Journal of the Japanese Society for Sonic Arts, Vol.5 No.2 pp.1–8

Research Report

PRAGMATIC AESTHETIC EVALUATION OF ABSTRACTIONS FOR LIVE CODING

Renick Bell
Tama Art University

ABSTRACT

Abstraction, meaning aggregation, generalization, and instantiation to facilitate programming, can improve live coding performances in some regards, but it also carries disadvantages. This becomes apparent through an analysis of live coding with a pragmatic aesthetic theory, both conceptually and more concretely through an examination of some abstractions in the Conductive live coding library for the Haskell programming language. That analysis shows the convenience of using those library functions, but it also exposes their potential opacity for audiences. As the level of abstraction increases, the likelihood that audiences understand the purpose or mechanism of the abstractions employed can decrease. From a pragmatic viewpoint, whether this impacts the value of the abstractions depends on factors including the performer's goals and the target audience.

1. INTRODUCTION

Improvising with generative processes is believed to make possible much more than a solo performer can achieve on traditional musical instruments. To do so, a library for the Haskell programming language called Conductive was developed [4]. It facilitates live coding in Haskell by abstracting control data for triggering other synthesizers. The library implements abstractions, the need for which became apparent through a pragmatic aesthetic analysis.

An explanation of how they resulted relies on definitions of abstraction and live coding and a pragmatic aesthetic theory. Following that explanation, the system is applied to live coding abstractions in a general or conceptual way. An examination of some abstractions in the Conductive library for live coding makes the application of the aesthetic theory more concrete. Finally, some conclusions and directions for future research are described.

2. DEFINITION OF ABSTRACTION

The following working definitions of abstraction are synthesized from various sources [13, 15, 18, 20, 25, 26, 28]

1. (uncountable) Abstraction is the process of aggregating or generalizing to achieve a representation of something in order to make programming easier
2. (countable) an abstraction is a representation of something achieved through aggregation or generalization that allows instantiation of itself in order to make programming easier

2.1. Higher-level Abstractions

In programming, abstraction exists in layers. It be said that there are low-level abstractions and high-level abstractions. Dijsktra recommends progressively refining the entities of a program in layers to ensure the correctness of the resulting program. Keller describes program structure as "a layered hierarchy of machines" [18].

Simonyi defines higher level abstractions as those covering the "essential qualities of the program" that directly work toward the targeted solution. These are a separate category from accidental detail (a phrasing he borrows from Brooks), which includes things changeable for reasons such as optimization and compatibility. Even higher-level abstractions can be developed which can handle two different but similar problem statements, leading to increased code reuse [27].

3. LIVE CODING

This paper defines live coding as the interactive control of algorithmic processes through programming activity, a definition derived from Brown, Collins, and Ward [10, 11, 31]. This paper will not consider programming in front of

others as a tutorial or make a distinction between public performances and solitary coding.

According to TOPLAP, live coding history dates back to the 1980s. In the 21st century, performers began projecting their programming activity on screens as a custom. Now an increasing variety of live coding systems are used in performances [24] and increased public live coding activity takes place [23, 34].

Live coding enables a more abstract manipulation of a representation of music than physical gestures used for playing instruments. It is also thought to be more convenient in many regards than windows-icon-mouse-pointer software [3]. From another perspective, it takes the potential of algorithmic composition and turns it into a live performance rather than a write/compile/run loop from traditional software development or electronic music composition.

Brown explains that time constraints in performances require concisely-notated and appropriate abstractions to produce coherent non-static musical content [10].

4. A PRAGMATIC AESTHETIC THEORY

This working version of an aesthetic system, based on Dewey's "Art as Experience" [12], is provided. Much of this section was originally presented in [5] with more discussion. Additional material on value has been added in this paper.

An affect is a kind of emotional state. An affectee is a person experiencing affects in an interaction with affectors. An affector is a percept that stimulates affects in an affectee. It can be a physical object or something abstract. A work of art is an affector which in some way was created, organized, or manipulated by a person for the purpose of being an affector. A person involved with the creation or arrangement of an affector is an artist.

An art experience is the experience of affects in an affectee as the result of the affectee's interaction with a network of affectors, with at least one of those affectors being a work of art. The art experience is the experience of those affectors either simultaneously or in sequence. Experience involves a possibly infinite number of affectors arrayed in a network structure in which they influence each other. Changing the perceived network of affectors changes the nature of the experience.

The value of an affector is related to the value of an art experience in which it is involved. The value of an art experience is determined by the affects experienced.

