Efectos de audio - Parte 2

17 de junio de 2015

1. Generalidades

La aprobación de este curso se consigue mediante la correcta implementación de dos pequeños proyectos de programación. Éstos son propuestos aproximadamente un mes antes de cada parcial, y entregados a través de una página web habilitada para tales fines, con fecha límite de entrega fijada poco antes de cada período de parcial. Cada entrega es complementada con una pequeña prueba escrita cuyo objetivo es evaluar aspectos más teóricos relacionados con el propio obligatorio.

Es importante recalcar que **tanto la prueba escrita como el proyecto entregado son individuales.** El sistema de recepción de entregas, además de almacenar los archivos enviados por los estudiantes, realiza un control de copias contra las entregas de otros estudiantes así como de programas similares que se encuentran en la web.

En casos de ser posible, el sistema intentará además compilar y ejecutar la entrega de cada estudiante, de modo de dar un mínimo de información al respecto de qué tan bien funciona la entrega. Dependiendo del proyecto, esta evaluación preliminar estará o no disponible.

En todo caso, la evaluación preliminar mencionada anteriormente **no** determina la nota obtenida en la prueba, siendo ésta definida por una evaluación manual por parte de los docentes.

1.1. Formato del archivo a entregar

El archivo entregado debe ser un archivo comprimido en formato **zip** (NO se pueden subir rar), de nombre **nnnnnnn.zip** donde **nnnnnnn** es su número de cédula, **sin** digito de verificacion, sin espacios ni puntos, ni guión. El contenido del archivo debe incluir los siguientes elementos bajo el subdirectorio **nnnnnnn** (no pueden haber subdirectorios):

- Todos los archivos fuente creados por el estudiante (**.h** y **.c**)
- La biblioteca precompilada entregada por los docentes, si es que existe.
- Un archivo **Makefile** para compilar el o los programas requeridos en el trabajo.

Por ejemplo, supongamos el obligatorio consiste en la generación de un ejecutable de nombre `oblig`, su cédula es 1234567-8, y usted implementó dicho ejecutable en tres módulos **a.c** (y su correspondiente encabezado **a.h**), **b.c** (encabezado **b.h**) y **main.c**. Además, los docentes le entregaron una biblioteca auxiliar con un encabezado **aux.h** y código objeto **aux.o**. Entonces debe subir un archivo de nombre `1234567.zip` con el siguiente contenido:

```
1234567/
  a.c
  a.h
  b.c
  b.h
  aux.h
  aux.o
  main.c
  Makefile
```

El **Makefile** podría ser así:

```
oblig: main.o a.o b.o aux.o
  cc -o oblig main.o a.o b.o aux.o -lm

main.o: main.c
  cc -c main.c

a.o: a.c
  cc -c a.c

b.o: b.c
  cc -c b.c
```

Nota: Pueden crear un zip desde la máquina virtual con el comando `zip`; la sintaxis es

```
$zip -r nombre_archivo.zip carpeta_a_comprimir
```

en el ejemplo anterior, sería `zip -r 1234567.zip 1234567`.

1.2. Metodología de trabajo

Algunas recomendaciones generales sobre cómo trabajar con proyectos como los que se proponen aquí:

- Simplicidad (KISS - Keep It Simple, Stupid). No complicar el código más allá de lo requerido.
- Prolijidad. No importa cuánto aburra, documentar bien lo que se hace es fundamental; es muy fácil olvidarse lo que uno mismo hizo.
- Incrementalidad. Implementar y probar de a pequeños pasos. “No construir un castillo de entrada”. Es muy difícl encontrar las causas de un problema si se prueba todo simultáneamente.
- No reinventar la rueda. Antes de implementar una función, buscar a ver si no existe una función en la biblioteca estándar de C que ya lo haga.

2. Descripción de la tarea

En esta segunda parte abordaremos los siguientes temas:

- definición y manejo de estructuras de datos asociadas al problema
- lectura y escritura de archivos WAV de tamaño arbitrario
- algunos efectos nuevos, cantidad de parámetros variable

2.1. Estructuras de datos

Tendremos dos tipos de estructuras: una para definir una pieza de audio, y otra para encapsular el concepto de buffer circular. Además, definiremos algunas constantes y etiquetas útiles para el resto de los programas.

