Técnicas de extracción en tiempo real del tempo de pistas de audio sobre dispositivos móviles

TRABAJO DE FIN DE GRADO
Doble Grado en Ingeniería Informática y Matemáticas

Jesús Javier Doménech Arellano

Dirigido por el Doctor
Pedro Pablo Gómez Martín
Codirigido por el Doctor
Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

Documento maquetado con TEXIS v.1.0.

Este documento está preparado para ser imprimido a doble cara.

Técnicas de extracción en tiempo real del tempo de pistas de audio sobre dispositivos móviles

Memoria que presenta para optar al
Doble Graduado en Matemáticas e Ingeniería Informática

Jesús Javier Doménech Arellano

Dirigido por el Doctor
Pedro Pablo Gómez Martín
Codirigido por el Doctor
Marco Antonio Gómez Martín

Departamento de Ingeniería del Software e Inteligencia Artificial
Facultad de Informática
Universidad Complutense de Madrid

Junio 2015

A Ana, mi pareja,
y
a mi extensa familia.

Siempre que te pregunten si puedes hacer un trabajo,
contesta que sí y ponte enseguida a aprender cómo se hace.
Franklin D. Roosevelt (1882-1945)

Agradecimientos

A todos los que la presente vieren y entendieren.

Inicio de las Leyes Orgánicas. Juan Carlos I

Este trabajo no habría sido posible sin muchas personas, ahora cito solo una pequeña parte de aquellas que me han dado lo necesario para realizarlo.

Por una parte mi pequeña gran familia que me ha soportado todo el desorden que he producido en casa, me han escuchado aunque no entendiesen de que les hablaba y se han adaptado a mis horarios; por eso y más gracias.

Mi pareja, Ana Pérez Gómez, cada día animando a terminar el trabajo siempre pendiente y ayudando en esa recta final del trabajo que se iba haciendo cuesta arriba.

D. Pablo Esteve y Joaquín Sánchez, que han colaborado con criterios y dando ese *plus* a los motivos por los que se hacen las cosas y también lo que podemos llamar un poco de *frikismo*.

Quiero agradecer también a todos los que han colaborado en la encuesta de precisión y en especial a la compañera, Estíbaliz Busto Pérez de Mendiguren, que me cedió una grabación para el análisis.

A mis profesores y amigos Luis Hernández Yáñez, Pedro Pablo Gómez Martín y Marco Antonio Gómez Martín, que no sólo han contribuido en mi formación todos estos años y han estado al pie del cañón durante todo el TFG, sino que también han contribuido a que se acreciente mi deseo de ser, en un futuro, profesor e investigador en la universidad.

Por último a mis compañeros: Luis María Costero, Jennifer Hernández, Alejandro Aguirre, Francisco Criado y Pablo Cabeza por ser el equipo *SWERC* acompañándome todos estos años.

Resumen

No podemos hacer grandes cosas, pero si cosas pequeñas con gran amor.

Beata Madre Teresa de Calcuta

En este trabajo se presentan dos análisis para la detección del ritmo en pistas de audio, llamados "Simple Sound Energy" y "Frecuency Selected Sound Energy" los cuales se describen como son en el Capítulo 3 y como han sido implementados en la Sección 4.2 y el Apéndice A.

Para llegar a ello primero se describe en la Sección 2.1 la manera en que el sonido es almacenado en un ordenador y como es reproducido, detallando dos de los formatos de archivos de audio más utilizados.

A continuación, en la Sección 2.1.2 se recorre la situación actual en lo que a detección de ritmo se refiere, que tipos de algoritmos existen y que proyectos o aplicaciones hacen este tipo de análisis.

También se muestra, en la Sección 2.2.1 la manera de integrar un código C/C++ dentro de una aplicación Android para obtener algo de eficiencia a la hora de ejecutar algoritmos complicados o que requieran grandes recursos y se desarrolla una aplicación para Android utilizando esta técnica en la Sección 4.3. En esa aplicación se han integrado los análisis de detección de ritmo en código C/C++ y se muestra el resultado de manera visual en la aplicación.

Posteriormente, en el Capítulo 5 se ha medido el nivel de precisión de ambos análisis con una encuesta *online* donde los usuarios encuestados han escuchado diversas pistas marcadas con un elemento sonoro en los instantes donde se han producido los golpes de ritmo.

Finalmente, en el Capítulo 6 se exponen las conclusiones obtenidas sobre todo el trabajo realizado y el futuro trabajo que podrá ser desarrollado.

Palabras clave: Android, detección de pulso, JNI, ritmo, sonido, transformada de Fourier.

Abstract

We can do no great things, only small things with great love

Beatified Mother Teresa of Calcutta

In this work two analysis are presented to detect the rhythm in audio-tracks. These analysis, called "Simple Sound Energy" and "Frecuency Selected Sound Energy", are described in Chapter 3 and they have been implemented in a method which is described in Section 4.2 and in Appendix A.

To achieve this, first, it is described in Section 2.1, the way the sound is stocked in a computer and how it is played (focusing on the two mainly used audio formats).

Later, in Section 2.1.2 we find a survey on the issue of rhythm detection, the types of algorithms we can find (ó there exist) and the apps making this kind of analysis.

In Section 2.2.1 it is also shown how to integrate code from C/C++ in an Android app in order to obtain efficiency when running complex algorithms or requiring large resources. Using this technique, in Section 4.3, an Android app is developed. In this app, the rhythm-detection analysis has been integrated, using C/C++, and the result is visually displayed on the screen.

Later, in Chapter 5 the accuracy level in both analysis has been measured with an online poll where the respondents have listened to some tracks marked with a sonorous item when the beat in the rhythm happens.

Finally, in Chapter 6 the conclusions obtained are exposed as the future work that will be developed.

Keywords: Android, beat detection, Fourier transform, JNI, rhythm, sound.

Índice

Capítulo 1

Motivación

La música es un arte que está fuera de los límites de la razón, lo mismo puede decirse que está por debajo como que se encuentra por encima de ella.

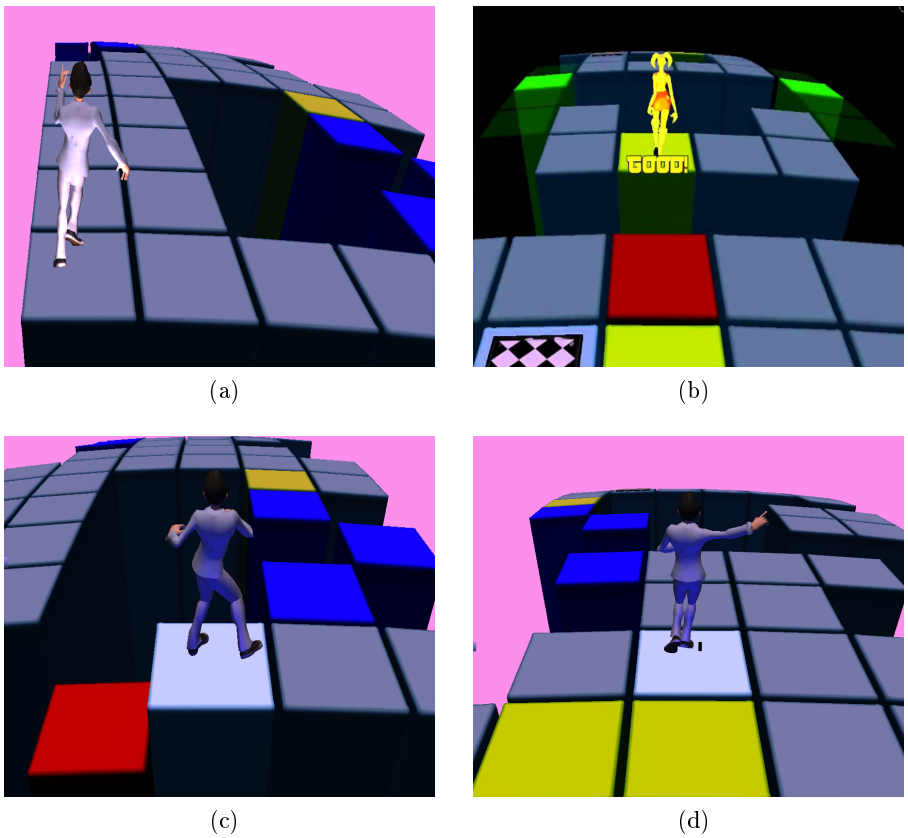
Pío Baroja

El entretenimiento es la gran demanda de hoy en día, todo se mueve en torno a él, el mundo que busca el *Estado de Bienestar* quiere hacer su vida amena y divertida. Dentro del entretenimiento existen unas grandes potencias como podrían ser la televisión o el fútbol, pero también encontramos los videojuegos. Estos requieren cada vez un desarrollo más logrado para que lleguen a sorprender a los usuarios y el videojuego sea aceptado.

Para esto hay que cumplir una serie de requisitos, entre los cuales se encuentra el tener una buena idea para el juego. Hace un tiempo surgió una idea: que cada vez que juegas el nivel sea diferente, que la música haga cambiar el escenario, pero no solo eso, sino que puedas escoger cualquier música. Con esta idea se desarrolló un videojuego (fig. 1.1) en el que tienes que recorrer un escenario que va cambiando al ritmo de la música hasta llegar al final, pero cómo adaptar los cambios a la música se convirtió en un problema y se propuso inicialmente preanalizar manualmente unas pocas canciones entre las que se pueda elegir. Con la intención de más adelante conseguir permitir que el usuario pueda elegir cualquier audio y jugar con él.

En este trabajo se pretende realizar un análisis automático para poder ampliar, en un futuro, el juego para cumplir con esta última idea. De manera que se analice la canción que el usuario elija y se extraigan las características necesarias de la canción para poder modificar el nivel con ellas.

El aumento de uso de móviles hacen que el juego fuese desarrollado pen-



(a) (b)

(c) (d)

Figura 1.1: Capturas del Juego

sando en estos dispositivos, y por tanto el análisis debe ser realizado para ellos. En este trabajo se desarrollará dicho análisis atendiendo a las limitaciones que estos dispositivos presentan, como son la memoria y la baja capacidad de computación. Por tanto, se plantea realizar dicho análisis en tiempo real o *streaming* si fuese necesario para minimizar los tiempos de espera al usuario.

1.1. Plan de trabajo

Para realizar la tarea mencionada se ha elaborado el siguiente plan de trabajo.

Se inició con una primera toma de contacto con toda la materia nueva que el trabajo iba a requerir, como puede ser:

- Probar el juego.

- Estudiar acerca del sonido.
- La modelización del sonido en ficheros.
- Programación para Android.
- Comunicar código para Android con C/C++.

Tras realizar una iniciación en estos puntos, se procedió a la investigación y estudio de la situación actual en lo respecto a la detección del tempo en pistas de audio. ¿Cuáles son algunas herramientas actuales que lo realizan? ¿Qué tipos de algoritmos hay? ¿Cuánto tiempo medio tardan los diferentes algoritmos? Han sido algunas de las preguntas que se ha buscado responder. Además de ejecutar los programas que se han encontrado para ver de manera subjetiva su precisión.

Un tercer paso que ha continuado al anterior, ha sido realizar una implementación de los algoritmos encontrados más relevantes, adaptándolos al resultado que se quiere obtener para su incorporación al juego. Pero solamente se ha pretendido ejecutar estos algoritmos en un ordenador para poder evaluar su precisión y rendimiento.

Tras unos resultados satisfactorios, se ha procedido con el desarrollo de una aplicación en Android que incorpora el algoritmo. Se ha buscado programar el algoritmo en código nativa del dispositivo para lograr una eficiencia mayor.

Con todo esto se da por finalizado el trabajo y la integración del algoritmo en el videojuego que está desarrollado sobre *Unity*, se dejará para más adelante tal y como se explica en el Capítulo 6 de este trabajo.

Capítulo 2

Situación actual

No estoy diciendo que voy a cambiar el mundo, pero garantizo que encenderé la llama del cerebro que si lo hará.

