Interactive Audiovisual Rendering of Recorded Audio and Related Data with the WavesJS Building Blocks

Benjamin Matuszewski
CICM/musidanse EA1572, université Paris 8,
STMS Lab IRCAM-CNRS-UPMC
Paris, France
benjamin.matuszewski@ircam.fr

Norbert Schnell, Samuel Goldszmidt
STMS Lab IRCAM-CNRS-UPMC
Paris, France
norbert.schnell, samuel.goldszmidt@ircam.fr

ABSTRACT

This article presents a set of components for the interactive audiovisual rendering of recorded audio signals and related data streams (e.g. audio descriptors and annotations) together with a set of example applications. The components are based on SVG graphics and the Web Audio API. The construction of both, the graphical user interface and the audio rendering of an application relies on a small hierarchical structure of classes that formalize different aspects of the rendering and facilitate both the implementation of complex applications using the provided components and the extension of the library by further defined graphics and audio rendering components. The library and the example applications described in the article are freely available.

CCS Concepts

•Applied computing → Sound and music computing;
•Software and its engineering → Software libraries and repositories;

Keywords

HTML 5; Web Audio API; Audio Visualization; Audio Processing; Graphical User Interface

1. INTRODUCTION

The manifold applications of interactive audio enabled by the Web Audio API [9] spread over a wide range of different domains. Many other digital technologies such as audio plugins have applications in very different domains such as music performance, composition, and audio post-production. In this context, the Web Audio API particularly contributes to the convergence of these applications with the domain of online publishing and education. This convergence is well illustrated by projects like the BBC's interactive web site on the *BBC Radiophonic Workshop* [1], the *Noteflight Worksheets* of *The Musician's Guide to Theory and Analysis* [14] or Google's *Inside Abbey Road* [5].

While the projects cited above focus on interactive audiovisual representations of sheet music and analog audio devices, many other applications use temporal representations of audio data. Available libraries are specifically designed for the visualization of audio waveforms [6, 4], audio recording, editing and composition [2, 8] or the visualization and archivation of multimodal data [7]. Other than these libraries, *WavesJS* offers low-level components and formalizations for the development of specific interactions for a very large range of applications.

In this article we present a set of flexible building blocks that support the audiovisual rendering of recorded audio signals and related data streams that are usually represented by `Array`, `ArrayBuffer` or `AudioBuffer` objects. Typical applications of the library feature the aligned display of audio segments and annotations that are associated to one or multiple tracks as well as their synchronized playback. The library facilitates the construction of complex interactions with the rendered graphics and audio. The graphical user interface components have been designed to easily create aligned superposed and/or juxtaposed representations of audio signals and related data streams such as audio descriptors, motion capture signals and descriptors as well as event or segmentation markers and annotations. The related audio components provide a formalism for synchronizing the rendering of buffered data and other data streams. Using these components, the temporal alignment of different data segments and different rendering techniques can be arbitrarily defined in advance and dynamically changed during the rendering.

Since the presentation of earlier versions at the Web Audio Conference in 2015 [10, 12], the components described in this article have been profoundly revised and integrated into a single library, *WavesJS*, that has been published on GitHub.[1] The library is written in ES6/ES2015 and implements a light hierarchy of classes free of external dependencies. It includes unit tests with code coverage reports.

After a brief overview of the library's features and implementation, the article briefly details example applications based on the described components.

2. BUILDING BLOCKS

The building blocks provided by the *WavesJS* library are separated into a graphical user interface and an audio part, *wavesjs-ui* and *wavesjs-audio*. The development of both

Web Audio Conference WAC-2016, April 4–6, 2016, Atlanta, USA.

---

[1] https://github.com/wavesjs/

sides of the library have followed independent paths forged by inherent constraints of their respective environments (i.e. *Scalable Vector Graphics* rendering and *Web Audio API*). While the library does not impose any predefined relationship between graphics and audio rendering, the APIs have been designed to work together and follow similar modular approaches.

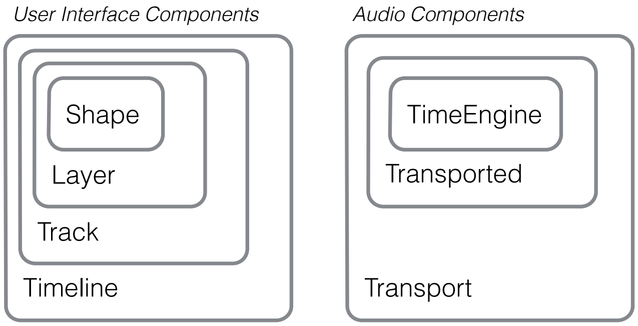

Figure 1: Overview of the hierarchical structures of user interface and audio components.

