iZotope Web Audio Considerations

I. Overview

The current web audio specification is suitable for implementing small, lightweight signal processing algorithms. The use case of this API is a very important consideration when it comes to its architecture. If the intent is to provide a simple interface for implementing inexpensive, lightweight signal processing routines, it would seem the existing architecture is sufficient. If the intent is to build a robust environment that can handle the adoption of more intensive DSP, we are of the opinion that the underlying technology will impose limitations that render the implementation of intensive real-time DSP impossible. The two main focal points to bear in mind when constructing a robust real-time DSP architecture are:

1. Non-pre-emptible processing – The DSP must be allowed to process continually without interruption or dropouts will occur.
2. Worst case analysis – Average CPU consumption metrics are not important in a real-time signal processing system. Only the worst case metrics should be used to evaluate whether or not a DSP system can reliably run.

We will elaborate further on these two topics in section III of this document. Prior to this, in section II, we discuss some general modifications and things to keep in mind for the API itself. Section IV presents an example architecture that we believe would allow for a robust real-time DSP environment. Section V presents matters of intellectual property protection that will allow for high quality DSP to be developed for this platform.

II. API Considerations

As far as the API itself is concerned, the majority of the implementation seems sound. The AudioNode and AudioParam interfaces generally make a lot of sense. There are, however, a few additions and considerations for the API we'd like to point out.

1. Addition of a query and notification system on latency changes. This should provide the DSP developer and the environment with a way to remain in sync as the parameters of a DSP algorithm change. As there are several different forms of latency, we shall clarify the type of latency we are discussing here is latency from input to output in a given DSP algorithm. The amount of latency is the number of invalid initial samples that should be removed from the beginning of the DSP output.

   As an example: a given DSP algorithm may report a latency of 1024 samples. If the audio system's buffer size is 512 samples, the first 2 buffers (1024 samples) to come out of the DSP algorithm in the first buffer should be thrown away. The first valid output sample will be the first sample of the 3rd buffer. Additionally, the end of the audio stream should be extended to obtain an additional 1024

samples from the DSP. In this example, it can be done by streaming an extra 2 buffers of zeros as input to the DSP. The DSP will output the final 1024 valid samples in those 2 buffers. This behavior is considered the 'system delay' or 'latency' of a DSP algorithm.

Latency reporting is perhaps best done by way of high precision floating point seconds. Something similar to implementation used in Mac OS X Audio Units is best. For more details, see 'kAudioUnitProperty_Latency' in: https://developer.apple.com/library/mac/#documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/TheAudioUnit/TheAudioUnit.html

2. CPU Monitoring should be done on a 'worst case scenario' basis. The cpu load of signal processing algorithms is not usually constant. Ideally, DSP should be developed with an eye on keeping the CPU load even across buffers. In practice, this can be difficult for some algorithms and there may be CPU spikes during processing. Poorly designed signal processing algorithms may have extremely large CPU spikes that prevent real-time processing. To ensure that CPU load will never go above 100% and the audio system will remain glitch-free, CPU levels should be measured on a worst-case basis. To illustrate this, let's take an example algorithm that requires some audio analysis to take place at some fixed time interval (say every 8192 samples). Let's assume the audio buffer size is 1024 samples and we are trying to process in realtime. Extra processing is going to take place inside the algorithm every 8 buffers and depending on the intensity of this processing, it may well be over 100% while processing every 8th buffer. As a consequence, you will see a CPU spike every 8 buffers and an audio dropout every 8 buffers. If the analysis portion of this algorithm is very heavy on CPU load and the other processing portion is very light, you could see an average CPU load that looks as if it should work in real-time. But don't be fooled, worst-case CPU is what matters for real-time DSP.

3. Many DSP algorithms have boolean and integer parameters in addition to floating point parameters. As we can only see floating point representations of parameters in the current API draft. This can work perfectly fine, but you may want to think about how to standardize this further. Many DSP algorithms also use boolean and integer parameters. Perhaps a solution similar to the VST interface would be satisfactory. There, Steinberg forces parameters to have a range between 0.0f and 1.0f (inclusive) and other types are handled by typecast. You can find more details on the VST interface itself here: http://www.gersic.com/vstsdk/.

III. The DSP Environment

True real-time DSP implemented in a web browser could change the way musicians, gamers and media producers work in the future. In order to make the API scalable and future-proof, a highly performant environment is necessary. We feel that Javascript itself cannot provide such an environment. There are a few key issues we

have noted in our discussions on the topic and the common thread running through all these issues is the assurance that sufficient processing is available at all times. Predictable CPU availability is absolutely critical to the effectiveness of real-time digital signal processing. As stated in the overview, non-pre-emptible processing and worst case analysis are the main issues to overcome. To that end, there are some key considerations:

1. A dedicated DSP processing thread is required to ensure the audio has the highest priority possible during processing. The thread should never be interrupted due to non-DSP code.

2. The general overhead of Javascript will limit the scalability and ability of developers to implement intensive, high performance DSP.

3. Garbage collected languages are not conducive to real-time audio applications as there is no guarantee of when the collector will run. This makes it impossible to predict what the actual CPU of the system will be and to determine whether or not a signal processing routine will be able to operate in real-time.

There are many effects that have mass market appeal which we believe would not be possible under the current architecture. The T-Pain Effect is a specific example of one such effect.

The general consensus is that audio going through a Javascript layer will likely end up with problems similar to the Android audio system. As the realization for the need for more performance becomes a problem, a push to huge audio buffers will be inevitable. That reliance on large audio buffers will make latency a real problem for anyone that needs a low-latency (5-10ms) solution. Low latency audio has been impossible for developers to achieve on Android without bypassing the audio architecture completely. There is also no way to take advantage of machine optimized functions. There are large speed ups in processing that could be achieved if optimized math functions could be used. The most viable solution for this is an LLVM-based just in time solution for the browser. Something similar to the behavior in Google's Native Client SDK: http://code.google.com/p/nativeclient/

IV. An LLVM-based proposal

An LLVM-based implementation would compile, link and run on-the-fly in the browser. This would allow developers to create independent byte-code that would be deployed to a user's machine through the browser and would run in a sandboxed area. Here is the process and how this approach is advantageous:

1. Developers write DSP code in C, C++, or related languages that are not garbage collected. Which has two distinct advantages:
    - **a.** Far less overhead than Javascript. There is no doubt the code will run much faster.
    - **b.** No garbage collection. This means writing real-time code is a possibility.

2. LLVM would turn the C/C++ code written by the developer into bytecode (intermediate form). This is intermediate form is distributed by the developer for use in the application.
3. When deployed, LLVM compiles intermediate form code into optimized intermediate form for the target platform. This can take advantage of optimizations made by the LLVM compiler and will produce code that runs significantly faster than what could run with the Javascript layer.

V. Intellectual Property Protection

For browser-based audio solutions, there should be some concern over intellectual property. As this code tends to be downloaded directly to a user's computer through the web browser, how will companies ensure protection over their IP? If there is no good defense for intellectual property, it will really limit the potential of high quality DSP to be produced for the platform. We should ensure that closed-source audio effects can be made available with no concerns over intellectual property.