Of Time Engines and Masters

An API for Scheduling and Synchronizing the Generation and Playback of Event Sequences and Media Streams for the Web Audio API

Norbert Schnell
Victor Saiz
Karim Barkati
Samuel Goldszmidt
IRCAM – Centre Pompidou, STMS lab IRCAM-CNRS-UPMC, Paris France
{norbert.schnell, victor.saiz, karim.barkati, samuel.goldszmidt}@ircam.fr

ABSTRACT

In this article we present an API and a set of Javascript modules for the synchronized scheduling and aligned playback of predetermined sequences of events such as notes, audio segments, and parameter changes as well as media streams (e.g. audio buffers) based on the Web Audio API logical time. The API has been designed to facilitate the development on both ends, the implementation of modules which generate event sequences or media streams as well as the integration of such modules into complex audio applications that require flexible scheduling, playback and synchronization.

Categories and Subject Descriptors

H.5.3 [**Group and Organization Interfaces**]: Web-based interaction; H.5.5 [**Information Interfaces and Presentation**]: Sound and Music Computing – Systems

Keywords

HTML 5; Web Audio API; Audio Processing; Scheduling; Synchronization

1. INTRODUCTION

Flexible and precise scheduling and sequencing of audio events and synthesis parameter changes is an important feature of any real-time audio synthesis authoring environment [4, 9, 2]. The Web Audio API [1] provides a logical time synchronized to the audio input/output system (i.e. `currentTime` of an `AudioContext`) and the possibility to schedule events and parameter changes with perfect accuracy (see also [8]). Nonetheless, real-time web audio processing applications that generate events which are required to impact audio synthesis with precise timing (e.g. to generate rhythmic or phase-synchronous signals) generally have to cope with the short-term synchronization between Javascript timers (i.e. `setTimeout`) and the Web Audio API logical time [10]. Multiple libraries propose APIs that formalize flexible scheduling and sequencing of audio events for musical applications based on the Web Audio API [3, 5].

The API that we present in this article, contributes a unified and interoperable formalization for the synchronized scheduling and aligned playback of predetermined sequences of events such as notes, audio segments, and parameter changes as well as media streams (e.g. audio buffers) based on the Web Audio API logical time. The API has been designed to facilitate the development on both ends, the implementation of modules, *"time engines"*, which generate events algorithmically or interpret prerecorded sequences or media streams as well as the integration of such modules into complex audio applications that require flexible scheduling, playback and synchronization.

2. TIME, POSITION, AND SPEED

As a very first step to properly formalize the different temporalities of a system-wide reference time and the playback time of predetermined event sequences or media streams, we propose to refer to the former as *time* and to the latter as *position*. This allows for intuitively defining the ratio between them as *speed*. While *time* is steadily increasing, the evolution of *position* can be interrupted (e.g. by stopping or pausing the playback), discontinuous (e.g. by rewinding or seeking) and reversed (e.g. by playing backwards). Nevertheless, *times* and *positions* are given in seconds.[1] Since they always refer to the logical time of the current Web Audio API context – directly or scaled through *speed* –, *time* and *position* can be regarded as perfectly precise when used in conjunction with Web Audio API calls such as the `OscillatorNode`'s or the `AudioBufferSourceNode`'s `start` method and the audio parameter automation methods (e.g. `setValueAtTime`). They can apply to events on any temporal scale including, for example, elementary waveforms synthesized through granular synthesis, the beats of a drum pattern or the macrostructure of a sound installation over several days, weeks or months.

[1]The *time-position-speed* nomenclatura we propose as a clear and intuitive alternative to denominations like *current time*, *playback time* and *playback rate* at least have the advantage of shortening the function and attribute names of the proposed API.

3. TIME ENGINES AND MASTERS

In the API we propose, a component that generates sequences of events or media streams extends the `TimeEngine` class. We have identified three cases that have been formalized as three different interfaces provided by the `TimeEngine`:

scheduled – the module generates events that are precisely scheduled in *time* (e.g. the grains generated by a granular synthesis engine)

transported – the module generates events that are precisely associated to particular *positions* (e.g. the percussive elements of a recorded beat pattern annotated by onset positions)

speed-controlled – the module provides its own synchronization to the current Web Audio context time as well as a way to control its playback rate (e.g. a Web Audio API `AudioBufferSourceNode`)

A `TimeEngine` can be added to a *master* module that uses one of these interfaces to drive the engine in synchronization with other engines. In apparent symmetry to the three interfaces provided by the `TimeEngine` class, we implemented three masters, the `scheduler`, the `Transport` class, and the `PlayControl` class. When adding an engine to a master, the master provides it with getters for the attributes `currentTime` and `currentPosition`. The engines can use these getters in their implementation to synchronize their processing to the master.[2]

The different interfaces provided by the engines and the getters provided by the masters allow for implementing engines that behave differently when added to different masters and for implementing masters that drive the same engines in different ways, while relying on a clear and simple API on both sides.