2.2. Lectura y escritura de archivos WAV

El principal cambio en esta parte es la lectura y escritura de archivos de audio. Previamente se utilizó una biblioteca provista por los docentes. En esta ocasión, la biblioteca será implementada por los estudiantes en el módulo "audio" mencionado anteriormente. Las funciones principales de ese módulo son `leer_audio` y `escribir_audio`. En ellas se utilizará lo aprendido en el curso sobre entrada y salida de datos para leer los datos de un archivo binario de audio, y generar un archivo de audio que sea legible por otras aplicaciones. Lo que falta para poder implementar dichas funciones es la descripción detallada del formato de archivos `wav`, byte por byte (a esto se le suele llamar el "byte stream"), que queremos manipular. Detalles sobre este formato se darán al final de este archivo como apéndice.

En este caso, además, nuestra aplicación (y por ende la biblioteca a escribir) debe ser capaz de manejar archivos de longitud, frecuencia de muestreo, y de cantidad de canales variable (*mono* y *stereo*). Sí nos restringiremos, sin embargo, a señales de 16 bits por muestra, tal como lo veníamos haciendo antes.

Cuando la señal es multicanal con C canales, llamaremos *frame* al conjunto de las C muestras tomadas en mismo instante de tiempo en todos los C canales. Si el archivo es monoaural (un canal), muestra y frame son lo mismo. En el caso de señales stereo (dos canales, izquiedo y derecho), el frame es un par de muestras (x_l, x_r).

El largo de una señal multicanal corresponde en este caso a la cantidad de frames, no muestras, que contiene la señal. Es decir, una señal de de C canales y largo N tendrá un total de $N \times C$ muestras.

2.3. Representación de señales multicanal

Las señales multicanal serán almacenadas en memoria en un único array unidimensional, en donde cada frame será almacenado secuencialmente en bloques consecutivos de C muestras. Para una señal de largo N, el array en cuestión deberá tener $N \times C$ elementos. Por ejemplo, para una señal stereo, el array tendrá largo $2N$, y su contenido será de la siguiente forma:

$$\{x_l[0], x_r[1], x_l[1], x_r[1], x_l[2], \ldots, x_l[N-1], x_r[N-1]\}.$$

Es importante notar sin embargo que muchos de los filtros que fueron implementados en la primera etapa operan muestra a muestra, independientemente de la cantidad de canales. Esto es una gran ventaja, ya que podrán operar sobre una señal multicanal sin modificación alguna de su código.

2.4. Nuevos filtros

El trabajo en este caso será bastante menor en carga relativa a los otros puntos, y consta de tres partes:

- acomodar las interfaces de los filtros para que tomen como argumento las estructuras definidas anteriormente en lugar de sus componentes "sueltos" como antes (por ejemplo, el buffer circular, en lugar de pasarse el buffer y el puntero, se pasa la estructura que los engloba),
- modificar los filtros para que operen con muestras multicanal; sólo uno de los filtros vistos anteriormente cambia significativamente por esto
- implementar un par de filtros nuevos y un nuevo bloque; este último es una variante mínima de otro visto anteriormente.

Bloque comb Este filtro es muy similar al eco que vimos anteriormente, sólo que la muestra retardada se retroalimenta a la entrada, de modo de generar una especie de reverberación infinita. La figura 1 muestra su esquema. La ecuación que relaciona la entrada con la salida de un canal es $y[j] = by[j - \Delta] + x[j]$ (el orden aquí importa: hay que leer el buffer antes de escribirlo!). El parámetro $b < 1$ controla la "intensidad" de la reverberación, mientras que el retardo $\Delta > 0$ controla la "profundidad" del efecto.

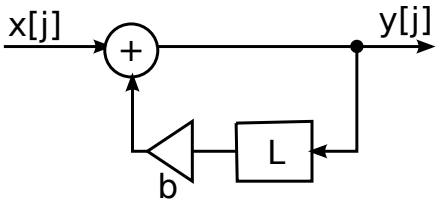

Figura 1: Bloque "comb" (peine en inglés) de profundidad L e intensidad b.

Chorus Este filtro combina el bloque de Eco que vimos en la primera parte del obligatorio con el LFO. La idea del filtro es generar una versión levemente "desafinada" de la señal de entrada, y sumársela a ésta, para dar la sensación de que hay más de una voz al unísono ejecutando la misma melodía. Este es un efecto muy usado por guitarristas y cantantes (ejemplos: "Come as you are", de Nirvana, "Message in a bottle", The Police, "Continuum", Jaco Pastorius). El efecto se logra haciendo que el eco tenga un retardo variable con el tiempo, $y[j] = (1 - a)x[j] + ax[j - \Delta - k(j)]$ en donde $k[j]$ es la salida de un LFO de amplitud Δ, usualmente sinusoidal (el que usaremos en nuestro caso).