For Dewey, judgment is both creative in that it generates new ends and transformative in that judgments can cause reevaluation of what we have up to that moment prized. In this way, what is valued constantly evolves in an experimental way as increasingly informed judgments are made [2].

Dewey would have evaluations of aesthetic experience used to enhance experiences through increased awareness of the causes and effects in the experience. Recognition of a unity of means and ends in an experience is an important factor in the assignment of value. What enhances an experience is context-dependent and subject to revision as new knowledge is apprehended [2].

5. APPLYING THE AESTHETIC THEORY TO ABSTRACTIONS IN LIVE CODING

Live coding consists of a network of affectors such as the musical output, including rhythms, timbres, harmony, and rate of change, programming languages and libraries used, projection contents, and performance space [5]. The network of affectors perceived by different affectees, such as the performer, music fans, programmers, or just the curious, and the affects they produce depend on each person's attention, knowledge, and experience.

Abstractions and their features are affectors in the network, either directly perceived or indirectly felt as a result of their influence on other affectors. Their difficulty of use can impair the other affectors, or enhance them when they are well-conceived. Second, they can also be opaque, hiding their contents from people lacking particular knowledge which would otherwise permit recognition of the true role of the abstraction in the experience. Other affectors can impact whether and to what degree abstractions are perceived. Affectors which impact that perception include the size and clarity of the projection, the type of sounds that are heard simultaneously, whether the text in the editor is scrolling or not, and so on.

5.1. the limiting influence of abstractions

Abstractions can be a negative limiting factor on the primary affectors in the musical experience: rhythm, form, melody, harmony, and so on. Abstractions are recognized as potentially hard to learn and use, and that difficulty affects what a performer can achieve. The inherent limitations posed by the representative nature of abstractions in programming for music is acknowledged in chapter 4 of Magnusson [22]. Further limitations described in Green and Blackwell include potential for

errors in creation, extended creation time, and effort of maintenance [15]. They emphasize the cognitive effort required for using them, making them possibly ill-suited for exploratory design. That cognitive effort was previously recognized by Brooks [17]. Selecting the correct abstraction is difficult, as is changing them without breaking the system [9].

This points to the commonly said phrase "all abstractions are leaky". Spolsky gives his "Law of Leaky Abstractions": "All non-trivial abstractions, to some degree, are leaky." [30] Spolsky explains that while abstractions make working more convenient, they take time to learn. The level of the abstraction contributes to its leakiness.

5.2. abstractions as affectors

The abstractions themselves might be perceived as affectors. Each individual abstraction, once perceived, is taken in a different way depending on the person. One major factor is the affectee's knowledge. An example is the use of >> in coding done by this author. First, someone who is not familiar with the symbol may just ignore it. However, some may guess. A non-programmer might interpret it as an arrow, pointing a direction. It might give an impression of movement. Another might think of it like a quoting symbol in an email. A programmer with Unix experience might initially think it is a redirect, as in: ls >> files.txt. Another might see it repeated several times in a line and wonder if it is a pipe. A Haskell programmer will know it has a type signature like this:

(>>) :: forall a b. m a -> m b -> m b

The documentation describes it as used to "Sequentially compose two actions, discarding any value produced by the first, like sequencing operators (such as the semicolon) in imperative languages." [35] Three types of people leads to three different experiences of a single symbol, pointing at the leaky nature of abstractions.

When implemented, abstractions are accompanied by notation. Green describes the cognitive dimensions of notation. Achieving proper role-expressiveness seems to rely heavily on the notation accompanying the abstraction: choosing the proper names, including characteristics such as morphology, length, and grammatical sensibility, for the abstractions and their parameters is important [21]. For example, a function can be named "raveBassDrop" instead of "rd1" to inform the audience of an intended musical effect. A performer intending to be cryptic might do the reverse, like in Zmoelnig's "Pointillism" [33].

For a simple example, consider a function producing a synthesis event requiring two parameters. Notation for the underlying abstraction (the same in both cases) can vary wildly. Each function call has its own advantages and disadvantages:

- myFMSynthesizer (SynthesizerEventData { amplitude = 1.0, modulation = 0.5})
- syn 1.0 0.5

Note that though the former says much more about how it used and to what purpose, it is unwieldy. The latter can be employed simply, but the audience is completely uninformed about the purpose of the two parameters.

