Adventures in scheduling, buffers and parameters : Porting a dynamic audio engine to Web Audio

Chinmay Pendharkar
Sonoport Asia Pte. Ltd.
Blk 71 Ayer Rajah Crescent
Ayer Rajah Industrial Estate,
Singapore
chinmay.pendharkar@sonoport.com

Peter Bäck
Sonoport Asia Pte. Ltd.
Blk 71 Ayer Rajah Crescent
Ayer Rajah Industrial Estate,
Singapore
peter.back@sonoport.com

Lonce Wyse
Communication and New Media Department
National University of Singapore
11 Computing Drive,
Singapore
lonce.wyse@nus.edu.sg

ABSTRACT

At Sonoport, we ported our Dynamic Sound Engine from Adobe's Flash technology to Web Audio. The difference in approaches to threading, scheduling and parameters between Flash and Web Audio created a few challenges for us. These differences and some peculiarities of Web Audio required workarounds to be able to implement our Dynamic Sound Engine in Web Audio. In this paper we discuss three of these workarounds dealing with creating Parameters, scheduling operations and playback position of buffers, and explain how these work-arounds, although not optimal solutions, allowed us to support our use cases. Finally we consider how the upcoming AudioWorker change in the Web Audio specification, is expected to impact these workarounds.

Categories and Subject Descriptors

H.5.5 [**Information Systems**]: Information Interfaces and Presentation (HCI)—*sound and music computing*

General Terms

Design

Keywords

Interactive Audio, W3C Web Audio API, Audio Synthesis

1. INTRODUCTION

Before the introduction of HTML5 and the modern web platform, plugins such as Adobe's Flash and Sun Microsystem's Java were the only means of creating and consuming rich interactive experiences on the web. HTML5 and the related web standards, especially the Web Audio specification, enabled the creation of rich interactive audio experiences using native browser capabilities instead of relying on plugins.

Sonoport's Dynamic Sound Engine was developed prior to the introduction of the modern web platform and hence was based on Adobe's Flash and related technologies. In the beginning of 2014, with a significant percentage of browsers now supporting the Web Audio API standard, we decided to port the dynamic sound synthesis engine from Flash written in ActionScript 3 to Web Audio developed in JavaScript.

Copyright ©2015 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

While the Web Audio specification is still evolving, many browsers such as Chrome, Safari and Firefox provide a stable implementation to develop and test the sound synthesis engine against. However, there were significant differences in some of the fundamental approaches these two platforms (Flash and Web Audio) take, that had to be overcome during this migration.

In this paper, we describe some of the major constraints we had to work around in this migration, the techniques we used to work around them as well as the effectiveness of these techniques in typical use cases. Finally, we also look at how the upcoming changes to the Web Audio specification may change these constraints.

Since the Web Audio specification is constantly changing, we will be referring to 2012 version of the specification [8] which is implemented by most of the browsers. The newer changes to the specification may or may not have been implemented by browsers at the time of writing.

2. FLASH TO WEB AUDIO

2.1 Dynamic Sound Engine

Sonoport's Dynamic Sound Engine is a library of audio functionality that exposes a collection of Sound Models. Sound Models [13] are parameterized algorithms for generating a class of sounds that run in real time. The Sound Models generally expose a set of parameters, which can be adjusted in real time, as well as actions, such as `play` and `pause`, that can be invoked on these Sound Models. The Sound Models can be either sample based, where they act upon a sample or texture, or completely synthesized based on various types of synthesis algorithms.

Sonoport's Dynamic Sound Engine was originally developed from a similar technology which was implemented in Java called ASound [12]. It was then migrated to Adobe's Flex Framework. The work done in those projects was migrated to work with Adobe's Flash on the Web platform creating the original Sonoport's Dynamic Sound Engine.

2.2 Web Audio

When we began to port Sonoport's Dynamic Sound Engine in early January of 2014, Web Audio implementations in the browsers were stabilizing, with Firefox having recently

moved from Audio Data API to Web Audio [6].

