Bringing the TidalCycles Mini-Notation to the Browser

Charles Roberts
Interactive Media & Game Development
Department of Computer Science
Worcester Polytechnic Institute
charlie@charlie-roberts.com

Mariana Pachon-Puentes
Department of Computer Science
Worcester Polytechnic Institute
mpachonpuentes@wpi.edu

ABSTRACT

TidalCycles has rapidly become the most popular system for many styles of live coding performance, in particular Algoraves. We created a JavaScript dialect of its mini-notation for pattern, enabling easy integration with creative coding tools. Our research pairs a formalism describing the mini-notation with a small JavaScript library for generating events over time; this library is suitable for generating events inside of an AudioWorkletProcessor thread and for assisting with scheduling in JavaScript environments more generally. We describe integrating the library into the two live coding systems, Gibber and Hydra, and discuss an accompanying technique for visually annotating the playback of TidalCycles patterns over time.

1. INTRODUCTION

In canonical live coding performances, performers create artistic output by programming it in front of an audience, often projecting their source code for the audience to follow [21]. Such performances have increased in popularity over the years; for example, organizers have promoted hundreds of concerts in one genre of live coding performance, Algorave[4], where performers take turns live coding (predominantly) dance music. The browser is a popular platform for developers creating live coding systems, with many environments that primarily target music, visuals, and choreography, in addition to hybrid systems enabling performers to generate both audio and visual content (see Section 2). Our research takes the pattern mini-notation of TidalCycles[11] and brings it to JavaScript as a library that can be incorporated into browser-based live coding environments. The TidalCycles mini-notation is a terse and expressive way to describe patterns of values over time, and we hope that bringing this popular representation to the browser will encourage adoption across many systems. Section 4 describes some of the integrations we and other developers have created to date.

But first we begin by describing the state of live coding in the browser, alongside a brief introduction to TidalCycles. We describe our implementation of the parser and accompanying library, and their integration into various browser-based creative coding systems. We then discuss techniques for visually annotating the playback of TidalCycle patterns, taking advantage of the browser's flexible markup capabilities. We conclude with a short discussion of open questions, and possible directions for future work.

2. BACKGROUND

Live coding is growing in popularity, with new systems introduced every year and an growing number of local interest groups and performances. The browser has become a popular target for many live coding systems, thanks to its ubiquity, powerful graphics capabilities, and increasingly powerful audio features. In this section we discuss existing live coding tools and their notations for music—with an emphasis on TidalCycles—and the state of current browser-based live coding systems.

2.1 TidalCycles

TidalCycles is a domain-specific language embedded within Haskell [11]. It has proven to be one of the most popular choices for live coding, particularly in the Algorave genre, perhaps due to its strong emphasis on time and rhythm [10]. For example, in a recent four-day streamed live coding event, sixty-five of approximately one hundred and sixty performances primarily used TidalCycles[19]. The next most popular end-user language in the event was SuperCollider[9], with eighteen live coding performances[1].

TidalCycles outputs OSC or MIDI messages, and performers have used it to control a wide variety of applications. It is most often used to remotely control *SuperDirt*, an audio sample playback and manipulation engine written in SuperCollider.

2.2 Live Coding and Patterns

There are a variety of techniques for generating patterns in live coding systems. In ixi lang[6], a mini-language that controls the SuperCollider audio server, patterns (or *scores* in ixi lang terminology) are defined using strings, with the location of each token determining the timing of notes and sample playback. In both Gibber[15] and in Conductive[3], a Haskell live coding environment where the programmer controls semi-autonomous agents, there are separate patterns for values in sequences and controlling the timing of

[1]Although live coding in the SuperCollider language is arguably not as popular as live coding in TidalCycles, many live coding systems provide an alternative language that uses the SuperCollider audio server for synthesis and processing.

sequence output. For example, in Listing 1, the pattern `[0,1,2,3]` selects notes from a default global scale while the pattern `[1/4,1/2]` specifies that these notes will have alternating durations of a quarter note and a half note.

```
syn = Synth()
syn.note.seq( [0,1,2,3], [1/4,1/2] )
```

