ScoreCard: Generative music programs as QR codes

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

Jason Freeman
Georgia Institute of Technology
Atlanta, Georgia, United States
jason.freeman@gatech.edu

ABSTRACT

QR codes are typically used to store small pieces of metadata (URLs, product codes, contact information) which serve as links or pointers to larger resources (websites, product databases, people). In this work, we explore the possibility of repurposing QR codes to store complete, self-contained generative music pieces, and the resulting technical and musical challenges and opportunities. We introduce a web application called ScoreCard, which enables users to scan and play QR codes (e.g. from "printed programs" on physical cards or from other users' screens) containing generative music pieces in the form of WebAssembly programs. We present several musical examples and pieces written for or ported to this medium.

1 Introduction

Music is distributed in many mediums. Scores capture notation on paper; analog media such as vinyl records and cassette tapes transmute sound from ephemeral pressure waves into more durable forms. Digital formats instead store audio as bitstreams which can be readily copied and shared, at the cost of the physicality and visibility of analog formats.

QR (Quick Response) codes serve as a bridge between the physical and digital. Per de Seta [10], QR codes were developed by Denso Wave for use in the automotive industry, but they have since become ubiquitous in media, advertising, and everyday transactions. In these contexts, QR codes are frequently deployed as a better way to direct an audience to a digital destination. Instead of reading a URL written as text and manually entering it, one simply scans a code.

Used in this way, QR codes serve as real-world hyperlinks—printed anchor tags readable by a user's smartphone. Unfortunately, this usage makes QR codes susceptible to link rot, and a QR code that worked yesterday may be a broken link tomorrow. Like a dangling pointer, the QR code remains readable, but the content it pointed to is lost.

QR codes are often used to share music in the form of links to streaming services or band websites. But sharing a link to music is not quite the same as sharing the music itself. As mentioned, links break, and even when they don't, access is mediated by whoever operates the linked site, and access is contingent on the service provider's terms.

Figure 1: ScoreCard playing a tiny generative music program (scanned from a QR code) in a mobile browser

In this work, we employ QR codes as a physical/digital medium for music storage. Rather than using a QR code to store a *link* to a piece of generative music, we propose to store *the piece itself* in the QR code. In this way, as long as the code can be scanned, the piece survives and may be played. We demonstrate this idea via *ScoreCard*,[1] a web application for playing generative music pieces from QR codes repurposed to serve as musical "score cards". We discuss the design and implementation of ScoreCard, the unique affordances and constraints of our approach, and the development of a few ScoreCard pieces as case studies.

[1]ScoreCard is available at https://ijc8.me/s, and its source code is available at https://github.com/ijc8/scorecard.

Licensed under a Creative Commons Attribution 4.0 International License (CC BY 4.0). **Attribution**: owner/author(s).

Web Audio Conference WAC-2024, March 15–17, 2024, West Lafayette, IN, USA.

© 2024 Copyright held by the owner/author(s).

2 Design

In ScoreCard, the piece *is* the program *is* the notation. The visual representation of a piece, as a QR code, contains the executable representation of a piece, as a WebAssembly binary, which encompasses all possible audio representations of a piece in its execution paths. This is roughly what we mean by "self-contained".

Of course, every piece needs a player. A vinyl record, though a "self-contained" musical representation, is mute without a turntable and speaker. Even a music box, which contains both "score" (a tiny piano roll) and "orchestra" (tiny tines), requires something to turn the crank and someone to hear the song. Nothing is truly self-contained; a statically-linked binary still depends on an OS, and even bare-metal embedded firmware still depends on the 'metal'!

Nonetheless, some things are more self-contained than others, and in ScoreCard we aim for pieces that are highly self-contained. In particular, the QR code contains an entire WebAssembly binary. That binary is expected to export up to two functions (`process()` and optionally `setup()`),[2] and no imports are allowed. This means that there is no additional JavaScript glue code (as typically generated by Emscripten or Rust's `wasm-bindgen`), and indeed, not even WASI imports. There is no I/O at all except through the two exported functions. Rather than storing or generating symbolic data (such as 'notes' or 'events') that must be further interpreted by a player, ScoreCard programs are responsible for generating audio directly: like a music box, they contain both score and orchestra, but unlike MUSIC-N languages [20], the question of how (or whether) to separate score and orchestra, or control and synthesis, is left up to the composer/programmer.