There were a few implementations of dynamic sound engines and sound synthesis frameworks in JavaScript based on Web Audio, which had started off in the experimental days of Web Audio and matured over time. audiolib.js [5] and flocking.js [3] were the most notable.

After reviewing these projects, we decided to write our own framework from scratch, instead of using one of the pre-existing frameworks. Firstly, we wanted to keep a similar API and usage pattern in the JavaScript version of the engine as the original Flash version. Secondly, the Web Audio API provided many high level primitives which reduced the need to rely on third party libraries for basic functionality. Building on top of a pre-existing framework would only add dependency without much utility.

2.3 Threading in Flash

Most interactive sound engines work by generating small chunks of audio data in real time. For example the Sound object in Flash uses a chunk size in the range 2048-8192 bytes. These chunks are sent to the underlying audio system to be played out. When the underlying audio system runs out of data to play, it requests, usually through a function callback or an event, more data from the sound engine. Given the constant sampling rate of the audio being played back, these requests for audio data occur at regular intervals. The response to this request for audio data needs to be completed within a specific amount of time, usually defined by the arrival of the next data request. If that constraint is not fulfilled then the audio may drop out or glitch.

The underlying threading model significantly affects the interaction of this audio data requests with the rest of sound engine as well as the rest of the application running on the system. If the request occurs in its own dedicated thread, a so called audio rendering thread, it doesn't interfere with or block the running of the rest of the system. However, this also mean that any data generated by the sound engine needs to be passed over to this audio rendering thread. Similarly, thread synchronization needs to happen for parameter values to ensure that parameters only get changed at appropriate instances (a-rate or k-rate [7]).

Adobe's Flash framework uses the other approach where the requests for audio data (from Flash Player version 10 onwards) [2] are made in a common thread (the so called Primordial Worker) which is shared by the entire application. While this reduces the complications of thread synchronization, this also means that any other part of the application, such as the user inputs, the graphics framework, etc, might block this common thread not allowing the audio engine to process the audio data request in time, thus causing clicks and glitches in the audio. To work around this, larger chunks can be used to allow for longer time between requests or ring buffers can be used to generate the data in advance, however, these approaches increase the latency of interaction in the audio engine, which is not preferable.

2.4 Web Audio and the ScriptProcessorNode

The Web Audio specification deals with threading of the audio rendering in two distinct ways. There are two Web Audio `AudioNodes` which generate audio, the `AudioBuffer-SourceNode` and the `OscillatorNode`. Both of these are implemented natively in the browser and handle the audio data requests in a separate audio rendering thread. The

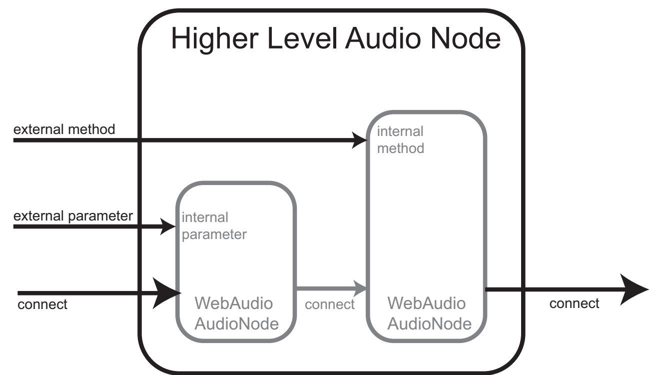

Figure 1: Composing higher level audio node using Web Audio AudioNodes

user application can interact with these nodes by assigning buffers of audio data to them or by setting various parameter on them. This approach hides the complexity of thread synchronization by exposing a much simpler interface to the user. However, that restricts the kind of audio that may be generated to oscillating sounds and pre-rendered buffers.

Although deprecated in the Web Audio specification as of August 2014, the `ScriptProcessorNode` takes another approach and provides a mechanism for the user application to reply to the audio data request (an `onaudioprocess` Event) in JavaScript. This is a very similar to the approach taken by Adobe's Flash framework, where the requests are handled in a common thread. In the case of Web Audio this common thread is the JavaScript event loop. Hence the `ScriptProcessorNode` suffers from the same vulnerability as the other parts of the user application blocking the event loop from processing the audio data request.