The developer using the library has to create a structure of hierarchical elements on both sides (see figure 1). At the root of this hierarchy, each side of the library provides a container that hosts the rendering of temporal data aligned to a common reference time (i.e. the `Timeline` on the graphics side and the `Transport` on the audio side).

The library allows for choosing among different graphical appearances and sound synthesis techniques for the same stream segment. While on the graphics side, the `Shape` class defines the appearance of the data on screen, a particular audio synthesizer is implemented as a `TimeEngine`. Both of these classes provide an interface that formalizes the rendering of atomic elements of a given stream in respect to their graphical alignment on screen and, respectively, their synchronisation to a common playback time and the timing of the audio subsystem. Based on this formalization, developers can integrate their own primitives into the framework.

On both sides of the library, the alignment of rendered stream segments to the reference time (i.e. `Timeline` and `Transport`) is determined by the attributes `start`, `offset`, `duration`, and `stretch` (see figure 2). Multiple segments associated to the same `Timeline` or `Transport` can arbitrarily overlap.

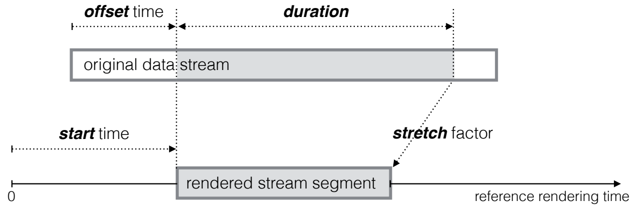

Figure 2: Parametrized alignment of a stream segment to the reference rendering time.

While the graphical components provide high-level abstractions for implementing graphical interactions with the constructed interfaces (i.e. the `EventSource` class), the audio components provide the necessary methods for the nonlinear rendering of recorded audio and related temporal data. The audio rendering of aligned data streams can be played back forwards or backwards at any speed and arbitrarily jump and loop. To support the implementation of complex interactions, the graphical and audio components allow for dynamically changing the data content, alignment, and boundaries of the rendered stream segments.

```javascript
import loaders from 'waves-loaders';
import ui from 'waves-ui';
import audio from 'waves-audio';
const loader = new loaders.AudioBufferLoader();

loader.load('assets/drum-loop.wav').then((audioBuffer) => {
  const $track = document.querySelector('#track');
  const visibleWidth = 1000, height  = 120;
  const pixelsPerSecond = visibleWidth / audioBuffer.duration;

  // setup ui rendering components
  const timeline = new ui.core.Timeline(pixelsPerSecond,
        visibleWidth);
  const axis     = new ui.helpers.TimeAxisLayer({ height: 12 });
  const waveform = new ui.helpers.WaveformLayer(audioBuffer, {
        height });
  const cursor   = new ui.helpers.CursorLayer({ height });

  timeline.createTrack($track, height, 'main-track');
  timeline.addLayer(axis, 'main-track', 'axis', true);
  timeline.addLayer(waveform, 'main-track');
  timeline.addLayer(cursor, 'main-track');

  // setup audio rendering components
  const playerEngine = new audio.PlayerEngine();
  playerEngine.buffer = audioBuffer;
  playerEngine.cyclic = true;
  playerEngine.connect(audio.audioContext.destination);

  const playControl = new audio.PlayControl(playerEngine);
  playControl.setLoopBoundaries(0, audioBuffer.duration);
  playControl.loop = true;
  playControl.start();

  // setup mouse interaction
  timeline.on('event', (e) => {
    if (e.type !== 'mousedown' && e.type !== 'mousemove') {
          return; }
    const position = timeline.timeToPixel.invert(e.x);
    playControl.seek(position);
  });

  (function updateCursor() {
    cursor.currentPosition = playControl.currentPosition;
    cursor.update();
    requestAnimationFrame(updateCursor);
  }());
}).catch((err) => { console.error(err.message); });
```

Figure 3: Code example for the hierarchical construction of graphical and audio rendering components as well as mouse interactions. The example features the looped playback of an audio file with a running cursor that the user can move to arbitrarily control the playback.

