Práctica NPI Android - Tutoriales

Anabel Gómez Ríos y Jacinto Carrasco Castillo

February 2016

1. appFotoVoz

La aplicación tiene dos pantallas: la inicial, en la que se muestra el botón que pulsar para hablar, y la que muestra la brújula en movimiento una vez se ha dicho por voz algo en el formato "Punto Cardinal", "Tolerancia". En caso de que se pulse el botón pero no se diga algo de este tipo, se mostrará por pantalla cuál debe ser el formato:

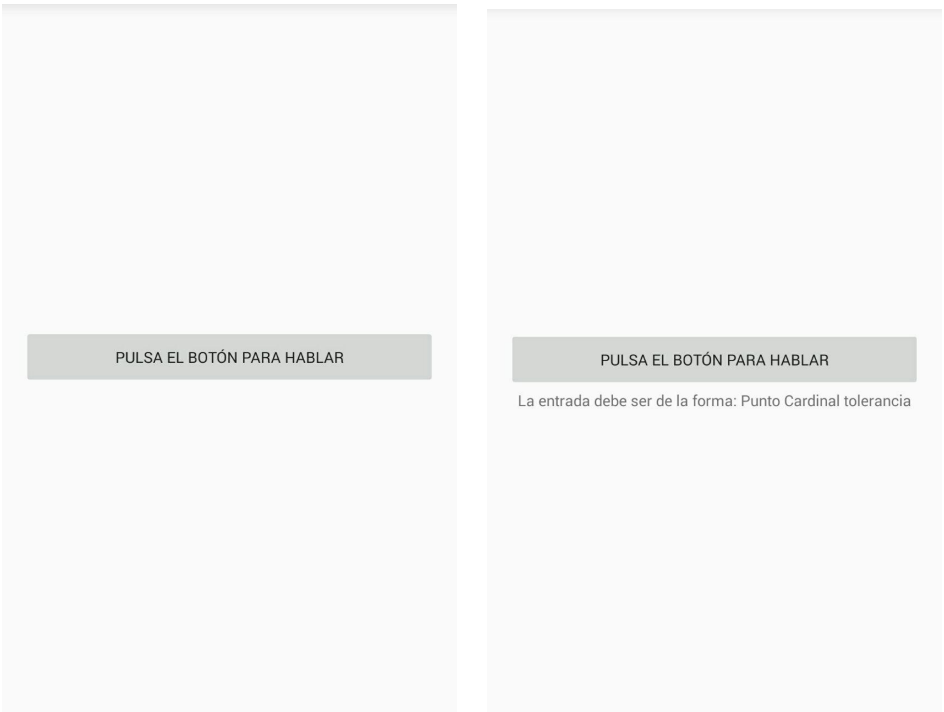

Figura 1: Primera pantalla

Esto se controla desde la clase *BrujulaVoz.java*, en el que creamos el botón en el fichero .xml y utilizamos los métodos `listen()` y `setSpeekButton()` para que se produzca el reconocimiento de voz. Para poder pasar lo que se escucha a la clase que contiene la brújula hemos tenido que hacerlo con un *Bundle*, dentro del método `sendMessage()`, ya que sólo con los *Intents* podíamos pasar *Strings* pero no floats,

que nos hacen falta para poder pasar tanto el punto cardinal (que es un *String*) como la tolerancia (que es un float).
También tuvimos el problema de que no siempre la que decía que era la mejor coincidencia para lo que se estaba diciendo por voz era realmente lo que se estaba diciendo, aunque lo que se estaba diciendo sí aparecía en la lista en un lugar más bajo (con menos confianza). Para ello lo que hacemos es coger una lista con los 10 mejores resultados y ahí buscar las cadenas que, primero, tengas dos palabras separadas por un espacio, y después, que la primera sea un punto cardinal y la otra un número. Una vez encontramos una cadena así (de no ser encontrada es cuando se muestra el mensaje que aparece en la imagen anterior) es cuando pasamos a la otra pantalla:

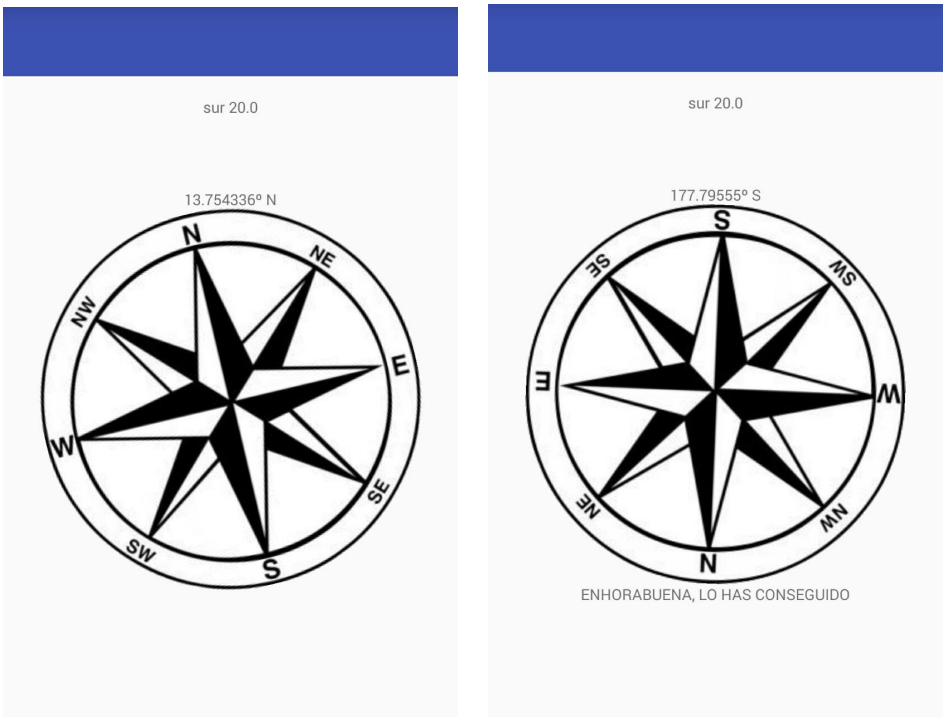

Figura 2: Segunda pantalla

En esta pantalla, como vemos, aparece arriba en un *TextView* lo que se ha reconocido que se ha dicho en la pantalla anterior por voz y por tanto, cómo tenemos que situarnos. Debajo tenemos la brújula en movimiento y encima de ella la orientación que tenemos en ese momento, que también está en continuo cambio. Una vez nos orientamos en la dirección indicada y con el margen de error indicado, aparece un mensaje debajo de la brújula indicando que ya lo hemos conseguido.

2. appGPSQR

En esta aplicación partiremos de una *Activity* donde tendremos dos botones. Pulsando el botón `New QR`, se iniciará un *Intent* para obtener el código QR. La obtención de este código la haremos a través de la aplicación *Barcode Scanner* y

usando el código de **XZING** para integrarlo. Para ello, descargaremos del repositorio oficial de XZING (https://github.com/zxing/zxing) las clases *IntentIntegrator* e *IntentResult*. En el método `onActivityResult()` se recibirá el resultado de la lectura del código QR, donde lo separaremos y añadiremos las coordenadas al *ArrayList* `coordinates`.
Este *ArrayList* será de la clase que hemos implementado *LatLong*. Esta clase implementa la interfaz *Parcelable*, para poder pasar el vector de coordenadas inicial a la siguiente *Activity* mediante un *Bundle*. *LatLong* simplemente tendrá dos *float*, que serán la latitud y la longitud. Cuando queramos mostrar el mapa para iniciar la navegación, tendremos que pulsar el botón `Navigate`, entonces se pasa a la *Activity Map* un *Bundle* con las coordenadas que hemos ido leyedo.