3.1 Scheduler and Scheduled

The `scheduler` object is provided as a global singleton that allows for driving engines which implement the *scheduled* interface.[3] *Scheduled* engines generate a sequence of events by implementing the method `advanceTime` which is called for the first time when the engine is added to the scheduler – or with an optional delay. The method `advanceTime` is called with the corresponding Web Audio context time of the event to be generated and returns the time of the next event (see figure 1). When returning `Infinity`, the engine's `advanceTime` method is not called again until it is rescheduled. The scheduler provides the scheduled engines with a function `resetNextTime` that allows for changing their next scheduling time.

We have implemented two versions of the global scheduler singleton. The more complex version maintains a priority queue of the currently scheduled engines. This variant allows for implementing complex systems where the precise order of events is crucial. A simplified version (i.e the *simple-scheduler*) is based on the principle laid out in [10] where the next scheduling time of all scheduled engines is tested in a regularly – and recursively – called `setTimeout` callback.

[2]In the `TimeEngine` base class, the getter of `currentTime` returns the current Web Audio context time and that of `currentPosition` returns 0.

[3]In addition, the scheduler allows for scheduling simple – one-shot – callbacks with arbitrary delay times.

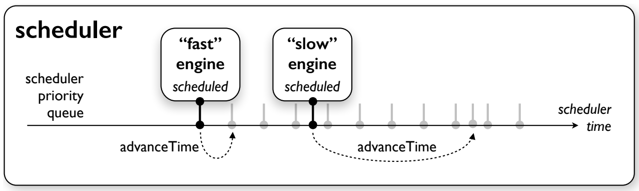

Figure 1: Two engines generating events at different pace. The scheduler successively calls the engines' `advanceTime` method just before their scheduled time and reschedules each engine at the returned time using a priority queue.

Both variants of the scheduler provide the same attributes to parameterize the scheduling behavior:

period – period (simple scheduler) or minimum period (scheduler) of the recursive `setTimeout` calls

lookahead – lookahead time by which all calls to the engines `advanceTime` method are anticipated in respect to the current Web Audio context time

3.2 Transport and Transported

The `Transport` class implements the core of a generic player that controls multiple engines. While the engines take over the generation or interpretation of event sequences and media streams, the transport master provides their synchronization and alignment to a common playing position that evolves as a function of the transport's external control. The user can determine how the engine's event or media positions are aligned to the transport position through a set of parameters including start, offset and end positions. While the *transported* interface is the preferred interface of the transport, it also accepts engines that implement the other two interfaces.[4]

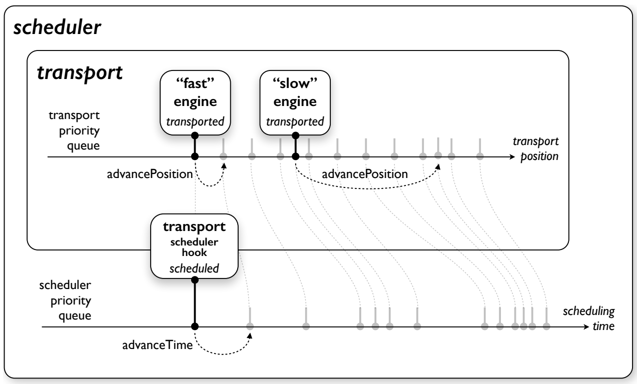

Figure 2: The same engines as in figure 1 controlled by a transport. The transport's *scheduled event* translates the engines' positions in scheduling times depending on the transport's playing controls.

[4]When adding an engine to the transport, the transport checks first whether the engine implements the *transported* interface followed by the *speed-controlled* and the *scheduled* ones.

With regard to *transported* engines, the transport can be seen as a translator that translates the engines' positions into scheduling times. The translation process is illustrated in the figures 2 and 3. Similar to the *scheduled* interface, the *transported* interface essentially consists of a method `advancePosition` that generates an event and returns the position of the next event regarding the current playing direction (i.e. whether the speed is positive or negative). To translate the event positions into scheduling times the transport adds an internal *scheduled* engine – its *scheduled cell* – to the global scheduler.[5] This *cell* is always scheduled – and rescheduled – at the time that corresponds to the position of the next transported engine (see figure 2). Similarly to the scheduler, a transport keeps a priority queue of the next positions of its *transported* engines. When playing in one direction, these positions do not depend on the transport speed, so that only the first position of the priority queue has to be rescheduled when the transport's speed changes. This allows for efficient scheduling of any number of *transported* engines.