Figure 3 shows the code for setting up the components for the audiovisual rendering of a sound file. In this example, the user can manipulate the cursor to control the playback of a sound file loaded into an audio buffer.[2]

2.1 UI Components

The UI part of the library provides primitives to display and to interact with temporal data or metadata such as audio and motion signals, descriptors, and annotations.

The `Timeline` class exposed by the API provides the context for any visualization of temporal data. As shown in figure 4, a `Timeline` usually contains a hierarchical structure of `Track`, `Layer`, and `Shape` objects to manage different aspects of the graphical interaction and rendering. Multiple `Track` objects added to the same `Timeline` are vertically juxtaposed and horizontally aligned to an abscissa which corresponds to a common reference time. Each `Track` can

---

[2] http://wave.ircam.fr/demo/playback-control

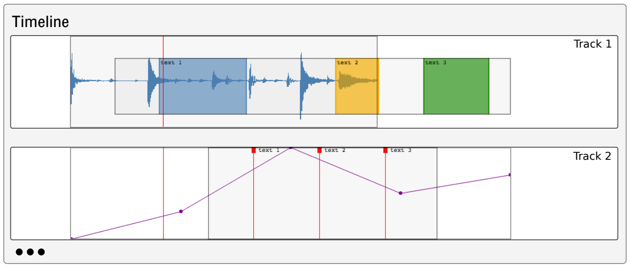

Figure 4: Overview of the hierarchical relations between the main classes of the user interface API. Provided shapes and their possible alignment among a shared `Timeline` through the `Track`.

contain one or multiple `Layer` objects each spanning over a specified time range (i.e. start time and duration). The `Layer` objects of a particular `Track` can arbitrarity overlap. Ultimately, each `Layer` contains `Shape` objects corresponding to temporal objects that may be associated to a single moment (e.g. a marker) or a time span (e.g. a segment).

To insert the graphical objects into an HTML document, `Track` objects are associated to DOM elements such as the items of a `<ul>`-list as shown in the code example 5. The rendering of `Track`, `Layer`, and `Shape` objects is based on the SVG standard which allows for the use of native features such as user interactions (e.g. mousedown, mousemove) and CSS styling.

```
<ul>
  <li id="track-1"></li>
  <li id="track-2"></li>
</ul>

<script>
  const $track1 = document.querySelector('#track-1');
  const $track2 = document.querySelector('#track-2');

  const timeline = new Timeline(pixelsPerSecond, visibleWidth);
  const track1 = new Track($track1, height);
  const track2 = new Track($track2, height);

  timeline.add(track1);
  timeline.add(track2);
</script>
```

Figure 5: Code example for the construction of two `Track` objects and their association to a `Timeline`. The example emphasizes how the `Track` objects are inserted into the HTML document through their relation to a given DOM element.

Timeline and Layer Time Contexts

The hierarchical composition of visualizations as `Layer` objects within a `Timeline`, implies two temporal contexts for the rendering of graphical primitives, the `TimelineTimeContext` and the `LayerTimeContext`.

The `TimelineTimeContext` of a `Timeline` essentially determines which time segment is displayed and how much space it occupies on screen. This mapping is parametrized through the attributes `zoom`, `offset`, and `visibleWidth`. It allows for interactively navigating through a given timeline while maintaining view consistency upon the DOM structure. The example code in figure 6 shows how the attributes of a `TimelineTimeContext` affect the `<svg>` and `<g>` tags created by the registered tracks.

```
<svg width="${visibleWidth}">
  <!-- background -->
  <rect><rect>
  <!-- main view -->
  <g class="offset" transform="translate(${offset}, 0)">
    <g class="layout">
      <!-- layers -->
    </g>
  </g>
  <g class="interactions"><!-- for feedback --></g>
</svg>
```

Figure 6: A pseudo-code example for the DOM structure created by a `Track` when rendered into the DOM and how the `visibleWidth` and `offset` attributes of the `TimelineTimeContext` are used by a `Track` to maintain its DOM structure.