The leakiness of abstractions for coders using them is greatly amplified for affectees other than the performer: coders unfamiliar with the abstraction are more likely to struggle to understand them, and that is increased to a much greater extent for non-coders.

Still, the elegance with which an abstraction solves its target problem can make the abstraction an affector in an art experience. For example, a live coder trying to stimulate an audience of coders might write functions live which exploit recursion, currying, or other techniques to elegantly express musical content, and do so from scratch to exhibit skill for affectees who are capable of understanding them.

5.3. conclusions about the use of abstractions

Kramer writes that the "level, benefit and value of a particular abstraction depend on its purpose." [19] Said another way, abstractions have to be chosen carefully with the goal firmly in mind. A continual refinement of those abstractions toward an evolving ideal is generally desired. A live coder choses purposes for the performance and then tries to achieve abstractions as close to ideal for those purposes as possible, revising both the abstractions and the purposes as a result of experiencing this experimental process. Though their value is contextual, abstractions can be judged generally insofar as those general judgments prove useful. Given that, some general conclusions about abstractions follow.

Two targets are making higher-level abstractions so that the intended musical output is easier to achieve and designing abstractions and their notation so that the intention of the programmer regarding the relationship between the abstractions and the affectees is achieved.

Despite the potential drawbacks, higher-level abstractions can significantly improve the performer's ability to manipulate the generation of musical material. Well-

chosen and appropriate abstractions can make expressing musical content easier [1] or even make possible what would otherwise not be [14]. That improves both the experience for the performer and the audience.

Blackwell and Collins compare the usability of programming languages for live music with the interfaces of commercial music software and explain the challenges that users of ChucK face [8]. These challenges can likley be attributed to other live coding environments. Of particular interest here is the characterization of ChucK as diffuse [8]. Additional abstractions are given by Blackwell as the solution to this problem [15].

In answer to the problem of exploratory tasks in Green and Blackwell, it is precisely the challenges and rewards of exploratory activity with this material that provides some of the aesthetic value of live coding.

Green says that a "system = notation + environment". Used with a model of behavior, the system requirements can be stated and differing dimensions may emerge. This makes trade-offs clearer. Alternatively, it may be possible to modify the notation to counter the deficiencies of the environment, and vice versa [16].

Revising Green, it could be thought that abstractions are the link between the notation and the environment. The notation makes the abstractions representing the environment employable. Conceiving of the right abstractions may simplify the notation needed to accomplish a programming goal and or counter the deficiencies of the environment. It seems that it may be useful to consider the abstraction and its notation as separate but linked.

With typing speed being an issue in live coding, Brown and Sorensen sought compact code constructs [29]. Given sufficiently high-level abstractions, the speed of typing should become much less of an issue.

Users of abstractions can mitigate the problems of leakiness by understanding the underlying abstracted matter and choosing abstractions appropriately. Following Simonyi's recommendation, choosing preexisting abstractions from the musical domain as targets is a good method to reduce the difficulty of the mental operations of the programmer when using them. Returning to the list given by Shutt and extrapolating, it can be asked what kind of higher-level data types and control structures would benefit programming for music.

Berg gives the suggestions of "reduced instruction set composition" and "evocative simplifications of compositional activity". In describing Xenakis, he mentions shapes, areas, densities, and contours. Koenig is described as expressing compositional rules and tendencies. Berg also "behavior that changes over time" [7].

Other targets could include things such as briefly-expressed constructs for managing state or methods for managing time at a levels including sub-beat, phrase, and full composition. This reflects Shaw's suggestion to develop abstractions at a system-organization level [25].

Wishart provides a strong account of such potential targets. He describes characteristics of sound and form, many of which are based on utterance and natural phenomenon. The role of the perception of sound and achieving freedom from Western notation are highlighted [32] (see p. 326). He refers to "lattice sonics" (sounds using organizational methods related to grids) and those that fall outside of grid-based expression [32] (see p. 7-8). Mixing the two categories, some particular aspects sampled at random and listed alphabetically include articulation, counterpoint, density, distribution in space, dynamism, gesture, grain, landscape, masking, mass, modulation, morphology, nesting, permutation, space nature, stability, timbre, time-based transformation methods, topology, types, and so on [32].

It can be argued that the flexibility of such abstractions should be kept in mind by library developers so that developed abstractions do not unduly color the resulting compositions. They should be general enough that a number of genres could employ them, and according to Brooks' warning, their frequency of use should be potentially high to avoid being esoteric. However, another programmer may desire very unique abstractions that would somehow give performances a distinct flavor. The intentions of the programmer can be seen as critical.

