Trane: Musical Janet on the Web

Anonymous Author(s)

Abstract

Trane is a domain specific language and environment for livecoding music on the web. It gives the programmer control over instruments, effects, their connectivity, and the ability to sequence well-timed events. In this paper we explore the motivation behind the language, its design, and implementation.

CCS Concepts: • **Applied computing** → **Performing arts**; **Sound and music computing**; • **Information systems** → *Web interfaces.*

Keywords: livecoding, music, lisp, janet, audio, web, browser

ACM Reference Format:
Anonymous Author(s). 2024. Trane: Musical Janet on the Web. In *Proceedings of ACM SIGPLAN International Workshop on Functional Art, Music, Modelling and Design (FARM '24).* ACM, New York, NY, USA, 6 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Live coding refers to a practise in which a programmer makes or modifies software live. [5] In the context of audio or music, the programmer might alter a process that generates sound and should be able to hear the modifications they are making. The number of live coding languages for audio has grown significantly since its first inception, with each offering different expressiveness to the programmer. Some languages might be a natural choice for complex drum sequencing, others might offer easy control over harmony generation, another could specialise in low-level DSP control.

2 Motivation

Trane was created from the desire for an ergonomic, mouldable, high-level language for audio sequencing, with easy control over sound-design. A subset of the language allows the programmer to define an audio graph declaratively, which was inspired by physical modular synthesizers. The other half of the language lets the programmer schedule events against this graph. Sonic Pi was a large influence in the design of the sequencing side of the language. Most notably the (`live_loop` ...) [2] and virtual time semantics [3] are borrowed.

Lowering the barrier to live-coding, audio synthesis, and lisp-style programming were also important motivations when implementing Trane, as was the ability to easily share patches and programs. The web was a natural target when considering this, with the added benefit of being cross-platform.

2.1 Previous Work

Previous artistic languages or environments have, directly or indirectly, influenced the design and implementation of Trane.

Several influential languages are dependent on the SuperCollider Environment [14]. ixilang [13] contains an interpreter inside of SuperCollider, and offers high-level control over synth definitions and samples. Tidalcycles [15] is embedded in Haskell, and allows for complex pattern creation using a cyclical notion of time. Sonic Pi [2] is a ruby-based DSL, and introduces musical notions of time and concurrency.

There are other languages that do not require the SuperCollider engine, and can be hosted elsewhere. Glicol [12] is a graph-oriented live-coding language implemented in Rust, which can be run in a multitude of environments. Strudel [1] is a javascript port of Tidalcycles, which uses Web Audio[16] for synthesis.

More recent work on Janet playgrounds for graphics have been the largest influence in the development of Trane. Bauble [10], is an interactive Janet playground for creating ray-marched Signed Distance Functions [9]. It runs in the browser, and translates a Janet DSL into GLSL fragment shaders. Toodle [11] is also a browser-based playground, in which the programmer writes Janet code to control turtle graphics.

3 Structure of a Trane program

Trane programs are conceptually programs of two halves. In one half, there is a declarative syntax that represents an audio graph. That is, modules and wires between them. On the other half, there are language constructs for sequencing well-timed audio events.

3.1 Defining an audio graph

Trane is inspired from physical modular synthesizers. Modules can generate audio signals, or they might process input signals to form a new signal.

FARM '24, Sept 02–07, 2024, Milan, Italy

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Modules can be wired together to form a patch. A patch can be represented abstractly as a directed graph, with nodes representing modules, and edges representing the wires between them. Similarly, some audio synthesis engines use a graph based structure to model the signal flow through a series of nodes.[17][14] Audio signals are generated in nodes, travelling through edges where they are processed by connected nodes, until finally they arrive at the at the buffer of an interface and your signal is made audible.

In Trane, to define such a graph, we can instantiate modules and the wires that connect them. For example, we might want to generate a sine wave, amplify it, then output the amplified wave to a speaker.

This can be achieved with the following code:

```
(chain
  (oscillator :hello-world "sine")
  (gain :hello-gain)
  :out
)
```

3.2 Modules

Modules are declared at the top-level of a trane program. They can be created using the relevant module instance macro.

Each module requires a name, so that events can be sent to them. The arguments that follow the name are module dependent. For example, the macro (`sample name URL pitch`) will create a new pitched sampler module, with an audio sample located at the location in `URL` and a reference pitch.