A `Layer` has to refer to a `LayerTimeContext` that determines the segment of the displayed data (i.e. through the attributes `offset` and `duration`) as well as its positioning on the timeline (i.e. through the attributes `start` and `stretch`). This mapping can be modified at runtime. Two `Layer` objects can share a common `LayerTimeContext` which allows for interactions such as the alignment of a waveform and its gain automation curve across arbitrary transformations of their common reference. The example code in figure 7 shows how the attributes of a `LayerTimeContext` affect the `<svg>` and `<g>` tags created by a layer.

```
<g class="layer" transform="translate(${start}, 0)">
  <svg class="bounding-box" width="${duration}">
    <g class="offset" transform="translate(${offset, 0})">
      <!-- background -->
      <rect class="background"></rect>
      <!-- shapes and common shapes are inserted here -->
    </g>
    <g class="interactions"><!-- for feedback --></g>
  </svg>
</g>
```

Figure 7: A pseudo-code example for the DOM structure of a `Layer` when inserted and rendered into a `Track` and how the `start`, `duration` and `offset` attributes of the `LayerTimeContext` bounded to the `Layer` are used to maintain its DOM structure.

The final mapping between the temporal layout of the data and its rendering on screen is determined by the compound application of these two levels of transformation.

Modularity and Extensibility

To allow for easily extending the library by user defined classes, all primitives — apart from `Timeline`, `Track` and `Layer` — are derived from a set of base classes sharing common interfaces. The following abstract classes[3] are provided:

BaseShape – defines specific shape to render the data. Provided extensions of this class include `Marker`, `Segment`, `Cursor`, `Dot` and `Line` (i.e. for breakpoint functions), `Waveform`, and `TraceDots` and `TracePath` (i.e. to display mean/range).

BaseBehavior – defines a specific interaction with a shape. Specific behaviors are defined for all provided shapes (e.g. the `SegmentBehavior` defines how to move and resize a `Segment`).

---

[3] These classes are considered to be abstract by convention.

EventSource – defines a new event emitter that allows for creating custom user interactions. The provided derived classes are `Surface` for mouse input on each `Track` and a global `Keyboard` listener.

BaseState – defines specific event handlers for the `Timeline` (e.g. selection, edition, zooming). This class, along with the `EventSource`, enables developers to implement their own interaction system in a unified way (see figure 8).

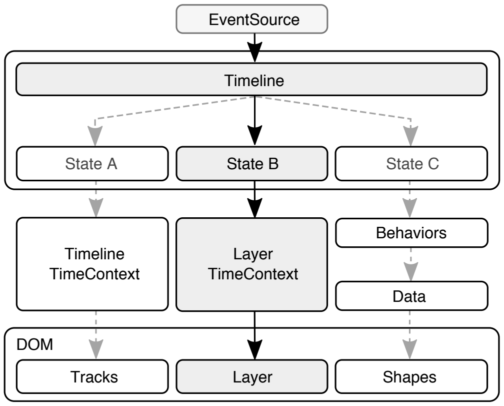

Figure 8: Propagation of an event emitted by an `EventSource` to the current state of a `Timeline`. Different states extending the `BaseState` abstract class can modify programmatically different aspects of the visualization.

Even if this architecture requires each `Layer` to be configured with its own `data`, `timeContext`, `shape`, and `behavior`, it enables the user to flexibly create high-level abstractions and reusable components such as plugins or WebComponents.

2.2 Audio Components

Similar to the user interface API, the audio side of the library provides a set of classes to build a hierarchical structure that defines the aligned synchronized of recorded data streams. The audio components are based on the concepts of *time engines* and *masters* that we already described in [12]. The `TimeEngine` class provides a general formalization of components that require precise and flexible scheduling and synchronization. For the rendering of prerecorded data, `TimeEngine` objects implementing the *transported* interface, such as a sample player or a granular player, are controlled by a `Transport` master. When being controlled by a common `Transport`, multiple `TimeEngine` objects generate synchronized data streams.

Other than on the graphics side of the library, `TimeEngine` objects are directly added to a `Transport` without the necessity to create an intermediate object (see figure 1). However, when adding a `TimeEngine`, the `Transport` creates and returns a `Transported` object that, similar to the `Layer` on the graphics side, manages the alignment of the temporal data according to the `Transport` position.[4] As for a graphics `Layer`, the alignment of a stream segment (e.g. an audio segment) to the `Transport` position is defined by its attributes `start`, `offset`, `duration` and `stretch` (see figure 2). All of these attributes can be changed dynamically during playback. Since the `Transport` itself implements a `TimeEngine` interface, a `Transport` object can be added to another. This allows for creating recursive structures in which complex compositions of stream segments may be included to another stream as a single segment.