Increasing the awareness of abstractions must be a factor in the depth of an experience of live coding. For some audiences, a performance could survive aesthetically on compelling use of advanced abstractions, while for other audiences the audio output would trump any consideration of the abstractions employed. It is up to the performer to adjust for those audiences accordingly. The intention of the performer then becomes very important. If the goal is clarity of expression of the mechanisms of the performance for the audience, then the abstraction and its notation need particular care. If the intention is simply to show the audience the activity of the performer through a projection (or even to perform with no projection at all), the abstraction just needs to suit the needs of the performer for expressing the desired output. It is not clear that these two are complementary, and cases where they may be in conflict can be imagined. In the "syn" example above, the performer can type "syn" and two numeric parame-

ters much faster than the longer function name and the notation for the data structure parameter. What is an advantage for the performer's speed is a disadvantage for the audience's understanding.

6. CONDUCTIVE CONCEPTS

Some abstractions from the Conductive library for the Haskell programming language will be presented below. They are intended to solve the time-constraint and other problems described in Brown above. For a more complete explanation of these abstractions, see [3] and [6]. The aesthetic theory presented above was used to judge the sonic output and the experience of using the functions from the library in live coding performance. The network of affectors were considered, and changes and additions were made to the library to get to the present state.

6.1. Players

The programming pattern of defining processes and having them repeat at various intervals seemed a fundamental one and a good target for abstraction. An abstraction called a Player was developed to manage these concurrent processes [3]. An analogy for a Player could be a solo performer or a person in an ensemble, or it can be thought of like a media player.

A Player initiates a process, waits for a period of time, initiates another process, waits for another period of time, and repeats this as long as it is running. The processes that a Player initiates are described in action functions. At present, the action functions in this author's performances are triggering sample-playing synthdefs in scsynth (SuperCollider synthesis engine).

It seemed desirable to create a separation of concerns in which the events and their timing could be dealt with independently, one changed without affecting the other and in which one, the other, or both could be reused by other concurrent processes. Another benefit of this separation is increased role-expressiveness, where the function for time is clearly labeled as an IOI function, as is the function generating events (as an action function).

The time that a player waits between initiating events is called an interonset interval (IOI). It is possible than an event has not finished running when the next event starts. An IOI is expressed in terms of beats, which in turn are based on a TempoClock. This IOI is determined by an IOI function which the Player refers to. It calculates the next beat that it should run its action function on.

A simple technique employed in an IOI function is looping through a list, successively returning each value as the current IOI, but the design is up to the user. Writing such lists by hand during a performance took too much time. As a result, abstractions were designed to make the generation of such lists easier.

A stochastic generator of IOI lists was developed which first makes a set of potential IOIs based on a core value. It then generates a set of subphrases based on those potential IOIs. It outputs a final phrase by choosing from those subphrases up to a user-specified length of time. This uses the concepts of repetition and variation to create arguably aesthetically-pleasing patterns.

6.2. density

A common activity which was observed was the specification of lists of IOI values to be used in sequence. These values used in sequence equate to a rhythm. For musical development, the density of events should be variable. In other words, some periods contain a higher ratio of events to length than others.

A higher-level abstraction was developed to generate a list and a set of rhythmically similar lists of greater and lesser density and store them in a table, called a density map, indexed by level of density. Doing so increased the likelihood that two lists would be perceived as having a rhythmic relationship and decreases the chance that an audience would perceive a kind of discontinuous state change when switching from one to another (as might happen with some techniques). The lists thus generated are stored in a table and retrieved when the table is queried with a density value.

6.3. TimespanMaps

Another commonly observed pattern was setting the timing of particular values. It is often desired that values change over time but at different rates. TimespanMaps are structures for handling such cases. Rather than specify the exact timing of a value, it specifies the range of time in which that value can occur. They are maps or dictionaries with intervals as keys to any kind of value. Another parameter of the structure is a specified length at which it loops. When a time is passed to the dictionary, the interval that time falls in is determined to be the key to use, and the corresponding value for that interval is returned. When the time value passed to the TimespanMap exceeds those for which it is defined, it

loops to return an appropriate value.

Convenience functions for TimespanMaps with random key values and interpolated TimespanMaps (in which fixed-length steps are linearly interpolated from a set of points in time) were developed to increase the speed with which TimespanMaps could be created.

