_ TELECOM ESCUELA TÉCNICA VLC SUPERIOR DE UPV INGENIEROS DE TELECOMUNICACIÓN

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

EFECTOS DIGITALES DE AUDIO CON WEB AUDIO API

Samuel García Chaparro

Tutor: José Javier López Monfort

Trabajo Fin de Grado presentado en la Escuela Técnica Superior de Ingenieros de Telecomunicación de la Universitat Politècnica de València, para la obtención del Título de Graduado en Ingeniería de Tecnologías y Servicios de Telecomunicación

Curso 2014-15

Valencia, 20 de mayo de 2015

Resumen

El presente trabajo consiste en un estudio de la capacidad de Web Audio API para el procesado de efectos de audio en tiempo real. De todos los efectos de audio posibles se han elegido el wah-wah, el flanger y el chorus, efectos ampliamente empleados con guitarra eléctrica. Se crean funciones en lenguaje JavaScript que modelan el comportamiento de los efectos de audio elegidos, haciéndolas funcionar sobre una plataforma web HTML5. Junto al análisis se propone el empleo de nueva terminología en el contexto del desarrollo de webapps con la librería.

Resum

El present treball consisteix en un estudi de la capacitat de Web Audio API per al processat d'efectes d'àudio en temps real. De tots els efectes d'àudio possibles s'han triat el wah-wah, el flanger i el chorus, efectes àmpliament empleats amb guitarra elèctrica. Es creen funcions en llenguatge JavaScript que modelen el comportament dels efectes d'àudio elegits, fent-les funcionar sobre una plataforma web HTML5. Junt a l'anàlisi es proposa l'ús de nova terminologia en el context del desenvolupament de webapps amb la llibreria.

Abstract

This paper is a study of the ability of Web Audio API for processing audio effects in real time. From all possible audio effects the wah-wah, flanger and chorus, widely used as electric guitar effects, have been chosen. Functions are created in JavaScript language that model the behavior of selected audio effects, making them run on a HTML5 web platform. Alongside the analysis, new terminology is proposed to employ in the context of development of webapps with the library.

Índice

Capítulo 1. Introducción

1.1 Objetivo

El presente trabajo se va a centrar en estudiar la capacidad de la librería Web Audio API y JavaScript de procesar efectos digitales de audio. Un aspecto importante que debe tenerse presente para que un efecto de audio sea viable es que debe tener una latencia reducida para utilizarse en tiempo real, si esto no sucediese se entendería que es inviable la utilización de las mencionadas herramientas en para hacer el procesamiento de efectos.

1.2 Metodología

La metodología de trabajo a seguir en el desarrollo de este trabajo se ha dividido en tres tareas diferenciadas:

1. **Documentación previa**. En la primera fase se ha procedido a aprender el lenguaje de programación necesario para el desarrollo, JavaScript, tomando la librería Web Audio API como eje. También se ha revisado la literatura sobre efectos digitales de audio.
2. **Implementación de efectos**. Se programan funciones que describan el comportamiento de los efectos haciendo uso de las herramientas obtenidas durante la documentación.

1.3 Estructura del documento

Capítulo 2. Bases para el audio en plataforma web

En este capítulo se hace un repaso de la historia del audio en los ordenadores personales, siendo el eje central las técnicas para el altavoz interno y las tarjetas de audio. Se introduce la historia del audio en la web, siendo objeto de estudio las soluciones de los navegadores web y las utilizadas por los usuarios. Se dedicará una parte a la descripción de las herramientas utilizadas.

Capítulo 3. Efectos digitales de audio

Este capítulo presenta el estudio de los efectos de audio, siguiendo la clasificación efectuada en el libro "DAFX: Digital Audio Effects" [Zol02]. Para cada tipo de efecto se realiza un estudio previo sobre el comportamiento y el modelado teórico, después se efectúa la implementación del efecto como una función en JavaScript haciendo uso de la librería Web Audio API. Por último se detallan los resultados obtenidos por cada modelo implementado.

Capítulo 4. Conclusiones y líneas futuras

Por último, se presentan una serie de conclusiones, las aportaciones realizadas con este trabajo y las líneas de continuación para un trabajo futuro.

Capítulo 2. Bases para el audio en plataforma web

2.1 Sonido para las masas

2.1.1 Breve historia del audio en ordenadores personales

La aparición de los computadores personales no estuvo ligada a la reproducción del sonido, eran máquinas dedicadas para generación de textos, resolución de cálculos tediosos y repetitivos, y renderización de imágenes monocromáticas tales como las de los videojuegos. Los primeros ordenadores incluían un pequeño altavoz incluido por el cual no eran capaces de reproducir más allá de una señal cuadrada, el *beeper*, que se mantiene en nuestros tiempos empleándose para emitir alertas de funcionamiento y/o error.

En los años ochenta se produjo la aparición de las tarjetas de sonido, que estuvo marcada por las diferencias de tecnologías de hardware entre los distintos modelos de ordenadores: IBM PC y compatibles, Atari, Commodore o Apple, por citar algunos. Las compañías de desarrollo de tarjetas de sonido, a falta de un estándar de Ley, debían decidir con qué tecnología fabricaban. Lo habitual fue crear tarjetas compatibles con IBM puesto que la mayoría de los equipos vendidos pertenecían a este colectivo.

Las aplicaciones de audio existentes en la época eran principalmente para síntesis muy básica de instrumentos, composición musical y síntesis del habla. Lo habitual era mezclar el mundo analógico de los equipos de la época con el mundo digital del ordenador personal, por ejemplo para grabaciones en cinta magnética o vinilo.

Hubo compañías que desarrollaron técnicas para reproducción de sonidos complejos basadas en la modulación por ancho de pulso (PWM), estas tecnologías no permitían la mezcla de audio de distintas fuentes o polifonía. Un ejemplo de esta aproximación al audio en computadores personales es la tecnología *RealSound*, de la empresa Access Software, utilizando audio PCM con 6 bits a través del *beeper* y sin tarjeta de expansión, patentado en 1989 y utilizado en varios videojuegos de la época.

Las soluciones para altavoces externos producían resultados mucho mejores y los fabricantes lo tomaban como solución real. Las tarjetas de audio hacían síntesis FM, permitiendo estéreo y polifonía, aunque la calidad seguía siendo notablemente sintética. Un ejemplo de esto es la *AdLib Music Synthetiser Card* de AdLib, que soportaba audio de 8 bits, pero no PCM. La empresa Creative Laboratories mejoró la tarjeta de AdLib añadiéndole un canal PCM y un puerto para conectar un teclado controlador MIDI, el resultado fue la *Sound Blaster v1.0* que se convirtió en el estándar *de facto* de las tarjetas de audio.