Listing 1: A sequence in Gibber

Other systems opt for more verbose temporal recursions (functions that invoke themselves over time[17]) or coroutines that provide low-level control of timing and output. For example, the *Extempore* and *Imporomptu* systems use verbose temporal recursions to generate patterns, often based on sampling continuous waveforms [18]. While these recursions are lengthy to type by hand (at least in the context of a live performance), the environments include text-editing macros that make insertion and editing fast and fluid. Sonic Pi[1] uses a terser design similar to a coroutine named the `live_loop`, where loops can sleep and are editable. Gibber and Gibberwocky[13] also support temporal recursions, in addition to their shared pattern affordances.

The mini-notation in TidalCycles is based on the Bol Processor (BP2)[2]. Polyrhythms, polymeter, repetition, and rhythmic subgrouping can all be described quite tersely. For example, the pattern string `'[kd sd, ch ch oh]'` describes a pattern where a kick drum and a snare drum are each half a *cycle* in duration, while the closed hihat and open hihat sounds are each a third of a cycle, creating a two against three polyrhythm. The expression of both timing and output values in the same text string provides a terseness that lends itself well to live coding. In our experience developing and performing with Gibber, we found that while defining output and timing separately has it merits, we often wanted the expressiveness of a combined notation similar to the TidalCycles mini-notation.

2.3 Browser-based Live Coding Environments

There are at least a dozen browser-based environments for live coding performance[2]. Some are geared heavily towards audio[3], while others emphasize graphics [5, 16], and at least one targets choreography[4]. Our hope is that the research presented here will be broadly applicable to a variety of such systems. In this vein, we discuss the integration of our library into two browser-based live coding environments, Hydra and Gibber, in Section 4.

Of particular relevance to this paper is *Estuary*[12], which provides a projectional editing interface for TidalCycles (in addition to other live coding systems), and runs in the browser. The authors developed the software in Haskell, which was then compiled into JavaScript. One notable advantage of this approach is that almost all of the features of TidalCycles were immediately available to the authors, while our approach requires implementing each of TidalCycles' features individually. For this reason, many features in our library are still in development / discussion, as described in Section 5. A comparative advantage of our library is that it is easy to incorporate into browser-based projects without requiring the use of a Haskell compiler[5], potentially easing adoption into other systems.

3. IMPLEMENTATION

We created two modules, designed to work in tandem, enabling the use of the TidalCycles patterns in the browser. The first module is a *parser* for the TidalCycles mini-notation. This parser was written using the *parsing expression grammar* formalism (or PEG); we have prior experience using this formalism to teach workshops on live coding language design [20]. The PEG.js library [6] translates the TidalCycles grammar into a parser that generates appropriate JavaScript data structures for *querying*, which is handled by our second module. Following the general structure of the TidalCycles library, this module exposes a `queryArc` function which accepts three arguments: a pattern, a start time, and a duration [7]. From these three arguments a list of *values* and *timestamps* is generated; the timestamps are offsets from the starting phase argument passed to `queryArc` and the values typically correspond to either indices in a scale denoting a pitch to be played or an identifier for a sample/sound to be triggered. The code example below shows these two modules in use employing the CommonJS module syntax.

```
parser   = require('./dist/tidal.js')
queryArc = require('./src/queryArc').queryArc

pattern = parser.parse( '0 [1 2]' )
events  = queryArc( pattern, 0, 1 )
/*
 * events = [
 *   { value:0, arc:{
 *     start:Fraction(0), end:Fraction(1,2) }
 *   },
 *   { value:1, arc:{
 *     start:Fraction(1,2), end:Fraction(3,4) }
 *   },
 *   { value:2, arc:{
 *     start:Fraction(3,4), end:Fraction(1) }
 *   }
 * ]
*/
```

Listing 2: Using the parser and query function

We combined the two modules together into a single `Pattern` object for easier use. The `Pattern` constructor accepts an argument pattern written using the TidalCycles mini-notation; the generated pattern object can then be queried by passing a starting phase and a duration to its `query` method. The `Pattern.query` method also sorts its output by the temporal proximity of each event to the start time of the query. Listing 3 shows this in action, assuming that the `Pattern` object has been imported into the global namespace.