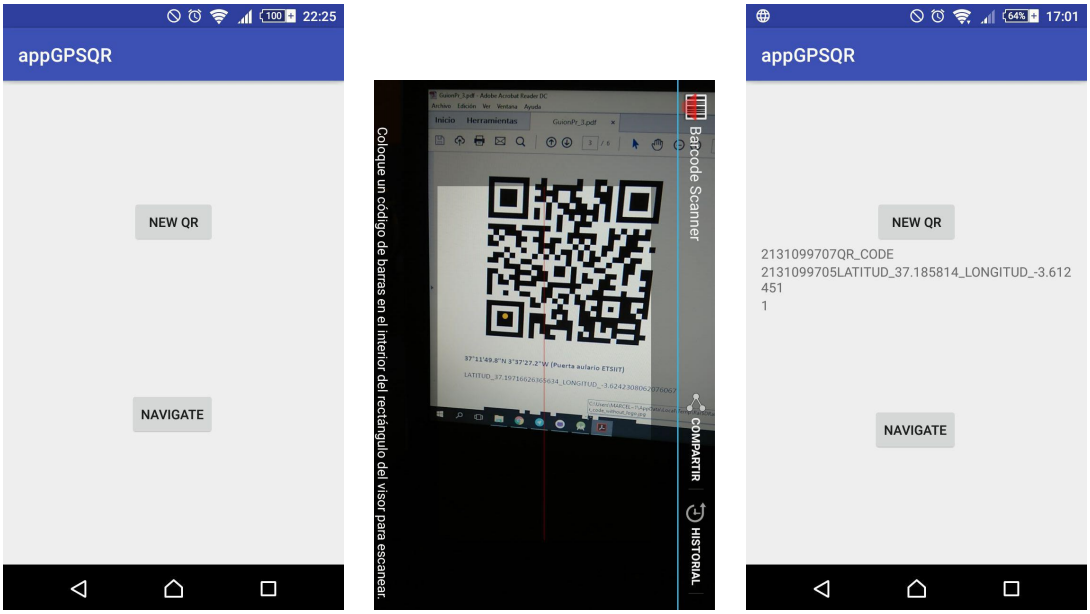

Figura 3: Primera pantalla

Cuando creamos la clase *Map*, debemos instanciar un objeto de la clase *GoogleApiClient*, que usaremos para obtener las posición del usuario. En el método `setUpMapIf Needed()` instanciamos y activamos un objeto de la clase *GoogleMap*, que nos permitirá mostrar el mapa en nuestra aplicación, así como hacer que se dibuje nuestra posición siempre que esté disponible, con `setMyLocationEnabled(true)`. Entonces creamos la petición de localización, indicándole que se actualice cada 4s y que queremos que nos detecte con una alta precisión, para lo que usará tanto el GPS del dispositivo como el Wi-Fi y los datos móviles. obtenemos las coordenadas del *Bundle* pasado por la *Activity QR* e instanciamos los botones que usaremos. Estos botones permitirán por una parte añadir un nuevo código QR y localización a la lista de localizaciones a visitar, y por otro, una vez que hayamos llegado al último punto de nuestro recorrido, visualizar todo el camino que hemos seguido.

