Sintetizador Básico en Javascript con Pico.js

Nikita Tchayka Razumov

Junio de 2015

Contenidos

1 Introducción

El objetivo de este trabajo era realizar un sintetizador digital con la posibilidad de utilizarlo como un plugin VST, de modo que se pudiese utilizar en programas de producción musical profesionales como FL Studio, Ableton Live, Apple Logic, etc...

Por motivos de tiempo y de dificultades con las diferentes librerias que permiten esta funcionalidad se ha optado a realizar un sintetizador simple en Javascript con la ayuda de la libreria Pico.js.

Además se ha intentado también realizar un sistema de control de lazo cerrado retroalimentando la salida de audio hacia la entrada con otros lenguajes, produciendose acoplamiento o silencio, dependiendo de que libreria se utilizase.

Pico.js no permite la grabacion de sonido externo, permitiendo tan solo la síntesis de este.

2 Descripción del trabajo realizado

Se ha realizado una interfaz de usuario básica con Twitter Bootstrap 3 y Bootstrap Slider.

Se basa en un "panel de mandos" con opciones para elegir:

- Forma de onda
- Amplitud de onda (Volumen)
- Frecuencia de corte de un filtro-paso-bajo
- Umbral de distorsión

También un botón para reproducir una onda con los parámetros seleccionados arriba a 440 Hz (equivalente al Do central en un piano o también C3 en la notación musical anglosajona).

2.1 Generación de las ondas

Para generar las diferentes ondas se han construido cuatro funciones con una estructura equivalente:

```
var phase = 0;
var phaseIncr = freq * PI2 / Pico.sampleRate;
var out = e.buffers;
for (var i = 0; i < e.bufferSize; i++) {
    var output = ...; // Calculo de la muestra segun onda
    out[0][i] = out[1][i] =
        process(output, cutoff, threshold) * volume;
    phase += phaseIncr;
    while (phase >= PI2) {
        phase -= PI2;
    }
}
```

En este codigo se hacen varias cosas:

La primera, se obtiene una referencia a los buffers de salida. Luego, se calcula la muestra actual dependiendo de la forma de la onda seleccionada. Según se van recorriendo los buffers se van introduciendo las muestras procesadas multiplicadas por la amplitud seleccionada.

Posteriormente, se incrementa la fase y, en caso de que sobrepase 2π , se le resta esta cantidad.

Este proceso sigue el mismo "esqueleto" con todas las formas de onda, cambiando tan solo el calculo de la muestra de salida.

Para las diferentes ondas tenemos las siguientes ecuaciones:

- Seno:
$$salida = sen(fase)$$
- Cuadrada:
$$salida = \begin{cases} 1 & \text{si } fase \leq \pi \\ -1 & \text{si } fase > \pi \end{cases}$$
- Diente de sierra:
$$salida = 1 - \frac{fase}{\pi}$$
- Triangular:
$$salida = 2\times \mid 1 - \frac{fase}{\pi} - 0,5 \mid$$

Implementar estas ecuaciones en código es algo trivial.

2.2 Procesamiento de señal

A la hora de procesar la salida de audio lo que se hace es pasar la muestra a una funcion **process** que se encarga de aplicarle un efecto de distorsión con un cierto umbral y luego un filtro-paso-bajo:

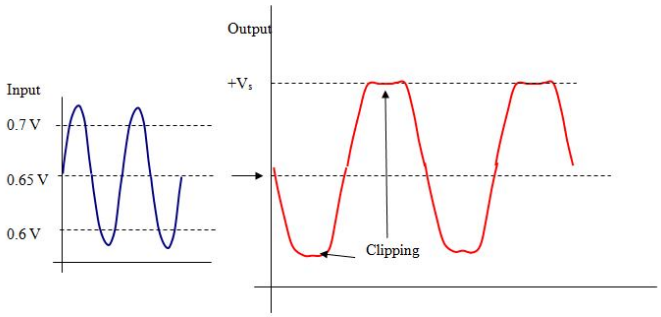
```
function process(value, cutoff, threshold) {
    return filter(distort(value, threshold), cutoff);
}
```

Un detalle interesante es como se podria resolver esto en un lenguaje funcional con una **composición de funciones**, por ejemplo en *Haskell* bastaría con hacer esta funcion de la manera:

```
process = filter . distort
```

2.2.1 Distorsión

Para hacer el efecto de distorsion simplemente tenemos que hacer que la señal se sature al pasar el umbral establecido.



Con el siguiente código lo conseguimos:

```
var output;
if (value >= 0) output = Math.min(value, threshold);
else output = Math.max(value, -threshold);
return output / threshold;
```

Simplemente, al pasar el umbral, "forzamos" a la muestra al valor del umbral.

Posteriormente, multiplicamos la muestra de salida por el valor del umbral inverso para que no disminuya la amplitud con respecto al umbral y se mantenga constante.

2.2.2 Filtro-paso-bajo

En este caso, la verdad que un filtro no tiene mucho sentido, ya que no es que hayan muchas frecuencias altas en un oscilador simple, pero sin embargo nos sirve para recortar algunas frecuencias de la serie armónica de ondas como la de diente de sierra.

Se implementaran dos *buffers* que utilizaremos para el filtro y posteriormente se implementara la función filter:

```
filterBuffer1 += cutoff * (value - filterBuffer1);
filterBuffer2 += cutoff * (filterBuffer1 - filterBuffer2);
return filterBuffer2;
```

En el caso de querer implementar un filtro paso alto, simplemente devolveríamos el *buffer* número uno.

3 Conclusión

Implementar sistemas de procesamiento de audio, o incluso de síntesis es una tarea bastante sencilla, ampliable casi hasta el infinito, o al menos, hasta donde nos permita la capacidad de procesamiento de las CPUs de hoy en día.

A este proyecto se podrían añadir muchisimas mas cosas, como por ejemplo n osciladores concurrentes, un oscilador de baja frecuencia (LFO) que se pueda ligar al tono del sonido o incluso a la frecuencia de corte del filtro, una envolvente y infinidad de cosas mas.

Con respecto a sistemas de control en lazo cerrado, no tiene mucha aplicabilidad, por no decir ninguna, en este campo ya que estamos hablando de sistemas generativos.

Sobre las tecnologías utilizadas, se demuestra que Javascript no es para nada un lenguaje optimizado para eficiencia, ya que con este procesamiento básico se observaban fallos de eficiencia debido a que el motor web es incapaz de realizar calculos tantas veces por segundo (44100 veces por segundo). Pico.js es un buen sistema para implementar sonidos sencillos como la generacion directa de una forma cuadrada o senoidal, sin llegar a su procesamiento.