A multitrack reactive music player for the browser

Daniel Martín
dmartinmartinez@gmail.com

ABSTRACT

Accompanying systems provide musical accompaniment to a human performer. These systems can adapt to music contextual changes like the tempo or the tonality, which makes them an interesting tool for music students. In this paper I describe a browser-based multitrack reactive music player. This player enables bringing up automatic accompanying systems to the browser. The player is implemented in TypeScript, using the web audio API and MIDI.js soundfonts.

1. INTRODUCTION

The goal behind the current work is the idea of a browser-based application equivalent to accompaniment music systems like *iReal Pro*[1] or *Band in a box*[2] which are native and desktop apps, respectively.

Accompanying systems are designed to play music that serves as background music for a human performer. Unlike simple audio backing tracks, they can adapt to contextual changes. Accompanying systems usually consist of several tracks, one per instrument, and they generate the music for each track in real-time, based on a fixed music form with an harmonic structure. The actual notes played on each track depend on a *music generation algorithm* that will take into account, at each interval of time, the current chord of the harmonic structure, and other contextual parameters like the tempo or the music style. Additionally, the algorithm can add some randomness to certain aspects of the musical output (e.g. the rhythm, or the pitch of some notes) so that the output is less repetitive, hence, less boring. Some accompanying systems go one step further: they provide interaction by analyzing the human's performance and use it as an input into the music generation algorithm [6][8][9][11].

Since the introduction of Web Audio API, many efforts have been focused on porting desktop applications to the browser. There are some examples of Digital Audio Workstations (DAWs) [5] or live coding systems [10]. However, little attention has been paid to accompaniment systems.

In the next section, I describe a multitrack reactive player for browser based accompanying systems.

Web Audio Conference WAC-2021, July 5–7, 2021, Barcelona, Spain.

[1] https://irealpro.com/
[2] https://www.pgmusic.com/

2. SYSTEM'S DESIGN

The multitrack player consists basically of two schedulers (named *music scheduler* and *time scheduler*) and several tracks. Each track is a *music buffer,* a buffer of notes associated with an instrument.

The *music scheduler* generates and periodically updates the notes for each track. These notes are events for the *time scheduler* to play.

A *music buffer* represents a short musical phrase, usually one bar (or measure) long. It is a sequence of note events where each note event has an onset (*start*) and a *duration* both specified in beats, and a set of pitches. See the type definitions in TypeScript:

```
interface TimeEvent {
 start: beats;
 duration: beats;
}
interface MusicEvent extends TimeEvent {
 notes: Pitch[]; //e.g. ['A4','C5','E5']
}
interface MusicBuffer {
 instrument: 'guitar' | 'bass' | 'drums';
 duration: beats;
 events: MusicEvent[];
}
```

2.1 Instruments

Instruments are loaded as soundfonts. Soundfonts consist of an audio sample for each note. The soundfonts are loaded in JSON format. This JSON file contains all base64 encoded notes.

After fetching the JSON file from a server, each base64 note is converted to an *ArrayBuffer* and saved as an *AudioBuffer*. The whole structure is saved to a Javascript object where the key is the note name (e.g. *A1*, which stands for pitch *A* in octave *1*) and the value corresponds to the *AudioBuffer*.

2.2 Music scheduler

For a specific implementation of the player, the *music scheduler* is told to run one or more functions. Each function is supposed to be run recurrently at a given period of time. The functions generate music notes for one or several instrument tracks.

In a more detailed level, the *music scheduler* has a default *resolution* of 0.25 beats; i.e. it runs every 0.25 beats, checking if there is any function to be run given the periodicity associated to that function. For example, for a given implementation we could define a function that generates a guitar pattern to be run at every

beat. The *music scheduler* would check every 0.25 beats, and would run the function the 4th time (beat 1), the 8th time (beat 2) and so on.

