifically designed for use in music from the ground up. One prominent example is **Extempore**, a live programming environment developed by Sorensen [46]. Extempore consists of Scheme, a Lisp-based language, and xtlang, a meta-implementation on top of Scheme. While xtlang requires users to write hardware-oriented type signatures similar to those in C, it leverages the compiler infrastructure, LLVM [47], to just-in-time (JIT) compile signal processing code, including sound manipulation, into machine code for high-speed execution.

The expressive power of general-purpose languages and compiler infrastructures, such as LLVM, has given rise to an approach focused on designing languages with math-

ematical formalization that reduces black-boxing. **Faust** [48], for instance, is a language that retains a graph-based structure akin to UGens but is built on a formal system called Block Diagram Algebra. Thanks to its formalization, Faust can be transpiled into various low-level languages, such as C, C++, or Rust, and can also be used as external objects in Max or Pure Data.

Languages like **Kronos** [49] and **mimium** [50], which are based on the more general computational model of lambda calculus, focus on PCM-based signal processing while exploring interactive meta-operations on programs [51] and balancing self-contained semantics with interoperability with other general-purpose languages [52].

DSLs are constructed within a double bind; they aim to specialize in a particular purpose while still providing a certain degree of expressive freedom through coding. In this context, efforts like Extempore, Kronos, and mimium are not merely programming languages for music but are also situated within the broader research context of functional reactive programming, which focuses on representing time-varying values in computation. Most computing models lack an inherent concept of real-time, operating instead based on discrete computational steps. Similarly, low-level general-purpose programming languages do not natively include primitives for real-time concepts. Consequently, the exploration of computational models tied to time —a domain inseparable from music— remains vital and has the potential to contribute to the theoretical foundations of general-purpose programming languages.

However, strongly formalized languages come with another trade-off. Although they allow UGens to be defined without black-boxing, understanding the design and implementation of these languages often requires expert knowledge. This can create a deeper division between language developers and users, in contrast to the many but small and shallow divisions seen in the multi-language paradigm, such as SuperCollider developers, external UGen developers, client language developers (e.g., TidalCycles), SuperCollider users, and client language users.

Although there is no clear solution to this trade-off, one intriguing idea is the development of self-hosting languages for music—that is, languages whose compilers are written in the language itself. At first glance, this may seem impractical. However, by enabling users to learn and modify the language's mechanisms spontaneously, this approach could create an environment that fosters deeper engagement and understanding among users.

5. CONCLUSION

This paper has reexamined the history of computer music and music programming languages with a focus on the universalism of PCM and the black-boxing tendencies of the UGen paradigm. Historically, it was expected that the clear division of roles between engineers and composers would enable the creation of new forms of expression using computers. Indeed, from the perspective of post-acousmatic discourse, some scholars, such as Holbrook and Rudi, still consider this division to be a positive development:

> Most newer tools abstract the signal processing routines and variables, making them easier to use while removing the need for understanding the underlying processes in order to create meaningful results. Composers no longer necessarily need mathematical and programming skills to use the technologies. [10, p2]

However, this division of labor also creates a shared vocabulary (as exemplified in the UGen by Mathews) and serves to perpetuate it. By portraying new technologies as something externally introduced, and by focusing on the agency of those who create music with computers, the individuals responsible for building programming environments, software, protocols, and formats are rendered invisible [53]. This leads to an oversight of the indirect power relationships produced by these infrastructures.

For this reason, future research on programming languages for music must address how the tools, including the languages themselves, contribute aesthetic value within musical culture (and what forms of musical practice they enable), as well as the social (im)balances of power they produce.

The academic value of the research on programming languages for music is often vaguely asserted, using terms such as "general," "expressive," and "efficient." However, it is difficult to argue these claims when processing speed is no longer the primary concern. Thus, as with idiomaticity [3] by McPherson et al., we need to develop and share a vocabulary for understanding the value judgments we make about music languages.

In a broader sense, the development of programming languages for music has also expanded to the individual level. Examples include **Gwion** by Astor, which is inspired by ChucK and enhances its abstraction through features, such as lambda functions [54]; **Vult**, a DSP transpiler language created by Ruiz for his modular synthesizer hardware [55]; and a UGen-based live coding environment designed for web, **Glicol** [56]. However, these efforts have not yet been incorporated into academic discourse.

Conversely, practical knowledge of past languages in 1960s, as well as real-time hardware-oriented systems from the 1980s, is gradually being lost. Although research efforts such as *Inside Computer Music*, which analyzes historical works of computer music, have begun [57], an archaeological practice focused on the construction of computer music systems themselves will also be necessary.

Acknowledgments

This research was supported by JSPS KAKENHI (Grant No.23K12059). I would also like to thank the many anonymous reviewers.

6. REFERENCES

[1] J. McCartney, "Rethinking the Computer Music Language: SuperCollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, Dec. 2002.