Furthermore, we avoid additional dependencies which might be used to cram larger executables into the QR code at the expense of player complexity, such as gzip or Brotli. For the same reason, we avoid structured binary schemes such as BSON or protobuf, which could be used to pack in additional metadata with the WebAssembly. (We do permit some metadata to optionally be included in the Wasm binary itself, as described in §5.1.)

In addition to self-containment, we also design ScoreCard for ease of use. The QR codes that encode the executable double as valid URLs that open the program in the ScoreCard web app when scanned by a smartphone. (The way this is accomplished is described in §4.2.) ScoreCard pieces may be distributed on paper (such as stickers or physical "score cards"), or they may spread from person-to-person by pointing one user's camera at another's screen. For simplicity, we restrict ScoreCard programs to a single QR code (instead of permitting them to consist of arbitrarily many QR codes scanned in sequence). This sharply limits the size of ScoreCard programs, which serves as a technical challenge and an aesthetic constraint, as elaborated in §4.3.

[2]To simplify things further (with the aim of reducing code size), we expect `process()` to return the next sample (rather than filling a block of samples), and we assume a fixed sample rate of 44100 and format of 32-bit floats. If a piece exports `setup()`, we pass it a 32-bit seed for e.g. initializing a PRNG. To save space, the actual export names are single characters (`p` for `process`, etc.), but we use the long names in discussion and source code for readability; build tools handle the renaming.

3 Related Work

Our work on ScoreCard is inspired by work on *fantasy consoles*, modern game platforms that adopt retro aesthetics and intentional technical constraints. Particularly relevant are Francesco Cottone's *Rewtro* [8] which supports loading games from "QR-Carts" (paper game cartridges consisting of three QR codes), and Bruno Garcia's *WASM-4* [12], in which games take the form of WebAssembly binaries. Compared to Rewtro, ScoreCard aims for greater simplicity, both in specification and in usage. Simple specification means that ScoreCard pieces have both more responsibility (as they must implement synthesis) and more freedom (as they are not limited to preset sounds). Simple usage means that users only need to scan a single QR code, and that code *also serves* as a link to the ScoreCard player (at the expense of some executable space, as discussed in §4.2).[3] But, as in Rewtro, the use of QR codes means that anyone can produce their own score cards using a printer, and music can be "pirated" using a copy machine—or otherwise shared any other way that people share images.

ScoreCard draws inspiration from the notion of "printed programs" as in collaborative computing environments Dynamicland [13] and Folk Computer [7]. In these environments, programs are invoked by pulling out a piece of paper, which contains both the human-readable program source code and a computer-readable marker (such as an AprilTag) that points to the corresponding program in a database. In contrast, in ScoreCard the computer-readable QR code instead contains the entire program, and the program is self-contained rather than fitting in as a piece of a larger system.

ScoreCard is informed by our previous work on Alternator, a general-purpose generative music player [5]. Like ScoreCard, Alternator features a web-based player for generative music bundles that may incorporate WebAssembly. In Alternator, these bundles may be arbitrarily large and include as many assets as needed, possibly including an entire language runtime (such as Csound, ChucK, libpd, or CPython). Thus, any ScoreCard piece may easily be turned into an Alternator bundle, but not vice-versa: score cards are a more constrained and exacting medium for generative music. The two projects share a conception of a generative music piece as an audio-generating program, but diverge in philosophy. ScoreCard turns away from the familiar aesthetic of Alternator (which seeks to emulate streaming music players with regard to convenience and usability, implicitly presenting music as something infinitely available on-demand) towards the perhaps quixotic approach of storing music on physical cards, with a corresponding emphasis on visibility, tangibility, and personal interaction. To that end, ScoreCard eschews the notion of a central repository for musical works, instead offering a decentralized model of musical creation and propagation that bears more resemblance to the cassette era than the streaming era.