Tupac Shakur

Resumen: En este capítulo, describiremos brevemente lo que es un sonido sus características y como se almacena digitalmente en diferentes formatos de archivos de audio. También mostraremos la situación de los últimos años en el ámbito de la extracción de tempo, los diferentes algoritmos y plataformas sobre las que se ha implementado. Por último, nos asomamos a la programación para Android, en especial la programación con código nativo o C/C++.

2.1. El sonido

El sonido es cualquier fenómeno que implique una propagación de ondas, generalmente producido por un movimiento vibratorio de un cuerpo. La propagación del sonido implica un transporte de energía sin transporte de materia a través de un medio comúnmente el aire y no se puede propagar por el vacío.

Como el sonido se produce por un movimiento ondulatorio al aplicarle la transformada de Fourier podemos expresarlo por una suma de curvas sinusoidales que corresponden a tonos puros que se pueden caracterizar por las magnitudes de cualquier onda como son:

- *Período ($\mathcal{T}$)*: es el tiempo transcurrido entre dos puntos equivalentes de una onda. Esto es, si la onda sigue la función $c(t)$, donde t indica

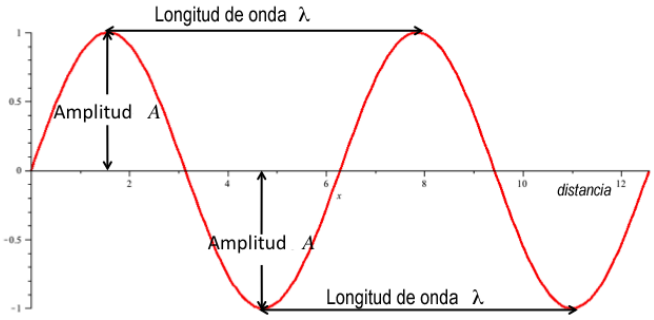


Figura 2.1: Características de una onda

el tiempo. El *período* cumple que $c(t+\mathcal{T}) = c(t)\,\forall t$, por tanto existen infinitos valores para $\mathcal{T}$ que satisfacen la condición anterior, se toma como valor al menor positivo no nulo que cumpla la condición. Por ejemplo el *período* de la función coseno es 2π.

- *Longitud de onda (λ)*: se trata de la distancia real que recorre una onda entre dos crestas o dos valles, la cresta es el punto más alto que alcanza la onda y el valle el más bajo.

- *Frecuencia (f)*: es la magnitud que mide el número de repeticiones sucede un fenómeno en un tiempo. En las ondas, la frecuencia es inversamente proporcional a la *longitud de onda*, a mayor *frecuencia* menor *longitud de onda* y viceversa. La *frecuencia* sigue la relación $f = \frac{v}{\lambda}$ donde v es la velocidad. Cuando una onda cambia de medio, por ejemplo del aire al agua, la *frecuencia* se mantiene constante variando la velocidad y la *longitud de onda*.

- *Amplitud*: es la medida que marca la distancia entre el punto más alejado de la onda con el punto de equilibrio. Esto es la mitad de la distancia entre la cresta y el siguiente valle, anteriormente definidos. Aunque se puede entender como *amplitud* para un valor concreto de tiempo, la distancia de la onda al punto de equilibrio en ese momento.

Con estas características un sonido audible por los seres humanos, es aquel que está entre los 20 y 20000 Hz. Para poder almacenarlos en un ordenador se requiere de un punto de entrada, es un transductor que transforma las ondas de presión de aire, es decir, las ondas sonoras, en señales eléctricas.

Este transductor puede ser de diferentes tipos como son el *electrostático*, *dinámico*, *piezoeléctrico*, de *carbono*, etc. Se explica el primero para ver una idea del funcionamiento.

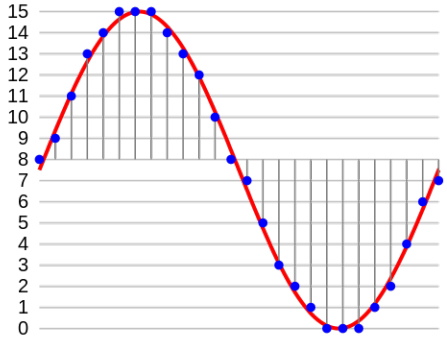


Figura 2.2: Muestreo y cuantificación de una onda senoidal en código PCM de 4-bits de profundidad

La vibración de la onda provoca en el transductor el movimiento oscilatorio en una membrana llamada diafragma que produce una variación en la energía almacenada en un condensador. Esa variación genera una tensión eléctrica que es análoga en amplitud y frecuencia a las ondas sonoras iniciales. La señal eléctrica ya puede ser almacenada.

Para almacenarla se procede a tomar en intervalos uniformes de tiempo unas muestras de la señal eléctrica analógica. La calidad de los datos almacenados dependen de la frecuencia de muestreo y la profundidad o cantidad de bits por muestra, como se aprecia en el ejemplo de la figura 2.2. El número de bits por muestra determinará el error de precisión que se comete por almacenar un valor contínuo (amplitud de la onda en un instante dado) en un valor discreto (entero de 8 o 16 bits normalmente, de 4 bits en la figura anterior). Por otro lado la frecuencia de muestreo determina la separación (en tiempo) entre una muestra y otra y por tanto los huecos en los que no hay muestreo y en los que los cambios de onda no son detectados. Esta modulación se conoce por PCM (*Pulse Code Modulation*, Modulación por impusos codificados). A modo de ejemplo, los CD de audio utilizan una frecuencia de muestreo de 44.100Hz y 16 bits por muestra.

Para recuperar la señal original se realiza un proceso inverso, convirtiendo las señales eléctricas en energía mecánica y esta en ondas audibles. Al igual que el transductor de entrada, un transductor de salida tiene también diferentes tipos el *electrostático* recibe en el condensador la señal eléctrica y al producirse una variación de energía se produce un efecto de atracción o repulsión eléctrica que mueve la membrana móvil también llamada diafragma, este mueve el aire que tiene situado frente a él, generando las variaciones de presión en el mismo que son las ondas sonoras.

<table>
<tr><td>'R'</td><td>'I'</td><td>'F'</td><td>'F'</td><td>Chunk Size
(4 bytes)</td><td>'W'</td><td>'A'</td><td>'V'</td><td>'E'</td></tr>
<tr><td>'F'</td><td>'M'</td><td>'T'</td><td>' '</td><td>Subchunk Size
(4 bytes)</td><td colspan="2">Audio
Format
(2 bytes)</td><td colspan="2">Num
Channels
(2 bytes)</td></tr>
<tr><td colspan="4">Sample Rate
(4 bytes)</td><td>Bit Rate
(4 bytes)</td><td colspan="2">Block
Align
(2 bytes)</td><td colspan="2">Bits per
Sample
(2 bytes)</td></tr>
<tr><td>'D'</td><td>'A'</td><td>'T'</td><td>'A'</td><td>Subchunk Size
(4 bytes)</td><td colspan="4">Samples...</td></tr>
</table>

Figura 2.3: Estructura de la cabecera de un archivo WAV

En esta sección se ha tomado la información principalmente del libro (Gold et al., 2011).

2.1.1. Archivos de sonido

Los archivos de sonido se presentan en formatos muy variados, en función de la tarea que se va a realizar con ellos. Existen formatos con y sin pérdida de información, como pueden ser: MIDI (*Musical Instrument Digital Interface*, Interfaz Digital de Instrumentos Musicales), WAV (*Waveform Extensión*, Extensión de forma de onda), MP3 (*MPEG Audio Layer III*), OGG, WMA (*Windows Media Audio*), 3GP (*3rd Generation Partnership Project*), etc. A continuación se describen WAV y MP3.

2.1.1.1. WAV

El formato de archivo WAV fue desarrollado en colaboración por Microsoft e IBM. Y es un estándar para los archivos de audio.

Estos archivos son un tipo de archivos en formato "RIFF", que poseen en su cabecera un bloque "FMT" donde se indica el formato concreto del archivo WAV. Todo el archivo esta en *little-endian*.

Si el campo *Audio Format* tiene valor "01 00" el audio se presenta sin comprimir, en "crudo", en formato PCM. Para obtener calidad de CD se graba el sonido a 44.100 Hz, campo *Sample Rate*, y a 16 *bits per sample*. Cada minuto de sonido llega a ocupar 10 MB. Esto se convierte en una limitación a la hora de enviar archivos, pero no presenta pérdida de calidad,

<table>
<tr><td colspan="5">Frame Sync
(11 bits)</td></tr>
<tr><td colspan="2">Frame Sync (cont.)</td><td>Audio Version
(2 bits)</td><td>Layer description
(2 bits)</td><td>Protection bit
(1 bit)</td></tr>
<tr><td colspan="2">Bit Rate
(4 bits)</td><td colspan="2">Sampling Rate
(2 bits)</td><td>Padding bit
(1 bit)</td><td>Private bit
(1 bit)</td></tr>
<tr><td>Channel Mode
(2 bits)</td><td>Mode Extension
(2 bits)</td><td>Copyright
(1 bit)</td><td>Original
(1 bit)</td><td colspan="2">Emphasis
(2 bits)</td></tr>
</table>

Figura 2.4: Estructura de la cabecera de un archivo MP3

por lo que es adecuado para analizar un sonido o tratarlo de forma avanzada e incluso editarlo.

2.1.1.2. MP3

El MP3 es el formato de audio más popular debido a su gran calidad de sonido y bajo tamaño por el algoritmo de compresión que utiliza.

El audio es comprimido con pérdidas, se denomina "Lossy", esto es que se elimina del sonido partes que luego serán irrecuperables, pero que son irrelevantes para el sonido dado que pertenecen a la parte imperceptible por el oído humano.

Además presenta la opción de configuración de calidad, comprimiendo más el archivo sacrificando calidad de sonido.

La cabecera estándar de un archivo MP3 responde a la figura 2.4. Los campos que más nos interesan son:

- *Bit Rate*, que nos indica la velocidad de la pista en kbps.
- *Sampling Rate*, lo ideal es que tenga valor "00", que corresponde con una frecuencia de 44.100 Hz.
- *Channel Mode*, que indica el modo en que esta la pista como por ejemplo si esta en *stereo* o en *mono*.

2.1.2. Extracción del tempo

El *tempo* es el ritmo o compás de una acción. Concretamente en el ámbito musical, es el ritmo de una canción o pieza. Este se suele medir en

BPM (*Beats per minute*, pulsos por minuto), esta medida es general a una pista de audio o a un fragmento de la misma del que se extrae el tempo y determina aproximadamente en qué momento va a haber un golpe de ritmo, pero no es una medida local que te indica el momento exacto del golpe. Nosotros nos centraremos en este análisis local para detectar el momento exacto.

Este análisis se conoce habitualmente por *Beat Tracking*, *Foot Tapping* o *Beat Detection*. Este último nombre será el que usemos en adelante.

La detección de los golpes de ritmo, los *beats*, en una pista de audio es un ejercicio prácticamente trivial para el oído humano. Pero a la hora de formalizar se convierte en una tarea verdaderamente laboriosa.

"The experience of rhythm involves movement, regularity, grouping and yet accentuation and differentiation" (Handel, 1989). No hay un camino fiable para encontrar el ritmo de un sonido o canción, la única forma de ver si se ha encontrado es que el ser humano que lo oiga este de acuerdo. Por tanto no se puede automatizar la verificación de los algoritmos utilizados para este análisis.

Al investigar sobre este tema se encuentran muchos proyectos que lo han llevado a cabo, y lo han implementado en diferentes lenguajes y para diferentes funcionalidades, como las mencionadas en diferentes artículos:

- IBT (*tempo Induction and Beat Tracker*, Inducción sobre el tempo y seguimiento del pulso) es un sistema de extracción de los golpes de ritmo automático implementado en C++ e incorporado en el *framework* del sistema llamado MARSYAS (*Music Analysis, Retrieval and Synthesis for Audio Signals*), esta basado en un sistema multiagente que obtiene el ritmo considerando unas hipótesis de manera paralela sobre el tempo y los golpes de ritmo que luego va corrigiendo con una cierta tolerancia. El sistema sigue el denominado algoritmo "BeatRoot". Esta implementación permite hacer el análisis en tiempo real, para sistemas *online*, para archivos e incluso recibiendo la entrada por micrófono. Se puede encontrar más información en el artículo (Oliveira et al., 2010b) y en el artículo original (Oliveira et al., 2010a), de este último se obtuvo la idea para finalizar uno de los análisis implementados en este trabajo.

