kilobeat: low-level collaborative livecoding

Ian Clester
Georgia Institute of Technology
Atlanta, Georgia, United States
ijc@gatech.edu

ABSTRACT

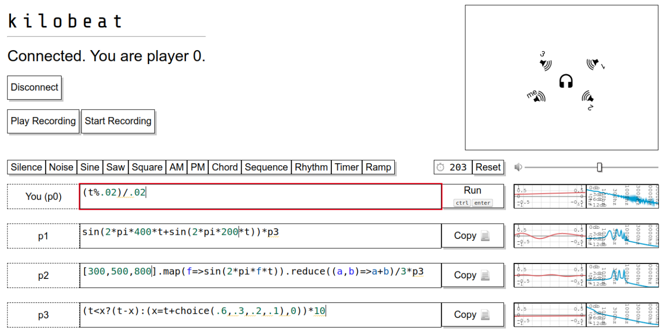
This paper presents **kilobeat**, a collaborative, web-based, DSP-oriented livecoding platform. Players make music together by writing short snippets of code. Inspired by the practices of *bytebeat*, these snippets are low-level expressions representing digital audio signals. Unlike existing platforms, kilobeat does not adapt existing livecoding languages or introduce a new one: players write JavaScript expressions (using standard operators and math functions) to generate samples directly. This approach reduces the amount of background knowledge required to understand players' code and makes kilobeat amenable to synthesis and DSP pedagogy. To facilitate collaboration, players can hear each other's audio (distributed spatially in a virtual room), see each other's code (including edits and run actions), and depend on each other's output (by referencing other players as variables). Additionally, performances may be recorded and replayed later, including all player actions. For accessibility and ease of sharing, kilobeat is built on Web Audio and WebSockets.

1. INTRODUCTION

The scenario is this: some people get together to make music. Each person has an instrument to play. When someone plays, everyone in the room can hear them play, and everyone can see how they are playing. This instrument is portable, so people can move around the room while they play, and this changes where the sound comes from. In good sessions, people work together, and aside from the occasional solo, people keep their output down to a certain level of complexity—so they do not dominate the sound, and so they can dedicate most of their attention to listening and responding to the other players.

kilobeat is a virtual environment for making music together remotely. It attempts to make possible the scenario described above, even when players are physically dispersed and latencies are too high for real-time approaches. In contrast to existing remote livecoding platforms, players in kilobeat perform by writing short expressions that generate audio directly, at the sample-level.

1.1 Related Work



```
You (p0)  (t%.02)/.02
p1        sin(2*pi*400*t+sin(2*pi*200*t))*p3
p2        [300,500,800].map(f=>sin(2*pi*f*t)).reduce((a,b)=>a+b)/3*p3
p3        (t<x?(t-x):(x=t+choice(.6,.3,.2,.1),0))*10
```

Figure 1: The kilobeat interface.

There are many projects which deal with remote music-making. Some attempt to preserve the possibilities of proximate music-making by relaying audio between participants in real-time (or as close to real-time as network latency allows). This category includes JackTrip [1] and Jamulus,[1] along with some commercial products. These are most viable when participants are physically located reasonably close together (or the music performed does not require tight synchronization).

Other projects are more flexible with latency, as they synchronize audio to a common timebase. NINJAM[2] does this with audio, while other projects—such as Troop [3], Estuary [5], and QuaverSeries [4]—do this by having players manipulate code rather than transmit audio directly. kilobeat, as a networked livecoding platform, falls into this latter category: players' actions are sent to other players with no timing guarantees, but player code may optionally synchronize to a common clock present in each player's instance. In terms of livecoding language and platform, kilobeat is also related to projects such as Gibber [6], EarSketch [7], and Jazzari,[3] which run in the browser and allow the player to make music with JavaScript.

Conceptually, kilobeat is inspired by the practices of *bytebeat*. To briefly summarize, the term "bytebeat" refers to music generated by programs—typically, terse programs consisting of a single expression, mostly comprised of arithmetic operators with access to a single variable (`t` for time). Participating in bytebeat involves devising or discovering these

[1]https://github.com/corrados/jamulus
[2]https://www.cockos.com/ninjam/
[3]https://github.com/jackschaedler/jazzari

short musical programs, trying out, appreciating, and tweaking others' programs, and sharing these programs with the community. Typically, bytebeat expressions are evaluated to produce each sample of an 8-bit, 8kHz mono audio stream[4], with access to a variable that is incremented with each sample. The low sample rate and bit depth give bytebeat its retro, chiptune aesthetic, but sites that run bytebeat code generally allow modifying these parameters.[5] For a more complete introduction to bytebeat, see [2].

kilobeat inherits from bytebeat the focus on short, single-expression programs for generating audio at the sample level. In the interest of clarity (particularly for those who know something about audio synthesis but not about bytebeat), it eschews unsigned 8-bit integer samples and bitwise arithmetic in favor of floating-point samples and arithmetic (including familiar math functions such as `sin` & co.).

