Live Coding Using a Visual Pattern Composition Language

Roger B. Dannenberg
roger.dannenberg at cs.cmu.edu, www.cs.cmu.edu/~rbd, 412-268-3827
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract. Live coding is a performance practice in which music is created by writing software *during* the performance. Performers face the difficult task of programming quickly and minimizing the amount of silence to achieve musical continuity. The *Patterns* visual programming language is an experimental system for live coding. Its graphical nature reduces the chance of programming errors that interfere with a performance. *Patterns* offers graphical editing to change parameters and modify programs on-the-fly so that compositions can be listened to while they are being developed. *Patterns* is based on the combination of pattern generators introduced in Common Music.

Introduction

Live coding is a music performance practice where music generation software is programmed in real time (Nilson, 2007). Often, the software development process itself is featured along with the resulting sound, usually by projecting the developer's computer screen for the audience to see. One of the interesting and enjoyable aspects of live coding is that the audience can anticipate new sounds and changes in sounds by following the code development process. The composer also has the opportunity, by typing comments and simply by exposing the software structure, to communicate musical intensions directly to the audience as part of the performance.

Many musicians like to read scores while listening to music. The score can reveal structures in music that are not immediately apparent upon listening, and the score gives the reader/listener some ability to look ahead and perhaps understand the music, not only in terms of what has been heard already, but in terms of where the music is going. Live coding brings yet another aspect to the perception of music, which is that deeper layers of structure in music can be understood from the program (if visible to the audience) and related to the sound. As a simple example, a loop statement that generates some randomized notes might cause the audience to think about seemingly unrelated random notes as repeated variations of just one underlying sound. This perception might not occur without knowledge of the program structure.

Typically, live-coding performances are developed using a real-time system in which software can be modified incrementally, allowing for the continuous generation of sounds while the program is modified and extended. However, live coding has used a variety of languages including assembler, C, Perl (McLean, 2004), Lisp (Sorensen, 2005) and Python as well as more obvious choices of computer music languages such as Max MSP (Zicarelli, 1998), ChucK (Wang and Cook, 2004), and SuperCollider (Collins et al., 2003).

One of the problems of live coding is that program development is usually a slow and tedious process. Like traditional composition with music notation, algorithmic composition or programmed music specification rarely takes place at real-time speeds. But this is exactly what is required in a live coding performance. It is often the case that live coding music suffers because the programmed specification must be accomplished quickly. Furthermore, the coder is more-or-less obligated to keep the audience's interest from the beginning to the end of the performance. This means that taking even 10 minutes to carefully design some compelling sounds is not practical. The live coder can, in part, deal with this

Originally published as: Roger B. Danneberg, "Live Coding Using a Visual Pattern Composition Language," in *Proceedings of the 12th Biennial Symposium on Arts and Technology*, March 4-6, Ammerman Center for Art & Technology, Connecticut College, 2010.

problem by making the initial software development something of a prelude to the initial sound. Like a music prelude, the coder can create a visual, textual, and conceptual framework that is already evolving in time even before the "music" begins. When done well, the audience can marvel at the challenge and forgive the coder for a slow start.

One the other hand, even the best live coding performances seem to proceed at a slow pace. Another problem is that unless the coder commits large amounts of code to memory or is a coding virtuoso, there are likely to be frustrating mistakes. In music, mistakes are often a matter of partial failure: a note is a bit out of tune or an entrance is a bit late. In software, mistakes are usually serious and must be corrected to move forward. Obvious mistakes that are easily corrected are merely annoying, but sometimes mistakes are subtle, leaving the coder to try repeatedly to understand and fix them while the audience wonders what is going on and the performance loses its momentum.

After some experience with live coding, I became interested in better ways to express music as a form of program that could be developed quickly and with a relatively small chance of making significant programming errors. While part of the entertainment factor in live coding is watching music arise from very primitive foundations, it seems that another part of the enjoyment is relating the logic of the program to the sounds that emerge. In this sense, a higher-level language that can be understood by the audience might have some advantage over a low-level language where, in spite of the heroic efforts to coax out music, the audience is left staring at unintelligible text and having no clue how it relates to the sound.