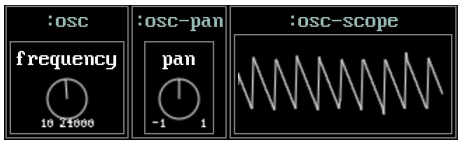

Figure 1. The Trane user interface: Each box represents a module. Here the programmer has instantiated a sawtooth oscillator, a panner, and a scope.

3.3 Wires

Modules can be wired together in two ways.

The (`wire from to param`) macro will wire two modules together. The output signal in `from` will be connected to the default input in `to`. The `param` argument is optional, and will override the default input in `to` to one of its parameters. An example of this is given in figure 3.

Modules can also be wired together using the (`chain & modules`) macro. Each module in `modules` will be wired from the previous module to the next module to form a chain.

3.4 Time, Notes

The unit of time in Trane is the beat. This means all events, durations and time-based instrument parameters are defined in beats.

MIDI notes are represented via a keyword or by their integer value. [`:Cs3 :cs3 :db3`] all refer to the same note. Chords can be represented by mutable arrays or tuples of notes. If a programmer wants to do arithmetic on notes they can convert to an integer with (`note :cs3`).

A (`play module note`) macro will schedule a note to play on a particular module. Optionally this macro will take a named `:dur duration` parameter, which defines how long the note will play for in beats.

3.5 Events

Events in Trane are timed operations on the audio graph. They have a destination (a module or parameter in the graph) and allow for precise scheduling of notes, or parameter changes. Each module in the graph may have a number of parameters, for example an oscillator module will have a frequency parameter. Other modules may accept play events, which might represent the start of a note, or the start of a particular sample.

3.6 Sequencing events

Once we have a valid graph, we can send audio events to any of the modules in the graph in order to change parameters of the modules. In order to send notes at a specific, accurate time, we can send events using the (`live_loop name & body`) macro.

```
(live_loop :gain-changer
  (lin :hello-gain :gain 0)
  (sleep 1)
  (lin :hello-gain :gain 1)
  (sleep 1)
)
```

Figure 2. Demonstrates scheduling parameter changes on the audio graph. This live_loop will schedule changes to the `:gain` knob on the `:hello-gain` module. Each loop body will schedule a linear interpolation of gain from a value of `0` at time t to `1` at time $t + 1$. It implements a low-frequency triangle wave oscillator with a period of 2 beats.

A (`live_loop ...`) can be viewed as a function of time. A virtual time is set as a thread-local variable when a loop body is evaluated. Each call to (`sleep t`) will advance the local time variable by `t` beats. This local variable is then used in each event producing macro. Event producing macros will add to a hidden array of events that each (`live_loop ...`) yields once its evaluation completes, along with the final virtual time.

4 Parameter changes

Parameters are represented in the interface as knobs. They can be changed with a mouse or can be mapped to MIDI CC controllers. Knobs may have a logarithmic scale where they represent a quantity that is more uniformly perceived in the logarithmic domain [18], such as frequency and volume.

4.1 From the code

The parameter changing macros all accept module and parameter options. These must be set. The module specified is checked to exist in the graph, and the parameter of parameter-name is checked to exist on the module.

(`lin module-name param-name p`) will linearly interpolate the specified module's parameter from its current value to the target value `p` at the scheduled time.

(`expo module-name param-name p`) behaves like (`lin ...`), but the interpolation behaviour is exponential instead of linear.

(`target module-name param-name p time-constant`) will approach the target `p` exponentially, starting at the scheduled time. The rate at which it approaches is given by `time-constant`

(`itarget module-name param-name p`) will instantaneously change the target parameter to `p` at the specified time.

4.2 From the graph

In addition to scheduling parameter changes from live_loops, parameters can also be changed using a modulator signal from the graph.

4.3 Knobs

Module parameters can also be changed via the graphical interface. Each module parameter will have an associated knob, whose value can be changed via clicking and dragging. 4. Parameters can also be mapped to MIDI controls by shift-clicking a knob and then moving an associated MIDI CC emitter.

Each knob will advertise its parameter name, and a minimum and maximum range. Apart from being a useful performance and compositional aid, these also act as documentation to the programmer.