2. DESIGN

2.1 Musical Expressions

2.1.1 *Language*

In kilobeat, players perform by writing and running code expressions. Once a player runs code, the expression is evaluated to generate each consecutive sample of output audio, one-by-one. These expressions are written in JavaScript, as supported by players' browsers. The differences from vanilla JavaScript are as follows:

- Math functions are in global scope, and thus do not require the `Math.` prefix.
- `Math.E` and `Math.PI` are available as `e` and `pi`, respectively, and `Math.random` is available as `rand`.
- The utility function `choice` (which chooses randomly among its arguments) is available.
- The special variables `x`, `y`, `z`, `acc`, `t`, `dt`, `sr`, and `now` are available. These are described in more detail below.
- There is an additional syntax for using `sin` in a way that accumulates phase, described below.

kilobeat exposes the `Math` functions globally (and in a few cases with shorter names) for convenience; this is also the rationale for including the non-standard (but trivial) function `choice`. `x`, `y`, `z` have no predefined purpose; they are provided in case the player needs to keep track of things between samples. `t`, `dt`, `sr`, and `now` are all provided for timing. `t`, as in bytebeat, represents the time of the current sample; that is, the sample the expression is generating. Unlike bytebeat, `t` is measured in the sample-rate-independent unit of seconds rather than samples. `sr` is the sample rate of the output audio, and `dt` is the time in seconds between samples (equal to `1/sr`). `now` is the time at which the expression was submitted by the player to replace the old code.

It is worth mentioning one special feature, included to simplify some common synthesis operations. In addition to the standard invocation `sin(x)`, where `sin` is a pure (stateless) function of phase, `sin` may also be invoked as `sin[i](dx)`. In this invocation, the function acts like a phase-accumulating oscillator (like `osc~` in MSP or Pure Data); this variant is a shorthand for `sin(acc[i] += dx)`, and this syntax is included to simplify frequency modulation. `acc` is an array of eight floats; it is used internally by the alternate `sin` syntax, but may also be accessed by the player directly.

[4]Suitable for redirecting into `/dev/dsp` or piping into `aplay` on Linux.

[5]Gregg Tavares's HTML5 Bytebeat (https://github.com/greggman/html5bytebeat) also supports a variant called "floatbeat," which, like kilobeat, uses floating-point arithmetic.

Code	Description
`0`	Silence
`rand()*2-1`	Noise (from -1 to 1)
`sin(2*pi*200*t)`	Sine tone (200 Hz)
`(t%.005)/.005`	Sawtooth wave (200 Hz)
`(t%.005)>.0025`	Square wave (200 Hz)
`sin(2*pi*400*t)* sin(2*pi*200*t)`	Ring modulation
`sin(2*pi*400*t)* (1+sin(2*pi*200*t))/2`	Amplitude modulation
`sin(2*pi*400*t+ sin(2*pi*200*t))`	Phase modulation
`sin[0](2*pi*dt*(400+ 10*sin(2*pi*200*t)))`	Frequency modulation (using phase-accumulating oscillator)
`[300,500,800] .map(f=>sin(2*pi*f*t)) .reduce((a,b)=>a+b)/3`	Chord comprised of three sine tones
`[.3,.4,.5][floor(t%3)]`	Sequence (cycles through the list, one element per second)
`t<x?(t-x):(x=t+ choice(.6,.3,.2,.1),0)`	Random rhythm
`t-now<5`	Timer; goes off five seconds after the player pressed "Run"
`min(t-now,1)`	Envelope: ramps from 0 to 1 over the course of one second

Figure 2: Examples of kilobeat expressions.