Los problemas de compatibilidad de hardware desaparecieron por el completo dominio de Microsoft en el mercado, que basaba su audio en el modelo de IBM, por lo que los demás fabricantes de ordenadores no tuvieron más remedio que seguirle. En los sucesivos años se fueron mejorando, se aumentó la resolución, la tasa de bits, al respuesta en frecuencia y las propiedades y características tales como distorsión armónica, relación señal a ruido, interferencia, número de canales, eficiencia energética o formato.

2.1.2 Historia del audio en la web

La *World Wide Web* (WWW), creada en 1990 y liberada en 1991 por Tim Berners-Lee, se basa en el transporte de ficheros entre ordenadores mediante redes de comunicaciones. El trabajo de interpretación de los contenidos recibidos recae sobre el cliente, quien debe decidir qué herramienta utilizar para el visionado o la escucha, habitualmente aplicaciones específicas de decodificación de multimedia.

Los lenguajes para identificación de elementos web se centran en el transporte de texto formateado. El iniciador de estos lenguajes es el *Generalized Markup Language* (GML), creado por Charles Goldfarb, Edward Mosher y Raymond Lorie –trabajadores de IBM– entre 1969 y 1970, con soporte únicamente para texto. Surgió una extensión del lenguaje que se convirtió en estándar en ficheros contenedores de hiperenlaces, el *Standard Generalized Markup Language* (SGML), que modificó la manera de escribir el lenguaje anterior a una manera más legible, nacían las etiquetas (en inglés *tags*) caracterizadas por ser una marca de texto entre (<marca>). En 1990, como siguiente evolución de SGML, aparecen los lenguajes *HyperText Markup Language* (HTML) y *Extensible Markup Language* (XML) que dominan el panorama actual.

En 1997 se lanzó la segunda versión del navegador Netscape, su planteamiento de añadir multimedia con contenido de audio y/o vídeo se produjo con la etiqueta de embebido (<embed>), usada habitualmente para insertar contenido flash. Otra cosa que incluyeron, y que fue clave en el desarrollo futuro de toda la WWW, fue el lenguaje script; para insertarlo en el documento HTML se utilizó la etiqueta de script (<script>). La primera vez que se plantea el audio puro insertado en una página web surge a mitad de 1996 con la tercera versión de Internet Explorer. Esta versión incluía una etiqueta fuera del estándar HTML llamada sonido de fondo (<bgsound>) que permitía reproducir automáticamente música de fondo al abrir la página, aunque no fue un éxito.

El script ha estado siendo durante muchos años una solución empleada aunque, si bien no es mala, no es nativa y requiere de programación para poder llevarla a cabo. Muchos scripts han aparecido como opciones a emplear para reproducción de audio. Alternativas open source y de pago, con todo tipo de funciones y de diferencias que hacían que cada script fuese adecuado para ciertos casos o tareas. Sin embargo, las aplicaciones flash han sido las que han dominado este terreno y era raro el sitio web donde no hubiese al menos una llamada al reproductor flash (Flash Player). Los hackers –no confundir con ciberterroristas–, tras centenares de investigaciones, llegaron a la conclusión de que Flash tiene demasiadas vulnerabilidades demasiado graves como para ser considerado seguro y recomendaron no utilizarlo a no ser que fuese estrictamente necesario y sólo en sitios seguros.

La comunidad de estudiosos se puso a trabajar y el resultado ha sido que en HTML5 se incluyen por primera vez las etiquetas para audio (<audio>) y vídeo (<video>), lo que resuelve el problema de la no-natividad del multimedia en un documento HTML. Sin embargo, esta etiqueta tiene grandes limitaciones que no la convierten en apta para implementar aplicaciones interactivas o juegos complejos. Algunas de esas limitaciones son: no se tiene un control preciso del tiempo, el número de sonidos reproducidos a la vez es muy bajo, no hay manera de estar seguro de que un sonido se ha pre-cargado en el buffer, no es posible aplicar efectos en tiempo real y no hay ninguna manera de analizar la señal.

La solución tenía que pasar por crear una librería de funciones con las que controlar el audio. Mozilla y Google trabajaron por su cuenta en su propia API:

- Mozilla desarrolló la *Audio Data API*. En ella se parte de la etiqueta de audio y mediante scripts en lenguaje JavaScript se ejecutan instrucciones para la gestión y control del audio.
- Google desarrolló la *Web Audio API*. En ella todo el trabajo se queda en el programa JavaScript, sin ser necesario incluir la etiqueta de audio en el documento HTML. Se tiene conexión con la librería *Web Real-Time Communications* (WebRTC).

El *World Wide Web Consortium* (W3C) creó en 2011 el *Audio Working Group* (grupo de trabajo en audio). Su trabajo consiste en decidir qué estándar se debe seguir, y decidieron seguir el producto desarrollado en Google. *Web Audio API* comenzó un proceso de *Working Draft* (borrador), en el que aún sigue, donde se tiene que escribir la especificación, por lo que debe ser probada y evaluada por la comunidad para ser mejorada.

En W3C decidieron nombrar como *Audio Processing API* a la conjunción de: *Web Audio API*, ya explicada; *Web MIDI API,* que ofrece las herramientas para la conexión de dispositivos MIDI con una página web; y *MediaStream Processing API*, creada por Mozilla y utilizada para trabajar con flujos de audio en tiempo real.

2.2 Herramientas

2.2.1 JavaScript

JavaScript es un lenguaje de programación interpretado que se utiliza para crear páginas web dinámicas. Fue desarrollado por Netscape e incluido por primera vez en la segunda versión de Netscape Navigator, en 1995. Su primer uso fue el de comprobación de formularios antes de enviarlos, muy útil dado que la velocidad de acceso a Internet de entonces no superaba los 28.8kbps. En 1997 la *European Computer Manufacturers Association* (ECMA) definió el lenguaje ECMAScript, que sería el lenguaje de script estándar multiplataforma e independiente de cualquier empresa; JavaScript no es más que la implementación que realizó la empresa Netscape del estándar.

Las características de JavaScript como lenguaje de programación interpretado deben ser muy tenidas en cuenta para la implementación. Las más importantes son:

- **Interpretado**. No necesita ser compilado previamente a la ejecución.
- **Orientado a objetos**. Es un paradigma de programación que engloba en una misma entidad, el objeto, las propiedades de estado, comportamiento e identidad.
- **Prototípico**. Los objetos no son creados mediante la instanciación de clases sino mediante la clonación de otros objetos.
- **Imperativo**. Cada instrucción indica cómo se debe realizar una tarea.
- **Débilmente tipado**. No es necesario indicar a la variable qué tipo de dato va a contener.

Uno de los puntos fuertes del lenguaje JavaScript en la actualidad es la capacidad de modificar el código HTML mediante el uso de la API *Document Object Model* (DOM).