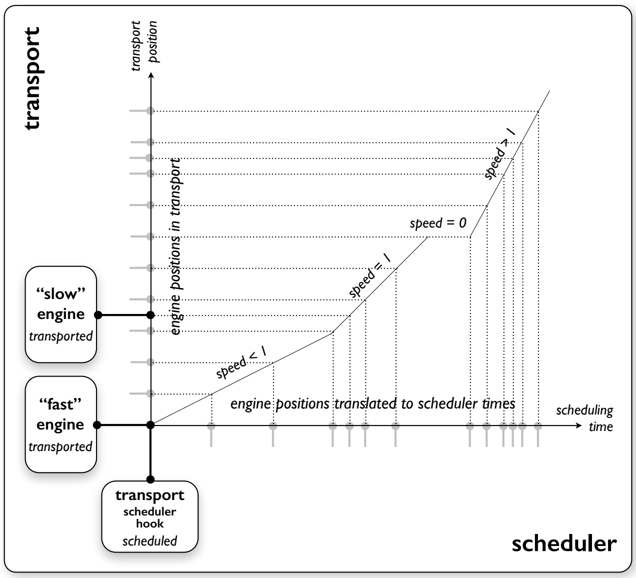
transport
transport position
engine positions in transport
speed > 1
speed = 0
speed = 1
speed < 1
"slow" engine transported
"fast" engine transported
engine positions translated to scheduler times
scheduling time
transport scheduler hook scheduled
scheduler

Figure 3: The transport translates the positions of the transported engines into scheduling times depending on the transport's variable speed. In the illustrated example corresponding to figure 2, the transport position evolves continuously (no *seek*) and without reversing the playing direction (*speed* ≥ 0).

However, the transport's priority queue is reconstructed each time the transport starts playing, seeks to a new position, or reverses its playing direction. Therefore, each *transported* engine provides a method `resetPosition` that returns its next position regarding the master's current position and playing direction.[6]

[5]The `Transport` class requires the version of the scheduler based on a priority queue in order to properly synchronize the transported engines and their control.

[6]The transport provides the engines under its control with a function `resetNextPosition` that allows for changing their next transport position.

When adding an engine that only implements the *scheduled* interface, the transport delegates the scheduling of the engine to the global scheduler and generates an adapter. The adapter consists of a minimal *transported* engine adds the engine to the scheduler at its start position and removes it when the transport stops or reaches the engine's end position. To align its processing to the transport's position, the engine can obtain a precise position corresponding to the current time through the `currentPosition` attribute getter provided by the transport (see figure 4).

```
scheduler.add(engine, Infinity, () => {
  // get currentPosition
  return (this.transport.currentPosition
     - this.offsetPosition) *
    this.scalePosition;
});
```

Figure 4: The `currentPosition` attribute getter provided by the adapter for *scheduled* engines added to a transport. The getter is given as third argument – an ECMAScript 6 arrow function –, when adding the engine to the global scheduler. The `Infinity` given as scheduling delay (second argument) makes that the engine is not called until it is actually started – and rescheduled – by the transport.

The third kind of engines that can be added to the transport are those implementing the *speed-controlled* interface. Similar to the case of *scheduled* engines, the transport automatically generates an adapter that controls the engine through the provided interface. The *speed-controlled* interface is described in the next section. An example that illustrates the functioning of a *speed-controlled* engine added to a transport is given below as the `PlayerEngine` (see 4.4).

The `Transport` class itself extends the `TimeEngine` class and implements two of its interfaces. Through its *transported* interface, a transport can be added to another transport, which allows for constructing hierarchical structures of transports being slaved to transports and, ultimately, to compose sequences of sequences. The *speed-controlled* interface allows for keeping the transport class to its essential functionalities. Play control methods and attributes can be added to a transport by associating it to a `PlayControl` master.

3.3 PlayControl and Speed-Controlled

The `PlayControl` class allows for controlling a single engine that is given as an argument of its constructor. Like transports, instances of the `PlayControl` class accept engines of all three `TimeEngine` interfaces. The basic idea of the `PlayControl` is to transform any engine into a player that provides common media player control methods and attributes such as `start`, `stop`, `seek`, and `speed`.