- Un proyecto analizado consiste en hacer bailar robot usando el sistema anterior, este ha sido integrado en el hardware del robot y conectado para realizar el análisis de ritmo en tiempo real. Se dispone de más información en el artículo (Santiago et al., 2011).

- Otro caso de implementación del algoritmo "BeatRoot" lo encontramos

en otro artículo (Goto y Muraoka, 1995), el cual nos presenta una forma más sencilla de implementarlo ya que no está integrado en un sistema de gran tamaño como es MARSYAS, además es bastante anterior al ya mencionado.

- Existe una familia de algoritmos basados en estimar un tempo global para toda la pista usar esa estimación para calcular con programación dinámica la posición del resto de golpes. Podemos encontrar más detalles en el artículo (Ellis, 2007).

- Por otro lado, se han desarrollado unos análisis basados en un estudio estático y estadístico sobre la amplitud de onda del audio. El manual (Patin, 2003)[1] describe cómo poder realizar una detección de golpes de ritmo fácilmente en un tiempo estático.

2.2. Android

La programación para dispositivos móviles está en auge, es la época del smartphone y las tablets. El SO (*Sistema Operativo*) predominante es Android. Y el desarrollo de aplicaciones para este SO es cada vez más sencillo con los diferentes manuales y paquetes de ayuda que se elaboran y desarrollan.

La arquitectura de los sistemas Android ha cambiado en sus últimas versiones, pero mantienen compatibilidad con la anterior. Por claridad y sencillez se utiliza la estructura antigua.

Esta arquitectura está construida sobre el *kernel* de Linux. Sobre este núcleo se pueden ejecutar programas y librerías en C/C++. A su vez, uno de estos programas es una Máquina Virtual llamada DVM (*Dalvik Virtual Machine*, Máquina Virtual de Dalvik) específica para Android aunque no cumple los estándares para ser llamada propiamente como JVM (*Java Virtual Machine*, Máquina Virtual de Java). Se facilita una API (*Application Programming Interface*, Interfaz de programación de aplicaciones) de Java que incluye paquetes especiales para facilitar el acceso a los diferentes componentes del dispositivo. En la última capa encontramos las aplicaciones en Java. Estas diferentes capas pueden apreciarse en la figura 2.5.

Lo más normal es desarrollar aplicaciones en el último nivel, aprovechando la API de Java. Se deben crear diferentes *layouts* que distribuyen los componentes de la aplicación. Se desarrolla un *activity* por cada tarea que la

[1](Patin, 2003) será el manual que seguiremos en capítulos sucesivos para desarrollar nuestro análisis

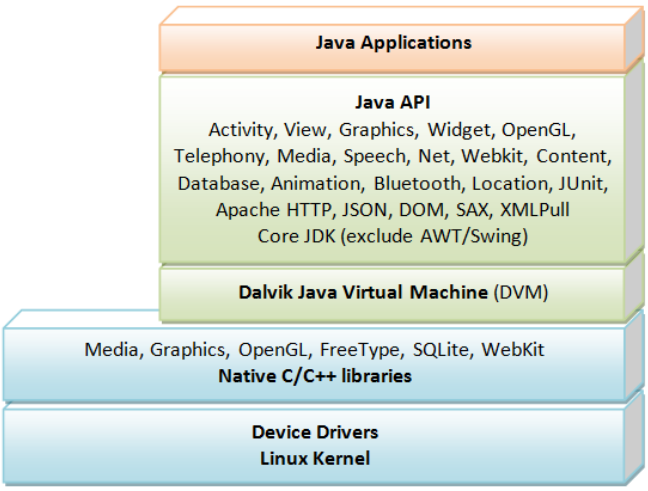


Figura 2.5: Arquitectura de la Plataforma Android

aplicación debe realizar. Para facilitar esta tarea se dispone del SDK (*Software Developer Kit*, Kit de Desarrollo de Software), que proporciona toda la documentación y acceso a la API.

Cuando se quiere optimizar la aplicación en uso de recursos y tiempo de ejecución se utiliza el lenguaje nativo, C/C++, y se ejecutan las aplicaciones directamente sobre el *kernel* sin pasar por la DVM. Pero realizar una interacción con el usuario del dispositivo sin la ayuda de la API de Java es una tarea complicada por lo que sólo se utiliza para programar "demonios" o "librerías".

Por otro lado, se pueden combinar ambos métodos de desarrollo de aplicaciones dejando los cálculos y algoritmos más pesados al código nativo y la interacción con el usuario a Java. Para fusionar ambos métodos se dispone del NDK (*Native Developer Kit*, Kit de Desarrollo Nativo) que junto con el JNI (*Java Native Interface*, Interfaz para código nativo y Java) nos facilita un conjunto de tipos de datos y la conversión de variables de Java a C.

2.2.1. La interfaz JNI

A veces es necesario conectar Java y C/C++, como hemos dicho JNI es una interfaz que facilita esta integración entre los dos lenguajes. La capacidad de JNI es mucha lo que complica su utilización en muchos aspectos. En lo sucesivo se describe brevemente la forma de utilizarlo para realizar tareas sencillas como es llamar a una función desde Java a C/C++ y el paso de

argumentos básico. Para más detalles o temas más avanzados se recomienda la lectura o consulta de (Liang, 1999).

En primer lugar, es necesario comprender los tipos de las variables usadas en el lado de C para intercambiar información con el de Java. Por ejemplo, cuando queremos recibir lo que para Java es un tipo `int`, en el lado de C/C++ no lo recibiremos como un `int` nativo (de C) porque los tamaños podrían no coincidir. En lugar de eso, JNI define tipos específicos que aseguran que tienen el ancho de la representación correcto:

- Tipos primitivos: `jint`, `jshort`, `jlong`, `jfloat`, `jdouble`, `jboolean`, `jchar`.
- Tipos referencia: `jstring`, `jobject`, `jintArray`, `jshortArray`, `jlongArray`, `jfloatArray`, `jdoubleArray`, `jbooleanArray`, `jcharArray`.

Primero necesitaremos crear una clase en Java que llame al método `System.loadLibrary()` que recibe un `String` como parámetro indicando el nombre de la librería en C que se quiere cargar, por ejemplo `Native`, esta librería en windows estará en el archivo `Native.dll` y en Unix como `libNative.so`. A la hora de compilar se debe incluir estas librerías como información para el compilador. También se debe añadir como métodos nativos los métodos a los que se quiere tener acceso de dentro de la librería. La clase Java de ejemplo puede verse a continuación:

```
1  public class claseJNI {
2    static {
3      System.loadLibrary("Native");
4    }
5
6    private native int suma(int x, int y);
7
8    public static void main(String[] args) {
9      claseJNI jni = new claseJNI();
10     int sum = jni.suma(3,2);
11   }
12 }
```

Ejecutando `javah` a la clase que acabamos de crear se obtiene el archivo cabecera para C/C++, aunque también podemos escribirlo manualmente, el resultado es el siguiente:

```
1  #include <jni.h>
2
```

```
3   #ifndef _Native_H
4   #define _Native_H
5   #ifdef __cplusplus
6     extern "C" {
7   #endif
8
9   JNIEXPORT jint JNICALL Java_claseJNI_suma(JNIEnv *,
10                                            jobject,
11                                            jint, jint);
12  #ifdef __cplusplus
13    }
14  #endif
15  #endif
```

Como puede apreciarse, el método suma nativo está declarado con el siguiente formato:

```
JNIEXPORT <returnType> JNICALL
Java_<NombreClase>_<NombreFunc>(<argsJNI>, <argsFunc>);
```

Los argumentos `<argsJNI>` son dos:

- Uno de tipo `JNIEnv` que referencia al entorno de JNI con una estructura de punteros a función, de modo que es usable en C sin problema, y en C++ ocasiona una "sintáxis de objeto" en las invocaciones que resulta natural.
- Un segundo argumento de tipo `jobject` que referencia el `this` del objeto de Java que llama al método.

Además, se escribe `extern C` para que el compilador de C++ lo pueda reconocer y compile esas funciones usando C. La implementación de `Native.c` se haría del siguiente modo:

```
1  // Native.c
2
3  #include <jni.h>
4  #include "Native.h"
5
6  JNIEXPORT jint JNICALL Java_claseJNI_suma(JNIEnv * env,
7                                            jobject thisObj,
8                                            jint x, jint y){
9    jint resultado;
10   resultado = (jint) x + y;
11   return resultado;
12 }
```

Observamos que la suma puede hacerse directamente sin problemas. Por otro lado, si queremos que la suma se haga en código C++ y no C, el método en C debe hacer de intermediario entre Java y C++ siendo él quien llame al método suma implementado en C++.

Si los atributos del método en C son `strings` de Java, se complica el proceso por lo que el argumento `JNIEnv` nos proporciona métodos para tratar los `strings` como por ejemplo para convertirlos en cadena de tipo C:

```
const char* GetStringUTFChars (JNIEnv*, jstring, jboolean*);
```

O para crear un `jstring` a partir de una cadena de tipo C:

```
jstring NewStringUTF(JNIEnv*, char*);
```

Podemos ver a continuación un ejemplo de archivo en C, donde se recibe un nombre en un `jstring` y se devuelve una cadena que le saluda.

```
1  // Native.cpp
2  #include <jni.h>
3  #include <string>
4  #include "Native.h"
5  using namespace std;
6
7  JNIEXPORT jstring JNICALL Java_Native_saludar(JNIEnv *env,
8                                                jobject thisobj,
9                                                jstring jnombre)
10 {
11     // jnombre a nombre en C: "cnombre"
12     const char *cnombre;
13     cnombre = env->GetStringUTFChars(jnombre,NULL);
14
15     string saludo = "Hola "+ cnombre;
16
17     // pasar saludo de string a cadena-C y de esta a jstring
18     return env->NewStringUTF(saludo.c_str());
19 }
```

Por último, vemos como es el uso de arrays de tipos básicos. Anteriormente se ha enumerado los tipos de array a los que JNI nos da soporte. Los arrays (`jxxxArray`) se convertirán a arrays de tipo C (`jxxx[]`) para poder ser utilizados con el método:

```
   <TipoPrimitivo> * Get<Tipo>ArrayElements
(JNIEnv *env, <TipoArray>array, jboolean *isCopy);
```

Se recibe un puntero al array del tipo nativo que corresponde el de entrada. También se dispone de un método para el proceso inverso de convertir un array de elementos (`jxxx[]`)de un tipo nativo a un tipo array (`jxxxArray`):

```
<TipoArray> New<Tipo>Array(JNIEnv *env, jsize len);
```

También se dispone de dos métodos para copiar de un tipo a otro seleccionando una región pero se requiere que se haya reservado el espacio del array destino previamente, estos métodos son:

```
void (Get|Set)<Tipo>ArrayRegion(JNIEnv *env,<TipoArray>array,
jsize start, jsize len, <TipoPrimitivo> *buffer);
```

`Get` copia de tipo array (`jxxxArray`) a array nativo (`jxxx[]`), y `Set` de array nativo a tipo array.

Con todo esto se tienen las herramientas suficientes para utilizar JNI en la integración de C/C++ en Java teniendo en cuenta los requisitos vamos a tener descritos en la Sección 4.3.

Capítulo 3

Análisis

Si quieres ser sabio, aprende a interrogar razonablemente, a escuchar con atención, a responder serenamente y a callar cuando no tengas nada que decir.

Johann Kaspar Lavater

Resumen: En este capítulo se describe el análisis para la detección de ritmo de una pista de audio. Se presentan dos modalidades, una directa y otra a través de la fragmentación en bandas de frecuencia.

Para realizar el análisis se ha tomado como guía el manual (Patin, 2003). En la explicación del análisis se asume que los audios están en modo *stereo*, por lo que disponen de canal izquierdo y derecho, y tienen de frecuencia de muestreo 44.100 Hz, esta frecuencia no corresponde con la frecuencia de onda, sino que está relacionada con el número de muestras tomadas por segundo.