2.2.2 Web Audio API

Web Audio API es una interfaz de programación que permite crear aplicaciones web de procesamiento y síntesis de sonido. Es un estándar aprobado por el W3C que sigue en revisión. Lo principal a tener en cuenta es que JavaScript no es código nativo, al contrario que un programa en C++, y la velocidad de ejecución del programa es inferior –un cálculo optimizado de la FFT en JavaScript en comparación a C++ se estima que es veinte veces más lento– por lo que puede no ser suficientemente eficiente para procesos pesados tales como convolución o espacialización 3D con muchas fuentes. Además, en JavaScript no se ejecuta en un hilo de procesamiento en tiempo real por tanto otros hilos que se ejecutan en el sistema pueden anticiparse en la cola de ejecución.

El paradigma empleado es el rutado por grafos; se establecen nodos, que son entidades con una función específica (oscilador, retardador, …), cada uno de los cuales con un número de entradas y salidas determinado que se van interconectando de modo que el flujo de audio va siendo procesado en cada nodo y pasando al siguiente hasta llegar al destino, la salida de audio. En la figura 1 se retrata el proceso. Se puede comprobar utilizando la *webapp Web Audio Playgound*[1].

[1] Web audio playground es una aplicación web creada por Chris Wilson, actual encargado de la edición del estándar de Web Audio API, liberada con la licencia open source del MIT. En ella se interconectan módulos con funciones específicas en el procesamiento del audio: entradas, procesamientos y salida. La última versión se puede visitar en http://webaudioplayground.appspot.com/

Todo el rutado se hace sobre una interfaz llamada contexto de audio (audioContext), que contiene la información de frecuencia de muestreo, destino, estado, tiempo actual y oyente – empleado para la espacialización 3D. El contexto contiene los prototipos para la creación de los nodos y otros controladores.

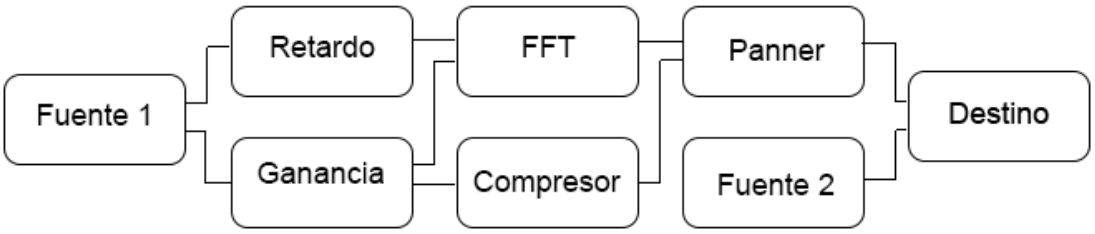

Figura 1. Ejemplo de rutado por grafos.

Los nodos y los controladores que contiene Web Audio API son los siguientes:

- AudioNode. Es la unidad de objeto, la mayor parte de nodos se encapsulan dentro de éste. Tiene la función de controlar el número de entradas y salidas, y las conexiones y desconexiones.
- AudioParam. Es la unidad de control de cada parámetro. El valor puede ser cambiado de manera inmediata o automatizar variaciones.
- AudioDestinationNode. Es el objeto de salida de audio y hay únicamente uno por cada contexto.
- GainNode. Nodo optimizado para el control de ganancia.
- DelayNode. Nodo optimizado para crear retardos.
- AudioBuffer. Es un objeto donde se almacena en memoria un clip de sonido. Se hace en coma flotante de 32 bits PCM. Se espera que el clip almacenado tenga corta duración, menos de un minuto.
- AudioBufferSourceNode. Es el nodo desde el cual se reproduce un audioBuffer. Tiene las funciones de start y stop a partir del momento elegido, y permite hacer bucles.
- MediaElementAudioSourceNode. Es el nodo que representa una fuente de audio desde un elemento de audio o video insertado en la página.
- AudioProcessEvent. Es un objeto con el que se puede procesar el audio directamente entre la entrada y la salida, o sintetizarlo directamente estableciendo sólo salidas. Se pueden definir parámetros para el manejo. Puede ser la solución más sencilla a la hora de crear un efecto o sintetizar un sonido, pero tiene problemas.
- PannerNode. Es un nodo en el que se procesa la posición y espacialización en un espacio tridimensional.
- AudioListener. Es la representación de la referencia para la espacialización, se crea con el contexto. Además de la posición, se pueden variar las constantes físicas de efecto doppler y velocidad del sonido.
- StereoPannerNode. Es similar al PannerNode, pero el algoritmo está diseñado y optimizado para una sola dimensión.
- ConvolverNode. Este nodo procesa la convolución lineal dada una respuesta al impulso desde un AudioBuffer con la entrada.
- AnalyserNode. Con este nodo se obtiene la información en el dominio del tiempo y/o frecuencia en tiempo real. Se emplea una ventana tipo Blackman de tamaño entre 32 y 32768 seleccionable para el cálculo de la FFT. Es útil para crear visualizadores.
- ChannelSplitterNode. Separa una señal de audio de N canales en N señales de audio de un solo canal.
- ChannelMergerNode. Une N señales de audio con un canal en una señal de audio con N canales.
- DynamicsCompressorNode. Es un nodo que implementa el efecto de compresión dinámica. No proporciona los efectos de expansor, limitador o puerta de ruido. Tiene los controles habituales de envolvente de este tipo de efectos.

- BiquadFilterNode. Es el nodo que procesa filtros. Todos los filtros incluidos son de segundo orden. Se incluyen los filtros: paso bajo, paso alto, paso banda, *shelving low*, *shelving high*, resonador, *notch* y paso todo.
- WaveShaperNode. Es el nodo capaz de implementar efectos de distorsión no lineal. Tiene la capacidad de hacer *oversampling* para aplicar el efecto.
- OscilatorNode. Es el nodo que representa una fuente generadora de señales periódicas. Las formas de onda son: sinusoidal, cuadrada, sierra, triángulo y personalizado.
- PeriodicWave. Es un control para crear una forma de onda personalizada.
- MediaStreamAudioSourceNode. Es el nodo que representa una fuente de audio recibida desde un *MediaStream*[2]. Esto abre la posibilidad al empleo del estándar *Web Real-Time Communications between browsers* (WebRTC) para el envío de audio en tiempo real.
- MediaStreamAudioDestinationNode. Es el nodo que representa la salida de audio mediante un *MediaStream*.

Una de las opciones más interesantes es la posibilidad de conectar la salida de un nodo a un parámetro de otro nodo, o incluso a un parámetro del mismo nodo –el sistema pasaría a ser inestable en la mayor parte de los casos.