ScoreCard also participates in a lively tradition of size-constrained creative coding. Examples include Peter Salomonsen's work on "WebAssembly Music" [18], bytebeat (in which musical pieces consist of very short code expressions) [14], sctweets [16], code golfing, and of course the demoscene, which is well-known for packing a lot of art into little programs ('demos') using every possible trick [3].

[3]In Rewtro, the QR codes are purely binary data, and thus cannot be read directly by smartphone camera apps.

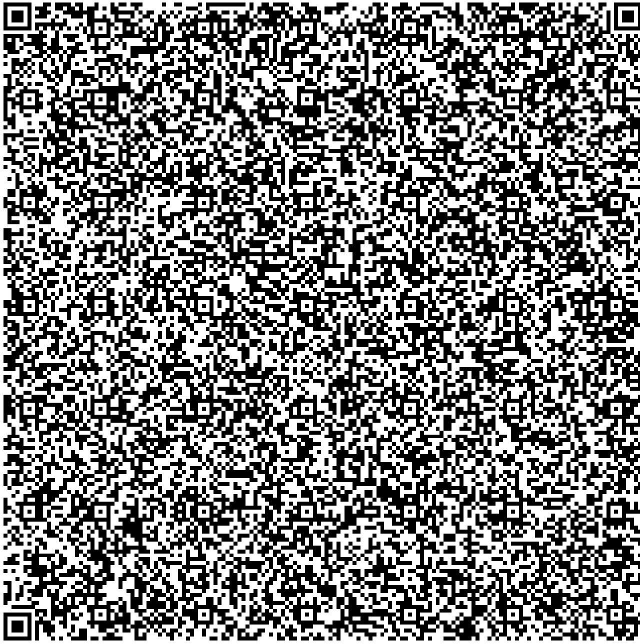

Figure 2: An implementation of Terry Riley's *In C* in C, compiled to WebAssembly, encoded in a QR code.

The way we have gone about writing ScoreCard pieces is informed by our previous work on Aleatora [4], in which everything is a stream, from samples on up. Writing in C, we lack the luxury of Python's generators, but we use Simon Tatham's technique [19] to fake coroutines to similar effect as described in §4.3.

Finally, we note an affinity with other projects that repurpose utilitarian supply-chain technologies to creative ends. Examples include Electronicos Fantastico!'s *Barcoder* [11], which transforms ordinary barcode scanners into electronic instruments, and the first author's work repurposing RFID tags as input devices for interactive systems [6].

4 Implementation

4.1 The ScoreCard Player

The ScoreCard player takes the form of a web application, as shown in Figure 1. It can play score cards passed in via query parameter (as in the case of URLs from QR codes), as well as scan codes directly from the browser.[4] The query parameter is decoded into a binary blob, which is passed to an AudioWorklet where it is loaded as a WebAssembly module and instantiated. The player also allows for rudimentary creation and modification of score cards by writing WebAssembly (in text format) directly; we intend to expand this functionality in the future.

The "Listen" tab, used for playback, displays the score card on the phone screen (re-encoded with the player's URL). Showing the QR code means that score cards can be passed around by scanning one person's screen with another person's camera. The interface also displays the title (if provided) and size of the loaded piece, as well as the current playback time. Playback controls include play/pause and reset. Resetting generates a new random seed. The user can also edit the seed directly, or *lock* the seed by tapping the die icon. Once locked, the seed survives resets and goes into the QR code along with the program, so that an interesting seed may also be shared by scanning.

4.2 Encoding Issues

Storing a generative music program in a QR code presents several challenges, the most obvious of which is size. A QR code (in its largest configuration, with 177x177 pixels and minimal error correction) can contain at most 2,953 bytes. Thus, from the outset, we are limited to encoding binaries smaller than 3 kilobytes.