In the case of a dedicated music app, or a game where the various aspect of the application are under the users control, it might be possible to guarantee that all other aspects of the application would not block the event loop from processing the `onaudioprocess` Event. However, when creating a generic dynamic audio engine which might be used in various, as yet unknown, applications, such a guarantee cannot be made. This opens up the possibility of a glitch or dropout in the audio, which makes for a terrible audio experience.

Hence, we decided while we were porting Sonoport's Dynamic Sound Engine to Web Audio to avoid using the Web Audio `ScriptProcessorNode` to generate audio data. The `ScriptProcessorNode` however was used in certain specific cases to synchronize the playing of other `AudioNodes` as described below.

2.5 Higher Level AudioNodes

`AudioNodes` are the fundamental building blocks of Web Audio. While the `ScriptProcessorNode` allows building of an `AudioNode` that does custom audio synthesis or processing, since we were avoiding using that, we had to create Sound Models by composing multiple other `AudioNodes` as shown in Figure 1.

Sound Models were designed as higher level audio nodes, which behaved just like other `AudioNodes` in terms of functionality they exposed such as connection methods, action methods and parameters. These higher level audio nodes internally managed multiple interconnected `AudioNodes` to

synthesize and process the audio as required. This implied that Sound Models could be used as new `AudioNodes` as long as they were first in the chain of Nodes. This approach works well as all Sound Models are designed as sound creating Nodes rather than processing Nodes.

This is similar to the *jsnode* approach implemented by Subramanian S. [9] although *jsnode* uses a `ScriptProcessorNode` to create higher level nodes. But since we wanted to avoid the use of `ScriptProcessorNode`, our approach allowed wrapping around any other `AudioNode`. This, however, made managing Parameters more complicated.

```
var looper = new Looper (context, "/audio.
    wav");
looper.easeIn.value = 3;
looper.easeOut.value = 2.5;
looper.start(0);
looper.playSpeed.value = 5;
```

Listing 1: Using Sound Models

This approach also ensured that the Sound Models API retained a structure similar to the Web Audio API for familiarity. And yet the API exposed the same functionality as the Flash version. Listing 1. shows an example source code for initializing a Looper Sound Model, changing it's `easeIn` and `easeOut` parameters, starting the playback of the Sound Model and then setting the `playSpeed` parameter.

3. PARAMETERS IN WEB AUDIO

`AudioNodes` in Web Audio have parameters of type `AudioParam`, which can be used to change the behaviour of the `AudioNodes`. These `AudioParams` are defined on the `AudioNode` based on the `AudioParam` Interface defined in the Web Audio specification. However currently, no mechanism exists to implement custom `AudioParams` on a generic JavaScript object. We needed custom `AudioParams` defined on the higher level audio nodes or Sound Models.

To keep the API of our Sound Models and the native `AudioNodes` consistent, we devised a wrapper around `AudioParams`. These pseudo-`AudioParams` are plain JavaScript objects that expose an `AudioParam` like interface. However, they interact with `AudioParams` on internal `AudioNodes` inside a Sound Model. In such cases where there are no appropriate underlying `AudioParams`, these pseudo-`AudioParams` will emulate `AudioParams` in JavaScript.

3.1 Wrapped AudioParams

Within the browser implementations of Web Audio, the `AudioParams` are native JavaScript Objects which conform to the `AudioParam` Interface definition of the Web Audio specification. The `AudioParam` Interface defines a couple of properties on the `AudioParam` and a set of parameter value automation methods, listed in Table 1. These method automatically change the value of the parameter over time. The methods are executed in the rendering thread and work synchronously with respect to the actual audio data being generated or synthesized.

To work around the inability to create `AudioParams` we have to mimic the properties and the methods exposed by the `AudioParam` interface. This wrapper around `AudioParam`, which we call *SPAudioParam*, can then be exposed on a Sound Model to be used similarly to an `AudioParam`.