La suposición anterior no pierde generalidad si el audio se encuentra en modo mono. Este modo posee un solo canal y la transformación a dos canales podría realizarse duplicando el único canal o estableciendo como nulo o cero todo el segundo canal y se procede con total normalidad sin afectar a los resultados tal y como veremos a continuación. En la implementación descrita en la Sección 4.1.1 se toma la opción de tomar como ceros el segundo canal ya que simplifica la implementación de la entrada de datos.

En cuanto a la suposición de la frecuencia concreta tampoco varía el resultado. La frecuencia de muestreo de una pista puede ser reducida eliminando muestras o ampliada recuperando valores intermedios entre una muestra y otra, por medio de una interpolación o un proceso similar. Sin

embargo existe otra solución menos costosa que sería la modificación de algunas constantes en el análisis, como por ejemplo el valor de 43 instantes, que se toma y define más adelante, se puede modificar por el número de instantes que corresponden a un segundo de tiempo en la frecuencia de la pista que se esté analizando.

Se han desarrollado dos análisis basados en una estadística local del *streaming* (flujo) de la amplitud de onda del audio.

3.1. Simple Sound Energy

Una persona escucha el sonido y lo transforma en una señal eléctrica que el cerebro interpreta, esta señal tendrá más o menos energía en función del sonido escuchado. Un golpe de ritmo la persona lo detectará cuando se produzca un aumento significativo de la energía del sonido y por un periodo relativamente breve. Por esto se podría decir que el golpe de ritmo depende de un historial de cantidad de energía recibida. Obviamente no es lo mismo la energía que se recibe al escuchar rock que al escuchar una pieza suave de música clásica.

En este primer análisis consideraremos que un golpe de ritmo sucede cuando el instante de energía supera en cierta cantidad a la media de energía recibida últimamente. Formalizando un poco más esto decimos:

Sean (a_n) y (b_n) dos listas de valores que contienen la amplitud del sonido cada cierto tiempo T_e siendo (a_n) los valores del canal izquierdo y (b_n) los valores del canal derecho con ambas listas de tamaño $n = \frac{duración}{T_e}$ que corresponde con el número de muestras de que hay tomadas en la pista completa. Definimos *instante* de tiempo a la correspondencia con 1024 muestras de las listas anteriores, esto serán 23.2 milisegundos (ms) una frecuencia de muestreo de 44.100 Hz. Sea (E_m) la lista que indica la energía del audio en cada *instante*, esta energía responde a la ecuación 3.1, que es simplemente la suma de los cuadrados de las amplitudes:

$$E_i = \sum_{k=i*1024}^{(i*1024)+1024} a_k^2 + b_k^2 \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.1}$$

Tomemos ahora la media local, definida como (M_m) que es una lista de valores de la media de los 43 instantes anteriores (43 bloques de 1024 muestras son aproximadamente un segundo a 44.100 Hz). La ecuación 3.2 expresa este cálculo formalmente:

$$M_i = \frac{1}{43} * \sum_{k=i-43}^{i} E_k \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.2}$$

Siendo en la ecuación 3.2, $E_k = 0$ si $k < 0$. Con lo anterior definimos un golpe de ritmo en el instante i si la energía del dicho instante supera una cierta proporción a la media de los 43 instantes anteriores:

$$E_i > C * M_i \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.3}$$

Donde $C \in (1'0, 2'0)$. El coeficiente C depende directamente de la canción que se analice siendo por ejemplo, $C = 1'4$ un buen valor para música *tecno* o *rap*, mientras que para *rock & roll* que tiene mucho más ruido, un buen coeficiente sería $C = 1'1$. Estos valores se han obtenido mediante pruebas, son aproximados y tienen cierta aceptación. Por otro lado, el intervalo de C es bastante fácil entender que si un instante tiene menos energía que la media no será un golpe de ritmo y que exigir que duplique la media es excesivo.

Pero, seleccionar un coeficiente diferente para cada canción no es una buena solución por eso se debe hacer que el propio análisis calcule la mejor constante dependiendo de la pista de audio. Para ello podemos usar una regresión lineal de C con la *varianza muestral* de la energía en cada instante. La varianza que está definida por la ecuación 3.4 toma valores en todo el espacio real, por eso será necesario normalizar sus valores para incluirlos en un rango limitado donde poder hacer la regresión lineal de C en su intervalo.

$$V_i = \frac{1}{43} * \sum_{k=i-43}^{i} (E_k - M_i)^2 \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.4}$$

Igual que en el caso anterior se toma $E_k - M_i = 0$ si $k < 0$. De modo que la constante C ha pasado a depender del instante en que es evaluada. Se decide tomar una normalización de la varianza que la deje en el intervalo $(-200, 200)$, entonces se quiere que cuando V_i tome su valor mínimo, se la normalice y quede con valor -200, con ese valor se quiere obtener el valor máximo posible para C_i. En el caso contrario cuando la varianza alcance el valor máximo, normalizada tome valor 200 y la C_i correspondiente tome su valor mínimo. Es decir, buscamos una ecuación lineal que dependa de la normalización de la varianza y de valores entre 1 y 2 cuando recibe entre -200 y 200, de la forma $C = (A * V) + B$. Existen multitud de ecuaciones que cumplen estos requisitos, en la ecuación 3.5 se toman los valores para A y para B más aceptados.

$$C_i = (-0{,}0025714 * V_i) + 1{,}5142857 \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.5}$$

Con esto finaliza el análisis, podemos ver su implementación en la Sección 4.2.1.

3.2. Frecuency Selected Sound Energy

El problema principal del algoritmo anterior es intentar detectar variaciones significativas de energía en canciones con mucho ruido como el *rock* o el *pop*. Por eso, vamos a buscar esas variaciones en ciertas bandas de frecuencia. Así podremos dar diferente importancia a cada banda en función de lo que nos interese. Esto podemos hacerlo gracias al *Teorema de Parseval*:

Teorema 1 (Teorema de Parseval) *La Transformada de Fourier es unitaria, esto es, la suma (o la integral) del cuadrado de una función es igual a la suma (o a la integral) del cuadrado de su transformada.*

*Si la función $f(t)$ es continua y siendo $\mathcal{F}(\alpha)$ la CFT (*Continuous Fourier Transform*, Transformada Continua de Fourier) de $f(t)$.*

$$\int_{-\infty}^{\infty} |f(t)|^2 dt = \int_{-\infty}^{\infty} |\mathcal{F}(\alpha)|^2 d\alpha \tag{3.6}$$

*Para una muestra discreta x_n, tomando X_n como la DFT (*Discrete Fourier Transform*, Transformada Discreta de Fourier) de dichas muestras:*

$$\sum_{k=0}^{n-1} |x_k|^2 = \frac{1}{n} \sum_{\alpha=0}^{n-1} |X_\alpha|^2 \tag{3.7}$$

Entonces, según el teorema no perdemos información de la energía de la canción por lo que podemos continuar con el análisis. Teníamos (a_n) y (b_n) dos listas de valores que contienen la amplitud del sonido cada cierto tiempo T_e siendo (a_n) los valores del canal izquierdo y (b_n) los valores del canal derecho, con ambas listas de tamaño $n = \frac{duración}{T_e}$ que corresponde con el número de muestras de que hay tomadas en la pista completa. Llamaremos (c_n) a los valores complejos tales que $c_k = a_k + ib_k$. Entonces aplicando la FFT (*Fast Fourier Transform*, Transformada Rápida de Fourier) cada 1024 muestras de (c_n) tenemos un espectro de frecuencias que dividimos en $\mathcal{B}$ bandas. Cuantas más bandas se usen más sensible será el análisis. La energía por cada banda se obtiene como se expresa en la ecuación 3.9, esto es, el módulo al cuadrado del espectro de frecuencias obtenido dividido en cada banda. Siendo $s < \mathcal{B}$ el número de la banda las ecuaciones quedan del siguiente modo:

$$K_i = [c_{i*1024}, \ldots, c_{(i+1)*1024-1}] \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.8}$$

$$E_i^s = \frac{\mathcal{B}}{1024} * \sum_{k=i*\mathcal{B}}^{(i+1)*\mathcal{B}} |(FFT(K_i))[k]|^2 \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.9}$$

Para cada banda $s < \mathcal{B}$ se calcula la media local:

$$M_i^s = \frac{1}{43} * \sum_{k=i-43}^{i} E_k^s \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.10}$$

Y definimos el golpe de ritmo en los instantes i en que:

$$E_i^s > C * M_i^s \qquad \forall i \in (0, \frac{n}{1024}) \tag{3.11}$$

Estos últimos cálculos corresponden con las ecuaciones 3.2 y la ecuación 3.3 sólo que se generaliza para cada banda.

La constante C puede volver a ser calculada con la varianza muestral en cada banda siguiendo la ecuación 3.4 del mismo modo que en el caso anterior.

Por tanto tenemos, $\mathcal{B}$ listas ($beat_i^s$) indicándonos si la banda s tiene un golpe de ritmo en el instante i o no. Para convertir estos *beats* locales a una banda en *beats* globales a todo el audio, se precisa tomar la decisión de cuándo ha de considerarse que es un golpe de ritmo global para ello lo ideal es ponderar cada banda con un valor que marque la importancia de dicha banda y si en un instante se obtiene un cierto porcentaje de bandas con un golpe de ritmo se dice entonces que el *beat* es global.

Capítulo 4

Implementación

La idea detrás de los computadores digitales puede explicarse diciendo que estas máquinas están destinadas a llevar a cabo cualquier operación que pueda ser realizada por un equipo humano.

Alan Turing

Resumen: En este capítulo vemos como se ha llevado a cabo la implementación de este análisis por medio de dos métodos diferentes y como se han incorporado ambos métodos a una aplicación Android que permite ejecutar el análisis en un dispositivo móvil.

4.1. Entrada y salida del análisis

Todo análisis presenta la estructura básica de Entrada-Algoritmo-Salida, en el primer paso se tratan los datos para amoldarlos a las condiciones del análisis, en el segundo se realiza propiamente dicho análisis y en el tercero se trata la respuesta para poder mostrar los resultados de una manera amigable y legible.

4.1.1. Entrada de datos

El primer paso cuando se quiere realizar cualquier análisis es obtener los datos sobre los que se va a trabajar, en este caso los datos se obtienen de un archivo de sonido. Este archivo puede estar en cualquier formato ya sea con un algoritmo de compresión o en su estado "puro".

Para el análisis asumimos que la entrada de audio son dos buffers con la información relativa a la amplitud de la onda de audio en cada muestra, el primer buffer referente a un canal izquierdo y el segundo al derecho, asumiendo también que van a una frecuencia de 44.100 Hz.

Dadas estas precondiciones, se han estudiado los formatos más comunes dos de ellos descritos en la Sección 2.1.1. De este estudio se plantean diferentes alternativas para realizar la entrada al análisis, una de ellas fue implementar un lector de archivos de audio que soportase al menos la lectura de WAV y MP3, pero fue descartada llegando a tomar la decisión de utilizar una librería de audio. Después de buscar entre varias opciones se eligió la llamada "FMODex"[1], que facilita la lectura y reproducción de los archivos, principalmente por estar disponible para la mayoría de SO incluido Android. Además, a esta librería se le puede pedir que devuelva la información del audio cumpliendo las precondiciones del análisis, como son el número de canales.

4.1.2. Salida del análisis

Como se ve más adelante en la Sección 4.2, el resultado del análisis es un buffer donde se nos indica si un determinado instante tiene un golpe de ritmo o no. Para visualizar el resultado del análisis se presentan tres formas diferentes.

La primera y más sencilla consiste en mostrar los resultados por consola o en un archivo, donde se muestra el milisegundo de la canción en que se produce el *beat* con un resultado por línea. Además para la consola se ofrece la posibilidad de escribir este resultado mientras se escucha la canción, viendo de este modo que se escribe el milisegundo en el momento en que se produce el *beat*. Este es el primer mecanismo que se ha utilizado para verificar la corrección del algoritmo. El archivo que se puede generar será el utilizado cuando se proceda a la integración del algoritmo en el videojuego mencionado en el Capítulo 1.