However, there are further constraints. Given the design requirement that our codes be readable by standard readers (such as those built in to smartphone camera apps), we cannot just store a binary blob. Ideally, the user should be able to scan the code from their camera app, and then tap once to open the generative music program in the web-based ScoreCard player. Accomplishing this kind of flow requires encoding a URL which in turn encodes the WebAssembly executable.[5]

Naively, we could encode the executable using base64, which is commonly employed on the web for wrapping binary data in a text package, resulting in a URL like `https://ijc8.me/s?c=aGVsbG8gd29ybGQh...` Base64 encodes data as a sequence 6-bit Base64 digits; represented as an ASCII or UTF-8 string, it uses four bytes of output data per three bytes of input data, incurring an overhead of $33.\overline{3}\%$. Thus, we lose 25% of our storage space for the executable, leaving us with at most 2,214 bytes even before we subtract the space for the fixed part of the URL.

Fortunately, the QR code specification affords some tricks. In particular, QR codes store data as a sequence of *segments*, each of which can be encoded in one of four modes. In byte mode, each byte takes (unsurprisingly) 8 bits, but other modes include numeric mode (which encodes the ten decimal digits, taking $3\frac{1}{3}$ bits per digit) and alphanumeric mode (which encodes the 26 uppercase letters, 10 digits, and 9 symbols, using $5\frac{1}{2}$ bits per character).[6] By choosing the mode carefully, we can store the program in a way that is both readable as a valid URL *and* more efficient than encoding a base64 string in byte mode.

For example, we could simply encode the entire Wasm blob as a long string of digits. As noted by Adam Langely [15], the QR code scheme of encoding a digit in $3\frac{1}{3}$ bits (3 digits in 10 bits) is $\log_2 10/\frac{10}{3} \approx 99.66\%$ efficient—far better than the 75% efficiency offered by base64. Indeed, we can store up to 2,943 binary bytes this way, losing just 10 bytes to encoding overhead.

Unfortunately, this scheme is foiled by the quirks of real readers, as we discovered while testing. On iOS, scanning a QR code that results in a long URL (>4000

[4]The player also supports dragging in .wasm files directly, which is convenient for score card development.

[5]We also considered the possibility of storing the link and the binary as separate segments, so that the code could be scanned outside the player to get the player URL and inside the player to load the binary. Unfortunately, this approach is sensitive to reader implementation details: on Android, the URL scans, and the binary data is replaced with "Unknown encoding", which is workable. But iOS refuses to scan such a code.

[6]Note that this mode is unfortunately *not* sufficient for encoding base64 strings due to the lack of lowercase letters.

Base	Max size (bytes)	String length	Efficiency ▽
10	2943	7088	99.66%
43	**2913**	**4295**	**98.65%**
64	2214	2952	74.97%
16	2148	4296	72.74%

Table 1: Efficiency comparison of encoding schemes for embedding a binary in a QR code such that it can be scanned by smartphone camera apps as a URL query parameter.

characters) may, in some circumstances, silently drop trailing components; a code that scans successfully as `https://ijc8.me/s?c=10639365947...` on Android may scan as `https://ijc8.me/s?c` on iOS, discarding the executable entirely. Thus, there is an additional surprise constraint to keep the URL relatively short.

As a result, ScoreCard employs a base43 encoding. Why such as unusual base? We use the 45 characters supported by the QR spec's alphanumeric mode, minus the two (% and space) that are URL-unsafe in query parameters, leaving us with the 43-character alphabet `0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ$*+-./:` and a resulting efficiency of $\log_2 43/5.5 \approx 98.66\%$. This is not quite as good as digit encoding (we lose 30 more bytes of executable space) but still much better than base64 (we get 699 bytes back), and the resulting URLs scan more reliably on iOS. We present a comparison of different encoding schemes in Table 1.

All told, after the base43 encoding and the URL prefix (`https://ijc8.me/s?c=`), we are left with 2,892 bytes for the executable, sacrificing about 2% of our storage capacity in exchange for codes that scan in ordinary QR code readers. Note that, although the QR code decodes to a valid URL for usability as described in §2, the URL may be still decoded by a ScoreCard player hosted on a different domain (or indeed a standalone player or progressive web app used without Internet), since it encodes the entire program in a self-contained manner.

4.3 Musical/Technical Issues

With the executable encoding scheme settled, we are left with challenge of building a small binary in the first place. Given just under three kilobytes, how much music can we make?