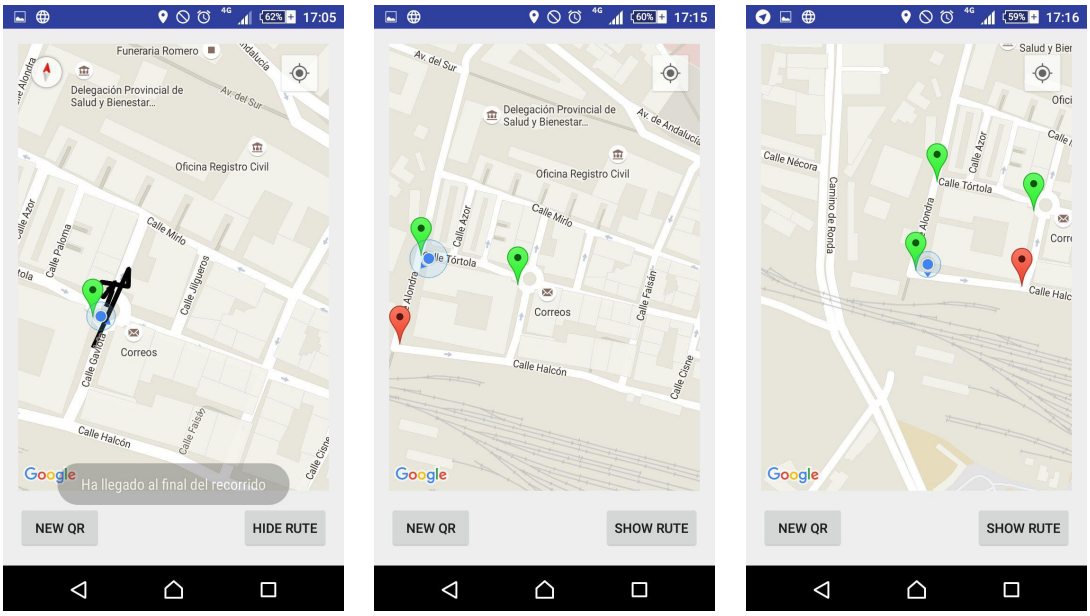

Figura 4: Dibujamos el recorrido hasta el primer punto y añadimos los otros tres

Comenzamos estableciendo como primer destino el primer punto de nuestro recorrido, añadiendo un *Marker* al mapa. Es importante resaltar que en los métodos `onStart()` y `onStop()` tendremos que conectar y desconectar el cliente de la API de Google. En el método `onConnected()` tomaremos la última posición conocida y centraremos el mapa en esta posición. El método que cobra especial relevancia es `onLocationChanged()`, que se activa cada vez que cambia nuestra posición. Lo que haremos será añadir esta nueva localización a la polilínea que pintaremos en el mapa. Comprobamos en cada cambio de localización si hemos llegado al siguiente destino, en cuyo caso cambiamos el color del *Marker* a verde. Si el destino era el último, mostraremos un mensaje diciendo que hemos acabado el recorrido, y en caso contrario mostraremos un mensaje diciendo que hemos llegado al siguiente punto en nuestro camino y cambiaremos el destino al siguiente.