The preferred interface of `PlayControl` masters is the *speed-controlled* interface – followed by the *transported* interface. This interface aims at controlling engines that, other than *scheduled* and *transported* engines, ensure their synchronization to the current Web Audio context time independently. This is for example the case for transports, which are internally driven by the global scheduler, as well as for audio players based on the Web Audio API `AudioBufferSourceNode`.

The interface consists of a single method `syncSpeed` that is called with the master's current (Web Audio context) time, position, and speed as well as with a flag `seek`, that indicates whether the engine has to jump to a new position. From the calls to the `syncSpeed` method, a *speed-controlled* engine can easily derive whether it should (re-)start, halt, reverse its playing direction, or jump to a new position (see 4.4 below).

The way the `PlayControl` class handles *scheduled* and *transported* engines is very similar to the `Transport`. However, since instances of `PlayControl` only control a single engine with a fixed alignment – at position 0 –, the implementation remains considerably simpler.

4. IMPLEMENTED ENGINES

We have developed a set of engines that implement different interfaces provided by the `TimeEngine` class. Here, they are included to further illustrate both the functioning of the `TimeEngine` API and its masters as well as the motivations for its development.

4.1 A Metronome

The `Metronome` class extends `TimeEngine` and implements both the *scheduled* and the *transported* interface. In both cases, the engine periodically generates click sounds using the Web Audio API. A set of attributes allows for setting the period and adjusting the sound of the click. As a *scheduled* engine, the metronome produces clicks of the given period as soon as it is added to the scheduler. When added to a transport, the metronome clicks are aligned to the transport position and occur when the transport reaches their positions depending on its external control. Figure 5 shows the implementation of the interface methods of the metronome.

```
// TimeEngine scheduled interface
advanceTime(time) {
  this.click(time); // generate sound
  return time + this.period;
}

// TimeEngine transported interface
syncPosition(time, position, speed) {
  var next = (Math.floor(position /
      this.period) + this.phase) *
      this.period;

  if (speed > 0 && next < position)
    next += this.period;
  else if (speed < 0 && next > position)
    next -= this.period;

  return next;
}

// TimeEngine transported interface
advancePosition(time, position, speed) {
  this.click(time); // generate sound

  if (speed < 0)
    return position - this.period;

  return position + this.period;
}
```

Figure 5: Implementation of the *scheduled* and *transported* interface methods of the `Metronome` class.

4.2 The GranularEngine

The `GranularEngine` is an example of a `TimeEngine` that only implements the *scheduled* interface but nevertheless can be aligned to the position of a transport. The engine performs granular synthesis on a given `AudioBuffer`. A set of attributes determines the engine's synthesis parameters such as the grain period, duration, and resampling (i.e. pitch transposition) as well as the *position* of the grains in a given `AudioBuffer`. When added to the `scheduler`, an engine generates grains with the given period at the given the position – that usually is randomly varied for each grain within given boundaries.

```
get currentPosition() {
  return this.position;
}
```

Figure 6: The default `currentPosition` attribute getter of the `GranularEngine` class.

Internally, the engine uses the `currentPosition` attribute to generate a grain, that by default is defined as the value of the position attribute (see figure 6). When added to a transport, the getter of `currentPosition` is redefined by the transport to return the current transport position transformed through the alignment parameters defined for the engine (see figure 4 in 3.2).

This way, the engine becomes a granular player that is automatically aligned to other engines controlled by the same transport. To create a simple granular player, the user can create a `GranularEngine` instance with `PlayerControl` instance (see figure 7).

```
var engine = new GranularEngine();
var player = new PlayControl(engine);

engine.period = 15;
engine.duration = 120;
player.start();
```

Figure 7: Example code composing a granular player from a `GranularEngine` and a `PlayerControl`.

4.3 The SegmentEngine

The `SegmentEngine` synthesizes sound segments that are defined by an array of onset positions and durations referring to an `AudioBuffer`. The engine's implementation and parameters largely resemble those of the `GranularEngine` class described in 4.2. Like the `metronome` (see 4.1), the `SegmentEngine` implements both the *scheduled* and *transported* interfaces. In the scheduler, the engine produces sound segments selected through the `segmentsIndex` attribute at regular time intervals while allowing for controlling their period, duration, and resampling by a set of attributes. When added to a transport, the segment onset positions are aligned to the transport position, which allows for playing the segments with different tempi – also backwards – and for rearranging their order by appropriately controlling the transport speed and position.