El filtro tiene tres parámetros: la proporción de mezcla a que ya vimos en el filtro eco, la amplitud de la sinusoide Δ (de modo que el retardo mínimo es 0 y el máximo 2Δ), a la que llamaremos "profundidad" y la frecuencia del LFO, θ, que suele ser muy lenta (1-5Hz).

Flanger La idea de este filtro es la misma que el anterior, pero en lugar del bloque de eco utiliza el bloque "comb" definido anteriormente, y en lugar de una sinusoide utiliza una onda triangular, de frecuencia aún más baja (0.1-0.3Hz). ¡El resultado es radicalmente distinto! Ejemplos clásicos de este filtro son temas como "Are you gonna go my way", Lenny Kravitz, (solo minuto 2:15) "Head over heels" (minuto 3:00). Todo el resto es idéntico al Chorus, por lo que es prácticamente copiar y pegar.

Reverb La idea del reverb es la de simular la reverberación acústica dentro de una sala. Para ello, se combinan en paralelo un número de bloques tipo "comb" de distinto retardo cada uno, simulando el tiempo de rebote entre paredes. Por ejemplo, dos paredes paralelas de 10m entre ellas generan un rebote de 30ms. En nuestra implementación, tomaremos cuatro bloques comb, de modo que

$$y[j] = (1 - a)x[j] + \frac{a}{4}\left[\text{comb}_1(x[j]) + \text{comb}_2(x[j]) + \text{comb}_3(x[j]) + \text{comb}_4(x[j])\right],$$

y elegiremos los retardos de dichos bloques como $\Delta_2 = 0{,}73\Delta_1$, $\Delta_3 = 0{,}57\Delta_1$, $\Delta_4 = 0{,}37\Delta_1$, y la intensidad de cada bloque como $b_1 = b_2 = b_3 = b_4$ con Δ_1 (profundidad), b_1 (intensidad) y a (mezcla) los tres parámetros seleccionables del filtro.

3. Especificación de requerimientos

3.1. Módulo audio

Debe crearse un módulo "audio" a partir de dos archivos, **audio.h**, **audio.c**. En **audio.h** deben declarse los siguientes tipos:

- una constante de preprocesador `MAX_VAL` de valor 32767
- una constante de preprocesador `MIN_VAL` de valor -32768
- un tipo enumeración de nombre `wav_resultado_t` con las siguientes etiquetas:
  1. `WAV_OK`
  2. `WAV_ERROR_LECTURA`
  3. `WAV_ENCABEZADO_INVALIDO`
  4. `WAV_ERROR_ESCRITURA`
- un tipo enumeración de nombre `canales_t` con las etiquetas `MONO` y `STEREO` en ese orden
- una estructura de nombre `audio_t`, con los siguientes campos:
  - `canales`, de tipo `canales_t`
  - `frec_muestreo`, de tipo entero sin signo
  - `largo`, de tipo entero sin signo
  - `muestras`, de tipo puntero a entero corto

Además, deben declararse en **audio.h** y definirse en **audio.c** las siguientes funciones:

- `void inicializar_audio(audio_t* audio);`
  Estando bien definidos todos los valores de la estructura `audio` pasados a la función menos el arreglo de `muestras` en sí, esta función reserva la memoria necesaria según los valores anteriores (canales, frecuencia, largo de la señal) y la asigna al campo `muestras`.
- `void destruir_audio(audio_t *audio);`
  Libera la memoria apuntada por `muestras`, y pone dicha variable a `NULL`
- `wav_resultado_t leer_wav(const char* ruta, audio_t *audio);`
  Lee el archivo WAV referido por `ruta` y "rellena" la estructura `audio` con los datos de dicho archivo. Ver más adelante por detalles del formato WAV.
- `wav_resultado_t escribir_wav(const char* ruta, const audio_t *audio);`
  Escribe los datos de audio contenidos en `audio` como un archivo en formato WAV ubicado en la ruta UNIX especificada por `ruta`. Ver más adelante por detalles del formato WAV.