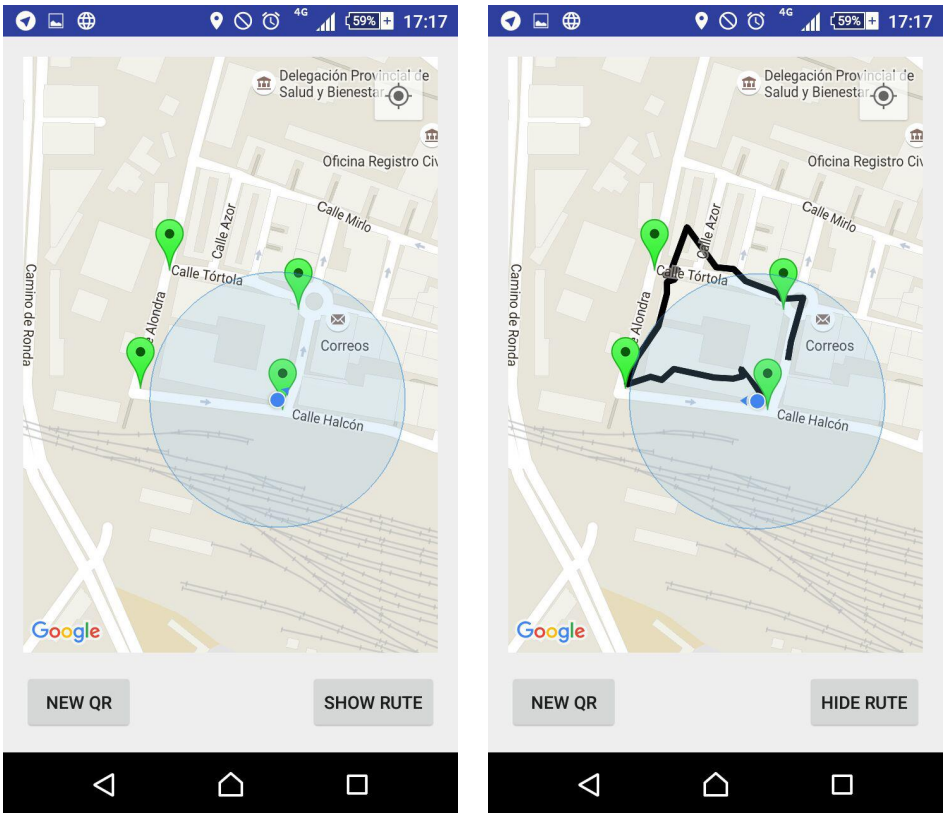

Figura 5: Final del recorrido

3. appGestosFoto

Para desarrollar esta aplicación ha sido necesario, en primer lugar, guardar un gesto que reconozcamos posteriormente para lanzar la foto. Nos hemos ayudado de la aplicación GestureBuilder para crear este gesto (guardado en res/raw/), que es el dibujo simplificado de una cámara (un simple rectángulo con una circunferencia en el centro).
Esta aplicación consta de dos pantallas: una primera en la que dibujaremos este gesto, y otra que será la que se encargue de manejar la cámara y guardar la foto tomada.

La actividad *Gesto* implementará la interfaz *OnGesturePerformedListener*, es decir, se encargará de recoger los gestos y procesarlos. Necesitamos que nuestra clase tenga una referencia a la librería de gestos almacenada. Además, necesitamos la capa donde dibujaremos los gestos (*GestureOverlayView*). Es importante destacar que, ya que el gesto que hemos dibujado tiene dos trazos (*GestureStroke*), debemos indicarle que el tipo de trazo será múltiple con la línea `gestureOverlayView.setGestureStroke Type(GestureOverlayView.GESTURE_STROKE_TYPE_MULTIPLE)`

Una vez hecho esto y correctamente instanciada la librería de gestos, tenemos que implementar el método `onGesturePerformed()`, activado cuando se recoge un gesto. Entonces se genera un *ArrayList* con las predicción dada para cada gesto de

la librería, y entenderemos que se ha dibujado ese gesto si la puntuación obtenida es buena. En nuestro caso hemos determinado que el valor 6.5 es un valor aceptable para dar por válidos siempre los casos en los que se dibuja la cámara y rechazar un buen número de situaciones en las que no se dibuja la cámara sino otra cosa. Si se produce esta situación, tendremos que generar la actividad *Foto*.