4.4 Which to choose?

With a few different methods to change module parameters inside of the graph, it could be confusing to decide which to choose when programming live. It's really up to the programmer or performer which of these to choose based on their goal. If strict timing is desired, for example in the attack phase of a note envelope, then parameter changes via code might be the right choice. However, graph based parameter changes can work in tandem with parameter changes from code or the knob interface. For example, we might wish to twist a frequency knob on a low-frequency-oscillator that is wired up to the volume of another audible-range oscillator.

```
(lfo :lfo-tremolo "sine")

(chain
  (oscillator :drone "sawtooth")
  (gain :drone-gain)
  :out
)

(wire :lfo-tremolo :drone-gain :gain)
```

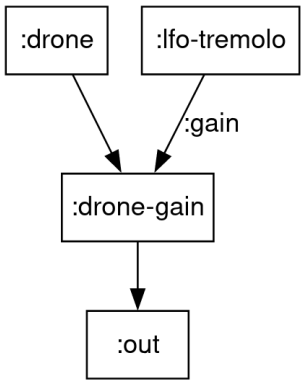

Figure 3. The code, and audio graph showing a low-frequency oscillator modulating the gain on another oscillator to produce a tremolo effect.

Figure 4. The default **gain** module. Its value can be changed with a mouse, or by mapping it to a midi CC event.

5 Implementation

Trane is implemented using modern web standards such as ECMAScript 2015, Web Audio [16], and WebAssembly [8]. The next section will go over particular parts that might be non-standard for a web-application. It will follow the sequence of events that are triggered from a change to the code, and lead up to an audible sound.

5.1 Compilation

Changes to the code will trigger compilation of the buffer in the editor. During compilation, macros are expanded and certain compile time checks are made. A feature of trane, designed to avoid confusion and bugs during performance, is that connectivity constraints on the audio graph are made when the buffer is compiled. The compilation step will not complete if wires do not have valid sources or destinations.

Additionally, event destinations will be checked against the graph. For example a (`play ...`) macro will throw an

error if the destination module it is sending to does not exist at compile time.

The compilation step yields a marshalled representation of the environment called an *image*. An *image* contains the environment data that is the result of any compile time processes, along with function and macro definitions. In Trane this will consist of audio graph connectivity information. Other information contained in the *image* are (`live_loop ...`) names and marshalled representations of their body, so that they can be scheduled and executed when required.

5.2 Code (re)Loading

Successful compilation yields an executable image. It's at this point that the user can swap out existing code for the new code. Once a code reload is triggered by the user we switch the images that are used by the loop manager. The next time a loop is scheduled, the loop manager will use this new code to generate events until the (`live_loop ...`) function yields a result. This method of hot code reloading was inspired by Erlang/OTP [4]

Each (`live_loop ...`) that isn't present in the new image will be cleaned up by the loop manager, but any events that were scheduled before its deletion will still be audible.

5.3 Graph Manager

The audio graph is implemented using the Web Audio standard [16]. During each compilation a set of modules and wires are generated. Once a code reload is triggered, a diff operation is made between the existing, running graph and the new one. Any modules or wires that are not present in the new image are disconnected and deleted from the running graph. Any new modules are created and wired up.

During this graph change, audible popping might be discernible if a part of the graph that is being changed contributes to the final, audible signal. While methods to change the graph with fewer such artefacts exist, for instance by smoothly mixing between old and new graphs after a code change, we welcome such artefacts: Particularly being analogous to real audio equipment, which might produce pops as instruments are being wired together.

Performers with an aversion to these pops might choose to introduce new sub-graphs behind a mixer or zeroed gain node if they choose.

5.4 Scheduling

A given executable (`live_loop name & body`) will produce a list of events. These all have scheduling times, in beats, which are routed to the respective graph nodes ahead of time.

Typically, in a browser, the ability to schedule events at a particular time in the future is provided to the programmer through the `setTimeout(fn, delay)` function. This API accepts a function and a delay in milliseconds. However,

```
(chain
  (oscillator :drone "sine")
- (gain :drone-level)
  :out
)

(chain
  (sample :piano "piano_c3.wav" :C3)
- (reverb :piano-verb "hall_impulse.wav")
+ (Dlay :piano-delay 1.5)
  (gain :piano-level)
  :out
)
```

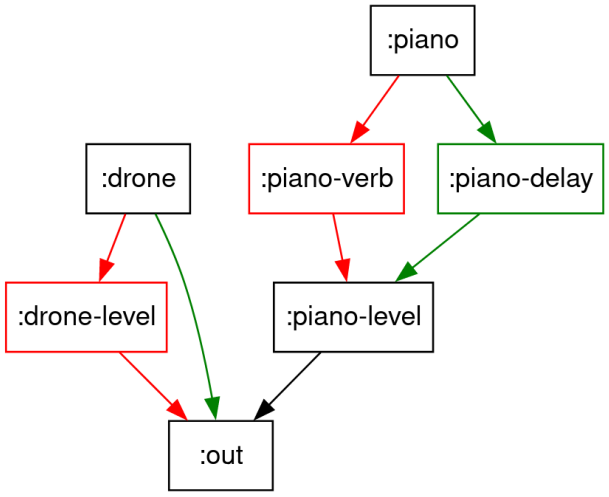

Figure 5. Operations on the graph after a code change. Old (red) modules and wires are discarded before new (green) modules are wired up.