Toward this goal, I designed and developed a new graphical language system for live coding. The goals of the system are to provide an interesting visual interface that an audience can see and enjoy and the coder can manipulate easily, and to introduce a minimum of constraints and syntax rules to allow rapid manipulation of algorithmic music compositions without awkward debugging. The language is named *Patterns.*

Model of Data and Computation

The path I took combines pattern generators introduced in Rick Taube's Common Music (Taube, 1997) with data-flow semantics and a graphical interface. Pattern generators take input parameters and generate a stream of values. For example, a cycle pattern generator takes a list of items, for example `(a b c)`, and generates the items in sequence, for example `a b c a b c a b c`…. The real power of patterns arises from the ability to combine them hierarchically. To facilitate this, patterns have the concept of *period*. The output of a pattern generator is not just a sequence of individual values but rather is a sequence of *periods*, where each period is a sequence of values.

In the cycle case, one iteration of the list, a b c, constitutes one period. Different pattern generators define their periods in different but usually intuitive ways. Consider the pattern P that generates random permutations of its input set `{x y z}`. Each permutation is one period, and P might generate the sequence `y x z, z x y, x y z, x y z`. (Here, periods are separated by commas.) Now suppose we use P as an item in a cycle generator. The complete input will be `(a b P)`. The standard rule is: *when an item is selected by a generator and the item itself is a generator, rather than output the item, output one period from the item.*

Thus, the cycle generator might generate `a b y x z, a b z x y, a b x y z,` …. Notice that whenever P is encountered, it is replace by a permutation from P. Notice also that the output of the cycle generator is organized into periods and that this cycle pattern generator could be an element in yet another pattern.

Patterns are something like regular expressions and context free grammars (Hopcroft and Ulman, 1969) where patterns represent non-terminals, although patterns can include a rich set of operations on sets and streams of items that are useful in music composition. To express this example as a grammar, we could write these production rules:

O → C O	P → x y z	P → y x z	P → z x y
C → a b P	P → x z y	P → y z x	P → z y x

Here, the non-terminal O represents the output. This grammar generates exactly the same output as the pattern generators. It should be noted that pattern generators are more general than context free grammars, so it is not always possible to rewrite a pattern computation in this way.

Pattern generators in Common Music, and those reimplemented in Nyquist (Dannenberg, 2008), are combined in expressions to yield pattern objects. These objects can then generate items incrementally. This idea is used in *Patterns* so that streams of parameters can be generated incrementally in real time. To make the system more expressive, there are two types of values: numbers and notes. A number is a scalar value, and a note is a structure that specifies pitch, duration, dynamics, midi channel, and inter-onset interval. The reason for introducing notes is to allow sequencers, musical scores, and other high-level musical objects and processes as a special case of a pattern generator.

The Graphical Language

To change the pattern behavior in Common Music, one must write another expression and try again. In my live-coding system, *Patterns*, a visual programming language gives a graphical picture of the pattern generators and their inputs.

This representation relies on the fact that music is *Patterns* is specified by a single nested expression. Returning to our example, the textual representation would be something like Cycle([‘a’, ‘b’, Permutation([‘x’, ‘y’, ‘z’]]), where the square brackets denote sequence construction. The important point here is that no matter how complex the pattern becomes, it can always be represented by a nested expression.

In the graphical *Pattterns* language, colored circles represent pattern generators such as Cycle and Permutation. Pattern generators have two classes of parameters: A parameter is either another pattern generator or a constant value such as a number of list of numbers. Pattern generator parameters are represented as circles within circles (see Figure 1.) Connections between patterns are made clear by the graphical representation. Non-pattern parameters are displayed in a control panel to the left of the nested circles. When a particular pattern generator (circle) is selected, the panel is updated to display its parameters. Parameters provided by another, nested pattern generator are drawn as a colored rectangle that matches the corresponding circle. Non-pattern parameters are drawn as sliders (for numbers) or text boxes (for number lists).

One can select any pattern generator and edit its parameters directly. One can also “plug in” new generators to control parameters, add generators to item sets, and remove generators. Figure 1 shows the graphical interface. Parameters for the selected pattern are visible on the left side of the panel. Buttons at the bottom appear when applicable and allow the current pattern to be inserted into the computation as an input to another pattern, to be deleted, or to be spliced into the computation in front of another pattern.

Run-Time Semantics

Although the syntax of the language is most easily thought of in terms of nested expressions, the behavior, or semantics, of *Patterns* is perhaps best thought of in terms of a tree of objects. (The expression view is almost correct if one considers the output or “value” of a pattern generator to be an infinite stream, but a further complication is that the user can actually modify the nested pattern structure – the “expression” – on the fly.

In fact, the nested circles have a tree of objects as the run-time representation, and computation is demand driven. The output is a sequence of notes, so a scheduler waits for the inter-note onset interval to expire and then requests a note to be generated by the top-most pattern object. (The largest circle at the “back” of the stack.) In order to obtain parameter values, each pattern object may invoke nested parameter objects, so computation flows up and down the tree structure.