2.1.2 *Expressions*

Some examples may help put all this in context. The simplest kilobeat expression (and each players' initial code when they join a session) is `0`. Each time this expression is evaluated, it evaluates to 0, so this code corresponds to endless silence. A slightly more lively expression, `sin(2*pi*400*t)`, produces a pure 400 Hz tone. Additional examples are included in Fig. 2. To provide the player with a reference and starting point, kilobeat includes most of these examples as "presets": simple expressions that perform common operations.

Expressions may be combined in all the usual ways. At the outermost level, `+` will mix two audio signals, while `*` will perform ring or amplitude modulation (or equivalently, apply an envelope). Nesting expressions, as in the argument to `sin`, enables techniques such as FM synthesis. Expressions may be chained via the comma operator, which may be useful when performing assignments. Branches are available through the ternary (`?:`) operator (and implicity by short-circuiting `&&` or `||`), and anonymous functions are allowed. In short, players may construct arbitrarily complex programs as single JavaScript expressions—but the short codebox is intended to encourage players to keep their expressions relatively simple.

2.1.3 *Execution*

When a player clicks the "Run" button (or presses `Ctrl+Enter`), kilobeat compiles their code in a new audio worklet. If the code is syntactically valid, the new expression is executed to generate all subsequent samples, and success is indicated with a brief green flash. If the code has a syntax error, this is indicated by a brief red flash and shake, and the player can get more detail by hovering over their expression. In this case, the last valid code continues running, so as to minimize audible disruptions.

2.2 Networking & Collaboration

kilobeat's name is intended to suggest many concurrent instances of bytebeat. Musical collaboration is at the heart of kilobeat, and this manifests itself in both kilobeat's interface and in player code.

2.2.1 *Interface*

In addition to hearing other players' output, each player can *see* other players' output visually on an oscilloscope and spectrum analyzer. Beyond observing their output, each player can see what every other player is doing. Whenever a player edits their code, selects text, or moves their cursor, the action is made visible to every other player, Google Docs-style. Additionally, players can see when others run their code and the result (acceptance or rejection). These features are intended to facilitate some of the intentionality and transparency which is conveyed visually when playing together in-person.

Furthermore, kilobeat provides a virtual room to play in. Each player generates a single channel of audio, but each player has a position and orientation in a virtual 2D room, and players' mono audio streams are combined according to these positions to create a stereo output (using the Head-Related Transfer Function). Players are free to move around the room at any time (by dragging their speaker on a canvas and scrolling to turn around), and, as with the code editor, these motions are visible to all other players. This spatialization is another feature inspired by the experience of playing together in-person, and it serves as another way to aurally distinguish and isolate each player's stream.

2.2.2 *Dependence*

Collaboration can also occur at a more integral level in kilobeat. By referring to other player IDs, players can write code that directly depends on another player's audio. For instance, the expression `p0` will simply replicate Player 0's output (with a small delay), while `sin[0](2*pi*(440+100*p0)*dt)` will modulate a tone's frequency by Player 0's output. This feature can be exploited to create complex webs of synthesis without requiring any one player to take on the full burden of that complexity. Dependences provide another avenue for complexity to emerge from simplicity, in addition to the traditional strategy of

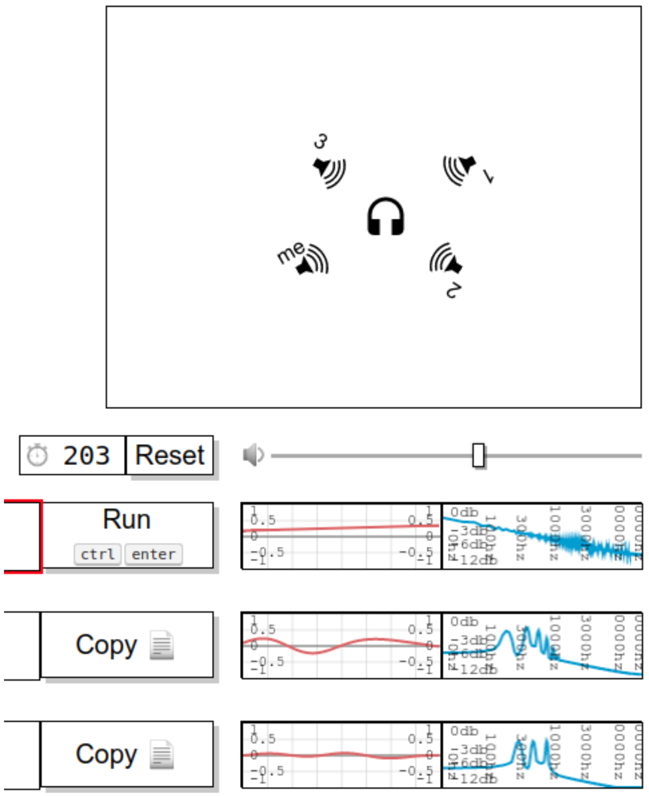


Figure 3: Right side of kilobeat interface, including speaker positions, run/copy buttons, oscilloscopes, and spectrum analyzers for each player.

playing simple things together.