For each data stream to be rendered, the user has to choose or implement an appropriate `TimeEngine` that generates the desired rendering. For the rending of audio data stored in an `AudioBuffer` the library provides three different `TimeEngine` classes:

PlayerEngine – functions like a conventional audio player implementing different playback speeds through resampling (based on a single `AudioBufferSourceNode`)

GranularEngine – uses granular synthesis to realize arbitrary changes of playback time and speed

SegmentEngine – requires an array of markers to trigger atomic sound segments (e.g. percussive events, syllables) in synchronization to an arbitrarily evolving playback time

The current version of the library does not provide any ready made component to render other data streams than recorded audio. However, the `TimeEngine` interface makes it very simple to create components that render data streams without having to deal explicitly with the synchronization to other data streams nor with user interactions.

3. EXAMPLE APPLICATIONS

The *WavesJS* library has been designed with a wide range of contexts and applications in mind. To illustrate the flexibility of the API, this section presents a set of applications from different domains including music information retrieval, musicology, pedagogy, and music performance.

3.1 Music Information Retrieval

The first example application illustrates a use case in the field of music information retrieval. It enables the validation and demonstration of music information retrieval algorithms applied to a music recording. The graphical user interface (see figure 9) displays an overview of the song's waveform above a more detailed segment of the stereo waveform aligned to metadata (i.e. visualizations of beats, chord progression, and song structure).[5]

The region of the detailed audio segment is displayed as a highlighted rectangle superposed to the overview waveform. The region can be moved around to scroll through the whole song. When pressing the play button on the top of the interface, the recording is played back and two cursors show the current position of the playback, one in the overview waveform and another on the detailed segment.

---

[4] While the API uses *time* to refer to the real-time audio clock, it refers to the reference rendering provided by a `Transport` as *position*. This also allows for an intuitive definition of playback *speed*.

[5] The data has been automatically extracted from the recording by three different algorithms, *Ircambeat*, *Ircamchord*, and *Ircamsummary* (http://anasynth.ircam.fr/home/english/software).

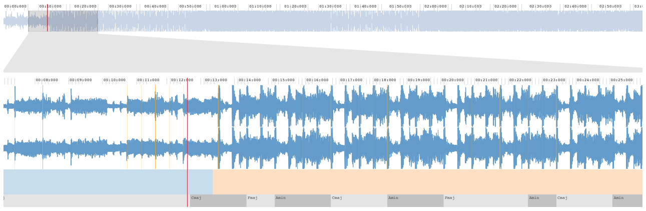

Figure 9: Screenshot of the *Phoenix - 1901* application

The application that uses the audio data and annotations for the song *1901* by Pheonix has been published at the following URL:
▷ http://wave.ircam.fr/demo/phoenix-1901/.

3.2 Musicography

Similar to the previous example, this application renders aligned audio waveforms and descriptors. In this case, the data is not only used for the visualization but also for the interaction with audio recordings. The application features recordings of 10 different interpretations of the *First Prelude* of *The Well-Tempered Clavier* by Johann Sebastian Bach performed by different piano, organ and harpsichord players. The recordings' audio waveforms and the extracted audio descriptors are displayed in 10 juxtaposed tracks. In addition, a pianoroll representation of the piece's score is displayed at the bottom of the interface (see figure 10).

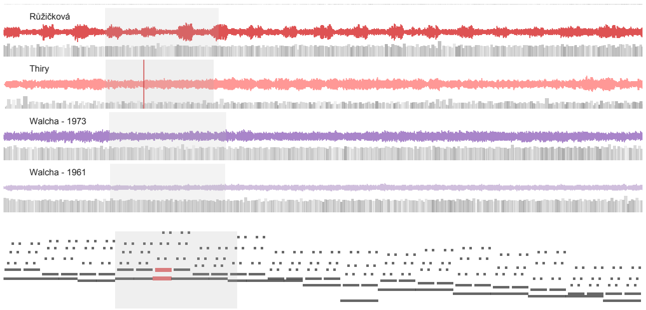

Figure 10: Screenshot of the *Bachotheque* application (reduced to 4 of 10 tracks).