3.2. Modulo buffer

Se implementará un módulo "buffer" en dos archivos **buffer.h** y **buffer.c**. En **buffer.h** se declararán los siguientes elementos:

- una constante de preprocesador `BUFFER_SIZE` de valor 8192
- una constante de preprocesador `BUFFER_MASK` de valor 8191
- una estructura de nombre `buffer_t`, con los siguientes campos:
  - `posicion` de tipo entero sin signo; apunta a la última muestra guardada
  - `almacenamiento` puntero a enteros cortos; almacena las muestras del buffer

Ademas, se declararán en **buffer.h**, y definirán en **buffer.c**, las siguientes funciones:

- `void inicializar_buffer(buffer_t *buffer);`
  Reserva espacio para un buffer (siempre tienen el mismo tamaño, especificado por la constante `BUFFER_SIZE`), lo asigna al campo `almacenamiento`, y luego lo inicializa con ceros en su totalidad. Finalmente, se pone a cero el campo `posicion`
- `short leer_buffer(float retraso, buffer_t *buffer);`
  Lee una muestra del buffer, con posición relativa a la última muestra almacenada dada por el parámetro `retraso`.
- `void escribir_buffer(short muestra, buffer_t *buffer);`
  Incrementa la `posicion` en uno (módulo el tamaño del buffer) y escribe allí la nueva muestra recibida como argumento.
- `void destruir_buffer(buffer_t *buffer);`
  Libera el espacio reservado para el buffer, apuntado por el campo `almacenamiento` de la estructura pasada como argumento.

3.3. Módulo bloques

Se deberá agrupar los bloques anteriormente escritos, y los nuevos, en un nuevo módulo definido por un encabezado **bloques.h** e implementación **bloques.c**. A continuación se listan **todos** los bloques (tanto los ya existentes como el nuevo):

- `float bloque_eco(float muestra, float ret, float mezcla, buffer_t *buffer);`
  Recibe un `buffer_t`
- `float bloque_lut(short muestra, float* T);` Sin cambios.
- `float bloque_lfo(int tipo, float f0, float a, double *ppsi, unsigned frec_muestreo);`
  En esta segunda parte recibe también la frecuencia de muestreo, que ya no es constante. Además, para minimizar errores numéricos, utilizaremos un `double` para la fase.
- `float bloque_comb(float muestra, float ret, float mezcla, buffer_t *buffer);`
  Nuevo en esta parte.

3.4. Módulo filtros

Se deberá agrupar los filtros anteriormente escritos, y los nuevos, en un nuevo módulo definido por un encabezado **filtros.h** e implementación **filtros.c**. A continuación se listan **todos** los filtros (tanto los ya existentes como el nuevo):

- `void filtro_amp (const audio_t* entrada, float amp, audio_t *salida);`
  El filtro en sí no cambia: se procesan todas las muestras sin tener en cuenta el canal.
- `void filtro_norm(const audio_t* entrada, float factor, audio_t *salida);`
  Idem a `filtro_amp`.
- `void filtro_over(const audio_t* entrada, float dist, audio_t* salida);`
  Idem a `filtro_amp`.
- `void filtro_clip(const audio_t* entrada, float a, audio_t * salida);`
  Idem a `filtro_amp`.
- `void filtro_eco(const audio_t* entrada, float retardo, float mezcla, audio_t * salida);`
  Este filtro cambia: se precisa un bloque de eco para cada canal.
- `void filtro_chorus(const audio_t * entrada, float profundidad_seg, float frecuencia_hz, float mezcla, audio_t * salida);`
  Nuevo filtro. Ver la descripción por una explicación de sus parámetros.
- `void filtro_flanger(const audio_t * entrada, float profundidad_seg, float intensidad, float frecuencia_hz, float mezcla, audio_t * salida);`
  Nuevo filtro. Ver la descripción por una explicación de sus parámetros.
- `void filtro_reverb(const audio_t * entrada, float profundidad_seg, float intensidad, float mezcla, audio_t * salida);`
  Nuevo filtro. Ver la descripción por una explicación de sus parámetros.

Los filtros `filtro_sen` y `filtro_tri` no se incluirán en esta parte, pero puede ser de utilidad mantenerlos en caso de que se precise depurar las LFOs.

3.5. Interfaz de usuario

Al igual que en la primera parte, el resultado de la tarea será un ejecutable de nombre `obligatorio`. En este caso, sin embargo, la sintaxis del comando será modificadad para permitir la especificación de más de un parámetro, siendo la cantidad de parámetros dependiente del filtro a aplicar. Concretamente, la sintaxis será de la forma:

```
./obligatorio entrada.wav salida.wav filtro par1 par2 ... parn
```

donde

- `entrada.wav` es el archivo de entrada al programa y
- `salida.wav` el nombre del archivo de salida del programa.
- `filtro` es el nombre del filtro a aplicar,
- `par1` es el primer parámetro numérico del filtro, de existir
- `par2` es el segundo parámetro numérico del filtro, etc.