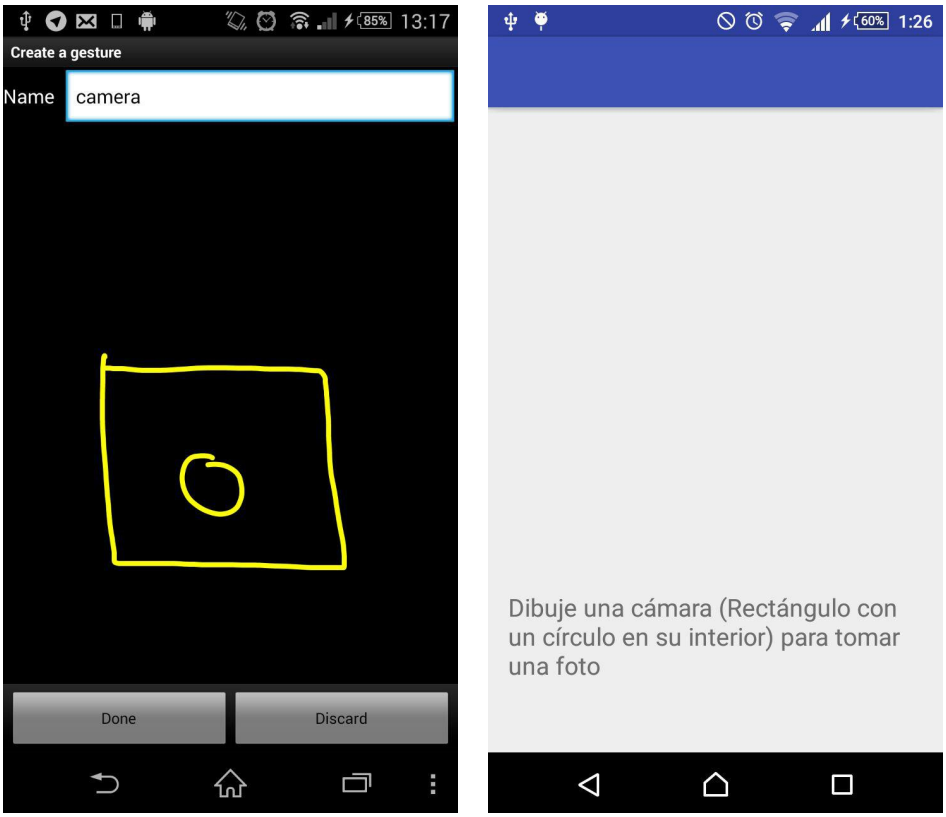

Figura 6: Preparación

En la actividad *Foto* la primera tarea será obtener una instancia de la cámara (*Camera*). Esto lo hacemos mediante el método `getCameraInstance()`, en el que tratamos de abrir la cámara. En caso de error, devolveremos `null`. Aquí ha estado una de las dificultades principales a la hora de realizar la aplicación, ya que la clase *Camera* está obsoleta y no sabíamos si realizar la aplicación con la clase *Camera2*, aunque la falta de información sobre ésta nos ha hecho continuar con *Camera*. Debemos crear también un objeto de la clase *CameraPreview* para poder ver lo que capturará la cámara, así como añadírselo a un *FrameLayout* y que sea visible. Por defecto la cámara está en horizontal, así que debemos girar la cámara 90 grados. Entonces, pondremos un temporizador para que cuando acabe se tome la foto.

Este temporizador lo haremos creando un objeto de la clase *CountTimer* que tome una foto cuando pasen tres segundos. Para guardar la imagen tendremos que implementar una *Camera.PictureCallback*, que tendrá el método `onPictureTaken()` para cuando el contador finalice y se tome la foto. En este método tomamos la fecha y hora actual para añadirlo al nombre del archivo y lo escribimos en la ruta establecida.

Debemos liberar el recurso una vez tomada la foto, con lo que tenemos que hacer `mCamera.release()` en los métodos `onDestroy()` y `onPause()`, así como volver a obtener el recurso de la cámara cuando volvamos a dibujar la cámara.