The user can select and playback arbitrary segments of the recordings. Since the recordings are temporally aligned to the score, an audio segment selected at one track is automatically mapped to the corresponding segments of the other 9 tracks and the pianoroll. This enables the user to compare selected musical passages across the different recordings.
The application has been published at the following URL:
▷ http://wave.ircam.fr/demo/bachotheque/

3.3 Musicology

Figure 11 shows a screenshot of a musicological application that presents the third movement of the piece *Voi(rex)* by Philippe Leroux from 2002 [3]. The application recreates the organisation of sounds excerpts recorded from a chamber orchestra in a multitrack audio editor. Each of the 9 tracks refers to a specific operation (e.g. applying a filter or a harmonizer) that the composer applied to the sound segments.

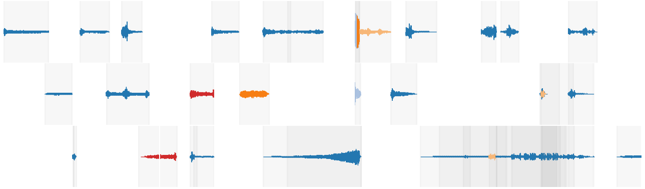

Figure 11: Screenshot of the *Voi(rex)* application

By moving the sound segments within their track, users can create their own version of the movement. The rendering of the segments continues consistently when they are moved during playback.
The application has been published at the following URL:
▷ http://wave.ircam.fr/demo/leroux-voirex/

3.4 Pedagogy

Mixed electro-acoustic music works (i.e. combining live instruments and electronic sounds) often require a complex technical setup which represents a considerable barrier for the appropriation of this repertoire by musicians — especially students. The application shown in figure 12 proposes an incomplete web implementation of the piece *Jupiter* (1987) by Philippe Manoury. It allows a flutist and a second performer to play the first page of the score directly from a browser. While the flutist interprets the soloist score, the other performer controls audio effects using a graphical user interface embedded into the score.

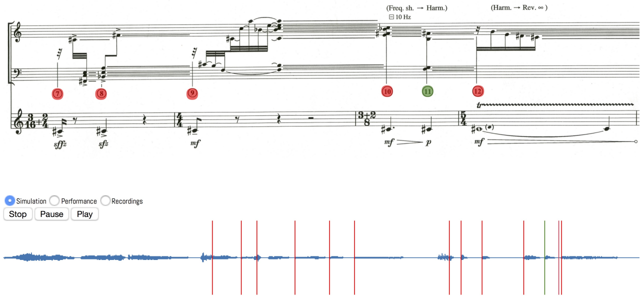

Figure 12: Screenshot of the *Jupiter* application.

Even if the audio processing is not yet as complete as in the original Max/MSP patch, the example demonstrates the basic concept the application. We imagine that mixed music works can be developed on the web platform which in the near future should allow for high-quality rendering in a concert situation as well as for web publishing in a pedagogical context. The implementation of such works based on shared and open standards may also better guarantee their preservation than current solutions.
The application has been published at the following URL:
▷ http://wave.ircam.fr/demo/manoury-jupiter/.

3.5 Performance

As already suggested above, the tools provided by the library are also suited for the context of music performances and interactive installations. The example shown in figure 13 allows for performing with recorded audio content by shaking a mobile device. The concept of the application has been originally developed for a participative live performance [13] and later adopted for a cycle of workshops with teenagers [11]. Entering the application, the user is first asked to record a fragment of sound. The recorded audio (i.e. using the `getUserMedia` API) is segmented using an onset-detection algorithm and the obtained segments are ordered according to their overall intensity. When shaking a mobile device, the percussive sound segments extracted from the recording are played in a steady tempo whereby the intensity of the segments is controlled according to the intensity of shaking (i.e. using the `DeviceMotion` API). At the same time, the user can interact with a visual representation of the segmented audio waveform on the device's touch screen.

The application has been published at the following URL: ▷ http://cosima.ircam.fr/apps/shaker/.

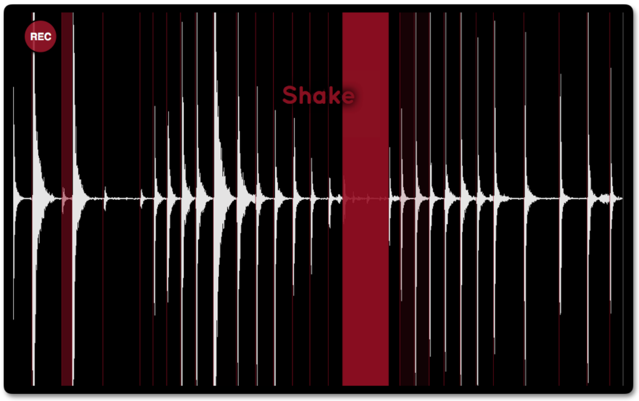

Figure 13: Screenshot of the *Shaker* application.