7. EVALUATION

The Player data structure has been very useful in testing over the past two years. The flexible usage of action functions and IOI functions appears to make editing safer and more convenient, from the author's experience. This arrangement also made group operations on multiple Players possible, opening the possibility for abstractions that deal with sets of Players. It is a particularly appropriate abstraction if the user employs a representation of music similar to that of a traditional musical ensemble, like a four-piece band. However, it is not limited to such models, preserving a high degree of generality.

Assigning samples by using TimespanMaps reduced the number of Players. Varying rhythm became much easier. It became easier to control performances. The results became less random and more musical. Performances are more interesting than they were previously. The generality of TimespanMaps also appears to be high. It proves through usage to be a flexible way to represent time, and was seen as a useful tool even in places it was not initially intended for use. That is the result of the widespread occurrence of values that change with time.

Patterns from the pattern generator are processed into density maps, with interpolated density values held in TimespanMaps. A Player then uses an IOI function designed to retrieve an IOI using that stack of abstractions. This enables relatively fast generation of large amounts of rhythmic material and variations. However, this particular setup sometimes feels too specific to one compositional style, partly because the method for generating the variations of the input pattern was fixed. At the same time, it is almost totally opaque for audiences without prior knowledge of the abstractions. Such knowledge is only available by reading papers on the library or through personal conversation or correspondence with the author.

This set of representations alone is still seen to be too low-level, and consequently making changes with the desired level of liveness and fluidity using only these abstractions remains challenging. The complexity can also be hard to keep in one's head and manage. At the same time, audiences seem to have no understanding of how the abstractions work, and the font size at which they have been projected has further prevented their understanding. However, it is thought that even these implemented abstractions improved the usability of the associated live coding system, enabling performances that would have otherwise been much more difficult or even unachievable without them.

8. THE FUTURE

This heuristic for evaluating abstractions should be applied in a more analytical way to a greater number of abstractions and complete live coding performances. The nature of how value is assigned by an affectee should be investigated more deeply. The aesthetic theory can be refined further. Additional study needs to be done on what makes the intention or mechanism of an abstraction clear for affectees.

9. ACKNOWLEDGEMENTS

Much thanks goes to Akihiro Kubota and Yoshiharu Hamada for research support.

10. REFERENCES

[1] S. Aaron, A.F. Blackwell, R. Hoadley, and T. Regan, "A principled approach to developing new languages for live coding," in *Proceedings of New Interfaces for Musical Expression*, 2011, pp. 381–386.

[2] E. Anderson, "Dewey Moral Philosophy," in *The Stanford Encyclopedia of Philosophy*, Fall 2012., E.N. Zalta, Ed. 2012 [Online]. Available: `http://plato.stanford.edu/archives/fall2012/entries/dewey-moral/`. [Accessed: 03-Sep-2013]

[3] R. Bell, "An Interface for Realtime Music Using Interpreted Haskell," in *Proceedings of LAC 2011*, 2011.

[4] R. Bell, "conductive-base." dec-2012 [Online]. Available: `http://hackage.haskell.org/package/conductive-base`. [Accessed: 13-Feb-2013]

[5] R. Bell, "Towards Useful Aesthetic Evaluations of Live Coding," in *Proceedings of the International Computer Music Conference*, 2013.

[6] R. Bell, "An Approach to Live Algorithmic Composition using Conductive," in *Proceedings of LAC 2013*, 2013.

[7] P. Berg, "Abstracting the Future: The search for musical constructs," *Computer Music Journal*, vol. 20, no. 3,

pp. 24–27, 1996.

[8] A. Blackwell and N. Collins, "The Programming Language as a Musical Instrument," in *Proceedings of PPIG05*, University of Sussex, 2005.

[9] A.F. Blackwell, T. Green, and D. Nunn, "Cognitive Dimensions and Musical Notation Systems," *Workshop on Notation and Music Information Retrieval*, 2000 [Online]. Available: http://scholar.google.com/scholar?hl=en\&btnG=Search\&q=intitle:Cognitive+Dimensions+and+Musical+Notation+Systems\#0. [Accessed: 12-Feb-2013]

[10] A.R. Brown, "Code jamming," *M/C*, vol. 9, no. 6, 2007.

[11] N. Collins, "Live Coding of Consequence," *Leonardo*, vol. 44, no. 3, pp. 207–211, 2011.

[12] J. Dewey, *Art as Experience*. Perigee Trade, 2005.