This question is complicated by the fact that the program needs to not only encode musical material, but also synthesize it as audio. Calling `sinf()`, for example, immediately adds 5,600 bytes to the size of the executable; even `powf()` (useful for implementing a typical "MIDI to frequency" function) adds 578.[7] Similarly, heap allocation is a death sentence. Calling `malloc()` pulls in an allocator, blowing up the binary by 4,022 bytes. These issues necessitate custom implementations or alternate designs.

We also have the question of language. Using any language that comes with a large runtime is impossible, as the runtime must *also* fit in < 3kB. This immediately eliminates most modern computer music languages[8] and interpreted or bytecode-interpreted languages.[9] The general issue with such languages in size-constrained environments is that the runtime contains code for everything the language can do, even if the actual program uses only a fraction of it; a program that does nothing is still the size of the entire runtime. Paying for unused features is not viable in such a constrained medium, so this effectively limits us to compiled languages.

Thus, we're left with compiled languages with small or non-existent runtimes that can compile to WebAssembly, such as C, C++, Zig, Rust, and AssemblyScript. In our examples, we stick with C, using Emscripten [21] (with `--no-entry`) to generate a WebAssembly binary sans JavaScript glue code.

Given the choice of C, we must then write music in it. In order to concisely express music from the level of samples on up, we adapt Simon Tatham's work [19] on simulating coroutines in C (with modifications to avoid heap allocation) in order to attain an experience similar to using generators in languages such as Python or JavaScript, such that we may `yield` a sequence of values from a function, suspending and resuming control flow without explicitly writing a state machine. These generator-esque macros allow us to write relatively straightforward code despite the fact that the ScoreCard player is driving generation from the outside (by repeatedly calling `process()`). They create the appearance that the piece's code drives the music by yielding samples as they are generated, and they enable the composer/programmer to describe individual musical processes and compose them in sequence, in parallel, or in function (e.g. applying effects), as demonstrated in our examples.

5 Examples

5.1 Writing simple examples from scratch

Let's begin with a trivial example, a noise generator, to demonstrate all the parts of a ScoreCard piece/program:

```c
1  #include <stdlib.h>
2
3  const char title[] = "noise example";
4
5  void setup(unsigned int seed) {
6      srand(seed);
7  }
8
9  float process() {
10     return rand() / (float)RAND_MAX * 2 - 1;
11 }
```

Both `title` (which is exported as metadata for the player to display) and `setup()` are optional and may be omitted to save space. The only essential ingredient in a ScoreCard program is `process()`, as demonstrated in the next example,

[7]These are the implementation sizes from **musl**, a lightweight implementation of the C standard library, which is already much smaller than typical implementations such as glibc.

[8]With the possible exception of FAUST, which is compiled; we intend to investigate this possibility in the future.

[9]Even tiny runtimes designed for embedded use are generally too large: MicroPython takes well over 100kB, and the NanoVM for Java still takes 8kB.

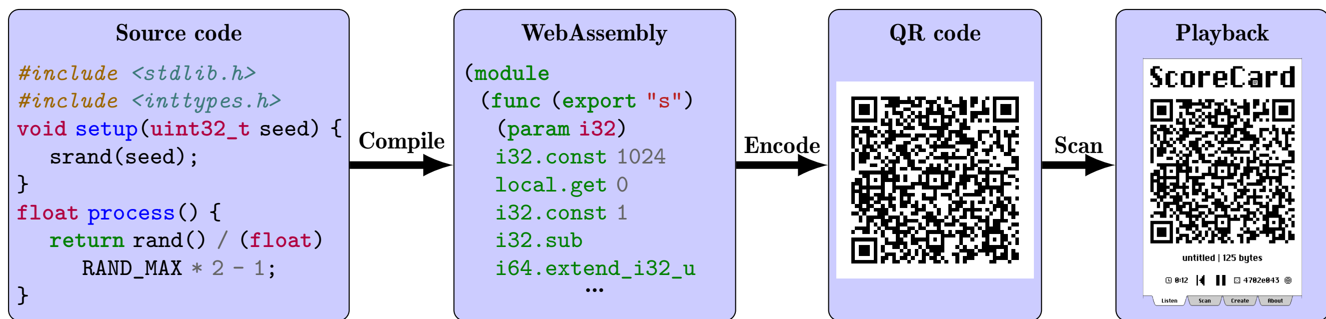

Figure 3: Progression of a ScoreCard piece, from composition/programming to playback in a browser

which plays a little tune using a bytebeat [14] expression.

```c
const char title[] = "bytebeat example";
int sample = 0;
float process() {
    int t = sample++ / 5;
    // Bytebeat expression:
    signed char x = t * (42 & t >> 10);
    return (float)x / 128;
}
```