[2]for a fairly comprehensive list of live coding systems, see https://github.com/toplap/awesome-livecoding/blob/master/README.md

[3]https://live.csound.com/

[4]https://github.com/sicchio/terpsicode

[5]Despite the unfamiliarity of Haskell for most web developers, the authors of Estuary make many interesting arguments for the security and type safety of Haskell in browser-based projects.

[6]https://pegjs.org

[7]In TidalCycles the combination of a start time and a duration is known as an `Arc`

```
const  pattern  =  Pattern('0  [1  2]')
const  events   =  pattern.query( 0,  1 )
//  same  event  output  as  Listing  1
```

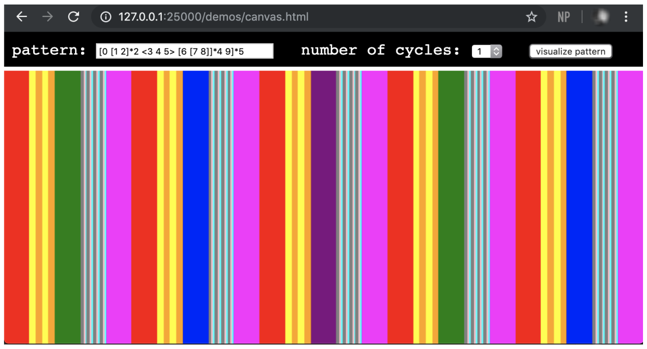
Listing 3: Use of `Pattern` object

In addition to these two modules and the `Pattern` object, the repository also contains a number of demos, and a set of over eighty unit tests covering the parser.

4. INTEGRATION

A demonstration using the `Pattern` object described in Sec. 3 with an HTML `<canvas>` element is included in the repository for the project and shown in Fig. 1. This figure draws the pattern `'[0 [1 2]*2 <3 4 5> [6 [7 8]]*4 9]*5'`, where each number [8] is assigned to represent a different color. The demo queries a user-provided pattern for a selected number of cycles, and then loops through all of the generated events drawing rectangles based on the duration and value of each event.

We have also successfully integrated this library into two live coding environments: Olivia Jack's Hydra and Gibber, developed by the first author. Developer/performer Diego Dorado also integrated the library into his browser-based live-emojing playground [9]. In this playground, emojis are used as TidalCycles groups. A pattern such as `'[☺*2 ☺*2?]☹'` can be interpreted as a TidalCycles pattern, where each emoji corresponds to a predetermined sound. Dorado's work uses Tone.js[7] for sound generation. Our repository also includes an audiovisual representation of TidalCycles patterns written with p5.js[8]. These two examples help demonstrate that the integration with common JavaScript libraries is relatively simple, enabling developers to bring the TidalCycles mini-notation to various browser applications.


127.0.0.1:25000/demos/canvas.html

pattern: [0 [1 2]*2 <3 4 5> [6 [7 8]]*4 9]*5   number of cycles: 1

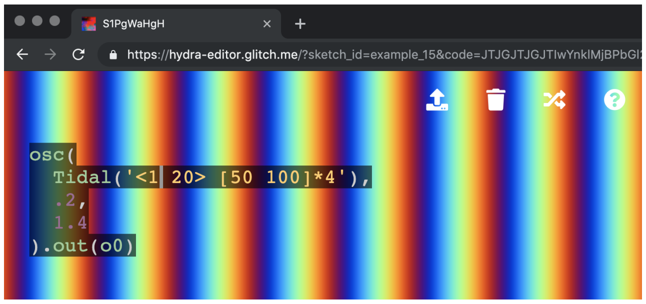
Figure 1: Using the library in conjunction with the HTML `canvas` object. In this example, numeric values are assigned to colors and patterned using the TidalCycles mini-notation. The `*5` at the end of the visualized pattern causes the entire pattern to repeat five times.

