Exposing Audio Data to the Web: an API and Prototype

David Humphrey
Seneca College
70 The Pond Road
Toronto, Canada
david.humphrey@senecac.on.ca

Corban Brook
Canadian Water Network
200 University Avenue West
Waterloo, Canada
corbanbrook@gmail.com

Alistair MacDonald
Bocoup
319 A St South
Boston, USA
al@bocoup.com

ABSTRACT

The HTML5 specification introduces the `<audio>` and `<video>` media elements, and with them the opportunity to change the way media is integrated on the web. The current HTML5 media API provides ways to play and get limited information about audio and video, but no way to programatically access or create such media. In this paper we present an enhanced API for these media elements, as well as details about a Mozilla Firefox implementation created by the authors, which allows web developers to read and write raw audio data.

Categories and Subject Descriptors

H.5.1 [**Multimedia Information Systems**]: Audio, Video and Hypertext Interactive Systems

General Terms

Experimentation, Standardization, Web

Keywords

HTML5, Audio, Firefox, FFT

1. INTRODUCTION

The HTML5 `<audio>` element allows sound or audio streams to be included in web documents, and played without the need for third-party plug-ins. Similar capabilities have been provided by native plug-ins, such as Adobe's Flash, for many years. The media elements (`<audio>` and `<video>`) are in flux, since the HTML5 specification is still under active development. As a result, we present a series of experiments, and a prototype of an enhanced media API with a view to expanding the discussions around audio in the browser. In particular, we will make the case that audio generation and access to raw audio data are an important step in the evolution of the web and web browsers.

Whereas the HTML5 media elements provide the ability to play sound and audio streams in web documents, Flash's enhanced audio APIs give developers the ability to generate real-time audio from script, extract raw audio data, obtain pre-calculated spectrum data (e.g., Fast Fourier Transform), filter and mix music. Prior to implementing the HTML5 media elements, modern web browsers lacked much of the functionality necessary to consider matching what Flash offers. However, modern browsers, such as Mozilla Firefox, now include native audio and video decoders, as well as a pipeline from the HTML document to the operating system audio interface. These inclusions provide a ready way to implement a thin API addition that leverages the work necessary to get HTML5 media generation and analysis working.

2. AN API FOR READING AUDIO DATA

We modified Mozilla Firefox in order to provide direct read and write access to the audio data available in the HTML5 media elements[1]. These changes make audio data available in real-time via an event-based API. As the audio is played, and therefore decoded, and before it is sent to the underlying audio library, each frame of audio is dispatched to content scripts via a DOM event: AudioWritten. Since this data is provided in synch with the audio itself, playing, pausing, and stopping the audio all affect the streaming of this raw audio data as well. The event use is demonstrated in Listing 1.

Listing 1: Event-based access to raw audio data

```
<audio src="song.ogg"
       onaudiowritten="audioWritten(event);">
</audio>

function audioWritten(event) {
  samples = event.mozFrameBuffer;
  // sample data is obtained using samples.item(n)
  for (var i=0; i < samples.length; i++) {
    processSample(samples.item(i));
  }
}
```

The raw audio framebuffer is a collection (e.g., array) of floating point values. Most data visualizations, or other uses of audio data begin by calculating a discrete Fourier transform by means of a Fast Fourier Transform. This can be calculated in JavaScript. However, we have included a native implementation for speed comparisons with Flash, which also includes this functionality. Both methods were used and found to be effective. Listing 2 shows how to access the pre-calculated spectrum data.

Listing 2: Event-based access to computed spectrum data

```
var spectrum;

function audioWritten(event) {
  spectrum = event.mozSpectrum;
  // spectrum data available via spectrum.item(n)
  for (var i=0; i < spectrum.length; i++) {
    processSpectrum(spectrum.item(i));
  }
}
```

3. VISUALIZING AUDIO SPECTRUM

The AudioWritten event makes it possible for web developers to create real-time visualizations of audio spectrum data. Figure 1 shows a simple example of such a visualization, and Listing 3 the

[1] See discussion and implementation code in Mozilla's Bugzilla https://bugzilla.mozilla.org/show_bug.cgi?id=490705

code to produce it. The technique shown here can be successfully used to produce much more complex visualizations.