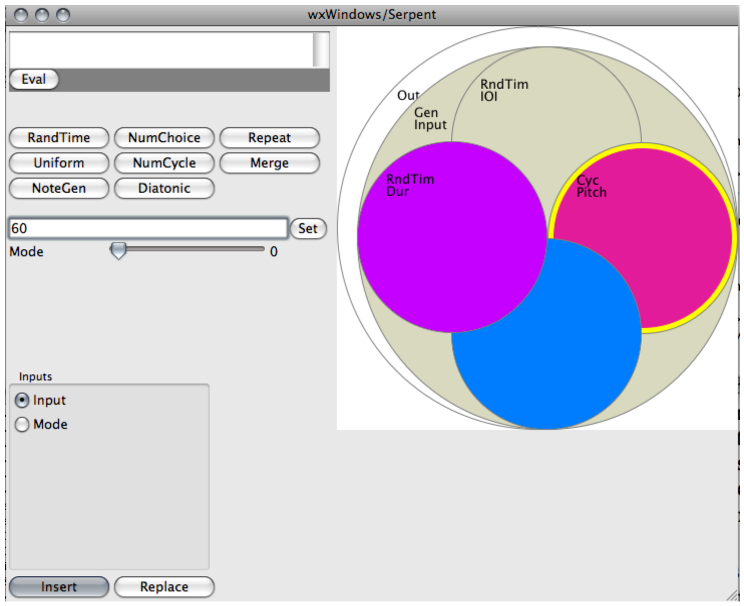


Figure 1. Graphical interface for the Patterns system.

The semantics of nested patterns requires some careful implementation. Returning yet again to our cycle-containing-permutation pattern example, the cycle object normally advances to the next element in the sequence each time it is called to produce a value. However, when the next element is the permutation pattern, a request is generated to the permutation pattern for its next item. In this case, the cycle pattern does not advance to the next element so that future requests for cycle items are also forwarded to the permutation. Only when the permutation finishes the generation of one *period* does the cycle object advance to the next element in its sequence.

The Implementation

At this time, there are a number of pattern types (see Table 1), but more need to be provided. The system outputs notes as MIDI, although in principle, there could be generators that gather parameters for any software synthesis system and initiate the generation and control of arbitrary sound objects.

Table 1. Pattern Generators

Name	**Type**	**Description**	**Parameters**
RndTim	Real	Returns a random duration using a negative exponential distribution	Mean duration
Unif	Integer	Returns a random integer using a uniform distribution	Lower bound, upper bound
Sum	Real	Forms the sum of inputs	Set of numbers and patterns to sum
Gen	Note	Forms note objects from parameters	Inter-onset interval, Pitch, Velocity, Duration, Channel
Choice	Real	Select randomly from a list	List of choices
Cyc	Real	Cycle through a list	List of numbers and patterns, Mode: forward, backward, or palindrome
Diat	Real	Quantize input value to the nearest pitch number in a diatonic scale	Input value, Transposition
Rep	Real	Repeat an input value	Input value, Repeat count
Merge	Note	Merge outputs from multiple patterns	List of note generators

Additional pattern generators are being added to the system on an as-needed basis. Common Music and Nyquist suggest possible additions:

- *line*: output list in order, repeat the last element indefinitely;
- *heap*: output all elements from an input list in random order in each period;
- *accumulation*: form the running total of all input values;
- *product*: form the product of input values;
- *length*: regroup inputs into periods according to a length parameter;
- *window*: form each output period from a sliding window over an input sequence, with parameters to control the window size and step size;
- *markov*: generate output according to a Markov model.

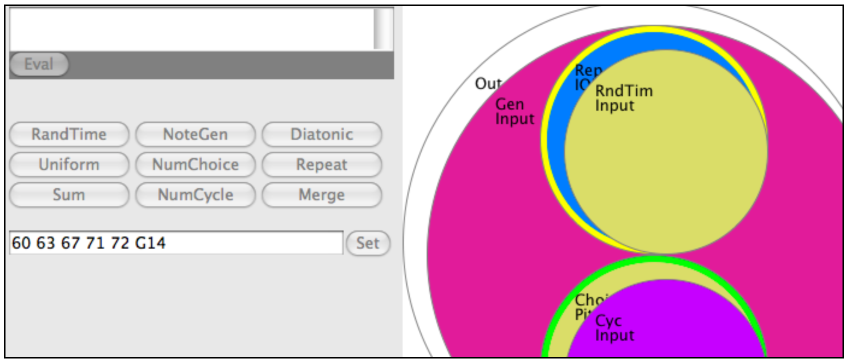
High-level operations on streams of notes are also possible and need to be explored. The *line, heap, length, window, and markov* patterns described above could apply to note sequences as well as numbers. Additional note-oriented pattern generators, include:

- *offset*: increment pitch, loudness, duration, onset time, or midi channel by some parameter;
- *scale*: multiply loudness or duration by some parameter;
- *replace*: set pitch, loudness, duration, inter-onset-time, or midi channel to some parameter
- *extract*: return the pitch, loudness, duration, inter-onset-time, or midi channel from the input (the output is therefore a stream of numbers)

Because *Patterns* has a representation for streams of notes as well as streams of numbers, there is the possibility of incorporating some sequencer elements into the program. Stored sequences seem to violate the spirit of live coding where everything is visible to the audience, but it should be possible to capture live musical motives, phrases, chords, or whatever and then output them from a special pattern generator. Other pattern generators could then be used to manipulate this material.

Example

Figure 2 shows a detail of a small Patterns program. The outermost white circle represents output. Within it, we see "Gen Input," which means a note generator ("Gen") serving as the "Input" parameter to its enclosing pattern. This in turn has two patterns as parameters. The IOI parameter is shown at the top ("Rep IOI," partially obscured), and the Pitch parameter ("Choice Pitch," also partially obscured) is shown at the bottom. The parameter to the Choice generator is a set of numbers, and since the Choice generator is selected, the numbers are shown at the left. Notice that one of the "numbers" is "G14." This is a text representation for a generator, which is this case is the "Cyc Input" generator.



60 63 67 71 72 G14

Figure 2. Detail of an example *Patterns* program.

The inter-onset intervals are chosen from a negative exponential distribution ("RndTim"), but each interval is repeated 8 times (a parameter that can be adjusted by selecting the "Rep IOI" generator and adjusting the slider that appears on the left.) Pitches are chosen at random from the list shown in the text box in Figure 2. When "G14" is chosen, a full period of pitches are output from the "Cyc Input" generator. This sequence of pitches is edited by selecting the "Cyc Input" generator and typing into a text box.

While this is not a complete composition, it shows that fairly interesting parameter generation can be described with a handful of nested circles and their associated parameters, which appear on a control panel at the left. In a more realistic example, there might be several "voices" running in parallel (using the "Merge" generator), and each might have more deeply nested expressions, resulting in more complex and interesting behavior.

Conclusion

In conclusion, *Patterns* is a system for dynamic, real-time, interactive programming and performance. It is especially intended for live-coding performances where a simple syntax and direct manipulation of music generation structures will allow rapid program development and evolution. The graphical interface is intended to allow an audience to follow the program development and anticipate the musical effects of program changes. *Patterns* is also fun to play with. I plan to make it available as free and open-source software. Because of the immediacy of the programming language and interface, I believe that *Patterns* might have some interesting educational applications.

References

Collins, N., A. McLean, J. Rohrhuber, and A. Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound*, 8(3), Cambridge University Press, pp. 321-330.

Dannenberg, R. 2008. *Nyquist Reference Manual.* http://www.cs.cmu.edu/~rbd/doc/nyquist/

Hopcroft, J. E. and J. D. Ullman. 1969. *Formal Languages and their Relation to Automata*. Addison-Wesley Longman.

McLean, A. 2004. "Hacking Perl in nightclubs." http://www.perl.com/pub/a/2004/08/31/livecode.html

Nilson, C. 2007. "Live coding practice." In *Proceedings of the 7th International Conference on New Interfaces For Musical Expression.* (NIME'07), pp. 112-117.

Sorensen, A. 2005. "Impromptu: an interactive programming environment for composition and performance." In *Proceedings of the Australasian Computer Music Conference*, pp. 149-153.

Taube, H. 1997. "An Introduction to Common Music." *Computer Music Journal*, 21(1), pp. 29-34.

Wang, G. and P. R. Cook. 2004. "On-the-fly Programming: Using Code as an Expressive Musical Instrument." In *Proceedings of the Internal Conference on New Interfaces for Musical Expression*. pp. 138-143.

Zicarelli, D. 1998. "An Extensible Real-Time Signal Processing Environment for Max." In *Proceedings of the 1998 International Computer Music Conference.* International Computer Music Association, pp. 463-466.