Debido a los cambios introducidos por nuevas versiones de la librería y a la adaptación de la librería en los diferentes navegadores, los programas antiguos pueden ser incompatibles con la especificación actual o los navegadores no haber implementado las actualizaciones de la librería. Para solucionar estas posibles incompatibilidades Chris Wilson, el actual encargado del estándar, ha creado un parche llamado AudioContextMonkeyPatch[3], que consiste en una librería javascript intermediaria que resuelve problemas de versiones y de soporte de navegadores.

El desarrollo de los efectos con esta librería se puede realizar de tres modos:

- *Unión*. Consiste en interconectar módulos y parámetros, de los ya existentes, hasta conseguir el efecto deseado. Es una solución sencilla para efectos simples o para los casos en los que no se necesiten funciones con funciones específicas no incluidas en la librería.
- *Generación*. En este caso se programa una función que será ejecutada con el módulo *AudioProcessEvent* que será la que realice el efecto.
- *Híbrido*. También es posible desarrollarlos empleando los bloques existentes en la librería y bloques programados; esta es, posiblemente, la manera más habitual y ventajosa.

2.2.3 Librerías de audio independientes

Dado que la W3C no ha sido capaz de solucionar con presteza el reto de la librería de audio, han ido apareciendo diversas librerías creadas por usuarios para satisfacer las necesidades que se tienen de forma general. Cada desarrollador ha utilizado su propia técnica y ha añadido las funciones que ha visto convenientes para su objetivo, por tanto no se pueden considerar librerías para uso general y, por ende, no pueden convertirse en un estándar. Todos los casos escogidos comparten la misma idea: usar JavaScript para gestionar y controlar la reproducción del audio.

- Audio.js[4], creado por Anthony Kolber en 2010. Su filosofía es aprovechar la etiqueta audio si está disponible y, en caso contrario, utilizar un reproductor de audio flash. Las funciones disponibles son escasas, de manera que su utilización queda relegada a un uso como reproductor de listas de reproducción.
- SoundJS[5], creado por Grant Skinner. Es la solución más sencilla para la reproducción inmediata de un sonido, la única función implementada es el play, sin embargo su fortaleza radica en que es multiplataforma y funciona en navegadores no actualizados.

---

[2] El interfaz MediaStream está descrito en el estándar Media Stream API.
[3] Se puede visitar en https://github.com/cwilso/AudioContext-MonkeyPatch (vis. 21 febrero 2015)

[4] La librería se puede visitar en la web: http://kolber.github.io/audiojs/ (vis. 4 marzo 2015)
[5] La librería se puede visitar en la web: http://www.createjs.com/SoundJS (vis. 4 marzo 2015)

- SoundManager 2[6], creado por Scott Schiller. Esta ya es una API de manejo de ficheros de audio completa. Contiene eventos y las funciones habituales de control: play, pause, stop, resume y selección de posición, entre otras.
- Flocking[7], creado por Colin Clark y Adam Tindale. La idea subyacente del proyecto es la generación de sonidos mediante síntesis mediante diferentes técnicas.
- Audiolib.js[8], ceado por Jussi Kalliokoski. Es una aproximación muy cercana a lo que es Web Audio API. Tiene capacidad para síntesis y procesado de audio y efectos. Incluye análisis de FFT, generador de ruido, secuenciador, envolvente y otros.

[6] La librería se puede visitar en la web: http://www.schillmania.com/projects/soundmanager2/ (vis. 4 marzo 2015)
[7] La librería se puede visitar en la web: http://flockingjs.org/ (vis. 4 marzo 2015)
[8] La librería se puede visitar en la web: https://github.com/jussi-kalliokoski/audiolib.js (vis. 10 abril 2015)

Capítulo 3. Efectos digitales de audio

El estudio de la utilización del procesamiento en señales de audio digital tiene, *grosso modo,* dos vertientes: los efectos de audio, con finalidades artísticas; y la codificación de audio, con finalidades técnicas.

"*Un efecto de audio es cualquier modificación que se efectúa sobre una señal de audio que provoca un cambio en la percepción del sonido. Los cambios introducidos por los efectos son desde muy sutiles […] hasta cambiar totalmente el sonido original […].".* [Dux12]

En el empleo de un efecto intervienen tres partes: algorítmica del efecto, parámetros de control y feedback audiovisual.

- **Algorítmica**

Es el fundamento matemático de un efecto. Dado un estado inicial de los parámetros y una entrada, siguiendo los pasos determinados sobre los que no cabe duda, se llega a un estado final y se obtiene una solución. Desde el punto de vista del algoritmo existen efectos simples, por ejemplo el retardo, y complejos, por ejemplo el autotune. La aparición de los DSP y el aumento de la potencia de cálculo de los mismos ha sido uno de los factores importantes en el empleo de efectos digitales frente a analógicos.

- **Parametrización**

Un efecto sin parámetros que el usuario sea capaz de controlar es prácticamente inútil, se pierde una parte fundamental del aspecto artístico del efecto y anula la capacidad de utilizarlo en múltiples escenarios. Los valores que toma un parámetro nunca son triviales y son objeto de estudio a la hora de crear y aplicar un efecto. La interfaz del efecto es muy importante, en el mundo del software se ha optado habitualmente por mantener más o menos la manera visual e intuitiva que tenían los efectos analógicos en los racks; si bien es cierto que las posibilidades son mucho mayores y muchos fabricantes quieren ofrecer tantas opciones que en un formato rack no es posible insertarlas, por ejemplo el Amp Designer incluido en Logic Pro, donde en una parte aparecen los controles de un amplificador tal cual son en la realidad y en la otra una representación gráfica de la pantalla de altavoces y la posición a la que se coloca un micrófono respecto a la misma.

- **Realimentación**

Esta es la parte en la que el usuario debe actuar. Dado que un efecto es algo artístico, es el artista quien debe controlarlo y adecuarlo. Un efecto, por sí mismo, no es nada. La finalidad del sonido procesado es ser oída, por tanto alguien debe decidir el grado de virtud del sonido procesado y modelarlo a las necesidades. En este punto entran en juego dos sentidos: la vista y el oído. El usuario tiene a su disposición visualizadores, que pueden ser tan simples como una luz parpadeando para marcar una saturación o tan complejos como espectrogramas de alta resolución. El usuario debe oír el efecto y cerciorarse de que el resultado es el que pretende con la utilización del mismo y, si oye algún malfuncionamiento, actuar en consecuencia.

De las muchas posibles clasificaciones de los efectos, se ha elegido la que utilizan Udo Zölzer *et alia* en el libro *DAFX: Digital Audio Effects* (en la bibliografía denotado como [Zol02]). Esta decisión se fundamenta en que la clasificación se realiza por la configuración en base a la cual se realiza la construcción del efecto, no por su finalidad de uso o características sonoras percibidas. Se dan casos de efectos que pueden ser generados mediante procesos con bases distintas, son los casos de *vibrato*, *chorus*, *flanger*, *phaser* y otros pocos.