El segundo método de salida consiste en generar un archivo de audio añadiendo un sonido *beep* en el instante del golpe de ritmo. Para ello se escribe un archivo en formato WAV, descrito en la Sección 2.1.1.1. Para poder insertar este *beep*, se toma la amplitud recibida en los buffers de entrada y se modifica haciendo una media con una amplitud superior, distorsionando así la muestra original.

Este sistema resulta inequívoco a la hora de juzgar si un golpe de ritmo

[1] FMODex: librería de audio que puede encontrarse en `http://www.fmod.org/`

ha sido bien detectado o no ya que se percibe la música y el *beep* al mismo tiempo. Este sistema es utilizado en la Sección 5 al realizar una encuesta a usuarios sobre la precisión del análisis.

El último sistema de salida, implementado en la aplicación Android consiste en un destello en el instante en que se produce el *beat* en el audio que se está reproduciendo; este sistema es bastante visual para comprobar sin esfuerzo que el análisis funciona correctamente, aunque puede producir imprecisión porque las pantallas van a 60 fps (*frames* por segundo). Se propone este sistema para ver los resultados en dispositivos móviles tras integrar el algoritmo en ellos a través del JNI que facilita el NDK de Android, tal como se detalla más adelante en la Sección 4.3.

4.2. Implementación de los algoritmos de análisis

En el Capítulo 3 se describió de manera teórica cómo se obtienen los golpes de ritmo para una canción cualquiera con dos métodos diferentes; en esta sección se explica como se ha llevado a cabo la implementación de ambos métodos. El análisis ha sido implementado en C++ e integrado en Android con ayuda de NDK.

Aunque inicialmente se iban a realizar las implementaciones para un análisis en tiempo real, es decir, durante la reproducción de la pista de audio, dada la eficiencia de las siguientes implementaciones, no se ha visto necesario llevar a cabo dicha implementación. Además se puede observar, más adelante, que para llevar a cabo un análisis completamente en tiempo real, lo único que habría que hacer es guardar de manera temporal las últimas 43 medidas de energía para poder decidir si en el instante actual hay un golpe de ritmo o no.

Ambos análisis, una vez se ha calculado las energías tienen todo el procedimiento en común menos la finalización. Este procedimiento puede verse en pocas lineas, los pasos son sencillos, el primero es calcular la media y posteriormente la varianza, esta varianza se normaliza para que pertenezca a un intervalo que se ha decidido sea de -200 a 200:

Función 4.1: Cálculo de los beats

```
1 void process(float* energia, int* beat, int length)
2 {
3   float C = 1.4;
4   float * media = new float[length];
5   float * varianza = new float[length];
6   calculoMedia(energia, media, length);
7   calculoVarianza(energia, media, varianza, length);
8   normalizar(varianza,length,200);
```

```
9     for(int i=0; i < length; i++)
10    {
11      C = (-0.0025714f * varianza[i]) + 1.5142857f;
12      beat[i] = (energia[i] > C * media[i]) ? 1 : 0 ;
13    }
14    delete media;
15    delete varianza;
16  }
```

Donde la función `calculoMedia` recibe un buffer de energía, su tamaño y un puntero al buffer donde se guardarán las medias:

Función 4.2: calculoMedia

```
1   void calculoMedia(float* energia, float* media, int length)
2   {
3     float suma_43=0.f;
4     // Iniciar cálculo de las medias
5     for(int i=0 ; i<43 ; i++)
6     {
7       suma_43 += energia[i];
8       media[i]=suma_43/43.f;
9     }
10    // para los demás
11    for(int i=43 ; i<length ; i++)
12    {
13      suma_43 -=  energia[i-43] + energia[i];
14      media[i] = suma_43/43.f;
15    }
16  }
```

La función `calculoVarianza`, para el cálculo de C (ecuación 3.5), recibe las energías, las medias, su tamaño y un puntero al buffer donde se guardarán las varianzas:

Función 4.3: cálculo de la varianza

```
1   void calculoVarianza(float* energia, float* media,
2                        float* varianza, int length )
3   {
4     float suma_43 = 0.f;
5     // Iniciar calculo de las varianzas
6     for(int i=0 ; i<43 ; i++)
7     {
8       suma_43 +=  (energia[i]-media[i])*(energia[i]-media[i]);
9       varianza[i]=suma_43/43.f;
10    }
11    // para los demas
12    for(int i=43 ; i<length ; i++)
13    {
14      suma_43 -= (energia[i-43]-media[i-43])
```

```
15                  *(energia[i-43]-media[i-43]);
16      suma_43 += (energia[i]-media[i])*(energia[i]-media[i]);
17      varianza[i] = suma_43/43.f;
18    }
19  }
```

Como puede observarse, en ambos casos se realiza una optimización, respecto a la ecuaciones 3.2 y 3.4, cuando al calcular la siguiente suma simplemente se resta el valor más antiguo y se suma el nuevo, sin volver a sumar todo.

4.2.1. Simple Sound Energy

Para el primer análisis el cálculo de las energías es muy sencillo siguiendo la ecuación 3.1. Con esas energías ya calculadas se procede a llamar a la función `process` (función 4.1). Recibiendo en el puntero *beat* un array de enteros (para la compatibilidad con C), con un 1 en los instantes en que se ha producido un golpe de ritmo y un 0 en los que no, lo que constituye directamente la salida del análisis, lista para ser utilizada por alguno de los métodos de presentación de la salida descritos en la Sección 4.1.2. A la hora de utilizar estos resultados hay que tener en cuenta que un golpe de ritmo puede durar algo más de un *instante* (23.2 ms), por tanto se debe controlar cuando fue el último recibido y dejar un pequeño épsilon, es decir, un pequeño intervalo de tiempo sin *beat*. Si se van a contar los *beats*, se deben contar por bloques continuos y no individualmente.

4.2.2. Frecuency Selected Sound Energy

Este análisis no es tan directo como el anterior pues requiere un tratamiento especial con la FFT la cuál se ha utilizado la versión de "Cooley" y "Tukey"[2].

El cálculo de la energía en este caso, sigue la ecuación 3.9. Tenemos entonces la energía de cada banda en cada instante. Y suponemos definido el número de bandas en `N_BANDAS`. Se calculan los golpes de ritmo de cada banda con la función `process` (Función 4.1) y se ha decidido de manera genérica contar el número de bandas que han tenido un golpe y para decidir si en un instante ha habido un *beat* o no a nivel general este número debe ser superior a un tanto por ciento, aproximadamente 60 % obtenido tras diferentes pruebas.

```
1  for(int i = 0; i < N_BANDAS; i++){
2    process(energia[i],beat_banda,length);
```

[2]Podemos encontrar una implementación para C fácilmente adaptable a C++, que en `http://en.literateprograms.org/Cooley-Tukey_FFT_algorithm_%28C%29` (Última visita, Junio 2015). Conviene notar que la función `fft_simple` deja *leaks* de memoria que se deben arreglar.

```
3     for(int j = 0; j < length; j++){
4       if( i == 0) cuenta[j] = 0;
5       cuenta[j] = beat_banda[j]>0 ? cuenta[j]+1: cuenta[j];
6     }
7   }
```

Con todo esto tenemos finalizado el algoritmo.

4.3. Aplicación Android

Para probar los análisis en dispositivos móviles se desarrolla una aplicación en Android que se ve dividida en tres partes que corresponde con cada uno de los lenguajes en que se han programado.

La primera parte corresponde propiamente a la aplicación y su interfaz, el usuario puede interaccionar con ella elegir la pista de audio que quiere analizar y el método por el cuál quiere que sea analizada. Esta interfaz está realizada con el SDK de Android y presenta la apariencia que se ve en la figura 4.1. Para ello es necesario crear un `Activity` principal con un `TextArea`, un rectángulo coloreable, un par de `radio Buttons` y otro par de botones.

El `Activity` comienza dando funcionalidad a los botones, uno de ellos para buscar el archivo a analizar y el otro para arrancar el análisis. El primero lanzará un `Intent` al SO solicitando abrir una aplicación que permita seleccionar archivos, esta aplicación variará en función de la versión de Android y de las aplicaciones instaladas por el usuario. La respuesta de este `Intent` se recogerá en la aplicación cuando vuelva a estar en primer plano y puede variar mucho el formato de respuesta de la aplicación, dado que no se ha establecido un estándar para ello. Lo más común es devolver un objeto de tipo *file* con la información del archivo y donde encontrarlo, aunque pueden devolver simplemente la ruta del fichero en una cadena de caracteres u otros formatos no controlados por la aplicación. Al final de esta acción tendremos la ruta de nuestra canción guardado y listo para ser abierto. El segundo botón lanzará la ejecución del análisis escrita en código nativo, para ello hemos debido notificar al `Activity` la existencia de dos funciones nativas con el formato:

```
1   public native Integer[ ] BeatDetectetosES(String path);
2   public native Integer[ ] BeatDetectetosFE(String path);
```

Y la librería donde encontrarlo:

```
1   static {
2     System.loadLibrary("Native");
3   }
```

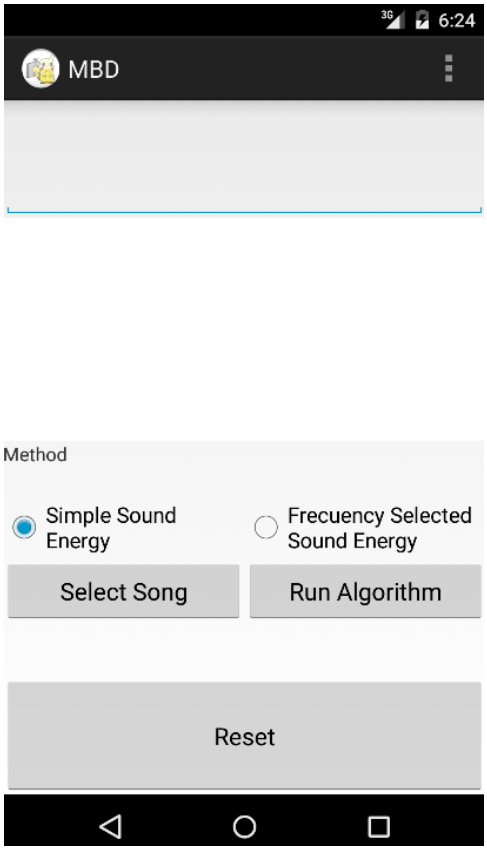


Figura 4.1: Interfaz de la Aplicación

Simplemente con estas líneas, la aplicación conoce por donde acceder al código que tengamos escrito en la librería que hemos denominado `Native`. Esta librería, escrita en C, se encarga de hacer de punto de entrada de los algoritmos de detección escritos en C++ tal como se han descrito en la Sección 4.2. En nuestro caso las cabeceras quedan de la siguiente manera:

```
1  #include <jni.h>
2
3  JNIEXPORT jintArray
4      JNICALL Java_com_friker_mbd_MainActivity_BeatDetectES(
5                                                 JNIEnv *env,
6                                                 jobject obj,
7                                                 jstring path_java
8                                               );
9
10
11 JNIEXPORT jintArray
12     JNICALL Java_com_friker_mbd_MainActivity_BeatDetectFE(
```

```
13                                                  JNIEnv *env,
14                                                  jobject obj,
15                                                  jstring path_java
16                                                );
```

Tal y como se explica en la Sección 2.2.1 estas cabeceras indican la clase que accederá a ellas y el nombre por el cuál pueden ser llamadas. Dado que la clase con acceso a estos métodos tiene la siguiente ruta de paquetes en Java `com.friker.mbd.MainActivity` Y las funciones se llaman `BeatDetectES` y `BeatDetectFE` respectivamente, tal y como se ha indicado antes.

En estos métodos escritos en C, se recibe un `string` de Java que indica la ubicación del archivo a analizar. Este string ha de pasarse a una cadena de caracteres al estilo C, para ello JNI nos facilita unos cuantos métodos en función de la codificación (UTF-8, ASCII, etc.) que se vaya a utilizar.