3.2 SPAudioParams

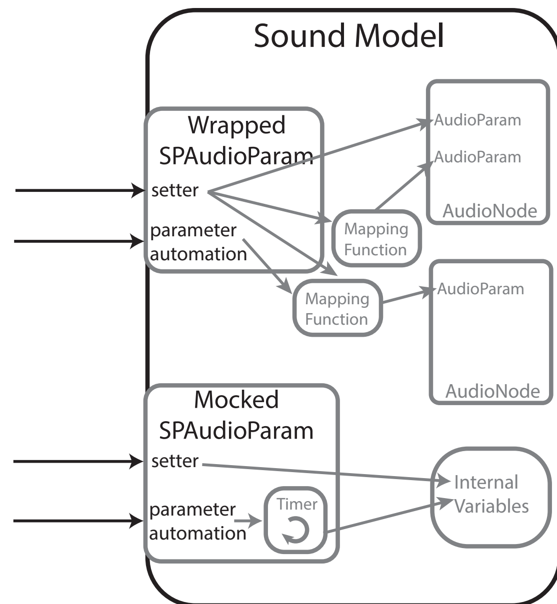

Figure 2: Comparison between the two approaches for wrapping AudioParams

A wrapped parameter, an *SPAudioParam*, can be easily created on any JavaScript Object, such as a Sound Model Object, using the JavaScript `Object.defineProperty` API. With this API, a setter method can be defined to either change the internal variables or set one or more `AudioParams` of the internal `AudioNodes` when the value of *SPAudioParam* is changed. This is an effective and clean parameter abstraction for the `value` and `defaultValue` properties of `AudioParams`.

However, wrapping around parameter automation methods is slightly more complicated. Two approaches can be taken for this. If the *SPAudioParam* can be linked to one or more `AudioParams` on `AudioNodes` internal to the Sound Model, then the wrapper can just pass along those method calls to the parameter automation methods to the internal parameters. We call this the *Wrapped SPAudioParam* approach.

In the *Wrapped SPAudioParam* approach there is no specific requirement on the relationship between the external *SPAudioParam* and the internal `AudioParam`. A mapping function can be specified at the point of creation to convert the value of the external *SPAudioParam* to the appropriate value for the internal `AudioParam`.

The other approach is to mock the parameter automation using JavaScript's `setInterval` timer functionality. We called this the *Mocked SPAudioParam* approach. The mathematical formula to calculate the value of the automated parameter at various time instances are given in the Web Audio specification [8]. These can be implemented in JavaScript using an interval timer triggering at some predefined small time interval. Figure 2. shows difference between the two approaches for wrapping the AudioParams

The *Mocked SPAudioParam* approach is a suboptimal and an inaccurate solution as `setInterval` is not guaranteed to fire with millisecond accuracy. Furthermore, the values of the internal properties are only updated every time the

Table 1: AudioParams Parameter Automation Methods

Method Name	Description
setValueAtTime	Schedules a parameter value change at the given time.
linearRampToValueAtTime	Schedules a linear continuous change in parameter to the given value.
exponentialRampToValueAtTime	Schedules an exponential continuous change in parameter value from to the given value.
setTargetAtTime	Start exponentially approaching the target value at the given time with given rate.
setValueCurveAtTime	Sets an array of arbitrary parameter values.

Table 2: Operations performed by a Queue

Operation	Description
PLAY	Starts playing a voice
STOP	Stops playing a voice
RELEASE	Decays the volume of a voice and stops
SETSOURCE	Sets a buffer as the source for a voice
SETPARAM	Sets a parameter on a voice Sound Model

timer fires, making it inaccurate during the time interval in between two timer callbacks. Hence, instead of the sample accurate parameter automation changes as defined by the Web Audio specification, these parameters change with a step function. Furthermore, the *Mocked SPAudioParam* approach can't emulate the interaction of overlapping parameter automations accurately and can cause unexpected behaviour.