5.2 deck.h: a library for composing cards

In order to avoid rewriting the same basic constructs for each piece, we have created a header file, `deck.h`, which contains a number of common building blocks and utilities, such as oscillators, envelopes, filters, and random(ness) functions. The following example demonstrates some of these:

```c
#include "deck.h"

card_title("Quirky FM ramps");

float t, dur, start, end, phase;
float mod_depth, mod_phase, mod_ratio;
float dur_options[] = {0.25, 0.5, 1.0, 2.0, 4.0};

void setup(unsigned int seed) {
    srand(seed);
    dur = choice(dur_options);
    start = uniform(0, 1000);
    end = uniform(0, 1000);
    mod_ratio = uniform(1, 100);
    mod_depth = uniform(0, 1);
}

float process() {
    if (t > dur) return 0;
    float freq = ramp(t, dur, start, end);
    float mod_freq = freq / mod_ratio;
    freq *= 1+mod_depth*sqr(&mod_phase, mod_freq);
    float out = sqr(&phase, freq) * env(t, dur);
    t += dt;
    return out;
}
```

`deck.h` also includes our adapted version of Simon Tatham's coroutine macros [19]. These provide an easy way to "invert" the apparent control from the player (which calls `process()` repeatedly to generate each sample) to the program, which in turn makes it easier to manage complexity as pieces gain structural layers, as in the following example:

```c
#include "deck.h"

card_title("lick spiral");

// A familiar tune
struct { char pitch; char dur; } notes[] = {
    {62, 1}, {64, 1}, {65, 1}, {67, 1},
    {64, 2}, {60, 1}, {62, 1},
};

float process() {
    static float phases[2];
    static float freq, dur, t = 0;
    static int offset = 0, i;
    gen_begin;
    for (;; offset = (offset + 1) % 12) {
        for (i = 0; i < SIZEOF(notes); i++) {
            freq = m2f(notes[i].pitch + offset);
            dur = notes[i].dur * 0.25;
            for (; t < dur; t += dt) {
                // Shephard scale shenanigans
                float p = t / dur;
                p = (i + p) / SIZEOF(notes);
                p = (offset + p) / 12;
                float x = p*tri(&phases[0],freq/2)
                    + (1-p)*tri(&phases[1],freq);
                // `yield()` suspends the execution
                // of `process()`, picking up from
                // where we left off next time.
                yield(x * ad(t, dur/8, dur*7/8));
            }
            // Subtract `dur` (instead of resetting
            // `t` to 0) to avoid accumulating
            // error from truncating dur/dt.
            t -= dur;
        }
    }
    gen_end(0);
}
```

In Appendix A, we include a more complete example: an implementation of Terry Riley's *In C* (appropriately named `in.c`). Figure 2 contains the resulting score card. Because *In*

C is intended to be played by multiple performers working their way through the same score semi-independently, `in.c` is an example of a score card that, although self-contained in a technical sense, benefits from being played at the same time as other cards (or other instances of the same card) on other devices.

6 Future Work

We plan to expand on the rudimentary card-creation functionality in the player (which currently allows users to write WebAssembly by hand in text format). A small compiler for bytebeat expressions or perhaps even Tidal-esque pattern notation [17] would enable users to create cards by editing short snippets on their phones. A web-based card creator with a built-in compiler would enable users to write cards in C (or other languages) without first installing a compiler toolchain. Either development would provide alternatives to writing WebAssembly, which seems more suitable for 'hacking' existing cards than composing cards from scratch.