there are no real guarantees from the specification or implementations that the function (and any sound producing side effects) will be evaluated at exactly that time [6]. This method to schedule events in the future does not give us sufficient accuracy for most audio applications. The situation is improved by the guarantees made by the WebAudio API. Certain events can be scheduled with sample-level accuracy if they are made far enough in advance.

To take advantage of this, Trane will try to execute the body of a live_loop some fixed time ahead of its virtual time. In this way we can generate notes or events for the future so that they can be accurately scheduled with the WebAudio API. Trane borrows the concept of virtual time semantics from Sonic Pi. live_loops are scheduled with sufficient lead time.

A sufficient lead time will be enough to evaluate the main body of the live_loop and avoid any timing jitter from the call to `setTimeout`. In Trane this is 250ms. A warning will be generated if the live_loop body exceeds its time budget so

that its events can be accurately scheduled. It's the responsibility of the programmer to keep to this budget. In the future we wish to give the programmer control over this timing constant.

6 Synchronization

Each (`live_loop ...`) is run in a separate thread, and do not easily communicate. To achieve synchronization between two live_loops the loop manager will schedule all new loops to start on every bar boundary.
Unlike Sonic Pi, Trane does not provide explicit cue or sync primitives in the language. Instead, programmers are encouraged to interpret live_loops as functions of time, so that they can eventually reach synchronicity through their output, or alternatively begin simultaneously on some future measure. Some utilities in the language might make this easier;

```
(til measure-length)
```

will return the time until the start of the next measure, where *measure-length* denotes the length of the measure. The example in figure 6 demonstrates its use.

Another useful utility for writing functions of time is

```
(timesel list change-every)
```

Which will select an item from *list* based on the current time. It will select the next element from *list* after *change-every* has elapsed. This selection wraps around the whole of list.

```
(live_loop :loop_a
  (sleep (til 64))
  ... play some notes
  (sleep 64)
)

+(live_loop :loop_b
+  (sleep (til 64))
+  ... play some notes in sync with :loop_a
+  (sleep 64)
+)
```

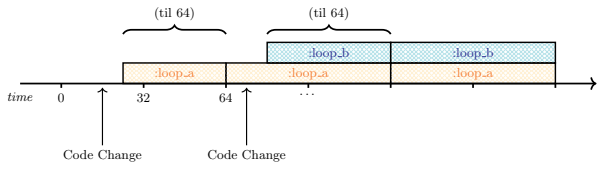

Figure 6. Demonstrates live_loop synchronisation in trane. :loop_a is run first, and sleeps until the next 64 bar boundary before scheduling any notes or events. :loop_b is then started at some point during the measure. To keep in sync with :loop_a it also sleeps until the next 64 bar boundary. After this they both produce events over the same virtual time.

7 Patterns

Trane has rudimentary support for musical patterns, similar to those seen in tidalcycles [15] or strudel [1]. A pattern in trane is encoded in an array or tuple of elements. These elements can represent notes, parameter values, or something more abstract. Patterns are evaluated over a time period, and the time between elements in the pattern is divided equally.

```
(P patt measure-length)
```

This macro will evaluate the pattern, dividing *measure-length* recursively between the elements of *patt*. It will return a list of tuples `[element, time]` that can be easily scheduled, manipulated, or combined in a (`live_loop ...`)

```
(P [[:c3 [:tie :e3]] :g3 [:c3 :e3 :d3] :c3] 4)
```

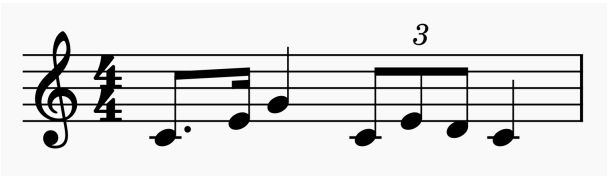

Figure 7. Demonstrates how time is divided up inside of a musical pattern.

7.1 Parsing Expression Grammars

The pattern evaluator takes as input a Janet tuple or array, which might be tedious, too verbose, or error prone to write manually. A string that matches a particular grammar might be a more enjoyable way to write patterns.