4.1 Integrating with Hydra

Hydra[10] is a popular browser-based environment for live coding visuals using the metaphor of analog video synthesis. When creating a video synthesis object, Hydra enables coders to assign functions to parameters instead of numbers; these functions are then evaluated every frame to determine the parameter's value. In order to integrate our `Pattern` object into Hydra, we first import the `pattern.js` file included in the source code repository, and then create a function that accepts TidalCycles mini-notation as an arugment. Invoking this function creates a function that updates an internal phase, queries an associated pattern for events, and returns values generated by the query. The code for the `Tidal` function shown in Fig. 2 (not counting on the included `Pattern` module) is given in Listing 4.

```
Tidal  =  function( pattern ) {
  let value = null,
      phase = 0,
      phaseIncr = 1/120,
      events = [],
      end = -Infinity

  const p = Pattern( pattern )

  const out = function() {
    phase += phaseIncr

    if( events.length <= 0 && phase > end ) {
      phase = 0
      events = p.query( phase, 1 )
    }

    const next = events[0]
    if( next !== undefined
      && phase > next.arc.start.valueOf() ) {
        const event = events.shift()
        value = event.value
        end   = event.arc.end.valueOf()
    }

    return value
  }

  return out
}
```

Listing 4: Hydra Integration



https://hydra-editor.glitch.me/?sketch_id=example_15&code=JTJGJTJGJTIwYnklMjBPbGl

```
osc(
  Tidal('<1 20> [50 100]*4'),
  .2,
  1.4
).out(o0)
```

Figure 2: Using the function from Listing 4 to create time-based patterns in Hydra.

4.2 Integrating with Gibber

Gibber is a browser-based tool for live coding both music and ray-marched visuals. Our integration enables creative coders to use the Tidal mini-notation to create patterns that can be applied both domains; the TidalCycles mini-notation can be used to generate musical patterns as well as control 3D geometries. We additionally provide a unique, animated

[8]The exceptions are numbers to the right of the `*` operator, which instead controls repetition
[9]https://diegodorado.com/en/labs/live-emojing
[10]http://hydra-editor.glitch.me/

annotation system that marks individual tokens in the mini-notation and reveals when events associated with each token are triggered.

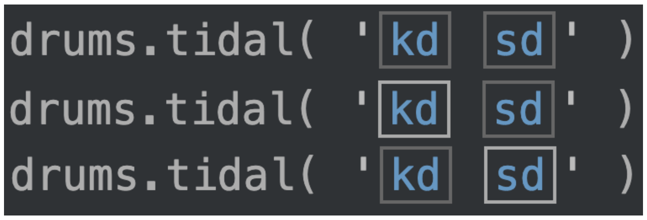
4.2.1 *Integrating with the AudioWorkletProcessor*

Audio in Gibber runs in an AudioWorkletProcessor thread; this includes all sequencing and scheduling, so that these systems can also be modulated using audio-rate signals. The use of the `Pattern` object needs to be as efficient as possible to fit in the tight time constraints specified for the `AudioWorkletProcessor` by the Web Audio API (128 samples per buffer). At first, we encountered problems with the speed of parsing patterns with deeply nested tokens; however, after specifying that the parser should cache results, these problems disappeared at the slight expense of additional processing time for the caching itself. The current unit test suite of over eighty tests completes in an average of 42 ms, indicating an average test time of about half a millisecond. We are optimistic that we can optimize the grammar to improve these results, but are also relatively unconcerned given that parsing—which typically takes much more time than querying—only occurs when a user creates a new pattern, a relatively infrequent event.