The smaller the periodicity associated with a function, the more reactive the player will be, but it will also be more expensive for the browser. The ability to run several functions at different periods of time results in a flexible and optimized scheduler. We could, for instance, tell the *music scheduler* to update the bass line at every 4 beats while running a function for the drums at every beat that will decide if a drum roll should be played or not.

The functions sent to the *music scheduler* are part of a *music generation algorithm*. This algorithm is specific to an implementation of the player. Usually, it generates the notes for the *music buffers* depending on the harmonic context as well as other contextual parameters such as the tempo or the music style. Also, it can also use randomness to produce a non-repetitive musical output. An example is described in *Section 3*.

Furthermore, the algorithm can take some parameters generated in real-time as input, thus, producing a reactive output. To remark,this design paves the way for an implementation of an interactive player that takes into account the human's performance.

2.3 Time scheduler

The time scheduler is in charge of playing the music events that are in queue in a precise time. Each track has its own music events generated by the *music scheduler*. When the user pauses and plays again, the system needs to resume correctly. This problem is very similar to those that DAWs face [7].

The time scheduler follows a scheduling technique inspired by [13] for playing notes using the web audio *AudioContext* clock. *AudioContext* clock runs in its own thread and it is independent from the browser's event loop[3], making it very precise. On the other hand, other Javascript timing functions like *setTimeout* or *setInterval* do depend on the event loop and can be affected by other events on the browser; hence, they have less precise timings.

The scheduler periodically checks at a given period *T*, for each track, what are the next music events to be played within a window of time ahead *w*. This periodic operation is triggered by *setInterval*. Each track has its own events.

For a music piece with 2 tracks, a period *T=50ms* with a window *w=60ms* has been proven to work well. Even though *T* is not very precise, the error is mitigated thanks to the window *w.* Another potential problem is that scheduled events cannot be cancelled. But in this case, a value of window time *w=60ms* represents an optimal one: if the user pauses and still listens to the music events it would be only for *60ms*. That is not long enough to get annoyed.

The Time Scheduler has to deal with mappings between beats and time [3]. There are 2 timers: the *currentTime* is the time played for the current song in seconds, whereas the *currentBeatTime* is the time played for the current song in beats, i.e. it is relative to the tempo.

To summarize, at each iteration the scheduler does the following:

1. From *currentBeatTime* it gets the next events to be played from within the time window (window value is transformed into beats)
2. For each event *e* found, it calculates the difference in beats, *diff_e* between the onset of that event and *currentBeatTime*.
3. Transforms beats to time (seconds) each of the found differences.
4. For each difference it schedules the event *e* to be played at *currentTime + diff_e*

Fig. 1 is an example where the tempo is 120bpm. It shows how notes are scheduled with a precise onset (3 beats in *beats* and 1.5 seconds in *time)* when they start within a window size from a certain moment, even though this moment has not a precise value (1.498 seconds).

Finally, due to the fact that *AudioContext* clock never stops, there is a third layer that maps the scheduler *currentTime* to *AudioContext*'s *currentTime.* Actually, events are scheduled at *AudioContext currentTime*.

The method described ensures time precision and robustness over tempo changes.

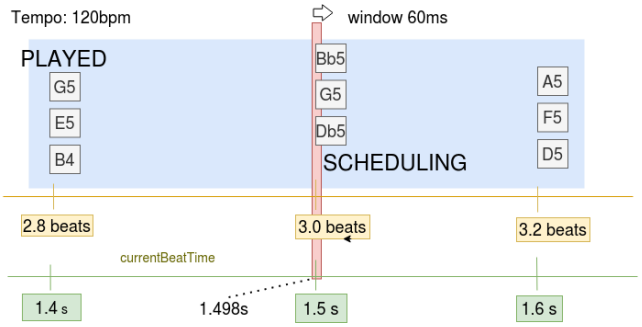

Figure 1. Time scheduler at instant 1.498s scheduling notes

The *time scheduler* performance has been tested in a *Firefox* browser in two different machines: a *Debian (1.8GHz Intel Core i3-3217U, 12 GB)* and in a *MacBook Pro (2.2GHz 6-Core Intel Core i7, 16 GB)*.