2.2.3 *Other Features*

kilobeat offers two features which do not facilitate collaboration but are related to it. The first is offline mode: kilobeat can be played offline, without requiring a server or other people. Naturally, playing kilobeat with friends is recommended for best results, but offline mode can be useful for experimentation or demonstration. In offline mode, the player may create new virtual players and control their codeboxes as well as their own.

kilobeat also offers a recording mechanism. This does not record audio, which is readily possible with external tools; rather, it records all player actions (edits, cursor movements, etc.), such that a performance may be repeated as it occurred. This recording/playback functionality comes with an interesting caveat/feature, which is further discussed in section 3.3.

2.3 Implementation

kilobeat consists of a client and server. The server is minimal: like a simple chat server, it relays messages between clients and performs light bookkeeping. The kilobeat server is implemented in Python using Flask[6] and Flask-SocketIO.[7]

The client is written in JavaScript and runs in each player's browser. It communicates with the server using

[6] https://flask.palletsprojects.com/
[7] https://flask-socketio.readthedocs.io/

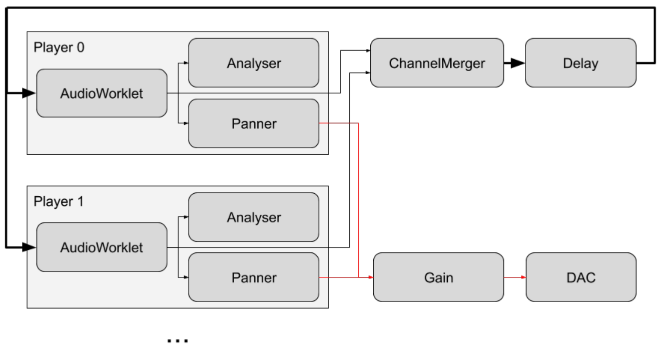


Figure 4: WebAudio graph on each client.

WebSockets and performs audio synthesis locally with Web Audio. For synthesis, each player's code is executed in a separate Audio Worklet; the source for the corresponding AudioWorkletProcessor is generated at runtime and filled in with the player's code expression. As depicted in Fig.4, each worklet node is connected to a panner node for spatialization, an analysis node for the oscilloscope and spectrum analyzer, and a channel merger node[8] for feedback/dependence. The client is built on the libraries socket.io,[9] CodeMirror,[10] JSHint.[11]

3. DISCUSSION

3.1 Composition & Performance Structures

From experience playing with kilobeat "at home" and in concert, certain usage patterns deriving from kilobeat's design have become apparent. These patterns are not explicitly a part of kilobeat's design, but nonetheless emerge from it.

It is often useful to converse in the midst of performance. These may be performance instructions ("let's move on to the next section"), improvisational remarks ("check this out!"), explanatory comments ("this does that like so"), or suggestions for tweaking details ("p2, speak up a little"). kilobeat has no built-in chat, but comments serve much the same function. Prepending a line starting with `//` (a JavaScript single-line comment) before an expression has no effect on the output but allows for chatting with or posting notices for other players, in much the same way a Google Doc can serve as an ad-hoc chatroom or bulletin board. Naturally, chat history is not shown (unless players continue to insert *additional* comments), but chat history *is* recorded with all other performance data by kilobeat's recording feature.

When playing with dependences, it's often useful to come up with a model for their use—for example, by having players take different roles. In kilobeat, all expressions are evaluated for every sample, so there is no built-in distinction between control and synthesis. One effective way to play with dependences is to reintroduce this distinction in some way that suits the composition, by having players take different roles and establishing dependences between them based on those roles. For example, one player might write an expression that chooses notes (e.g. outputting frequencies or MIDI pitch values) while another player, depending on the first, writes an expression that renders those notes, effectively specifying timbre. Or one player might output a value signaling a fundamental frequency or a chord that other players build upon and realize. In both of these cases, the first player takes on more of a "control" role while the second handles "synthesis," but these lines can quickly become fuzzy: the second player is always free to choose how much of an effect the first player has on their output, and the first player may have an expression that changes at audible rates, ultimately producing a timbral effect.