Desde estos métodos se llama a su correspondiente algoritmo escrito en C++ que, en principio, no presenta ninguna modificación especial para ser incorporado, aunque sus cabeceras deben estar declaradas como externas para que el código C pueda acceder a ellas. La librería "FMODex" nos devuelve los buffers de la amplitud del sonido en dos canales, y la longitud de la canción con lo que procede el análisis tal y como se ha descrito en la Sección 4.2.

Después de ejecutar el análisis tenemos un puntero a un array de enteros de tipo C++, con valores 1 o 0, compatibles con C que será transformado en tipo `jintArray` en el código C que llamó a nuestro análisis para proceder a devolver el resultado a la aplicación Java.

De vuelta en la aplicación, procedemos a reproducir el audio solicitándoselo al reproductor del propio Android[3]y lanzamos una hebra que pregunte por qué segundo va el audio y encienda o apague según corresponda el recuadro destinado a tal tarea.

Con esto se concluye la aplicación, para más detalles sobre la implementación consultar el Apéndice A.

[3] *Media Player*: Reproductor integrado en Android que funciona como una máquina de estados. Podemos encontrar su documentación en el siguiente enlace: `http://developer.android.com/reference/android/media/MediaPlayer.html`

Capítulo 5

Precisión

El hombre acepta sin problemas que una máquina corra más que él. Pero, difícilmente aceptará, que piense mejor que él.

Mijaíl Tal

Resumen: En este capítulo se pretende realizar un estudio de la precisión de los análisis desarrollados.

Los dos análisis que se han explicado en el Capítulo 3 necesitan ser medidos en cuanto a precisión, la única manera de realizar esta medición de modo automatizado es tomando archivos MIDI y comprobando que el golpe de ritmo se produce en el instante correcto del compás, dado que este tipo de archivos almacena esta información por ser un archivo centrado en instrumentos musicales, pero como nuestro objetivo va más allá de este tipo de archivos el método no es suficiente.

Una primera fase del análisis de precisión consiste en la comprobación manual y visual de la misma, realizando pequeñas variaciones en las constantes y condiciones del análisis hasta obtener un resultado suficientemente bueno para un número considerable de pistas de audio, las cuales se iban eligiendo notoriamente más complicadas para depurar el análisis.

Una vez se finaliza esta fase, se procede a realizar una verificación de los resultados por medio de una encuesta a una población no condicionada con el proyecto. La encuesta se realiza tal y como se describe en las secciones siguientes.

Figura 5.1: Captura de la encuesta realizada

5.1. Situación poblacional y del material

Se ha procedido a realizar una encuesta a una población genérica de 16 a 50 años para evaluar la precisión de los análisis desarrollados en este trabajo (Sección 4.2). La encuesta se ha realizado de manera *online* a través de una web desarrollada expresamente para este propósito que, atendiendo a la comodidad de los encuestados, tiene un diseño compatible con todo tipo de dispositivos móviles y ordenadores, tal y como puede apreciarse en la figura 5.1.

En la encuesta se presentan tres canciones con diferentes características a las que se le han aplicado ambos algoritmos y marcado los resultados de manera audible tal como se indica en la Sección 4.1.2. Por tanto, la encuesta dispone de un reproductor de audio con nueve archivos A, A_1, A_2, B, B_1, B_2, C, C_1 y C_2 que explicamos a continuación.

La canción A es *"Lucid"* de *"Dirty Doering"* que pertenece al género tecno, con ritmos muy marcados y golpes continuos. Por ser música tecno, no presenta ruidos fuera de la pista, esta generada por un ordenador y no hay interferencias de otros sonidos. Lo podríamos llamar una pista limpia y perfecta para encontrar los golpes de ritmo.

La canción B es *"Oh darling"* de *"The Beatles"* interpretada con ukelele por una conocida. Esta canción, aparte de ser más lenta que la anterior está interpretada con un instrumento de cuerda y el audio presenta ruidos por ser una grabación no profesional.

La canción C es *"Tanta Tinta Tonta"* de *"Estopa"*. Utilizan instrumentos de cuerda con mucho rasgueo pero se trata de una grabación profesional de estudio. Pertenece al género del Rock, por lo que hay una amplia presencia de sonido instrumental.

Si el nombre de la pista de audio termina por "_1" significa que ha sido analizado por el algoritmo "Simple Sound Energy" descrito en la Sección 4.2.1 y que termine por "_2" indica que ha sido analizado por el algoritmo "Frecuency Selected Sound Energy" descrito en la Sección 4.2.2.

A los encuestados se les pide escuchar los nueve audios, y responder que porcentaje de acierto creen que ha habido en las seis pistas modificadas con una marca sonora generada como se describe en la Sección 4.1.2 en los *beats* detectados por los diferentes algoritmos.

5.2. Hipótesis de la encuesta

Al realizar la encuesta contamos con una hipótesis preliminar que queremos verificar o validar al termino de la misma. Esta hipótesis está basada en la descripción de las pistas propuestas para la encuesta y en el conocimiento interno del algoritmo, así como su prueba previa.

El resultado que se espera obtener en esta encuesta para las diferentes pistas de audio dependen completamente de cada pista y cada algoritmo, por lo general se espera un mejor resultado del primer algoritmo que del segundo, dado que, como se explica en la Sección 4.2.2, el segundo no presenta un cierre del algoritmo suficientemente depurado, sino que presenta uno genérico para dar un buen resultado promedio.

Más detalladamente, podemos esperar de la primera canción de *"Dirty Doering"* una aceptación elevada por parte de los encuestados en ambos algoritmos, con un porcentaje muy alto en el primero por ser una canción tan limpia y sencilla que no tiene ruidos y toda la melodía está repartida por todo el espectro de frecuencias de manera más o menos uniforme, por lo que una separación en bandas como hace el segundo algoritmo resulta innecesaria. El segundo algoritmo también debería obtener una alta aceptación por los mismos motivos aunque menor por lo ya explicado.

Para la segunda canción, que recordamos presenta mucho ruido, es una grabación casera y suena principalmente un ukelele. Esta canción se espera que no de buenos resultados y que no sea muy ampliamente aceptada ya que los instrumentos de cuerda no suelen producir buenos resultados dado que confunden al algoritmo, por ejemplo en los momentos en que se produce un cambio de acorde sin realizar un rasgueo, el nuevo acorde continúa con la vibración que restaba al acorde anterior.

Por último, la tercera canción, la canción de Estopa que es Rock, como decíamos presenta mucha guitarra. Por ello se espera obtener un resultado no del todo correcto pero mucho más acertado que la pista anterior dado que el audio es de mayor calidad. Y al elegir entre el primer o segundo algoritmo, el segundo podría dar mejores resultados por ser Rock, pero no se tiene especial esperanza en ello.

Por tanto si ordenamos por orden de precisión esperada será:

$$A_1 > C_1 \geq C_2 > B_1 \geq A_2 > B_2 \tag{5.1}$$

5.3. Resultados y conclusiones

El resultado de la elaboración de la encuesta ha sido satisfactorio por los datos obtenidos que como se detalla más adelante verifican casi en totalidad la hipótesis desarrollada en la Sección 5.2.Además, en un breve periodo de tiempo se encuesto a una población bastante amplia. Se han encuestado a 57 personas de diferentes ámbitos en los que se incluyen: informáticos que conocen el funcionamiento de los algoritmos, músicos con un oído acostumbrado a los tempos y ritmos, y otros perfiles como sería profesores de colegio e institutos y estudiantes de diferentes niveles por lo que se puede garantizar que la encuesta no ha resultado sesgada por un tipo de población concreta.

Metiéndonos en los resultados obtenidos vemos que la hipótesis se ha cumplido casi totalmente. Adjuntamos una pequeña muestra resumida de las respuestas (los valores corresponden a un porcentaje de precisión) donde se puede observar que hay algunos resultados que generan un poco de ruido, se ha intentado compensar el ruido realizando la encuesta a un número elevado.

En la tabla también observamos la media por canción teniendo en cuenta el total de encuestas realizadas. Que ordenándolo queda:

$$A_1 > B_1 > C_1 > A_2 > C_2 > B_2 \tag{5.2}$$

#	A_1	A_2	B_1	B_2	C_1	C_2
1	99	80	98	98	99	89
2	80	60	80	80	30	50
3	98	0	70	0	90	50
4	85	60	25	10	60	85
5	97	85	60	10	90	95
6	92	100	85	80	98	95
7	100	90	80	5	100	10
8	98	90	80	30	70	60
9	100	80	70	10	90	80
10	80	90	55	30	50	35
11	98	85	60	60	80	75
12	97	85	60	10	90	95
13	75	80	45	1	40	60
14	96	75	88	98	88	98
15	90	70	70	20	75	80
16	95	90	90	80	60	70
17	95	90	70	50	70	60
18	90	50	100	0	50	50
19	75	44	32	12	54	34
20	75	80	35	50	60	80
...	...	...	...	...	...	...
Media	91.47	69.52	72.40	40.70	70.14	66.86

Rápidamente vemos que el primer algoritmo, descrito en la Sección 3.1, obtiene mejores resultados que el descrito en la Sección 3.2; esto se debe a que la implementación realizada no ha utilizado todo el potencial tal y como advertíamos en la Sección 4.2.2 y al comienzo de este capítulo.

Aunque presenta un desorden el resultado obtenido respecto a la hipótesis definida, podemos decir que se han cumplido las expectativas ya que se preveía todo igual, salvo B_1 que se esperaba por debajo de A_2 pero viendo los resultados numéricos la diferencia entre ellos es apenas un 3 %.

Capítulo 6

Conclusiones y trabajo futuro

He notado que aun la gente que dice que todo está predestinado y que no podemos hacer nada para cambiar nuestro destino, mira antes de cruzar la calle.

Stephen Hawking

Resumen: Terminamos el trabajo con unas conclusiones sobre la elaboración de dicho trabajo, una breve introducción a lo que serán los próximos pasos para completar este trabajo y una valoración personal de lo que ha supuesto realizar este proyecto.

6.1. Conclusiones

Tras finalizar el trabajo se obtienen las conclusiones siguientes derivadas del análisis de los resultados obtenidos y desarrollados en este documento.

Empezando por el tema central del trabajo, los análisis desarrollados resultan satisfactorios y presentan la compatibilidad suficiente con el proyecto que se quería desarrollar. Entre ambos análisis se elige inicialmente el más sencillo, *Simple Sound Energy*, descrito en la Sección 3.1, dado que al no realizar la FFT sobre el muestreo del audio obtiene unos resultados en tiempo mucho mejores que el segundo, además como se desarrolla en la Sección 5.3 las diferencias de precisión no son suficientemente notorias como para utilizar todos los recursos que requiere el análisis de la Sección 3.2, pero la toma de la decisión final se aplaza en espera de realizar un estudio para mejorar la finalización del segundo análisis tal y como se describe en la siguiente Sección 6.2.

Por otro lado, la integración de los análisis a la aplicación Android utilizando código nativo para ello ha resultado satisfactoria, por ello, a falta de comparar tiempos con un análisis realizado en la misma aplicación con Java, se da por buena esta metodología para incorporar el análisis seleccionado al videojuego.

Por tanto, se considera satisfactorio el trabajo y los resultados expuestos en este documento.

6.2. Trabajo futuro

Se ve ahora el modo en que se pretende continuar este trabajo con diferentes acciones que llevarán a un resultado todavía más depurado y preparado para la integración con el videojuego que se contaba en el Capítulo 1.

- *Mejoras*: Se prevé mejorar la finalización del algoritmo "Frecuency Selected Sound Energy", actualmente cuenta las bandas que han producido un golpe de ritmo tal y como se describe en la Sección 4.2.2, esta finalización es injusta con la capacidad del algoritmo. A su vez, se quiere optimizar el uso de recursos evitando precalcular y almacenar tantos datos como se hace en los análisis actuales de modo que se compute según se va necesitando ahorrando así memoria que es uno de los principales requisitos de los dispositivos móviles.

- *Tiempos*: Se quiere medir la eficiencia de los algoritmos con respecto al tiempo para tomar una decisión de incorporación de un algoritmo u otro. También se plantea la posibilidad de comparar una implementación para la parte nativa de Android, como la explicada en este trabajo, con una posible implementación en código Java.