4.2.2 *Annotating TidalCycles Patterns*

One relatively unique feature in Gibber is the use of animated annotations to reveal system state within the source code editor. These annotations and visualizations can take many forms, from changing the source code itself, to highlighting tokens in the editor, to using HTML canvas-based visualizations of modulation signals that animate over time.

In order to annotate TidalCycles patterns in a similar fashion we extended our parser to assign a unique identification number to each token depicting a value, and to also include the each token's location in the code editor. With this information we can create a `<span>` element wrapping the token, and assign a unique CSS class to it based on its ID number. This enables us to always know the precise location of each token in our editor, even as the code is actively being edited[11].

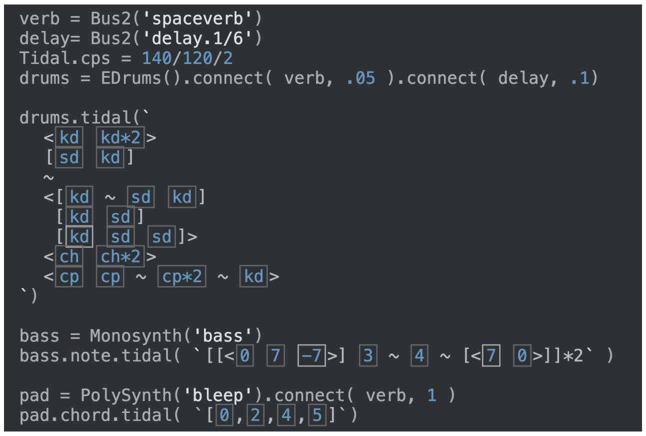


```
drums.tidal( 'kd sd' )
drums.tidal( 'kd sd' )
drums.tidal( 'kd sd' )
```

Figure 3: A simple TidalCycles pattern with three different annotated/animated states of activity. In the first line of code, no sounds are being triggered. In the second line, the kick drum (`kd`) has just been triggered, while the snare drum has been triggered in the final line.

The highlighting effect, in our opinion, is subtle, and while it is difficult to portray here in a static document, when animated the flashes of activity are clear and visible. Although we could use an annotation with higher contrast, our past research has indicated that many people prefer such annotations to be less distracting[14], so that they don't distract from the act of programming.

[11]Thanks to the exceptional http://codemirror.net/)CodeMirror library for providing editing interface that supports this



```
verb = Bus2('spaceverb')
delay= Bus2('delay.1/6')
Tidal.cps = 140/120/2
drums = EDrums().connect( verb, .05 ).connect( delay, .1)

drums.tidal(`
  <kd kd*2>
  [sd kd]
  ~
  <[kd ~ sd kd]
   [kd sd]
   [kd sd sd]>
  <ch ch*2>
  <cp cp ~ cp*2 ~ kd>
`)

bass = Monosynth('bass')
bass.note.tidal( `[[<0 7 -7>] 3 ~ 4 ~ [<7 0>]]*2` )

pad = PolySynth('bleep').connect( verb, 1 )
pad.chord.tidal( `[0,2,4,5]`)
```

Figure 4: Multiple annotated sequences running concurrently. One kick drum token and two notes in the bass line are highlighted to indicate activity.

5. CONCLUSIONS AND FUTURE WORK

We have created a JavaScript library for parsing and querying the mini-notation of TidalCycles, and successfully integrated it into two live coding environments, Gibber and Hydra. Our research also described a novel technique for annotating the playback of TidalCycle patterns to potentially improve audience and programmer understanding of the TidalCycles mini-notation.

However, decisions and challenges remain. TidalCycles contains functions to both specify patterns and to transform them over time; these transformations are missing from our current implementation. Some are relatively simple to implement and have been included in our Gibber integration, such as shifting the placement of patterns by adding or subtracting from the current phase of the pattern. Other transformations are more difficult. In regards to integrating the mini-notation into Gibber, it is unclear whether it is preferable to use Gibber's existing pattern manipulation API to also transform patterns generated by the TidalCycles mini-notation, or if we should include a separate API that matches the TidalCycles for manipulating patterns.