3.2 Pedagogy

kilobeat was developed with collaborative musical performances in mind, and it has been put to this task in two concerts with the MIT Laptop Ensemble. However, kilobeat may also have applications in the classroom. Its emphasis on generating audio at the sample-level, coupled with its hiding lower level details (by using floats instead of integers and abstracting away sample rate and bit depth) and immediate feedback (via speaker, oscilloscope, and spectrum analyzer), make it suitable for demonstrating synthesis concepts. Its collaborative nature and ease of access make it amenable to experimentation and discussion. I hope that kilobeat, or something like it, may serve as a pedagogical tool for DSP and synthesis education in the future, and I propose that future work evaluate its fitness for this purpose.

3.3 Caveats

As mentioned previously, kilobeat's server is minimal, serving only to broadcast messages between clients. All synthesis is strictly client-side, and player actions are not precisely timed. This loose timing can be compensated for by scheduling for the future using the global clock, or by using dependences to ensure synchronization between voices.

One other consequence of the client-side synthesis and loose synchronization of client state is that kilobeat performances are naturally aleatoric. In particular, kilobeat makes no effort to synchronize JavaScript's pseudorandom number generator between clients,[12] so invoking `rand()` or `choice()` will produce different results for different players, and it will produce different results when replaying a recorded performance. This effect is imperceptible when randomness is employed at the sample level (noise sounds like noise), but it may become increasingly pronounced with randomness at higher levels of organization (rhythms, phrases, sections).

4. CONCLUSION

kilobeat is a collaborative livecoding platform with some unusual features: sample-level audio generation, spatialization, dependences between players, and aleatoric replays. kilobeat's source code is available under the MIT license at https://github.com/ijc8/kilobeat, a demo video is available at https://youtu.be/SKCyLakDqkU, and a live demo is available at https://ijc8.me/kilobeat.

5. ACKNOWLEDGMENTS

[8]Followed by a delay node, to avoid creating an illegal cycle.
[9]https://socket.io/
[10]https://codemirror.net/
[11]https://jshint.com/

[12]Even if it used a shared seed at the start, the state would quickly become desynchronized due to the loose action timing discussed previously.

In addition to the libraries mentioned in Sec.2.3, kilobeat is particularly indebted to two open-source projects: I used Arthur Cabott's Audio DSP Playground[13] as my starting point and style guide, and adapted spatialization code from Boris Smus's WebAudio demos.[14]

I originally developed kilobeat at MIT for the course SOUND: PAST & FUTURE[15] in Spring 2020 and for the MIT Laptop Ensemble (FaMLE). Thanks to FaMLE for playing, Tod Machover for early feedback, Jason Freeman for later feedback, and Ian Hattwick for both.

6. REFERENCES

[1] J.-P. Cáceres and C. Chafe. Jacktrip: Under the hood of an engine for network audio. *Journal of New Music Research*, 39, 09 2010.

[2] V.-M. Heikkilä. Discovering novel computer music techniques by exploring the space of short computer programs, 2011.

[3] R. Kirkbride. Troop: A collaborative tool for live coding. In *Proceedings of the 14th Sound and Music Computing Conference*, 2017.

[4] Q. Lan and A. R. Jensenius. Quaverseries: A live coding environment for music performance using web technologies. In A. Xambó, S. R. Martín, and G. Roma, editors, *Proceedings of the International Web Audio Conference*, WAC '19, pages 41–46, Trondheim, Norway, December 2019. NTNU.

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

[6] C. Roberts and J. Kuchera-Morin. Gibber: Live coding audio in the browser. In *ICMC*, 2012.

[7] A. Sarwate, T. Tsuchiya, and J. Freeman. Collaborative coding with music: Two case studies with earsketch. In J. Monschke, C. Guttandin, N. Schnell, T. Jenkinson, and J. Schaedler, editors, *Proceedings of the International Web Audio Conference*, WAC '18, Berlin, Germany, September 2018. TU Berlin.

[13] https://github.com/acarabott/audio-dsp-playground
[14] https://github.com/borismus/webaudioapi.com
[15] http://spf.media.mit.edu/