The *Wrapped SPAudioParam* approach works well and was used throughout the implementation of the Dynamic Audio Engine. In a few cases we had to implement the *SPAudioParam* using the *Mocked SPAudioParam* approach, since there was no appropriate internal `AudioParam` available. Listing 2. shows the source code for initializing such a *Wrapped SPAudioParam.* This usually happens internal to the Sound Models defined in the Dynamic Sound Engine and is only exposed to end users as properties of type *SPAudioParam* on each Sound Model.

```
var buffer = context.createBufferSource();
var playSpeedParam = new SPAudioParam('
    playSpeed', buffer.playbackRate,
    mapFunction);
```

Listing 2: Creating a Wrapped SPAudioParam

However, in certain situations where specific *SPAudioParam* need not be automated, or does not need highly accurate scheduling (for example, a parameter for describing the number of times playing a buffer is to be looped before it is automatically stopped, maxLoops), the mocked *SPAudioParam* approach provides a viable alternative that still exposes a consistent interface for *SPAudioParam.*

4. QUEUES AND SCHEDULING

Queues are an essential aspect of any Dynamic Audio Engine. Queues allow operations, such as those listed in Table 2., defined on certain Sound Models to be scheduled for the future and processed just in time. While Web Audio supports scheduling of method calls and parameter changes, it doesn't provide for a way to unschedule a specific event or to change the timing, arguments or parameters of an event once it is scheduled.

Queues are also important in polyphonic audio synthesis when multiple voices are playing or synthesising different parts of the audio. With multiple voices, queues can be used to hold a series of operations, which can be assigned to different voices based on which voices are busy and which voices are free. A queue will ensure these operations are executed at the stipulated time.

Finally queues also allow dependent operations to be scheduled and executed in the correct order. This is important in scenarios where the operation such as `PLAY` can't proceed before a `SETSOURCE` operation is executed on a specific voice.

4.1 Queue Model

The queue model we implemented is based on a scheduler created using the JavaScript `requestAnimationFrame` API which was designed for use with HTML5 Canvas. The queue dispatches any operations which are due within 16ms (corresponding to a 60 millisecond refresh rate) using Web Audio to schedule the operations accurately. So the queue is responsible for a larger time frame aspect of the scheduling and the Web Audio methods do the accurate scheduling of the operations. This two clock approach originally highlighted by Wilson [11] yields a queue that is accurate as well as controllable.

The queue model works with all Web Audio operations which are schedulable and takes in a timestamp as a parameter. These operations are executed exactly at the instance of time in the timestamp. However certain operations are non-schedulable in Web Audio, for example, setting the buffer property of an `AudioBufferSourceNode` or `setPeriodicWave` method on an `OscillatorNode`. Hence it is difficult to incorporate these operations in the queue model.

Our first attempt was to mimic Web Audio style scheduling of these non-schedulable operations using JavaScript's `setTimeout` API. However, this approach failed to work in situations where the order of operations was critical. Since `setTimeout` API isn't millisecond accurate, there were situations where although a `PLAY` operation on a specific voice was scheduled after a `SETSOURCE` operation, the actual execution order was reversed and the `PLAY` operation was called before the `AudioBuffer` was assigned by the `SETSOURCE` operation. The result was a disruption to audio playback.

One workaround to deal with this could be to execute all non-schedulable operations, such as `SETSOURCE` significantly ahead of their stipulated timestamp. This ensures that they have been executed at the time defined by their timestamp. However this means the order of the queue is completely violated and could cause problems if that voice is being used for other operations exactly when this non-schedulable operation is executed.

4.2 Voices and Polyphony

Redesigning how voices and polyphony works was the solution for being able to accurately queue the non-schedulable

operation.

Traditionally a polyphonic synthesis engine would have a limited number of voices, usually limited by the hardware oscillators there exist, or on computer systems, limited by CPU speed or memory available. However, with the modern Web platform running on powerful CPUs, these are rarely limiting factors. Nonetheless polyphonic voice-based synthesis engines are still designed with a limited number of voices which are recycled when their previous set of operations are completed. This design also helps to reduce the memory footprint of the synthesis engine.