3.1 Filtros

3.1.1 Descripción

Filtrar es seleccionar objetos con ciertas características de un conjunto mayor de objetos. Traducido al lenguaje de audio digital, seleccionar frecuencias determinadas de entre todo el espectro de frecuencias existente. Los filtros son objeto de estudio en electrónica desde casi los comienzos de la misma y actualmente se tiene un gran conocimiento de los mismos.

Atendiendo a la respuesta en frecuencia hay cinco filtros fundamentales, a partir de los cuales se pueden obtener el resto de tipos de filtros:

- Paso bajo
- Paso alto
- Paso todo
- Oscilador
- Filtro peine (*comb*)

El resto de filtros habituales, combinación de los anteriores, son:

- Paso banda
- Elimina banda
- Rechaza banda (*notch*)
- Resonador (*peaking*)

La estructura de cada tipo de filtro no es objeto de estudio del presente TFG, *ergo* se obvia.

Este tipo de efectos son fáciles de comprender puesto que estamos habituados en nuestro entorno a encontrar filtros en cada rincón producidos por objetos materiales, *exempli gratia*, al abrir y cerrar una puerta o ventana, al cubrirnos las orejas con las manos o al alejarnos de una fuente sonora.

El efecto más común basado en filtros son los ecualizadores, mantienen una cierta cantidad de filtros con los que se genera una respuesta en frecuencia determinada. Si le añadimos la dimensión temporal, aparecen efectos como son: el wah-wah, a partir de un filtro paso banda; el phaser, a partir de filtros paso todo; el flanger, a partir de un filtro peine; o, incluso, ecualizadores variables con el tiempo.

3.1.2 Implementación de wah-wah

El efecto de wah-wah surge del empleo de la sordina 'desatascador' (*plunger mute*) en trombón y trompeta, aparece durante los años 20 y se usa principalmente en el jazz. En los 60 se extendió a instrumentos electrónicos como la guitarra o el órgano, para su manejo se desarrolló un encapsulado específico, el pedal wah-wah –más tarde se conoce genéricamente como pedal de efectos móvil, en contraposición a los efectos estáticos de on/off, puesto que contiene un elemento móvil que controla un parámetro.

La estructura del efecto se muestra en la figura 2. Se emplea un filtro paso banda, con cuya salida se hace una suma ponderada con la señal de entrada –habitualmente retardada para compensar la fase y que no se produzcan efectos indeseados. En alguna literatura se indica que el filtro usado es paso alto y en otra se indica que es paso bajo, sin embargo, puede ser por cuestión de gustos; la implementación tendrá en cuenta estos supuestos.

Es posible automatizar la variación de la frecuencia de corte del filtro utilizando un LFO modulado, este efecto se conoce como auto-wah. Si el LFO no se modula lo que se obtiene es el efecto de trémolo-wah.

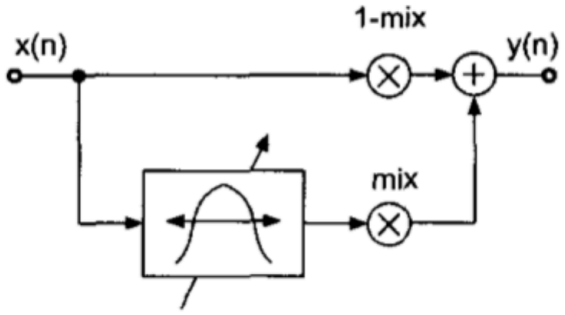

Figura 2. Estructura del efecto wah-wah [Zol02].

- **Parámetros**

Filtro:

- Factor de calidad, Q.
- Frecuencia de corte, fc.
- Tipo de filtro.

Genérico:

- Mezcla del efecto, mix.
- Retardo, r.

- **Algoritmo**

La ecuación que modela el comportamiento del wah-wah es:

$$\boldsymbol{y(n) = mix \cdot \big(x(n) * h(n)\big) + (1 - mix) \cdot x(n + r)} \qquad \textbf{(3.1)}$$

siendo '*h(n)*' la respuesta al impulso del filtro y el operador '$*$' la convolución.

El diagrama de bloques utilizado para programar se ve en la figura 3. El retardo compensador es opcional y las ganancias finales están relacionadas de manera lineal. El código javascript que modela el comportamiento se encuentra en el anexo 2.

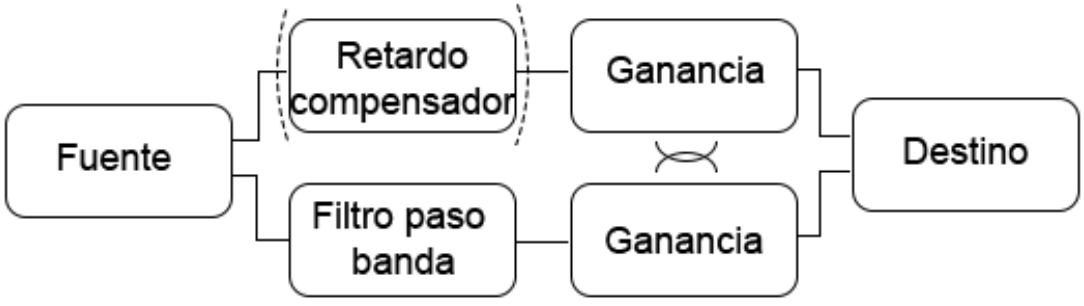

Figura 3. Algoritmo de Wah-Wah con Web Audio API.

3.2 Retardos

3.2.1 Descripción

El retardo es un parámetro en física que determina la diferencia de tiempo entre dos sucesos sincronizados. Que haya sincronía indica que debe haber un instante de referencia, se toma el tiempo de uno de los sucesos y, con el signo, se indica si el segundo es un suceso adelantado o retrasado.

$$\boldsymbol{retardo_t = t(suceso\ 1) - t(suceso\ 2)} \quad \textbf{(3.2)}$$

El sonido, por su naturaleza ondulatoria, posee una velocidad, $v=e/t$, dependiente del medio por el que se propaga y de las condiciones del mismo. De esta manera se puede establecer una analogía entre espacio y tiempo, se dice que el retardo es la diferencia de caminos entre dos sucesos.

$$\boldsymbol{retardo_t = [e(suceso\ 1) - e(suceso\ 2)]/v} \quad \textbf{(3.3)}$$

$$\boldsymbol{retardo_t = retardo_e/v} \quad \textbf{(3.4)}$$

Considerando en (3.3) que la velocidad es invariante en un medio en las condiciones de experimentación, se puede decir que el retardo temporal y el retardo espacial tienen una relación lineal. Teniendo esto en cuenta, la terminología posterior podrá hablar indistintamente de retardo espacial o temporal.

Hay varios efectos interesantes que se pueden conseguir utilizando retardos, entre ellos la reverberación y el filtro peine.