To run the tests, we have created 4 guitar music patterns, each of them consisting of a sequence of 64th notes (or *hemidemisemiquaver* notes). At a tempo of 60bpm, that means playing 16 notes per second for each track. Having 4 tracks, 64 notes will be played every second. If we increase the tempo to a very fast one like 300bpm (5 times faster), those 64 notes have to be played in 200ms. Using the period *T=50ms* and the window *w=60ms*, the *time scheduler* should process about 16 notes each time. Our tests show that in the worst cases only 20ms are needed to schedule 16 notes, so it is enough with our current settings.

The test song we have used is more typical of extreme styles like *black MIDI* rather than the music styles used in accompaniment systems such as jazz or pop. In conclusion, we can say that our *time scheduler* performance is suitable for accompaniment systems.

2.4 Audio loop music events

I have described the system to be used with soundfonts. However, there are cases where using an audio loop can result in a more realistic musical output. For instance, for a jazz brushes drums

[3]https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

track, an audio loop pattern would sound better. Audio loops have to be handled differently from music events based on soundfonts.

With soundfonts, duration needs to be handled but is not important to finish precisely, because the onset of the following note does not depend on it. Therefore, duration is not something to be concerned about. For loops, instead, duration has to be exactly equal to the pattern's duration. Consequently, if there are tempo changes, the loop duration has to be properly modified. Adapting the loop and its duration can be easily implemented thanks to the web audio API by changing the *AudioBufferSourceNode*'s *playbackRate*, but better results would be achieved by using time stretching techniques.

If one track has several loop patterns, each loop pattern will be associated with its own *AudioBufferSourceNode*. If the music scheduler decides to change the loop pattern used for the following iteration, the previous one needs to be stopped, and the new one needs to be started at the given offset time determined by *currentTime*. The same logic is applied when the music player is paused and resumed.

3. INTEGRATION INTO AN ACCOMPANIMENT SYSTEM

An automatic accompanying system can make use of this player system to play, for example, the chord progression of a popular song. Given the chord symbols, the accompanying system would generate the notes for the patterns consumed by the player system. An approach for that is explained in [1].

Here, we describe an accompaniment system prototype that plays *ii-V-I* progressions in all tonalities. At each time it transposes the whole chord progression one fourth, so it will play:

Dm7 | G7 | Cmaj7 | Cmaj7

Gm7 | C7 | Fmaj7 | Fmaj7

...etc.

The time signature is 4/4 so each measure is 4 beats long. The whole progression is 4 measures long; that is, 16 beats long.

The system will consist of two tracks: guitar and bass. The guitar track will play the chords in a given inversion. There is a predetermined inversion for each chord type: one for *iim7*, one for *V7* and one for *Imaj7*. Also, there are two predefined guitar rhythmic patterns. At each measure the algorithm will randomly choose between one or the other.

Fig. 2 shows how the algorithm gets as input the harmonic context, a pattern rhythm chosen randomly and a predetermined inversion type [3,5] (in this case, the third grade, F, and fifth grade, A, are transposed by one octave) and generates the music notes for one measure of the guitar track.

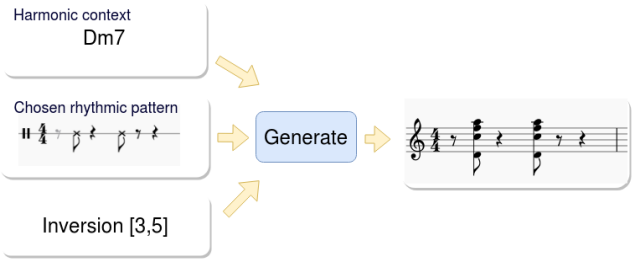

Figure 2. Guitar notes generation algorithm for one measure.