A Parsing Expression Grammar (PEG) is a concise, unambiguous set of rules that describe a top-down parser. [7] The Janet language comes bundled with a powerful Parsing Expression Grammar (PEG) engine that will match and capture a string which matches a particular grammar.
The (**peg/match** peg text) utility will match a string text to the grammar in peg. This utility is also available inside of trane, making experimentation with new musical grammars easy within the environment. The snippet of code below shows how a grammar can translate a small subset of the Tidalcycles mininotation [15][1] to a Trane pattern.

```
(def little-grammar
  ~{
    :rest (/ "~" :rest)
    :tie (/ "-" :tie)
    :note (cmt (<- (*
        (range "ag")
        (at-most 1 (+ "s" "b"))
        (range "09")))
      ,keyword)
    :subdivide (group (* "[" :sequence "]"))
    :repetition (cmt
      (*
        (+ :subdivide :note :rest :tie)
        "*"
        (number :d+))
      ,rep)
    :item (+ :repetition :note :subdivide :rest :tie)
    :sequence (+ (* :item :s :sequence) :item)
    :main (* :sequence -1)
  }
)

(peg/match
  little-grammar
  "a2*4 [g2 a2 b2 ~] b2 [a2 b2 c3 ~] c3"
)
```

8 Summary and Future Work

We've described Trane: a musical DSL embedded in the Janet language. Despite being a complete environment to experiment with livecoding music and audio synthesis, Trane has need for some improvements to make it a more fun and inclusive environment to make music in. Updates to the pattern evaluator, better error reporting, more and better quality modules, and improved interaction and accessibility features are all planned for future releases.

References

[1] 2023. *Strudel: Live Coding Patterns on the Web.* Zenodo. https://doi.org/10.5281/zenodo.7842142

[2] Samuel Aaron and Alan F Blackwell. 2013. 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*. 35–46.

[3] Samuel Aaron, Dominic Orchard, and Alan F Blackwell. 2014. Temporal semantics for a live coding language. In *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. 37–47.

[4] Joe Armstrong. 1996. Erlang—a Survey of the Language and its Industrial Applications. In *Proc. INAP*, Vol. 96. 16–18.

[5] Alan F. Blackwell, Emma Cocker, Geoff Cox, Alex McLean, and Thor Magnusson. 2022. *Live Coding: A User's Manual.* The MIT Press. https://doi.org/10.7551/mitpress/13770.001.0001 arXiv:https://direct.mit.edu/book-pdf/2239274/book_9780262372633.pdf

[6] MDN Contributors. 2023. *SetTimeout Documentation.* https://developer.mozilla.org/en-US/docs/Web/API/setTimeout

[7] Bryan Ford. 2004. Parsing expression grammars: a recognition-based syntactic foundation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 111–122.

[8] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[9] John Hart. 1995. Sphere Tracing: A Geometric Method for the Antialiased Ray Tracing of Implicit Surfaces. *The Visual Computer* 12 (06 1995). https://doi.org/10.1007/s003710050084

[10] Ian Henry. 2023. *Bauble.* https://bauble.studio

[11] Ian Henry. 2023. *Toodle.* https://toodle.studio

[12] Qichao Lan and Alexander Refsum Jensenius. 2021. Glicol: A Graph-oriented Live Coding Language Developed with Rust, WebAssembly and AudioWorklet. In *Proceedings of the International Web Audio Conference (WAC '21)*, Luis Joglar-Ongay, Xavier Serra, Frederic Font, Philip Tovstogan, Ariane Stolfi, Albin A. Correya, Antonio Ramires, Dmitry Bogdanov, Angel Faraldo, and Xavier Favory (Eds.). UPF, Barcelona, Spain.

[13] Thor Magnusson. 2011. The IXI Lang: A SuperCollider Parasite for Live Coding. (01 2011).

[14] James McCartney. 2002. Rethinking the computer music language: Super collider. *Computer Music Journal* 26, 4 (2002), 61–68.

[15] Alex McLean. 2014. 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*. 63–70.

[16] Hongchan Choi Paul Adenot. 2021. *Web Audio API Specification.* https://www.w3.org/TR/webaudio/

[17] Miller S Puckette et al. 1997. Pure data. In *ICMC*.

[18] S. S. Stevens and J. Volkmann. 1940. The Relation of Pitch to Frequency: A Revised Scale. *The American Journal of Psychology* 53, 3 (1940), 329–353. http://www.jstor.org/stable/1417526