Reverberación es el tiempo que necesario para que se reduzca sesenta decibelios el nivel de presión acústica en una sala cerrada. Los estudios de Sabine, Eyring y Millington se centraron en la estimación de este tiempo. La idea de la que todos partieron es que un rayo sonoro[9] en el aire cambia de medio hacia una superficie sólida en la que se producen los efectos de absorción y reflexión. Esta reflexión, llamada rebote, se produce en cada pared hasta que se absorbe toda la energía que porta la onda. El efecto de reverberación más básico está basado en múltiples líneas de retardo con amplitudes distintas para conformarlo. Actualmente es muy habitual utilizar la convolución de la señal con la respuesta al impulso de una sala, el resultado es mucho más natural.

El filtro peine es un caso interesante de estudio porque, pese a ser un filtro, el modelado del mismo es más sencillo utilizando una única línea de retardo . La forma del filtro es similar a una sucesión de filtros notch en la frecuencia seleccionada y en sus armónicos. Experimentalmente: utilizándose dos altavoces iguales a los que se les inyecta la misma señal, al desplazar en un eje uno de los altavoces se produce una variación de la diferencia de caminos entre las señales, produciéndose cancelaciones en las frecuencias múltiplo impar de $\lambda/2$ y refuerzo de 6dB en las frecuencias múltiplo par de $\lambda/2$.

3.2.2 Implementación de flanger

El efecto de flanger tiene como origen las grabaciones en cinta magnetofónica donde se mezclaba la grabación original con una copia de la grabación que se iba retrasando y adelantando levemente[10]. Al hacerlo aparece una diferencia de tiempos entre ambas señales que produce un filtro peine con frecuencia fundamental dependiente del retardo temporal, la frecuencia se puede calcular con la ecuación siguiente.

$$\boldsymbol{f = 2/retardo_t} \quad \textbf{(3.5)}$$

La explicación técnica del efecto es la variación temporal de la frecuencia central de un filtro peine. Cuando el peine tiene una frecuencia central muy alta, el efecto es escaso puesto que no aparecen los armónicos del filtro, pero al reducir la frecuencia central van apareciendo más y

[9] El rayo sonoro hace referencia a la estela que dejaría un diferencial de superficie de la esfera de emisión en su recorrido entre dos puntos cualesquiera.

[10] Un ejemplo se puede ver en http://www.youtube.com/watch?v=iSOf5uQg9nc (vis. 11 abril 2015)

más armónicos que cancelan las respectivas frecuencias. Este efecto tiene poco sentido si no hay variación de la frecuencia de muestreo puesto que la ausencia de variación sigue proporcionando un sonido natural; esto es fácil de probar puesto que sucede en un sistema estereofónico cuando la distancia de un oyente a un altavoz es distinta que la distancia del oyente al otro altavoz, la ausencia de ciertas frecuencias no es percibida a no ser que las distancias varíen.

- **Parámetros**

Retardo:

- Retardo, delay.

LFO:

- Frecuencia, speed.
- Ganancia, depth.

Genérico:

- Mezcla del efecto, mix.
- Feedback.

- **Algoritmo**

La ecuación que modela el comportamiento del flanger es:

$$\boldsymbol{y(n) = mix \cdot x(n) + (1 - mix) \cdot (y(n-1) + x(n - delay(depth \cdot speed(n))))} \quad \textbf{(3.6)}$$

siendo '*speed(n)*' la onda generada por el oscilador de baja frecuencia.

El diagrama de bloques utilizado para programar se ve en la figura 5. La señal del LFO se conecta con el parámetro *delayTime* del módulo de retardo y las ganancias finales están relacionadas de manera lineal. El código javascript que modela el comportamiento se encuentra en el anexo 3.

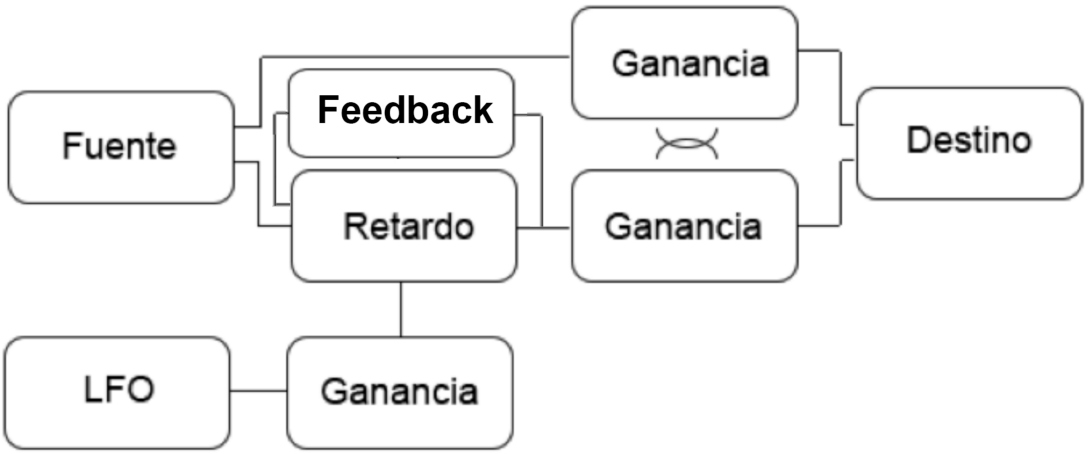

Figura 4. Algoritmo de Flanger con Web Audio API.

3.3 Moduladores y demoduladores

3.3.1 Descripción

La modulación es el proceso de variar una característica (amplitud, frecuencia y fase) de una onda portadora de acuerdo con la señal que transporta información. En telecomunicaciones consiste en desplazar el espectro de una señal a otra banda de frecuencias, para eso se requiere utilizar una frecuencia mucho mayor al ancho de banda de la señal. En el procesado de audio se distinguen dos tramos de frecuencia en los que el funcionamiento de una modulación varía:

- Por debajo de 20 Hz, el espectro muestreado no audible, las modulaciones producen variaciones en la señal moduladora sin que la señal portadora sea audible; esto lo hace perfecto para ser usado en efectos. Al oscilador que genera una señal portadora con esta característica se le denomina Oscilador de Baja Frecuencia (LFO).
- Por encima de 20 Hz, en este caso la señal portadora es audible y por tanto no es interesante para efectos. La modulación con señales en este rango de frecuencias se suele reservar para la síntesis de audio por modulación.

Lo interesante del audio es que no sólo se puede modular la señal de audio, sino cada uno de los parámetros de los efectos que se le apliquen a la señal.

3.3.2 Implementación de chorus

El efecto chorus pretende enriquecer el sonido emulando la suma de varias voces tocando exactamente lo mismo, como sucede en las agrupaciones vocales o coros. Para provocar el efecto se suman la señal original con la señal original modificada, aplicando un retardo y, en ocasiones, desplazando ligeramente el tono.