4.3 Garbage Collection

Web Audio has been designed to run on the modern day JavaScript Virtual Machines with advanced garbage collection. Fink S. [4] gives a great primer on how garbage collection works in Firefox's SpiderMonkey JavaScript Virtual Machine. In general the design encourages creation of as many `AudioNodes` as necessary and leaving them to be garbage collected when they're not needed anymore. This is further encouraged by having some `AudioNodes` enforce a single use policy. For example, the `AudioBufferSourceNode` can only be started and stopped once, after which, the `AudioBufferSourceNode` has to be discarded.

The Web Audio specification also implies that the native implementation of the `AudioNodes` would handle caching of buffers and memory to ensure that repeated allocation of `AudioBufferSourceNode` would be optimized for performance.

This approach makes the garbage collection algorithm critical and in cases where `AudioNodes` are being created rapidly, the ability of the garbage collection algorithm to quickly and effectively collect `AudioNodes` that have finished playing, ensures a smooth operation. The Web Audio specification also highlights the mechanism with which the garbage collection algorithm collects finished `AudioNodes`.

4.4 Single Use Voices

The redesigned polyphonic voices in our Dynamic Sound Engine used the approach that Web Audio suggests of creating as many `AudioNodes` as needed and ensuring that they are garbage collected by removing all references to them. Instead of limiting the number of voices, we can create more voices as and when we need them. By not having to recycle voices, some of the constraints on the timing of the operations can be relaxed. This allows operations such as `SETSOURCE` to be executed far before their due time since the voice they are being executed on is a new voice and there is no danger of affecting any previous operation in progress.

This fire and forget method for implementing voices allowed us to scale queues and schedule up to 60 operations per second with certain Sound Models with negligible jitter. However that requires creation of many new voices every second. With each voice having a `GainNode` to support ADSR Envelope the number of `AudioNodes` to be created per operation is doubled.

Taking this approach implied we had to be extremely careful to ensure that the voices that had been used up were garbage collected at the next garbage collector pass. We did this by ensuring the relevant voices did not have a playing `AudioBufferSourceNode` or a `OscillatorNode` and that all references to these voices and any internal AudoNodes were discarded. To ensure there were no memory leaks from this, memory profiling tools in Chrome and Firefox were used.

5. BUFFERS AND PAUSING

Pausing a playing sound at a given instant and restarting it from the exact playback position where it stopped is a common use case needed in implementing a Dynamic Sound Engine. However the single use design of the `AudioBufferSourceNode` in Web Audio makes implementing this behaviour difficult. While there is a mechanism for starting the playback of a `AudioBufferSourceNode` at a given playback position which is accurate to a sample, knowing exactly where a `AudioBufferSourceNode` stopped is not so straightforward.

5.1 Tracking Playback Position

A naive approach is to count the time between when the `AudioBufferSourceNode` is started and when it is stopped. This approach works when the `playbackRate` parameter of the `AudioBufferSourceNode` is unchanged, since the playback position only advances at a constant rate for the time interval the `AudioBufferSourceNode` is being played. Since the start and the stop operations are called with sample accuracy it is possible to get the accurate value for the time between them. However, with a changing `playbackRate` this calculation gets more complex. To support tracking of the current playback position, all operations on the `playbackRate` parameter of `AudioBufferSourceNode` need to be intercepted and tracked. Everytime the value of `playbackRate` is set, the current timestamp can be used to update the tracking of the playback position. However, it becomes much more complicated when trying to intercept parameter automation methods, not only because of the slight differences in internal implementations of these automation algorithms in various browsers but also the way multiple parameter automation requests interact when they are invoked with an overlapping time range.

As we implemented this we realized that we were basically reimplementing in JavaScript parameter automation as implemented natively by the browsers. This was an unsustainable approach, as we would have to learn and implement the nuanced differences in the native implementations of these automation algorithms in the browsers and also keep track of any changes in these implementations and change our implementation accordingly.

5.2 Counting Playback Position

In the approach we finally implmented, we composed a higher level `AudioNode`, which we call *SPAudioBufferSourceNode*, that contains the `AudioBufferSourceNode` whose playback position we are trying to track.

This higher level `AudioNode` contained a second `AudioBufferSourceNode`. This new `AudioBufferSourceNode`, which we call the *Counter Node*, is assigned a buffer which has monotonically increasing integer values which represent the index in the primary buffer.