- *Integración*: Como ya es de esperar un paso seguro es la integración del análisis en el videojuego; esto conlleva conectar la parte nativa de Android con *Unity* que es la plataforma donde esta desarrollado el videojuego, o si los tiempos favorecen incorporar el futuro desarrollo en Java del análisis con el videojuego.

6.3. Valoración personal

La elaboración de este trabajo ha pasado por diferentes etapas, unas más agradables y otras de peores condiciones, si tuviese que resumir todo el trabajo lo haría con algunas de las frases que están por el trabajo dado que han sido muy significativas para mí a la hora de ir elaborándolo. El trabajo comenzó cuando me disponía a elegir que hacer en ese momento se me propuso realizar un análisis de pistas de audio para sacar el ritmo y meterlo

en un móvil. No sabia de música, no sabía de ondas, no sabía de Android pero acepté el reto, este es el motivo por el que encabezo el trabajo con la frase de *Franklin D. Roosevelt*: "Siempre que te pregunten si puedes hacer un trabajo, contesta que sí y ponte enseguida a aprender cómo se hace."

Estoy verdaderamente contento de haber llegado a un buen resultado que lo he podido obtener gracias a todas las personas mencionadas al principio y a todo lo aprendido en ambas carreras. Informática me ha enseñado a producir con cabeza y Matemáticas a persistir ante lo desconocido o extraño, aparte de los conocimientos teóricos.

El trabajo continuó con muchas dificultades, y todo se vio marcado por la frase de *San Agustín*: "Es mejor cojear por el camino que avanzar a grandes pasos fuera de él. Pues quien cojea en el camino, aunque avance poco, se acerca a la meta, mientras que quien va fuera de él, cuanto más corre, más se aleja." Aunque a veces aparecía fuera del camino siempre encontraba la manera de volver a cojear dentro de él.

Apéndice A

Código implementado

En todas las cosas, naturales y humanas,
el origen es lo más excelso.

Platón

Resumen: Adjuntamos en este apéndice los archivos de la implementación más importantes del análisis y la aplicación, por si sirven de referencia para futuros trabajos.

A.1. Introducción

En este apéndice se incorpora la implementación concreta de la aplicación Android. Tal y como se describe en la Sección 4.3 tiene tres partes diferenciadas. No se incluye la definición de los ".h" que tienen los prototipos de las funciones dado que nos interesa más los archivos de la implementación explícita. En la Tabla A.1, podemos observar el número de líneas de código que se han escrito para la aplicación.

Lenguaje	Archivos	Comentarios	Código
Cabeceras C/C++	1	0	19
C++	1	10	168
C	1	0	18
Java	1	11	213
XML	4	0	126
SUMA:	8	21	544

Tabla A.1: Número de líneas de código de la aplicación Android

A.2. Android

En la parte de Android tenemos lo que corresponde a la vista y al controlador, siendo la vista el archivo `activity_main.xml` y el controlador el archivo `MainActivity.java`.

A.2.1. activity_main.xml

```
<LinearLayout
 xmlns:tools="http://schemas.android.com/tools"
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:layout_width="match_parent"
 android:layout_height="match_parent"
 android:orientation="vertical"
 tools:context="com.friker.mbd.MainActivity" >

    <EditText
        android:id="@+id/editText1"
        android:layout_width="match_parent"
        android:layout_weight="1"
        android:layout_height="0dp"
        android:ems="10" >
     </EditText>

    <View
        android:id="@+id/Rectangle"
        android:layout_width="match_parent"
        android:layout_height="50dp"
        android:layout_weight="1.33"
        android:background="#FFFFFFFF" />

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="0dp"
        android:layout_weight="1"
        android:orientation="vertical" >

        <TextView
            android:layout_width="match_parent"
            android:layout_height="0dp"
            android:layout_weight="1"
            android:text="@string/methods" />

        <RadioGroup
          android:id="@+id/radioGroup1"
          android:layout_width="match_parent"
          android:layout_height="0dp"
          android:layout_weight="1"
```

```
41              android:orientation="horizontal" >
42
43              <RadioButton
44                  android:id="@+id/ES"
45                  android:layout_width="0dp"
46                  android:layout_height="wrap_content"
47                  android:layout_weight="1"
48                  android:checked="true"
49                  android:text="Simple Sound Energy " />
50
51              <RadioButton
52                  android:id="@+id/FE"
53                  android:layout_width="0dp"
54                  android:layout_height="wrap_content"
55                  android:layout_weight="1"
56                  android:text="Frecuency Selected Sound Energy"
57              />
58          </RadioGroup>
59
60      </LinearLayout>
61
62      <LinearLayout
63          android:layout_width="match_parent"
64          android:layout_height="0dp"
65          android:layout_weight="1" >
66
67          <Button
68              android:id="@+id/button1"
69              android:layout_width="match_parent"
70              android:layout_height="wrap_content"
71              android:layout_weight="1"
72              android:text="Select Song" />
73
74          <Button
75              android:id="@+id/button2"
76              android:layout_width="match_parent"
77              android:layout_height="wrap_content"
78              android:layout_weight="1"
79              android:text="Run Algorithm" />
80      </LinearLayout>
81
82      <Button
83          android:id="@+id/button3"
84          android:layout_width="match_parent"
85          android:layout_height="0dp"
86          android:layout_weight="1"
87          android:text="Reset" />
88
89  </LinearLayout>
```

A.2.2. MainActivity.java

```
1  package com.friker.mbd;
2
3  import android.app.Activity;
4
5  public class MainActivity extends Activity {
6    private boolean cargado;
7    private String PATH;
8    private int[] BEAT;
9    private int PICK_AUDIO = 0;
10   private FileChooserListener listener;
11   private MediaPlayer mp;
12   final private Handler handler = new Handler();
13   private int color;
14   private int count=0;
15   private int constante = 1;
16   final private Color col = new Color();
17
18   @Override
19   protected void onCreate(Bundle savedInstanceState) {
20     super.onCreate(savedInstanceState);
21     setContentView(R.layout.activity_main);
22     cargado = false;
23     PATH = "";
24     mp = new MediaPlayer();
25     color = col.WHITE;
26     Button button1 = (Button) findViewById(R.id.button1);
27     button1.setOnClickListener(new View.OnClickListener() {
28       public void onClick(View v) {
29         getSong();
30       }
31     });
32     Button button2 = (Button) findViewById(R.id.button2);
33     button2.setOnClickListener(new View.OnClickListener() {
34       public void onClick(View v) {
35         ejecutar();
36       }
37     });
38     final Button button=(Button)findViewById(R.id.button3);
39     button.setOnClickListener(new View.OnClickListener() {
40       public void onClick(View v) {
41         if(mp.isPlaying()){
42      mp.stop();
43         }
44       }
45     });
46   }
47
```

```
48   void changeText(final String str){
49     runOnUiThread(new Runnable() {
50       @Override
51       public void run() {
52         EditText tx=(EditText)findViewById(R.id.editText1);
53         tx.setText(Area.getText()+str);
54       }
55     });
56   }
57
58   void runSong(){
59     final View salida = findViewById(R.id.Rectangle);
60     try {
61       mp.reset();
62       mp.setDataSource(PATH);
63       mp.prepare();
64       mp.start();
65       constante = (int)mp.getDuration()/BEAT.length;
66       new Thread(new Runnable(){
67         @Override
68         public void run() {
69      while(mp.isPlaying()){
70        runOnUiThread(new Runnable() {
71          public void run(){
72            int cur = mp.getCurrentPosition();
73            if(BEAT[(int)cur/constante]>0){
74         if(color != col.GREEN){
75           color = col.GREEN;
76           salida.setBackgroundColor(color);
77         }
78             }else{
79         if(color != col.WHITE){
80           color = col.WHITE;
81           salida.setBackgroundColor(color);
82         }
83             }
84          }
85        });
86      }
87         }
88       }).start();
89     }
90     catch (Exception e) {
91       e.printStackTrace();
92     }
93   }
94
95   void ejecutar() {
96     EditText Area= (EditText)findViewById(R.id.editText1);
```

```
97      if(!cargado || PATH == "" || PATH == null){
98        cargado = false;
99        Area.setText("Please Select a song");
100       return;
101     }
102     RadioGroup rg=(RadioGroup)findViewById(R.id.radioGroup1);
103     int option = rg.getCheckedRadioButtonId();
104     if(option == R.id.ES){
105       BEAT = BeatDetectES(PATH);
106     }else{
107       BEAT = BeatDetectFE(PATH);
108     }
109     int c = 0;
110     for(int i = 0; i < BEAT.length;i++)
111     if(BEAT[i]>0) c++;
112     runSong();
113   }
114
115   void getSong() {
116     askForAudio(new FileChooserListener() {
117       @Override
118       public void choose(String path) {
119         EditText tx=(EditText)findViewById(R.id.editText1);
120         PATH = path;
121         if (path == null) {
122      tx.setText("Error path no found");
123      cargado = false;
124         } else {
125      tx.setText("Path: "+path);
126      cargado = true;
127         }
128       }
129     });
130   }
131
132   public void askForAudio(FileChooserListener listener) {
133     this.listener = listener;
134     Intent intent = new Intent();
135     intent.setAction(Intent.ACTION_GET_CONTENT);
136     intent.setType("audio/*");
137     intent = Intent.createChooser(intent, "sound");
138     if(intent.resolveActivity(getPackageManager()) !=null){
139       startActivityForResult(intent, PICK_AUDIO);
140     } else {
141       listener.choose(null);
142     }
143   }
144
145   private String getPathFromIntent(Intent data) {
```

```
146     Uri uri = data.getData();
147     if ("file".equalsIgnoreCase(uri.getScheme())) {
148       return uri.getPath();
149     } else {
150       return getDataColumn(uri, null, null);
151     }
152   }
153
154   private String getDataColumn(Uri uri, String selection,
155         String[] selectionArgs) {
156
157     Cursor cursor = null;
158     String column = "_data";
159     String[] projection = { column };
160     try {
161       cursor = getContentResolver().query(uri, projection,
162                           selection, selectionArgs, null);
163       if (cursor != null && cursor.moveToFirst()) {
164         int index = cursor.getColumnIndexOrThrow(column);
165         return cursor.getString(index);
166       }
167     } finally {
168       if (cursor != null) {
169         cursor.close();
170       }
171     }
172     return null;
173   }
174
175   @Override
176   protected void onActivityResult(int requestCode,
177                               int resultCode, Intent data){
178     if (requestCode == PICK_AUDIO) {
179       if (resultCode == RESULT_OK) {
180         listener.choose(getPathFromIntent(data));
181       }
182     }
183   }
184
185   interface FileChooserListener {
186     void choose(String path);
187   }
188
189   private native int[] BeatDetectES(String path);
190   private native int[] BeatDetectFE(String path);
191   static {
192     System.loadLibrary("Native");
193   }
194
```

```
195 }
```

A.3. Nativo

La parte del análisis, `BeatDetector.cpp`, esta escrita en C++ pero se conecta con la aplicación por medio del archivo `Native.c` que esta escrito en C. Aparte se incluye el archivo `Android.mk` para una correcta compilación de los archivos. Todos estos archivos deben ir en la carpeta `jni` del proyecto.