En este caso, todos los parámetros que no se especifiquen de un filtro tendrán un valor por defecto, especificado más adelante en la sección que detalla su implementación. A continuación se listan los nombres y parámetros asociados a cada filtro, tal y como aparecerán en la linea de comandos, con sus respectivos valores por defecto en paréntesis, si no se especifican.

filtro	**nombre**	**parámetros**
amplificar	amp	factor (1.0)
normalizar	norm	factor (1.0)
overdrive	over	intensidad (3.0)
clip	clip	umbral_relativo (1.0)
eco	eco	retardo_seg (0.1) mezcla (0.5)
chorus	chorus	profundidad_seg (0.005) frecuencia_hz (0.5) mezcla (0.5)
flanger	flanger	profundidad_seg (0.001) intensidad (0.8) frecuencia_hz (0.2) mezcla (1.0)
reverb	reverb	profundidad_seg (0.15) intensidad (0.8) mezcla (0.9)

4. Recomendaciones

- Recuerden siempre cerrar los archivos al finalizar su escritura!
- La correcta escritura y lectura de valores numéricos de distinta precisión, desde y hacia archivos, es algo bastante delicado. Además, cualquier pequeño error en la escritura del encabezado torna al archivo ilegible. Por eso, es altamente recomendable practicar primero el lograr escribir y leer números enteros de 2 y 4 bytes en archivos binarios antes de proceder con la implementación del formato WAV.
- Para chequear el contenido binario de un archivo WAV se recomienda utilizar la herramienta **hexdump** de Linux.

A. El formato WAV

Como prácticamente todo formato de archivos, el formato `wav` divide el stream en dos bloques: un *encabezado* al principio, en donde se detallan aspectos generales del archivo (los llamados "metadatos") como largo, cantidad de canales, cantidad de muestras, frecuencia de muestreo, etc., y luego vienen los datos en sí.

El encabezado tiene partes que son de texto ASCII, intercalados con campos numéricos de distinto ancho (es decir, cantidad de bytes). Todos los números son enteros, algunos con signo y otros sin signo, y se almacenan utilizando la convención "little endian", es decir, en un entero de más de un byte (por ejemplo, un `short` de 16 bits), los 8 bits *menos* significativos aparecen *primero* en el stream, y luego aparecen los 8 más significativos, es decir, si el primer byte leido tiene valor (de byte) a y el segundo tiene valor b, el entero leido finalmente será $256 * b + a$.

Recordemos que en el caso de señales multicanal, llamamos *frame* al conjunto de muestras tomadas en todos los canales en un mismo período de tiempo. En un archivo WAV stereo (dos canales), cada frame es codificado secuencialmente, apareciendo primero la muestra izquierda x_l y luego la muestra derecha x_r.

A modo de ejemplo, y combinando lo que se dice anteriormente, los cuatro primeros bytes de un archivo WAV stereo de 16 bits son: $\text{LSB}(x_l), \text{MSB}(x_l), \text{LSB}(x_r), \text{MSB}(x_r)$ donde LSB y MSB son Least y Most Significant Byte respectivamente. La tabla 1 muestra el detalle del stream WAV.

pos	bytes	tipo	valor	comentario
0	4	texto	'R','I','F','F'	identificador de tipo de archivo
4	4	entero	$36 + B{\times}C{\times}N$	largo del resto del archivo
8	4	texto	'W','A','V','E'	identificador para archivo de audio
12	4	texto	'f','m','t',' '	identificador textual para encabezado
16	4	entero	16	largo del encabezado
20	2	entero	1	identificador de tipo PCM
22	2	entero	C	cantidad de canales
24	4	entero	F	frecuencia de muestreo
28	4	entero	$F{\times}B{\times}C$	
32	2	entero	$B \times C$	bytes por muestra de todos los canales
34	2	entero	$B{\times}8$	bits por muestra
36	4	texto	'd','a','t','a'	indicador de comienzo de bloque de datos
40	4	entero	$N{\times}B{\times}C$	cantidad de bytes de datos a continuación
44	$N{\times}B{\times}C$	entero	...	datos

Cuadro 1: Formato WAV. B es la cantidad bytes por muestra de *un* canal, C la cantidad de canales, N la cantidad de frames en el archivo y F es la frecuencia de muestreo.