Uno de los parámetros del chorus es el número de voces, a más voces, más rico el sonido pero más coste computacional; una manera de reducir el coste emulando varias voces es modular el retardo. Lo habitual para este caso es emplear una señal sinusoidal, sin embargo es posible utilizar ruido blanco de ancho de banda menor a 20Hz que produce un retardo múltiple con un único módulo de retardo.

- **Parámetros**

Retardo:

- Retardo, delay.

LFO:

- Frecuencia, speed.
- Ganancia, depth.

Genérico:

- Mezcla del efecto, mix.

- **Algoritmo**

La ecuación que modela el comportamiento del chorus es:

$$\boldsymbol{y(n) = mix \cdot x(n) + (1 - mix) \cdot x(n - delay + depth \cdot speed\,(n))} \qquad \textbf{(3.7)}$$

siendo '*speed(n)*' la onda generada por el oscilador de baja frecuencia.

El diagrama de bloques utilizado para programar se ve en la figura 5. La señal del LFO se conecta con el parámetro *delayTime* del módulo de retardo y las ganancias finales están relacionadas de manera lineal. El código javascript que modela el comportamiento se encuentra en el anexo 4.

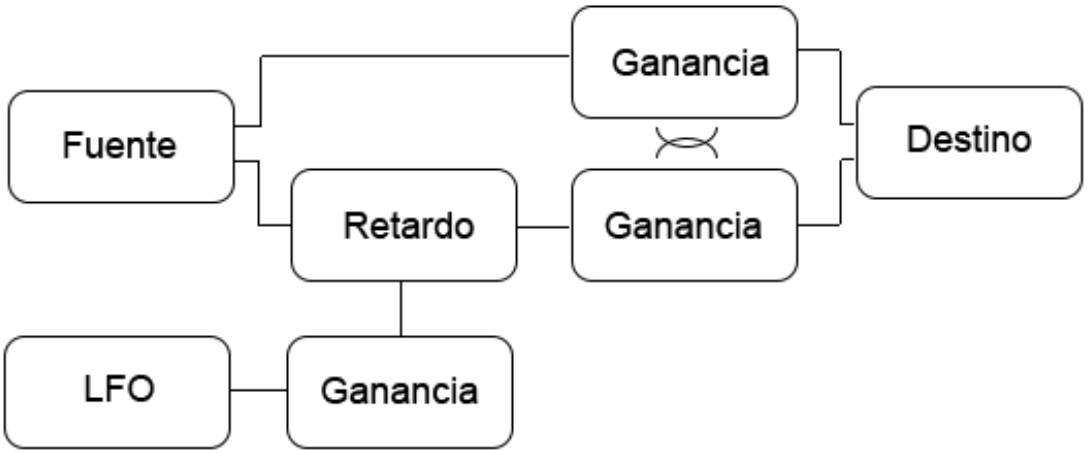

Figura 5. Algoritmo de Chorus con Web Audio API.

Capítulo 4. Conclusiones y líneas futuras

En el presente trabajo se han desarrollado una serie de algoritmos de efectos digitales de audio con Web Audio API. De todos los posibles efectos se han implementado algunos de los más utilizados en guitarra que son el wah-wah, el flanger, y el chorus.

La principal restricción que se ha tenido en cuenta ha sido que dichos efectos deben funcionar en tiempo real. Se toma la entrada predeterminada de micrófono del sistema y la salida predeterminada de altavoces para la captación y reproducción de los sonidos, de modo que es el usuario el que debe mantener los dispositivos lo suficientemente alejados para evitar el efecto de retroalimentación.

El método de implementación empleado ha sido por unión de bloques para todos los efectos, permitiendo analizar las características propias de la librería y liberar los juicios de la posible programación de funciones a nivel de buffer.

4.1 Conclusiones de Web Audio API

La idea de una API de audio 'estandarizada' es muy buena, ya existía un motor de gráficos estandarizado para web, OpenGL, que tiene buena aceptación y resultados muy aceptables. Como ésta está más fresca y en fase de *working draft*, es difícil prever la aceptación que tendrá. De momento me aventuro a decir que es un buen punto de partida, con algunas cuestiones que mejorar pero con varios puntos fuertes. En mi experiencia durante la ejecución de este trabajo he llegado a las siguientes conclusiones dignas de mención:

- El módulo de retardo no está pensado para retardos muy cortos, del orden de muestras. Funciona por buffers, el retardo mínimo es de un buffer, 128 muestras. Cuando se requieren retardos del orden de segundos funciona bien, puesto que unas pocas muestras no son relevantes. Sin embargo, para efectos en los que una pequeña variación puede ser determinante, por ejemplo el flanger, sus propiedades son insuficientes.

- JavaScript es un lenguaje que utiliza el reloj del sistema, pero no produce aplicaciones prioritarias en las colas de ejecución. Esto causa retrasos en la ejecución si el procesador está ocupado. Las aplicaciones que usen ritmos, por ejemplo, no producen buenos resultados con las funciones propias de JavaScript. Para solucionarlo se decidió que el *audioContext* manejase su propio reloj, pero la única función existente en relación a este reloj es para conocer el tiempo. Solucionar estos casos merece una solución, por ejemplo una función de *timeout* tal y como la propia de JavaScript pero utilizando el reloj propio

- Cuando se utilizan entradas externas, por ejemplo de micrófono, se necesita hacer uso de otra API. Dado que esta otra API sí está estandarizada sería muy buena idea que existiese un método para utilizarla en vez de tener que escribir más de 10 líneas de código en un proceso engorroso y poco claro para el programador.

- Existe un módulo oscilador donde se pueden generar señales con varias formas de onda posibles. Sin embargo, me parece un error que no exista un módulo generador de ruido de varios tipos.

- El motor de convolución es muy bueno, funciona genial y no tengo ninguna queja acerca del mismo con convoluciones simples, aunque haré una puntualización acerca del comportamiento del mismo. Cuando se cambia la respuesta al impulso se produce un salto brusco en vez de generar una salida donde, durante un tiempo, se haga la convolución de las dos respuestas al impulso. Esto hace que no se pueda utilizar el motor para un módulo emulador de reverberación de salas y poder oír los cambios de manera instantánea. Podría ser buena idea añadir un booleano en el módulo de la API para estos casos.

4.2 Líneas futuras de investigación

Posibles líneas de investigación para continuar con el tema que compete al presente trabajo son:

- Analizar el comportamiento de efectos desarrollados por generación (ver apdo. 2.2.2). Conociendo cómo se comportan tanto los efectos desarrollados por unión como por generación, es posible predecir el comportamiento de los híbridos.

- Crear una plataforma, a modo de web audio playground, donde poder incluir los efectos, concatenarlos y jugar con ellos.