This *Counter Node* is connected to a `ScriptProcessorNode`, which we call *Scope Node*. The *Scope Node* gets the audio data output from the *Counter Node* and keeps track of the value of the last frame of audio data it gets for every audio data request or chunk. The *Scope Node* also ensures that the audio data it sends forward in the processing chain is all zeros thus not affecting the output audio.

This *Scope Node* is connected to the same output node as

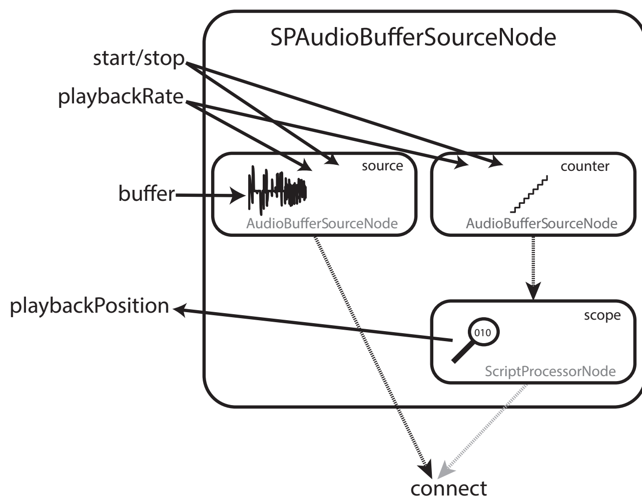

Figure 3: *SPAudioBufferSourceNode* - a work around to count current playback position

the original AudioBufferSourceNode whose playback position we are trying to track.

Web Audio ensures that both inputs, the *Counter Node* connected to the *Scope Node* as well as the original AudioBufferSourceNode are played out in sample accurate synchrony. Figure 3 demonstrates the connections of an *SPAudioBufferSourceNode*.

By tying the start and stop method calls as well as any changes to the playbackRate, including parameter automation methods, on this new *SPAudioBufferSourceNode* to both the internal AudioBufferSourceNodes we can ensure that the value of the audio data in the *Scope Node* will correspond to the current playback position in the original AudioBufferSourceNode.

We had to use a ScriptNode here as there was no other way to track the playback position of the AudioBufferSourceNode while still avoiding ScriptProcessorNode.

It is important to realize that the position we're tracking would be inaccurate while the AudioBufferSourceNode is playing since by the time we retrieve the position, the playback would have proceeded forward. But it is very accurate for finding out at which position an AudioBufferSourceNode stopped playing.

```
var extender = new Extender (context, "/
    audio.wav");
extender.pitchShift.value = 2;
extender.eventPeriod.value = 5;
extender.start(0);
window.setTimeout(function (){
      extender.pause();
}, 1000);
window.setTimeout(function (){
      extender.play();
}, 2000);
```

Listing 3: Playing and Pausing Sound Models

With this structure in place, a playing Sound Model could be paused by stopping it and storing it's playback position. When it was requested to continue playing, the playback position could be used as a starting position for a new AudioBufferSourceNodes. This approach allows for pause and unpause functionality, while not sample accurate, accurate to a single chunk length.

Listing 3. shows the source code that can be used to pause and unpause the playback of the Extender Sound Model. This will pause the Sound Model's playback 1 second after the start, and unpause after another second.

6. AUDIO WORKER

The Web Audio specification is being updated and there is an upcoming major change [1] to improve some of the functionalities. This change is still being discussed [10] and agreed upon and has not yet been implemented by the browsers at the time of writing this paper.

The most critical addition in this proposed change is the ability to create an AudioWorker. An AudioWorker would allow the audio data requests to be handled in a background thread using the WebWorker technology to implement threading. This would solve a lot of the issues discussed in Section 2.4 about ScriptNodes. With the AudioWorker, ScriptNode like functionality becomes viable without the risk of being blocked by other part of the application.