An easy extension of ScoreCard would allow pieces to take in audio input, enabling audio effects to be stored on QR codes.[10] Such an extension raises the possibility of chaining together ScoreCards in a modular fashion, treating each individual card as a synthesizer module or unit generator. In this work, we prefer to focus on the simplicity of ScoreCard and the feasibility of self-contained musical programs, but the prospect of modular, connectable ScoreCards is a tempting direction for future exploration.

We note that some cards, such as `in.c` (in Appendix A), are already intended to be combined (possibly with themselves) by being played simultaneously, forming a compound piece realized by multiple participants. In the future, we would like to explore the possibility of automatic synchronization between devices, as in Brian Barth's *Your Terms of Service* [2]. (This is more complex in our use-case because the devices may be playing different parts.)

Another intriguing direction involves expanding on the visual aspect of ScoreCard. Building on techniques described by Russ Cox [9], it may be possible to visibly embed 'album art' in score cards by including carefully-chosen blobs in WebAssembly binaries.[11] Another possibility is highlighting activated (read, written, executed) parts of the binary in the QR code itself, while the program is running. This would likely require instrumenting a WebAssembly interpreter (forgoing the browser's fast, built-in VM) or dynamically patching WebAssembly binaries themselves, as in Jack Baker's work on hacking WebAssembly games [1]. Neither approach is easy, but such a visualization might do a great deal to make the process feel more lively.

Finally, we intend to continue exploring how much can be done with just a little code. We believe ScoreCard presents a delightful opportunity to explore the expressive (and *compressive*) potential of computer music, because any technique that "cheats" by depending on large pieces of recorded or rendered data (assets such as samples or even MIDI files) simply won't fit.

[10] This could be accomplished in a backwards-compatible way by checking the signature of the exported `process()` function.

[11] This idea is complicated by the base43 encoding step but may be possible nonetheless.

7 Conclusion

In this paper, we presented ScoreCard, a system for repurposing QR codes as a musical medium by creating codes storing entire WebAssembly generative music programs. We discussed the challenges and affordances of this unusual medium in the context of usability and composition, and we believe they make ScoreCard a unique and playful way to create and share generative music.

In cities, we often find stickers with QR codes that link to an artist's presence on a streaming service or social media—ushering people in to centralized musical or social repositories operated by corporations where every play is tracked and stored away as fodder for algorithms. ScoreCard turns this relationship on its head, inviting people to leave visible traces encoding musical spaces in physical places, sharing complete pieces in a decentralized manner, using the web to promote access rather than surveillance and paper for sharing rather than advertising.

8 References

[1] J. Baker. Hacking WebAssembly Games with Binary Instrumentation. https://doi.org/10.5446/48379, 2019. Series: DEF CON 27 Published: DEF CON.

[2] B. Barth and J. van Tubergen. Your Terms of Service: Interactive Audio Installation. https://zenodo.org/records/6770101, June 2022.

[3] A. Carlsson. The Forgotten Pioneers of Creative Hacking and Social Networking – Introducing the Demoscene. In *Proceedings of the Third International Conference on the Histories of Media Art, Science and Technology*, Re:live Media Art Histories, 2009.

[4] I. Clester and J. Freeman. Composing the Network with Streams. In *Audio Mostly 2021*, pages 196–199. Association for Computing Machinery, New York, NY, USA, 2021.

[5] I. Clester and J. Freeman. Alternator: A General-Purpose Generative Music Player. In *Proceedings of the International Web Audio Conference*, WAC '22, Cannes, France, June 2022.

[6] I. J. Clester. *RFID localization for interactive systems*. Thesis, Massachusetts Institute of Technology, 2020.

[7] F. Computer. Folk Computer. https://folk.computer/.

[8] F. Cottone. Rewtro. https://www.kesiev.com/rewtro/, 2019.

[9] R. Cox. QArt Codes. https://research.swtch.com/qart, Apr. 2012.

[10] G. de Seta. QR code: The global making of an infrastructural gateway. *Global Media and China*, 8(3):362–380, Sept. 2023. Publisher: SAGE Publications Ltd.