[2] H. Nishino and R. Nakatsu, "Computer Music Languages and Systems: The Synergy Between Technology and Creativity," in *Handbook of Digital Games and Entertainment Technologies*, 2016.

[3] A. McPherson and K. Tahlroğlu, "Idiomatic Patterns and Aesthetic Influence in Computer Music Languages," *Organised Sound*, vol. 25, no. 1, pp. 53–63, 2020.

[4] R. B. Dannenberg, "Languages for Computer Music," *Frontiers in Digital Humanities*, vol. 5, Nov. 2018.

[5] V. Lazzarini, "The Development of Computer Music Programming Systems," *Journal of New Music Research*, vol. 42, no. 1, pp. 97–110, 2013.

[6] M. Mathews, "The Digital Computer as a Musical Instrument," *Science, New Series*, vol. 142, no. 3592, pp. 553–557, Nov. 1963.

[7] B. Ostertag, "Why Computer Music Sucks," https://web.archive.org/web/20160312125123/http://bobostertag.com/writings-articles-computer-music-sucks.htm, 1998.

[8] E. Lyon, "Do We Still Need Computer Music?" https://disis.music.vt.edu/eric/LyonPapers/Do_We_Still_Need_Computer_Music.pdf, 2006.

[9] M. Adkins, R. Scott, and P. A. Tremblay, "Post-Acousmatic Practice: Re-evaluating Schaeffer's Heritage," *Organised Sound*, vol. 21, no. 2, pp. 106–116, Aug. 2016.

[10] U. Holbrook and J. Rudi, "Computer Music and Post-Acousmatic Practices: International Computer Music Conference 2022," in *Proceedings of the International Computer Music Conference, ICMC 2022*, G. Torre, Ed. International Computer Music Association, Jul. 2022, pp. 140–144.

[11] T. Magnusson, "Of Epistemic Tools: Musical Instruments as Cognitive Extensions," *Organised Sound*, vol. 14, no. 2, pp. 168–176, Aug. 2009.

[12] M. Mathews and C. Roads, "Interview with Max Mathews," *Computer Music Journal*, vol. 4, no. 4, pp. 15–22, 1980.

[13] P. Doornbusch, "Early Computer Music Experiments in Australia and England," *Organised Sound*, vol. 22, no. 2, pp. 297–307, Aug. 2017.

[14] R. D. Woltman, F. B. Woltman, L. D. Wilson, A. B. Tonik, J. K. Swearingen, C. M. Shuler, J. E. Sberro, J. E. Sammet, H. W. Matter, D. W. Marquardt, F. K. Koons, M. W. Huff, F. E. Holberton, C. Hammer, D. B. Dixon, E. L. Delves, G. Danehower, M. P. Chinitz, L. S. Carter, J. Bartik, L. W. Armstrong, D. P. Armstrong, and A. E. Adams, "UNIVAC Conference." Charles Babbage Institute, Tech. Rep., 1990.

[15] D. Davis, "Very Early Computer Music," *Resurrection The Bulletin of the Computer Conservation Society*, vol. 10, pp. 19–20, 1994.

[16] S. Miyazaki, "Algorhythmic Listening 1949-1962 Auditory Practices of Early Mainframe Computing," in *AISB/IACAP World Congress 2012: Symposium on the History and Philosophy of Programming, Part of Alan Turing Year 2012*, 2012, p. 5.

[17] S. Levy, *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*, 1st ed. O'Reilly Media, May 2010.

[18] D. G. Loy, "Life and Times of the Samson Box," *Computer Music Journal*, vol. 37, no. 3, pp. 26–48, 2013.

[19] J. Sterne, *Diminished Faculties: A Political Phenomenology of Impairment*. Durham: Duke Univ Press, Jan. 2022.

[20] ——, *The Audible Past: Cultural Origins of Sound Reproduction*. Durham: Duke University Press, 2003.

[21] M. Puckette, "The Sampling Theorem and Its Discontents," *International Computer Music Conference*, pp. 1–14, 2015.

[22] M. V. Mathews, "Max Mathews Full Interview | NAMM.Org," https://www.namm.org/video/orh/max-mathews-full-interview, Mar. 2007.

[23] D. M. Innis, "Sound Synthesis by Computer: Musigol, a Program Written Entirely in Extended Algol," *Perspectives of New Music*, vol. 7, no. 1, pp. 66–79, 1968.

[24] G. Loy and C. Abbott, "Programming Languages for Computer Music Synthesis, Performance, and Composition," *ACM Comput. Surv.*, vol. 17, no. 2, pp. 235–265, Jun. 1985.

[25] M. V. Mathews and J. E. Miller, *The Technology of Computer Music*. M.I.T. Press, 1969.

[26] B. L. Vercoe, "Computer Systems and Languages for Audio Research," *The New World of Digital Audio (Audio Engineering Society Special Edition)*, pp. 245–250, 1983.