A.3.1. Native.c

```
1
2  #include <jni.h>
3  #include "BeatDetector.h"
4
5  JNIEXPORT jintArray
6  JNICALL Java_com_friker_mbd_MainActivity_BeatDetectES(
7                  JNIEnv *env, jobject obj, jstring str)
8  {
9    const char *path_c=(*env)->GetStringUTFChars(env,str,0);
10   unsigned int length;
11   int* beat = BeatDetectorEnergyS(path_c,&length);
12   //allocate
13   jintArray out = (*env)->NewIntArray(env, length);
14   // copy
15   (*env)->SetIntArrayRegion(env, out, 0 , length, beat);
16   return out;
17 }
18
19 JNIEXPORT jintArray
20 JNICALL Java_com_friker_mbd_MainActivity_BeatDetectFE(
21               JNIEnv *env, jobject thisObj,jstring str)
22 {
23   const char* path_c=(*env)->GetStringUTFChars(env,str,0);
24   unsigned int length;
25   int* beat = BeatDetectorFrequencyE(path_c,&length);
26   // allocate
27   jintArray beats = (*env)->NewIntArray(env, length);
28   // copy
29   (*env)->SetIntArrayRegion(env, beats, 0 , length, beat);
30   return beats;
31 }
```

A.3.2. BeatDetector.cpp

```
1
2
3  #include "BeatDetector.h"
4
5  unsigned int init_FMOD(const char * path,
6                         int ** Ldata, int ** Rdata){
7    unsigned int length=100;
8    FMOD_SYSTEM *system;
9    FMOD_SOUND *music;
10
11   FMOD_System_Create(&system);
12   FMOD_System_Init(system, 1, FMOD_INIT_NORMAL, NULL);
13
14   FMOD_System_CreateSound(system, path,
15               FMOD_SOFTWARE | FMOD_2D , 0, &music);
16
17   FMOD_Sound_SetLoopCount(music, -1);
18   FMOD_Sound_GetLength(music, &length, FMOD_TIMEUNIT_PCM);
19
20   void* ptr1;
21   void* ptr2;
22   unsigned int length1;
23   unsigned int length2;
24   *Ldata = new int[length];
25   *Rdata = new int[length];
26
27   FMOD_Sound_Lock(music, 0, length,
28                   &ptr1, &ptr2, &length1, &length2);
29   for (int i=0 ; i<length ; i++)
30   {
31     (*Ldata)[i] = ((int*)ptr1)[i]>>16;
32     (*Rdata)[i] = (((int*)ptr1)[i]<<16)>>16;
33   }
34   FMOD_Sound_Unlock(music, ptr1, ptr2, length1, length2);
35
36   FMOD_System_Release(system);
37   return length/1024;
38 }
39
40 int* BeatDetectorEnergyS(const char* path,
41                                 unsigned int* len){
42   int* Ldata;
43   int* Rdata;
44   int * beat;
45   unsigned int length;
46   length = init_FMOD(path,&Ldata,&Rdata);
47   *len = length;
48   beat = new int[length];
49   float* energia = new float[length];
```

```
50    for(int i = 0; i < length; i++)
51      energia[i] = energiaES(Ldata, Rdata,
52                             1024*i, 4096,length);
53    process(energia,beat,length);
54    delete energia;
55    return beat;
56  }
57
58  int* BeatDetectorFrequencyE(const char* path,
59                                  unsigned int* len){
60    int* Ldata;
61    int* Rdata;
62    int * beat;
63    unsigned int length;
64    length = init_FMOD(path,&Ldata,&Rdata);
65    *len = length;
66    beat = new int[length];
67    float** energy = new float*[length];//[N_BANDS];
68    float* energia = new float[length];
69    int* beat2;
70    beat2 = new int[length];
71    for(int i = 0; i < length; i++){
72      energy[i] = new float[N_BANDS];
73      energy[i] = energiaFE(Ldata, Rdata,
74                            1024*i, 1024,length);
75    }
76    int* cuenta;
77    cuenta = new int[length];
78    for(int i = 0; i < N_BANDS;i++){
79      for(int j = 0; j < length; j++){
80        energia[j] = energy[j][i];
81      }
82      process(energia,beat2,length);
83      for(int j = 0; j < length; j++){
84        delete energy[j];
85        if( i == 0) cuenta[j] = 0;
86        cuenta[j] = beat2[j] ? cuenta[j]+1: cuenta[j];
87      }
88    }
89    int last = -1;
90    for(int i = 0; i < length; i++){
91      if(last+0<i && cuenta[i] > N_BANDS*0.5){
92        beat[i] = cuenta[i];
93        last = i;
94      }
95    }
96    delete energia;
97    delete cuenta;
98    delete energy;
```

```
99    return beat;
100 }
101
102 float* energiaFE(int* left, int* right,
103                  int offset, int window, int length){
104   complex *fft_in;
105   complex *fft_out;
106   float energia=0.f;
107   float *B;
108   int j = 0;
109   fft_in = new struct complex_t[window];
110   B = new float[window];
111
112   // Creamos los complejos para FFT
113   for(int i=offset ; (i<offset+window)&&(i<length) ; i++){
114     fft_in[j].re = (double)left[i];
115     fft_in[j].im = (double)right[i];
116     j++;
117   }
118
119   fft_out = FFT_simple(fft_in, window);
120
121   // B tiene las amplitudes
122   for (int j=0; j<window;j++)
123     B[j] = complex_magnitude(fft_out[j]);
124   float * energy;
125   energy = new float[N_BANDS];
126   //calculamos la energia de cada banda
127   for (int j=0; j<N_BANDS;j++){
128     energia = 0.f;
129     for (int k=0; k<window/N_BANDS;k++)
130       energia += B[j*N_BANDS+k];
131     energy[j] = energia;
132   }
133   //No dejamos leaks de memoria
134   delete fft_in;
135   delete fft_out;
136   delete B;
137   return energy;
138 }
139
140 float energiaES(int* left,int* right,
141                  int offset, int window,int length){
142   float energia=0.f;
143   for(int i=offset ; (i<offset+window)&&(i<length) ; i++)
144     energia += (left[i]*left[i]+right[i]*right[i]);
145     energia = energia*1.0/window;
146     return energia;
147   }
```

```
148
149 void calculoMedia(float* energia, float* media,
150                     int length){
151   float suma_43=0.f;
152   // Iniciar cálculo de las medias
153   for(int i=0 ; i<43 ; i++){
154     suma_43 = suma_43 + energia[i];
155     media[i]=suma_43/43.0;
156   }
157   // para los demás
158   for(int i=43 ; i<length; i++){
159     suma_43 = suma_43 - energia[i-43] + energia[i];
160     media[i] = suma_43/43.0;
161   }
162 }
163
164 void calculoVarianza(float* energia, float* media,
165                      float* varianza, int length ){
166   float suma_43 = 0.f;
167   // Iniciar cálculo de las varianza
168   for(int i=0 ; i<43 ; i++){
169     suma_43 +=(energia[i]-media[i])*(energia[i]-media[i]);
170     varianza[i]=suma_43/43.0;
171   }
172   // para los demás
173   for(int i=43 ; i<length/1024 ; i++){
174     suma_43 -= (energia[i-43]-media[i-43])
175                *(energia[i-43]-media[i-43]);
176     suma_43 +=(energia[i]-media[i])*(energia[i]-media[i]);
177     varianza[i] = suma_43/43.0;
178   }
179 }
180
181 void normalizar(float* signal, int size, float max_val){
182   float max=0.f,aux = 0.f;
183   for(int i=0 ; i<size ; i++){
184     aux = signal[i]>0?signal[i]:(-1)*signal[i];
185     if (aux>max) max=aux;
186   }
187   float ratio = max_val/max;
188   for(int i=0 ; i<size ; i++){
189     signal[i] = signal[i]*ratio;
190   }
191 }
192
193 void process(float* energia,int* beat,int length){
194   float C = 1.4;
195   float * media = new float[length];
196   float * varianza = new float[length];
```

```
197     calculoMedia(energia,media,length);
198     calculoVarianza(energia, media,varianza,length);
199     normalizar(varianza,length,200);
200     for(int i=0 ; i<length; i++){
201       C = (-0.0025714f * varianza[i]) + 1.5142857f;
202       beat[i] = (energia[i]>C*media[i]) ? 1 : 0;
203     }
204   }
```

A.3.3. Android.mk

```
1  LOCAL_PATH := $(call my-dir)
2
3
4  include $(CLEAR_VARS)
5  LOCAL_MODULE := BeatDetector
6  LOCAL_SRC_FILES := BeatDetector.cpp
7  LOCAL_SHARED_LIBRARIES := fmodex complex_simple fft
8  include $(BUILD_SHARED_LIBRARY)
9
10 include $(CLEAR_VARS)
11 LOCAL_MODULE := Native
12 LOCAL_SRC_FILES := Native.c
13 LOCAL_STATIC_LIBRARIES := BeatDetector
14 include $(BUILD_SHARED_LIBRARY)
15
16 include $(CLEAR_VARS)
17 LOCAL_MODULE := fft
18 LOCAL_SRC_FILES := fft.cpp
19 LOCAL_SHARED_LIBRARIES := complex_simple
20 include $(BUILD_SHARED_LIBRARY)
21
22 include $(CLEAR_VARS)
23 LOCAL_MODULE := complex_simple
24 LOCAL_SRC_FILES := complex_simple.cpp
25 include $(BUILD_SHARED_LIBRARY)
26
27 include $(CLEAR_VARS)
28
29 LOCAL_MODULE := fmodex
30 LOCAL_SRC_FILES := \
31      ../api/lib/$(TARGET_ARCH_ABI)/libfmodex.so
32 LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/../api/inc
33
34 include $(PREBUILT_SHARED_LIBRARY)
```

Bibliografía

Y así, del mucho leer y del poco dormir,
se le secó el celebro de manera que vino
a perder el juicio.

Miguel de Cervantes Saavedra

Ellis, D. P. Beat tracking by dynamic programming. *LabROSA, Columbia University, New York*, 2007.

Gold, B., Morgan, N. y Ellis, D. *Speech and Audio Signal Processing*. Jhon Wiley and Sons, INC., 2011.

Goto, M. y Muraoka, Y. A real-time beat tracking system for audio signals. *International Computer Music Conference Proceedings*, 1995.

Liang, S. *The Java Native Interface: Programmer's Guide and Specification*. Versión electrónica, 1999.

Oliveira, J. L., Davies, M. E. P., Gouyon, F. y Reis, L. P. Beat tracking for multiple applications: A multi-agent system architecture with state recovery. , 2010a.

Oliveira, J. L., Gouyon, F., Martins, L. G. y Reis, L. P. Ibt: A real-time tempo and beat tracking system. *In Proceedings of International Society for Music Information Retrieval Conference (ISMIR)*, 2010b.

Patin, F. *Beat Detection Algorithms*. Versión electrónica, 2003.

Santiago, C. B., Oliveira, J. L., Reis, L. P. y Sousa, A. J. Autonomous robot dancing synchronized to musical rhythmic stimuli. *Conference: Information Systems and Technologies (CISTI)*, 2011.

Lista de acrónimos

3GP ..... *3rd Generation Partnership Project*

API ..... *Application Programming Interface*, Interfaz de programación de aplicaciones

BPM ..... *Beats per minute*, pulsos por minuto

CFT ..... *Continuous Fourier Transform*, Transformada Continua de Fourier

DFT ..... *Discrete Fourier Transform*, Transformada Discreta de Fourier

DVM ..... *Dalvik Virtual Machine*, Máquina Virtual de Dalvik

FFT ..... *Fast Fourier Transform*, Transformada Rápida de Fourier

IBT ..... *tempo Induction and Beat Tracker*, Inducción sobre el tempo y seguimiento del pulso

JNI ..... *Java Native Interface*, Interfaz para código nativo y Java

JVM ..... *Java Virtual Machine*, Máquina Virtual de Java

MARSYAS ... *Music Analysis, Retrieval and Synthesis for Audio Signals*

MIDI ..... *Musical Instrument Digital Interface*, Interfaz Digital de Instrumentos Musicales

MP3 ..... *MPEG Audio Layer III*

NDK ..... *Native Developer Kit*, Kit de Desarrollo Nativo

PCM ..... *Pulse Code Modulation*, Modulación por impusos codificados

SDK ..... *Software Developer Kit*, Kit de Desarrollo de Software

SO ..... *Sistema Operativo*

WAV ..... *Waveform Extensión*, Extensión de forma de onda

WMA ..... *Windows Media Audio*

–¿Qué te parece desto, Sancho? – Dijo Don Quijote –
Bien podrán los encantadores quitarme la ventura,
pero el esfuerzo y el ánimo, será imposible.

Segunda parte del Ingenioso Caballero
Don Quijote de la Mancha
Miguel de Cervantes

Es mejor cojear por el camino que avanzar a grandes pasos fuera de él.
Pues quien cojea en el camino, aunque avance poco, se acerca a la meta,
mientras que quien va fuera de él, cuanto más corre, más se aleja.

San Agustín