4. CONCLUSION

We presented a set of building blocks for the audiovisual rendering of recorded audio and related data that are provided by the *WavesJS* library. The description of the components has emphasized the similarity and differences of the graphical user interface and audio parts of the library. The flexibility of the library and its large range of applications has been illustrated by a set of examples that have developed as a proof of concept and validation of the library's concepts and implementation.

Our tests of the current version of the library have confirmed its stability and satisfying performance. Nevertheless, some performance problems could occur in very complex applications that accumulate a large number of tracks. Some of these issues could be solved by providing alternative canvas-based implementations for the graphical rendering of complex shapes like audio waveforms. However, currently these implementations require to work around remaining issues and inconsistencies across browsers (e.g. the joint use of a `<foreignObject>` and `<canvas>` elements in complex DOM structures).

Up to now, we have used the library in our own projects, which allowed us to develop and validate the key concepts as well as implementational details through several iterations. A fully functional and documented version of the the library is published on GitHub[6] and we hope that it will be adopted by a wide range of developpers.

5. ACKNOWLEDGEMENTS

The developments described in this article have been conducted in the context of the *WAVE* and *CoSiMa* research projects 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 Victor Saiz, Sébastien (Robi) Robaszkiewicz, Jean-Philippe Lambert, Karim Barkati, Riccardo Borghesi, Frédéric Bevilacqua, and Nicolas Donin for their precious contributions to this work.

6. REFERENCES

[1] BBC R&D. Recreating the sounds of the BBC Radiophonic Workshop using the Web Audio API. http://webaudio.prototyping.bbc.co.uk/, 2012.

[2] M. Buffa, A. Hallili, and P. Renevier. MT5: A HTML5 Multitrack Player for Musicians. In *WAC – 1st Web Audio Conference*, Paris, France, 2015.

[3] N. Donin, S. Goldszmidt, and J. Theureau. De Voi(rex) à Apocalypsis, fragments d'une genèse. Exploration multimédia du travail de composition de Philippe Leroux. *L'inouï, revue de l'Ircam*, (2006)(2), 2006.

[4] C. Finch, T. Parisot, and C. Needham. Peaks.js: Audio waveform rendering in the browser. http://www.bbc.co.uk/rd/blog/2013/10/audio-waveforms, 2013.

[5] Google. Inside Abbey Road. https://insideabbeyroad.withgoogle.com, 2015.

[6] Katspaugh. wavesurfer.js. http://wavesurfer-js.org/, 2012.

[7] O. Mayor. Web-based Visualizations and Acoustic Rendering For Multimodal Data From Orchestra Performances Using Repovizz. In *WAC – 1st Web Audio Conference*, Paris, France, 2015.

[8] J. Monschke. Building a Collaborative Music Production Environment Using Emerging Web Standards. Master's thesis, HTW Berlin, Germany, 2014.

[9] C. Rogers, P. Adenot, and C. Wilson. Web Audio API – W3C Editor's Draft. http://webaudio.github.io/web-audio-api/.

[10] V. Saiz, B. Matuszewski, and S. Goldszmidt. Audio oriented UI components for the web platform. In *WAC – 1st Web Audio Conference*, Paris, France, 2015.

[11] N. Schnell and S. Robaszkiewicz. Collective Sound Checks – Shaker. http://cosima.ircam.fr/2014/07/15/cosc-shaker/, 2014.

[12] N. Schnell, V. Saiz, K. Barkati, and S. Goldszmidt. Of Time Engines and Masters. In *WAC – 1st Web Audio Conference*, Paris, France, 2015.

[13] A. Tanaka, B. Caramiaux, and N. Schnell. Mubufunkscatshare: Gestural energy and shared interactive music. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, pages 2999–3002, New York, NY, USA, 2013. ACM.

[14] I. W. W. Norton & Company. The Musician's Guide to Theory and Analysis – Noteflight Worsheets. http://wwnorton.com/college/music/theory-analysis2, 2012.

---

[6] http://wavesjs.github.io/