[13] E.W. Dijkstra, "On the role of scientific thought," in *Selected writings on computing: a personal perspective*, New York: Springer, 1982 [Online]. Available: http://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html

[14] P. Feyerabend, *Conquest of Abundance: A Tale of Abstraction versus the Richness of Being*. University Of Chicago Press, 2001.

[15] T. Green and A. Blackwell, "Cognitive dimensions of information artefacts: a tutorial," 1998 [Online]. Available: http://www.cl.cam.ac.uk/\textasciitilde{}afb21/CognitiveDimensions/CDtutorial.pdf. [Accessed: 12-Feb-2013]

[16] T.R. Green, "Cognitive dimensions of notations," in *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, New York, NY, USA: Cambridge University Press, 1989, pp. 443–460 [Online]. Available: http://www.cl.cam.ac.uk/users/afb21/CognitiveDimensions/papers/Green1989.pdf

[17] F.P.B. Jr, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Anniversary. Addison-Wesley Professional, 1995.

[18] R.M. Keller, *Computer Science: Abstraction to Implementation*. Book manuscript in progress., 2001 [Online]. Available: www.cs.hmc.edu/~keller/cs60book/\%20\%20\%20All.pdf. [Accessed: 12-Feb-2013]

[19] J. Kramer, "Is abstraction the key to computing," *Communications of the ACM*, vol. 50, p. 2007.

[20] C.W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, jun 1992 [Online]. Available: http://doi.acm.org/10.1145/130844.130856. [Accessed: 09-Feb-2013]

[21] B. Liblit, A. Begel, and E. Sweetser, "Cognitive perspectives on the role of naming in computer programs," in *Proceedings of the 18th Annual Psychology of Programming Workshop*, 2006.

[22] T. Magnusson, *Epistemic Tools: The Phenomenology of Digital Musical Instruments*. 2009 [Online]. Available: http://eprints.brighton.ac.uk/6703/. [Accessed: 09-Oct-2012]

[23] A. McLean, "Computer Music Journal special issue on Live Coding." 2012 [Online]. Available: http://toplap.org/cmj/. [Accessed: 17-Feb-2013]

[24] A. McLean and Others, *TOPLAP website*. 2010 [Online]. Available: http://www.toplap.org/index.php/Main_Page

[25] M. Shaw, "Larger scale systems require higher-level abstractions," *SIGSOFT Softw. Eng. Notes*, vol. 14, no. 3, pp. 143–146, apr 1989 [Online]. Available: http://doi.acm.org/10.1145/75200.75222. [Accessed: 09-Feb-2013]

[26] J.N. Shutt, "Abstraction in Programming - working definition," 1999 [Online]. Available: http://citeseer.uark.edu:8380/citeseerx/viewdoc/summary?doi=10.1.1.40.1415

[27] C. Simonyi, "Intentional programming: Innovation in the legacy age," in *IFIP Working group*, vol. 2, 1996, pp. 1024–1043.

[28] J.M. Smith and D.C.P. Smith, "Database abstractions: aggregation and generalization," *ACM Trans. Database Syst.*, vol. 2, no. 2, pp. 105–133, jun 1977 [Online]. Available: http://doi.acm.org/10.1145/320544.320546. [Accessed: 09-Feb-2013]

[29] A. Sorensen and A.R. Brown, "aa-cell in Practice: An approach to musical live coding," in *Proceedings of the International Computer Music Conference*, 2007.

[30] J. Spolsky, *The Law of Leaky Abstractions*. 2002 [Online]. Available: http://www.joelonsoftware.com/articles/LeakyAbstractions.html. [Accessed: 09-Oct-2012]

[31] A. Ward, J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander, "Live algorithm programming and a temporary organisation for its promotion," in *Proceedings of the README Software Art Conference*, 2004.

[32] T. Wishart, *On sonic art*, vol. 12. Routledge, 1996.

[33] I.M. Zmoelnig, "Pointillism." dec-2012 [On-

line]. Available: http://vimeo.com/55085462. [Accessed: 06-Sep-2013]

[34] unknown, "live.code.festival 2013 – Call for Participation." sep-2013 [Online]. Available: http://imwi.hfm.eu/livecode/call/. [Accessed: 17-Feb-2013]

[35] various, "Haskell Hierarchical Libraries." 2013 [Online]. Available: http://www.haskell.org/ghc/docs/latest/html/libraries/index.html. [Accessed: 06-Sep-2013]

11. AUTHOR'S PROFILE

Renick Bell

Renick Bell is a doctoral student at Tama Art University researching live coding. His website can be found at http://renickbell.net.