[11] E. FANTASTICOS! Barcoder. https://www.electronicosfantasticos.com/,https://www.electronicosfantasticos.com/en/works/barcoder/, Mar. 2021.

[12] B. Garcia. WASM-4. https://wasm4.org/, 2021.

[13] D. M. Group. Dynamicland. https://dynamicland.org/.

[14] V.-M. Heikkilä. Discovering novel computer music techniques by exploring the space of short computer programs. http://arxiv.org/abs/1112.1368, Dec. 2011.

arXiv:1112.1368 [cs].

[15] A. Langley. Efficient QR codes. https://www.imperialviolet.org/2021/08/26/qrencoding.html, Aug. 2021.

[16] F. M. Martins and J. H. Padovani. Be Brief: Convergences and Possibilities of Live-Coding and sctweeting. In *Proceedings of the 7th International Conference on Live Coding (ICLC2023)*, Utrecht, Netherlands, Apr. 2023.

[17] A. McLean and G. Wiggins. Tidal–Pattern Language for the Live Coding of Music. In *Proceedings of the Sound and Music Computing Conference*, SMC '10, pages 331–334, Barcelona, Spain, July 2010.

[18] P. Salomonsen. WebAssembly Music. https://zenodo.org/records/6772287, June 2022.

[19] S. Tatham. Coroutines in C. https://www.chiark.greenend.org.uk/~sgtatham/coroutines.html, 2000.

[20] G. Wang. A History of Programming and Music. In J. d'Escrivan and N. Collins, editors, *The Cambridge Companion to Electronic Music*, Cambridge Companions to Music, pages 58–85. Cambridge University Press, Cambridge, 2 edition, 2017.

[21] A. Zakai. Emscripten: an LLVM-to-JavaScript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, OOPSLA '11, pages 301–312, New York, NY, USA, Oct. 2011. Association for Computing Machinery.

APPENDIX

A Extended Example: *In C* in C

```c
#include "deck.h"
#include "in.h" // Contains `notes` (array of note data) and `fragments` (lengths of sections in `notes`).
card_title("in.c");
setup_rand;

float play_pulse() {
    const int freq = m2f(84);
    const float dur = 0.25f;
    static float sqr_phase, t = 0;
    gen_begin;
    for (;;) {
        for (; t < dur; t += dt) {
            yield(env(t, dur) * sqr(&sqr_phase, freq));
        }
        t -= dur;
    }
    gen_end(0);
}

float play_score() {
    const float grace_note_frac = 0.05f;
    const float amplitudes[] = {0.44f, 0.66f, 1.0f};
    const osc_func osc_funcs[] = {sqr, saw, tri};
    static osc_func osc;
    static int num_reps, rep, fragment_index, fragment_start, fragment_end, note_index;
    static float osc_phase, freq, dur, amp, t = 0;
    gen_begin;
    osc = choice(osc_funcs);
    for (fragment_index = 0; fragment_index < SIZEOF(fragments); fragment_index++) {
        fragment_start = note_index;
        fragment_end = note_index + fragments[fragment_index];
        num_reps = rand() % 4 + 3; // Determine how many times to repeat this fragment before moving on.
        for (rep = 0; rep < num_reps; rep++) {
            for (note_index = fragment_start; note_index < fragment_end; note_index++) {
                dur = notes[note_index].dur / 4.0f;
                if (notes[note_index].pitch == 0) { // Rest: skip this note.
                    sleep(t, dur);
                    continue;
                }
                freq = m2f(notes[note_index].pitch);
                if (dur == 0) { // Grace note
                    dur = (notes[note_index + 1].dur / 4.0f) * grace_note_frac;
                } else if (note_index > 0 && notes[note_index - 1].dur == 0) { // Last note was a grace note
                    dur *= (1 - grace_note_frac);
                }
                amp = amplitudes[notes[note_index].velocity - 1];
                for (; t < dur; t += dt) { // Synthesize the note!
                    yield(env(t, dur) * osc(&osc_phase, freq) * amp);
                }
                t -= dur;
            }
        }
    }
    gen_end(0);
}

float process() {
    return (play_pulse() + play_score()*3)/4;
}
```