This addition to Web Audio would solve almost all the problems described above. With an open ended customizable AudioNode available, many of the schemes we had used in the original Flash design could be brought back to Web Audio without the risk of audio glitches. There may still be cases where using the natively implemented AudioNodes such as GainNode or even AudioBufferSourceNode is appropriate for performance reasons. If more granular levels of control are called for and performance is a lesser concern, then an AudioWorker can be used.

The AudioWorker updates will also add the ability to create AudioParams on the AudioWorker. These AudioParams will support parameter automation and provide sample accurate parameter values to the callback handling the audio data request. This will finally provide a means to have a complete, reliable and robust AudioParam on Sound Models.

While the playback position of an AudioBufferSourceNode might not be available, it can easily be implemented as a property on an AudioWorker. Since the playback of audio would be done internally in the AudioWorker, tracking the playback position and exposing it as an external property would be trivial.

Finally, all operations on Sound Models implemented with AudioWorker can be scheduled by passing in a time stamp. The actual performance of the implementation of operation dispatch is yet to be seen but this approach will definitely yield a more robust design with fewer chances of glitches and drop outs.

7. CONCLUSIONS

We ported a Dynamic Sound Synthesis Engine from Adobe's Flash platform to Web Audio. Due to the differences in the architecture many structures of the engine had to be redesigned to work with the constraints of Web Audio. Some peculiarities of Web Audio, such as single use AudioNodes, inability to create AudioParams and lack of access to playback position were worked around with various techniques. Most of these workarounds did not completely solve the problem, but did provide a solution in specific use cases that were needed for the Dynamic Sound Synthesis Engine. Finally, we believe the upcoming AudioWorker updates to

Web Audio can solve most of the issues we discussed. However, the performance of doing most of the computation in JavaScript using the AudioWorker as opposed to using native AudioNodes remains to be seen.

8. ACKNOWLEDGMENTS

This paper is based on technology developed at Sonoport, www.sonoport.com.

9. REFERENCES

[1] P. Adenot and C. Wilson. Web Audio API W3C Editor's Draft 15 October 2014. `http://webaudio.github.io/web-audio-api/#the-audioworker`, October 2014.

[2] Adobe. Sound - Adobe ActionScript 3 (AS3) API Reference. `http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/media/Sound.html`, October 2014.

[3] C. Clark. Flocking - creative audio synthesis for the web. `https://github.com/colinbdclark/Flocking`, October 2014.

[4] S. Fink. Clawing our way back to precision | javascript. `https://blog.mozilla.org/javascript/2013/07/18/clawing-our-way-back-to-precision/`, July 2013.

[5] J. Kalliokoski. audiolib.js is a powerful audio tools library for javascript. `https://github.com/jussi-kalliokoski/audiolib.js`, October 2014.

[6] Mozilla. Listen up! web audio api now in firefox - completes web as a platform for gaming. `https://blog.mozilla.org/blog/2013/10/29/listen-up-web-audio-api-now-in-\firefox-completes-web-as-a-platform-for-gaming/`, October 2013.

[7] M. D. Network. Audioparam - web audio api. `https://developer.mozilla.org/en-US/docs/Web/API/AudioParam`, September 2014.

[8] C. Rogers. Web Audio API - W3C Working Draft 13 December 2012. `http://www.w3.org/TR/2012/WD-webaudio-20121213/`, December 2012.

[9] S. Subramanian. steller/jsnode.js at experimental_buildsys - srikumarks/steller. `https://github.com/srikumarks/steller/blob/experimental_buildsys/src/models/jsnode.js`, January 2013.

[10] O. Thereaux. Worker-based scriptprocessornode. `https://github.com/WebAudio/web-audio-api/issues/113`, October 2014.

[11] C. Wilson. A tale of two clocks - scheduling web audio with precision. `http://www.html5rocks.com/en/tutorials/audio/scheduling/`, January 2013.

[12] L. Wyse. A sound modeling and synthesis system designed for maximum usability. In *Proceedings of the International Computer Music Conference*, pages 447–451, Singapore, 2003.

[13] L. L. Wyse. Generative sound models. *Proceedings of the Multimedia Modeling 2005 Conference*, pages 370–377, 2005.