For the bass track, we will use a simple bass walking generation algorithm that, given the harmonic context of one measure, will generate basically quarter notes. Finally, we will change the whole harmonic context at the end of each progression by transposing the chord progression by a fourth.

In summary, we are defining three functions for the *music scheduler*. Two of them will run every measure (4 beats), one for the guitar and one for the bass, and one function to transpose the whole progression that will run every 4 measures (16 beats).

In order to make the system reactive, it could communicate with an external MIDI instrument through *MIDIAccess* from the *Web audio API*. Alternatively, it could make use of recent web audio processing tools like *Essentia.js* [2]. In both cases, it would extract parameters that would be sent as input to the music generation algorithm of the player.

4. CONCLUSIONS AND FUTURE WORK

In this paper, I have described the details of the system's implementation of a browser-based multitrack music player and how it can be integrated into an accompaniment system. The system is flexible and could eventually use more sophisticated AI algorithms to generate the music notes for each track.

There are some improvements to be done in addition to audio loops described in *Section 2.4*. First, even though the system has proved to work well with Soundfonts and a few tracks, there is a chance that the use of *setInterval* can cause problems. The solution presented by [12] suggests offline rendering. However, it does not work for our case as the *music generation algorithm* decides what to play on the fly. A better approach to explore is the use of *web workers* as in [5], this way, the *music scheduler* could run in another thread.

Second, in order to provide the sound assets (soundfont, audio loops) several fetching strategies can be approached. To simplify, we can choose between pre-fetching assets, fetching on-demand or a combination of both. The optimal strategy depends on variables like the quantity of data to load for a given session, how likely it is that new data will be needed, or whether or not we want to support offline usage.

Finally, still regarding fetching optimization, some techniques can be applied to minimize the fetching payload. For instance, we could fetch just certain notes of each instrument and generate the rest by using pitch shifting. Thus, as usual, there is a trade-off between quality and optimization.

5. REFERENCES

[1] Bernardes, G., Cocharro, D., Guedes, C., & Davies, M. E. (2016). Harmony generation driven by a perceptually motivated tonal interval space. *Computers in Entertainment (CIE)*, *14*(2), 1-21.

[2] Correya, A. A., Bogdanov, D., Joglar-Ongay, L., & Serra, X. (2020). Essentia. js: a JavaScript library for music and audio analysis on the web.

[3] Dias, B., Pinto, H. S., & Matos, D. M. (2016). BPMtimeline: Javascript tempo functions and time mappings using an analytical solution.

[4] Gwartz, J., & Gold, N. Loop-Based Graphical Live-Coded Music in the Browser.

[5] Jillings, N., & Stables, R. (2017). An Intelligent audio workstation in the browser.

[6] Lewis, G. E. (2000). Too many notes: Computers, complexity and culture in voyager. Leonardo Music Journal, 33-39.

[7] Mahadevan, A., Freeman, J., Magerko, B., & Martinez, J. C. (2015). EarSketch: Teaching computational music remixing in an online Web Audio based learning environment. In *Web Audio Conference*.

[8] Moreira, J., Roy, P., & Pachet, F. (2013). Virtualband: Interacting with Stylistically Consistent Agents. In *ISMIR* (pp. 341-346).

[9] Nika, J., & Chemillier, M. (2012, September). Improtek: integrating harmonic controls into improvisation in the filiation of OMax.

[10] Roberts, C., & Pachon-Puentes, M. (2019). Bringing the tidalcycles mini-notation to the browser. In *Proceedings of the Web Audio Conference*.

[11] Rowe, R. (1992). Machine listening and composing with cypher. *Computer Music Journal*, *16*(1), 43-63.

[12] Sullivan, J. A tale of no clocks. *http://joesul.li/van/tale-of-no-clocks/*

[13] Wilson, C. (2013). A tale of two clocks-scheduling web audio with precision. *January*, *9*, 2013.