Having a separate implementation of the TidalCycles mini-notation also raises the possibility of creating a new dialect. For example, the mini-notation currently doesn't have a mechanism for specifying the loudness of individual notes or sample triggers. We plan to add this to our library, creating a minor fragmentation between the two notations. Having the language available in JavaScript might encourage more developers to experiment with the grammar and add their own additional features; perhaps some of these features will eventually make their way back to the canonical Haskell implementation to the benefit of the greater community.

6. ACKNOWLEDGMENTS

Our thanks to Alex McLean and the TidalCycles community for helping us to understand the intricacies of pattern parsing in TidalCycles via online discussion.

7. REFERENCES

[1] S. Aaron and A. F. Blackwell. From sonic pi to overtone: creative musical experiences with domain-specific and functional languages. In *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 35–46. ACM, 2013.

[2] B. Bel. *Rationalizing musical time: syntactic and symbolic-numeric approaches*, pages 86–101. Feedback Studio, 2001.

[3] R. Bell. An approach to live algorithmic composition using conductive. In *Proceedings of LAC*, volume 2013, 2013.

[4] N. Collins and A. McLean. Algorave: Live performance of algorithmic electronic dance music. In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pages 355–358, 2014.

[5] D. Della Casa and G. John. LiveCodeLab 2.0 and its language LiveCodeLang. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 1–8. ACM, 2014.

[6] T. Magnusson. ixi lang: a SuperCollider parasite for live coding. In *Proceedings of the International Computer Music Conference*. University of Huddersfield, 2011.

[7] Y. Mann. Interactive music with tone. js. In *Proceedings of the 1st annual Web Audio Conference*. Citeseer, 2015.

[8] L. McCarthy, C. Reas, and B. Fry. *Getting Started with P5. js: Making Interactive Graphics in JavaScript and Processing*. Maker Media, Inc., 2015.

[9] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[10] A. McLean. Making programming languages to dance to: live coding with tidal. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*, pages 63–70. ACM, 2014.

[11] A. McLean and G. Wiggins. Tidal–pattern language for the live coding of music. In *Proceedings of the 7th sound and music computing conference*, 2010.

[12] D. Ogborn, J. Beverley, L. N. del Angel, E. Tsabary, and A. McLean. Estuary: Browser-based collaborative projectional live coding of musical patterns. In *International Conference on Live Coding (ICLC) 2017*, 2017.

[13] C. Roberts and G. Wakefield. Live Coding the Digital Audio Workstation. In *Proceedings of the 2nd International Conference on Live Coding*, 2016.

[14] C. Roberts, M. Wright, and K.-M. JoAnn. Beyond editing: Extended interaction with textual code fragments. In *Proceedings of the New Interfaces for Musical Expression Conference*, 2015.

[15] C. Roberts, M. Wright, and J. Kuchera-Morin. Music Programming in Gibber. In *Proceedings of the International Computer Music Conference(ICMC)*, pages 50–57, 2015.

[16] J. Rodríguez, E. Betancur, R. Rodríguez, and A. de México. Cinevivo: a mini-language for live-visuals. In *Proceedings of 2019 International Conference on Live coding*, 2019.

[17] A. Sorensen. The many faces of a temporal recursion, 2013 (last accessed October 5th, 2019). http://extempore.moso.com.au/temporal_recursion.html.

[18] A. Sorensen and H. Gardner. Programming with time: cyber-physical programming with impromptu. In *ACM Sigplan Notices*, volume 45, pages 822–834. ACM, 2010.

[19] TOPLAP. *TOPLAP 15th Anniversary Concert Signup Form*, 2019 (last accessed October 5th, 2019). https://docs.google.com/spreadsheets/d/1Mksi_TReJpp9R3XunGCR0SALZRA03aMoiOEd3qUW0bo/edit#gid=0.

[20] G. Wakefield and C. Roberts. A virtual machine for live coding language design. In *NIME*, pages 275–278, 2017.

[21] 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*, volume 289, page 290, 2004.