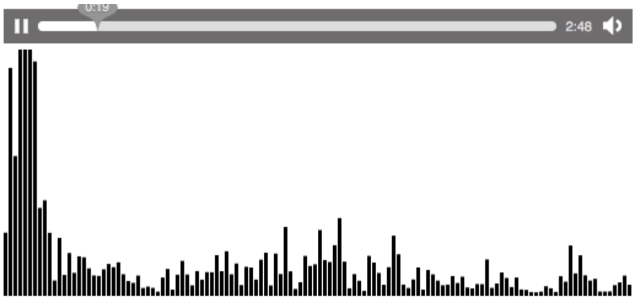

Figure 1. Audio Spectrum Visualization

Listing 3: Complete example visualizing audio spectrum data

```
<!DOCTYPE html>
<html><head>
  <title>JavaScript Spectrum Example</title>
</head>
<body>
  <audio src="song.ogg"
         controls="true"
         onaudiowritten="audioWritten(event);"
         style="width: 512px;">
  </audio>
  <div>
    <canvas id="fft" width="512" height="200"></canvas>
  </div>
  <script>
   var spectrum;
   var canvas = document.getElementById('fft');
   var ctx = canvas.getContext('2d');

   function audioWritten(event) {
     spectrum = event.mozSpectrum;
     var specSize = spectrum.length,
         magnitude;

     ctx.clearRect(0,0, canvas.width, canvas.height);

     for (var i = 0; i < specSize; i++) {
       magnitude = spectrum.item(i) * 4000;
       // Draw rectangles for each frequency bin
       ctx.fillRect(i*4, canvas.height, 3, -magnitude);
     }
   }
  </script>
</body>
</html>
```

4. AN API FOR WRITING AUDIO DATA

In the same way that the AudioWritten event provides raw audio data in the form of a collection of floats, two new methods were added to the media elements in order to allow content scripts to produce and write such data to the underlying audio layer. Typically, media elements include a **src** content attribute, which is the address of a media resource to show or play. Setting this attribute causes the browser to create an appropriate media channel for playing the audio resource, with suitable sample rate, number of audio channels, and volume. In the case of dynamically generated audio through script, another mechanism is needed in order to setup the media channel. After the channel is created, audio frames can be sent to the media channel to be played. Listing 4 demonstrates the setup and use of these new media methods.

Listing 4: Settup and writing to an audio element

```
var audioOutput = new Audio();
audioOutput.mozSetup(2, 44100, 1);
var samples = [0.242, 0.127, 0.0, -0.058, ...];
audioOutput.mozAudioWrite(samples.length, samples);
```

The **mozSetup** method takes three arguments, including: number of channels, sample rate per second, and initial volume. The **mozAudioWrite** method can then be called with an array of floats sufficient to represent one frame's worth of samples at the specified rate. Listing 5 shows a more complete example, allowing the user to dynamically generate and play a tone at the specifed Hz.

Listing 5: A simple HTML tone generator

```
<!DOCTYPE html>
<html><head>
  <title>JavaScript Audio Write Example</title>
</head>
<body>
  <input type="text" size="4" id="freq" value="440">
  <label for="hz">Hz</label>
  <button onclick="generateWaveform()">set</button>
  <button onclick="start()">play</button>
  <button onclick="stop()">stop</button>
  <script type="text/javascript">
   var sampledata = [];
   var freq = 440;
   var interval = -1;
   var audio;

   function writeData() {
     var n = Math.ceil(freq / 100);
     for(var i=0;i<n;i++)
       audio.mozWriteAudio(sampledata.length,
                           sampledata);
   }

   function start() {
     audio = new Audio();
     audio.mozSetup(1, 44100, 1);
     interval = setInterval(writeData, 10);
   }

   function stop() {
    if (interval != -1) {
      clearInterval(interval);
      interval = -1;
    }
   }

   function generateWaveform() {
     var f = document.getElementById("freq").value;
     freq = parseFloat(f);
     // Playing at 44.1kHz, figure out how many
     // samples will give us one full period
     var samples = 44100 / freq;
     sampledata = Array(Math.round(samples));
     for (var i=0; i<sampledata.length; i++) {
       sampledata[i] = Math.sin(2*Math.PI *
                       (i / sampledata.length));
     }
   }

   generateWaveform();
  </script>
</body>
</html>
```

5. DOM IMPLEMENTATION DETAILS

The API presented above required a number of additions and changes to the current DOM implementation of Mozilla Firefox. Two new DOM interfaces were created in order to support the AudioWritten event, and two new methods were added to the nsIDOMHTMLMediaElement interface.