4.4 The PlayerEngine

The `PlayerEngine` completes the set of `TimeEngine` classes that implement different interfaces to playback recorded audio aligned and synchronized through a transport by a simple audio player. The engine implements the *speed-controlled* interface and derives the control of an `AudioBufferSourceNode` from the `syncSpeed` method calls (see figure 8).

```
// TimeEngine speed-controlled interface
syncSpeed(time, position, speed, seek) {
  var lastSpeed = this.speed;

  if (speed !== lastSpeed || seek) {
    if (seek || lastSpeed * speed < 0) {
      this.halt(time);
      this.start(time, position, speed);
    } else if (lastSpeed === 0 || seek) {
      this.start(time, position, speed);
    } else if (speed === 0) {
      this.halt(time);
    } else if (this.bufferSource) {
      this.bufferSource.playbackRate
          .setValueAtTime(speed, time);
    }

    this.speed = speed;
  }
}
```

Figure 8: Implementation of the *speed-controlled* interface in the `PlayerEngine` class. The internal methods `start` and `halt` create, start and stop an `AudioBufferSourceNode` using the given parameters.

Associated to a `PlayControl` the `PlayerEngine` provides a simple audio player. When added to a transport, multiple instances of the class can be composed to multi-track audio player synchronized to engines of the other interfaces.

5. CONCLUSION

We have presented an API and a set of Javascript modules for the scheduling and synchronization of modules that generate or playback sequences of events or media streams using the Web Audio API. The article summarized the underlying concepts as well as the implementation of the provided `TimeEngine` base class and *masters*: the `scheduler` singleton, the `Transport` class and the `PlayControl` class. Finally, we have briefly described a set of example modules extending the `TimeEngine` class to further illustrate the presented concepts.

Since early on, the `TimeEngine` API and the presented master and audio processing modules have been used in the development of prototype applications for the *WAVE* and *CoSiMa* projects.[7]. These developments have guided the design and allowed for validating many of its aspects [7].

The developed software modules are available on GitHub, under BSD-3-Clause license, as separate repositories associated to the *Ircam-RnD* organization.[8] The repositories include documentation and working examples. The documentation of the `TimeEngine` API can be found in the `time-engine` repository [6].

[7]http://wave.ircam.fr/ and http://cosima.ircam.fr/
[8]https://github.com/Ircam-RnD

6. ACKNOWLEDGEMENTS

The developments described in this article have been conducted in the context of the *WAVE* and *CoSiMa* research projects, partially funded by the french National Research Agency (ANR, projects ANR-12-CORD-0027 and ANR-13-CORD-0010). We would like to thank our colleagues and project partners Sébastien (Robi) Robaszkiewicz, Benjamin Matuszewski, and Emmanuel Fréard for their precious contributions to this work.

7. REFERENCES

[1] Web Audio API – W3C Editor's Draft. http://webaudio.github.io/web-audio-api/.

[2] G. Essl. UrMus – An Environment for Mobile Instrument Design and Performance. In *Proceedings of the International Computer Music Conference*, ICMC '10, New York, 2010.

[3] S. Piquemal. WAAClock, a Comprehensive Event Scheduling Tool for Web Audio API. `https://github.com/sebpiq/WAAClock`, 2013.

[4] M. Puckette. Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*, 15(3):68–77, 1991.

[5] C. Roberts, G. Wakefield, and M. Wright. The Web Browser as Synthesizer and Interface. In *Proceedings of the Conference on New Interfaces for Musical Expression*, NIME '13, pages 313–318, 2013.

[6] N. Schnell. The WAVE Audio TimeEngine Base Class. `https://github.com/Ircam-RnD/time-engine`, 2014.

[7] N. Schnell, S. Robaszkiewicz, D. Schwarz, and F. Bevilacqua. Collective Sound Checks – Exploring Intertwined Sonic and Social Affordances of Mobile Web Applications, 2015.

[8] N. Schnell and D. Schwarz. Gabor, Multi-Representation Real-Time Analysis/Synthesis. In *COST-G6 Conference on Digital Audio Effects*, DAFx '05, pages 122–126, Madrid, Spain, Septembre 2005.

[9] G. Wang. *The Chuck Audio Programming Language. "A Strongly-timed and On-the-fly Environ/Mentality"*. PhD thesis, Princeton, NJ, USA, 2008.

[10] C. Wilson. A Tale of Two Clocks – Scheduling Web Audio with Precision. `http://www.html5rocks.com/en/tutorials/audio/scheduling/`, 2013.