[27] M. Puckette, "Pure Data," in *International Computer Music Conference Proceedings*. Michigan Publishing, University of Michigan Library, 1997.

[28] J. McCartney, "SuperCollider, a New Real Time Synthesis Language," in *International Computer Music Conference Proceedings*. Michigan Publishing, 1996.

[29] G. Wang, P. R. Cook, and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, 2015.

[30] S. Salazar and G. Wang, "ChuGens, ChubGraphs, ChuGins: 3 Tiers for Extending ChucK," in *International Computer Music Conference Proceedings*, 2012, pp. 60–63.

[31] I. Reese, "47 • Miller Puckette • Max/MSP & Pure Data," https://futureofcoding.org/episodes/047.html, May 2020.

[32] G. Born, *Rationalizing Culture*. University of California Press, 1995, no. 1.

[33] G. Taylor, "Article: An Interview With David Wessel | Cycling '74," https://cycling74.com/articles/an-interview-with-david-wessel, 1999.

[34] P. Théberge, *Any Sound You Can Imagine: Making Music/Consuming Technology*, ser. Music/Culture. Hanover, NH: Wesleyan University Press : University Press of New England, 1997.

[35] L. Emerson, *Reading Writing Interfaces: From the Digital to the Bookbound*. Univ of Minnesota Press, Nov. 2014.

[36] A. C. Kay, "American Computer Pioneer Alan Kay's Concept, the Dynabook, Was Published in 1972. How Come Steve Jobs and Apple iPad Get the Credit for Tablet Invention?" https://www.quora.com/American-computer-pioneer-Alan-Kay-s-concept-the-Dynabook-was-published-in-1972-How-come-Steve-Jobs-and-Apple-iPad-get-the-credit-for-tablet-invention, Apr. 2019.

[37] C. Kelly, *Cracked Media: The Sound of Malfunction*. Cambridge, Mass: MIT Press, 2009.

[38] A. Inglizian, *Beyond Bending - Triggering, Sequencing, and Modulating Circuit-Bent Toys*, 3rd ed. New York: Routledge, Jun. 2020.

[39] TOPLAP, "ManifestoDraft - Toplap," https://toplap.org/wiki/ManifestoDraft, 2004.

[40] T. Magnusson, "The IXI Lang: A SuperCollider Parasite for Live Coding," *International Computer Music Conference Proceedings*, vol. 2011, 2011.

[41] S. Aaron and A. F. Blackwell, "From Sonic Pi to Overtone: Creative Musical Experiences with Domain-Specific and Functional Languages," *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, pp. 35–46, 2013.

[42] A. McLean, "Making Programming Languages to Dance to: Live Coding with Tidal," in *FARM 2014 - Proceedings of the 2014 ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design*. New York, New York, USA: Association for Computing Machinery, 2014, pp. 63–70.

[43] R. Kirkbride, "FoxDot: Live Coding with Python and Supercollider," in *Proceedings of the International Conference on Live Interfaces*, 2016, pp. 194–198.

[44] R. B. Dannenberg and R. Bencina, "Design Patterns for Real-Time Computer Music Systems," in *ICMC 2005 Workshop on Real Time Systems Concepts for Computer Music*, 2005.

[45] G. Wakefield, W. Smith, and C. Roberts, "LuaAV: Extensibility and Heterogeneity for Audiovisual Computing," in *Proceeding of Linux Audio Conference*, 2010.

[46] A. C. Sorensen, "Extempore: The Design, Implementation and Application of a Cyber-Physical Programming Language," Ph.D. dissertation, The Australian National University, 2018.

[47] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004, p. 75.

[48] Y. Orlarey, D. Fober, S. Letz, and S. Letz, "FAUST : An Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*. Delatour France, 2009.

[49] V. Norilo, "Kronos: A Declarative Metaprogramming Language for Digital Signal Processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[50] T. Matsuura and K. Jo, "Mimium: A Self-Extensible Programming Language for Sound and Music," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design*, ser. FARM 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 1–12.

[51] V. Norilo, "Kronos Meta-Sequencer – From Ugens to Orchestra, Score and Beyond," in *Proceedings of the International Computer Music Conference*, 2016, pp. 117–122.

[52] T. Matsuura, "Lambda-Mmm: The Intermediate Representation for Synchronous Signal Processing Language Based on Lambda Calculus," in *Proceedings of the 4th International Faust Conference*, 2024, pp. 17–25.

[53] J. Sterne, "There Is No Music Industry," *Media Industries Journal*, vol. 1, no. 1, pp. 50–55, Jan. 2014.

[54] J. Astor, "Gwion," https://github.com/Gwion/Gwion, 2017.

[55] L. L. Ruiz, "Vult Language," http://modlfo.github.io/vult/, 2020.

[56] Q. Lan, "Glicol," https://glicol.org/, 2020.

[57] M. Clarke, F. Dufeu, and P. Manning, *Inside Computer Music*. New York: Oxford Univ Pr, Sep. 2020.