First, in the case of the AudioWritten event, both raw framebuffer data and computed spectrum data are returned in a pseudo-array named nsIDOMAudioData (see Figure 6). This is not as efficient as possible, but was chosen for ease of implementation in other browsers. In future this could be changed to use more efficient native array types, such as those being introduced for canvas pixel data.

Listing 6: Audio Data DOM interfaces (Mozilla XPIDL)

```
interface nsIDOMAudioData : nsISupports
{
  readonly attribute unsigned long length;
  float              item(in unsigned long index);
};

interface nsIDOMAudioWrittenEvent : nsIDOMEvent
{
  readonly attribute nsIDOMAudioData mozFrameBuffer;
  readonly attribute nsIDOMAudioData mozSpectrum;
};
```

The **length** attribute indicates the number of elements of data returned. The **item** method provides a getter for audio data, which are floats.

The AudioWritten event's **mozFrameBuffer** attribute contains the raw audio data (float values) obtained from decoding a single frame of audio. This is of the form [left-channel, right-channel, left-channel, right-channel, ...] (e.g., stereo interlaced). All audio frames are normalized to a length of 4096 or greater, with shorter frames padded with 0s (zeroes).

The **mozSpectrum** attribute contains a pre-calculated Fourier transform for the current frame of audio data. It is calculated using the first 4096 float values in the audio frame only, which may include zeros used to pad the buffer. We take the 4096 stereo interlaced samples, mix them down to a 2048 mono track, and then calculate a Fourier transform to get 1024 frequency bins.

Audio write access is achieved by adding two new methods to the nsIDOMHTMLMediaElement, as shown in Listing 7.

Listing 7: Additions to nsIDOMHTMLMediaElement (Mozilla XPIDL)

```
void mozSetup(in long channels, in long rate,
              in float volume);
void mozWriteAudio(in long count,
                   [array, size_is(count)] in float
                   valueArray);
```

The **mozSetup** method allows an `<audio>` or `<video>` element to be setup for writing from script. This method must be called once before **mozWriteAudio** can be called, since an audio channel has to be created for the media element. It takes three arguments:

1. channels - the number of audio channels (e.g., 2)
2. rate - the audio sample rate (e.g., 44100 Hz)
3. volume - the initial volume to use (e.g., 1.0)

The choices made for channel and rate are significant, because they determine the frame size that must be used in subsequent calls to **mozWriteAudio**. Sending the wrong amount of data for a frame will result in a DOM exception being thrown.

The **mozWriteAudio** method can be called after **mozSetup**. It allows a frame of audio (or multiple frames, but always whole frames) to be written directly from script. It takes two arguments:

1. count - the number of elements in this frame (e.g., 4096)
2. valueArray - an array of floats, which represent a complete frame of audio (or multiple frames, but whole frames).

A DOM exception is thrown if the audio frame size does not match what is expected based on the initial call to **mozSetup**.

6. DISCUSSION

In our experiments building web audio applications with the API and prototype discussed in this paper, a number of potential uses have emerged, from accessibility to gaming and rich media.

One area that would benefit right away from this work is web accessibility. Having access to raw audio data would mean new possibilities for visualizing sound for the hearing impaired, or for building text to speech or speech to text interfaces, which can be written and deployed as part of the web page itself. Similarly, as 3D becomes a more common feature of the web through WebGL, O3D, etc., sound provides a way for the visually impaired to interact with 3D web spaces through sound manipulation and 'seeing' with sound (e.g., depth and feature perception through echo).

Another area that would benefit from these techniques is rich media web applications, for example music sites and online games. Many web applications need to update or stay in synch with an audio track. This might mean using something like beat detection in order to drive a visualization or advance events in the page. Another common need is generating real-time audio, such as sound effects for games, or altering (e.g., mixing, filtering, etc.) audio as it is played. This also opens the door to the creation of in-browser interaction with sound, for example, creating instruments like pianos or synthesizers.

7. CONCLUSION AND FUTURE WORK

In this paper we discussed an enhanced audio data API to supplement the HTML5 media elements. We discussed a simple set of additions to the existing specification, which add an event-based, real-time data access API, as well as an API to generate audio data from script. We presented two complete examples of using these APIs in HTML5. We then discussed the technical implementation details necessary to add this functionality to the DOM. Finally, we identified a number of future applications for this API, and showed how these applications are broadly appealing to many types of users. We believe, and have shown through our own implementation work, that the additions to the HTML5 specification presented here are modest yet powerful.

8. ACKNOWLEDGMENTS

Our thanks to the Mozilla Corporation and the Processing.js community, who have provided valuable feedback and input on the API and its implementation.