- Una de las aplicaciones para la cual la API está pensada es para el entorno sonoro de videojuegos, podría crearse un juego donde se utilice esta API como controlador de sonido junto a un motor gráfico.

Capítulo 5. Bibliografía

[Iwa] Iwaki, T. “HTML-lint” http://www.letre.co.jp/~iwaki/htmllint/tagslist.html [Online].

[Moz] Mozilla Corp., “Audio Data API” https://wiki.mozilla.org/Audio_Data_API[Online].

[Smu13] Smus, B. *Web Audio API*. O’Reilly, 2013.

[WAu] W3C Audio Group “Web Audio API” Specification, Editor’s Draft 3 march 2015. http://webaudio.github.io/web-audio-api/ [online]

[Dux12] Duxans, H.; Costa-jussà, M. R. “Procesamiento de audio, módulo 5: Efectos digitales de la señal de audio” FUOC, Fundación para la Universitat Oberta de Catalunya, 2012.

[Zol02] Zölzer, U. *DAFX: Digital Audio Effects*. Wiley, 2002.

[Egu] Eguiluz, J. *Introducción a JavaScript*. LibrosWeb. http://librosweb.es/libro/javascript/

Capítulo 6. Anexos

6.1 Anexo 1: Código genérico

```
window.AudioContext = window.AudioContext || window.webkitAudioContext;
var audioContext = new AudioContext();
var audioInput = null,
   realAudioInput = null;

function convertToMono( input ) {
   var splitter = audioContext.createChannelSplitter(2);
   var merger = audioContext.createChannelMerger(2);

   input.connect( splitter );
   splitter.connect( merger, 0, 0 );
   splitter.connect( merger, 0, 1 );
   return merger;
}

function initAudio() {
   if (!navigator.getUserMedia)
      navigator.getUserMedia = navigator.webkitGetUserMedia || navigator.mozGetUserMedia;

   navigator.getUserMedia(
      {
         "audio": {
            "mandatory": {
               "googEchoCancellation": "false",
               "googAutoGainControl": "false",
               "googNoiseSuppression": "false",
               "googHighpassFilter": "false"
            },
            "optional": []
         }
      }, gotStream, function(e) {
         alert('Error getting audio');
         console.log(e);
      });
}
window.addEventListener('load', initAudio );
```

6.2 Anexo 2: Wah-Wah

```
// Modulos del efecto
var split = null;
var filter= null;
var delay = null;
var gain1 = null;
var gain2 = null;

// Parámetros del efecto
var paramQ=1;
var paramFc=300;
var paramTipo="bandpass";
var paramMix=1;
var paramRet=2/audioContext.sampleRate;

function gotStream(stream) {

    realAudioInput = audioContext.createMediaStreamSource(stream);
    audioInput = convertToMono(realAudioInput);
    split=audioContext.createChannelSplitter(2);

// Aplicación del efecto
    filter=audioContext.createBiquadFilter();
    filter.Q.value=paramQ;
    filter.frequency.value=paramFc;
    filter.type=paramTipo;

    delay=audioContext.createDelay();
    delay.delayTime=paramRet;

    gain1=audioContext.createGain();
    gain2=audioContext.createGain();
    gain1.gain.value=paramMix;
    gain2.gain.value=1-paramMix;


//Conexiones
    audioInput.connect(split);

    split.connect(filter,0);
    split.connect(delay,1);

    filter.connect(gain1);
    delay.connect(gain2);

    gain1.connect(audioContext.destination);
    gain2.connect(audioContext.destination);
}
```

6.3 Anexo 3: Flanger

```
// Modulos del efecto
var split = null,
    delay = null,
    osc= null,
    gain=null,
    gainFB=null,
    gain1 = null, //for wet
    gain2 = null; //for dry

 // Parámetros del efecto
var paramRet=0.005; //seconds
var paramFrecLFO=0.25;
var paramTypeLFO="sine";
var paramGainLFO=0.001;
var paramFB=0.5;
var paramMix=0.5;

function gotStream(stream) {

    realAudioInput = audioContext.createMediaStreamSource(stream);
    audioInput = convertToMono(realAudioInput);
    split=audioContext.createChannelSplitter(2);

// Aplicación del efecto
    delay=audioContext.createDelay();
    delay.delayTime=paramRet;

    osc=audioContext.createOscillator();
    osc.type=paramTypeLFO;
    osc.frequency.value=paramFrecLFO;

    gain=audioContext.createGain();
    gain.gain.value=paramGainLFO;

    gainFB=audioContext.createGain();
    gainFB.gain.value=paramFB;

    gain1=audioContext.createGain();
    gain2=audioContext.createGain();
    gain1.gain.value=paramMix;
    gain2.gain.value=1-paramMix;

//Conexiones
    audioInput.connect(split);

    split.connect(delay,0);
    split.connect(gain2);

    delay.connect(gain1);
    delay.connect(gainFB);

    osc.connect(gain);
    gain.connect(delay.delayTime);

    gainFB.connect(delay);

    gain1.connect(audioContext.destination);
    gain2.connect(audioContext.destination);

    osc.start(0);

}
```

6.4 Anexo 4: Chorus

```
// Modulos del efecto
var split = null,
    delay = null,
    osc= null,
    gain=null,
    gain1 = null, //for wet
    gain2 = null; //for dry
var whiteNoise=null,
    filtro=null;

 // Parámetros del efecto
var paramRet=0.005; //seconds
var paramFrecLFO=2;
var paramTypeLFO="sine";
var paramGainLFO=0.001;
var paramMix=0.5;

function gotStream(stream) {
    realAudioInput = audioContext.createMediaStreamSource(stream);
    audioInput = convertToMono(realAudioInput);
    split=audioContext.createChannelSplitter(2);

// Aplicación del efecto
    delay=audioContext.createDelay();
    delay.delayTime=paramRet;

    osc=audioContext.createOscillator();
    osc.type=paramTypeLFO;
    osc.frequency.value=paramFrecLFO;

    gain=audioContext.createGain();
    gain.gain.value=paramGainLFO;

    gain1=audioContext.createGain();
    gain2=audioContext.createGain();
    gain1.gain.value=paramMix;
    gain2.gain.value=1-paramMix;

    whiteNoise = audioContext.createBufferSource();
    whiteNoise.buffer = createWhiteNoise();
    whiteNoise.loop = true;
    whiteNoise.start(0);

    filtro=audioContext.createBiquadFilter();
    filtro.type="lowpass";
    filtro.frequency.value=15;
    filtro.Q.value=2;

//Conexiones
    audioInput.connect(split);

    split.connect(delay,0);
    split.connect(gain2);

    delay.connect(gain1);

    osc.connect(gain);
    gain.connect(delay.delayTime);

    gain1.connect(audioContext.destination);
    gain2.connect(audioContext.destination);

    osc.start(0);
}
```