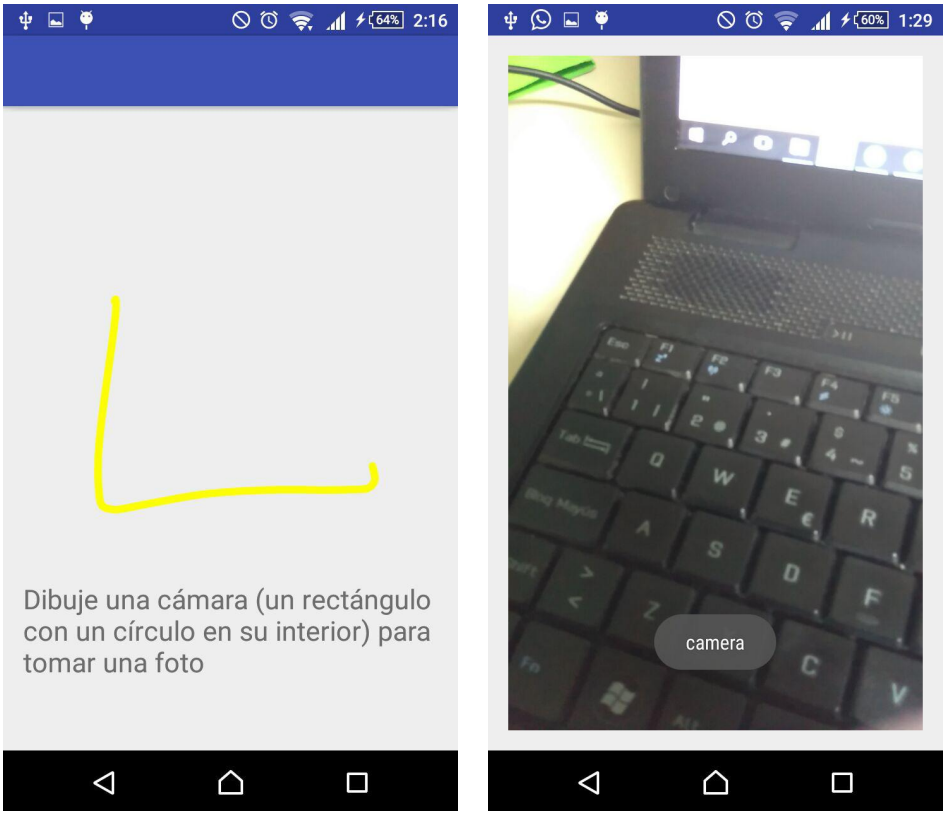

Figura 7: Tomando foto

4. appMovimientoSonido

Queremos realizar una aplicación que reproduzca un sonido cuando movamos el dispositivo. En concreto, haremos sonar un tambor cuando movamos el móvil hacia abajo con la pantalla perpendicular al suelo, y suenen unas maracas cuando lo giremos en el plano de la pantalla del dispositivo (ambas cosas son en el eje Z del dispositivo, que es el que sale perpendicular de la pantalla hacia afuera).

Para los sensores utilizamos un *SensorManager* y dos *Sensor*, uno de tipo *linear_acceleration* y otro de tipo *gyroscope*. Los declaramos globales a la clase:

```
// Administrador de los sensores
private SensorManager mSensorManager;

// Sensor con el que sabremos la aceleración que sufre el dispositivo
// y sensor con el que sabremos la velocidad angular (giroscopio).
private Sensor mAccelerometer;
private Sensor mGyroscope;
```

y los inicializamos en el método `onCreate()`:

```
// Instanciación del SensorManager
 mSensorManager = (SensorManager) getSystemService(SENSOR_SERVICE);

 // Obtención del acelerómetro y giroscopio
mAccelerometer = mSensorManager.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
mGyroscope = mSensorManager.getDefaultSensor(Sensor.TYPE_GYROSCOPE);
```

Para los sonidos utilizamos dos objetos `MediaPlayer`, uno para cada sonido. Los declaramos también globales a la clase:

```
// Reproductores del sonido
private MediaPlayer reproductor;
private MediaPlayer reproductorMaracas;
```

y los inicializamos en el método `onCreate()` también, cada uno con el sonido que va a reproducir:

```
// Creación de los reproductores
reproductor = MediaPlayer.create(this, R.raw.tomacoustic01);
reproductorMaracas = MediaPlayer.create(this, R.raw.maracas);
```

En el método `onSensorChanged()` es donde tenemos que controlar qué hacer cuando el móvil se mueve y distinguiendo entre los movimientos (girar o mover en un eje). Hemos tenido el problema de que es muy complicado girar o mover el móvil sólo en un eje, y los sonidos se reproducen siempre que hay un giro o un movimiento en el eje Z, pero puede que en realidad lo estemos haciendo principalmente en el eje Y y aun así el sonido se reproduzca porque en realidad sí está habiendo movimiento en el eje Z. Para arreglar esto lo que hemos hecho ha sido, en este mismo método, hacer que los sonidos suenen siempre que el movimiento principal sea en el eje Z, pero no en los demás:

```
public void onSensorChanged(SensorEvent event) {
switch (event.sensor.getType()){
   case Sensor.TYPE_GYROSCOPE:
     // Obtenemos la velocidad angular en todos los ejes
     gyr_Z = event.values[2];
     gyr_X = event.values[0];
     gyr_Y = event.values[1];

     // Mostramos por pantalla la velocidad angular del eje Z
     gyr_Z_text.setText(Float.toString(gyr_Z));
 break;

 case Sensor.TYPE_LINEAR_ACCELERATION:
     // Obtenemos la aceleración en todos los ejes.
     accel_Z = event.values[2];
     accel_X = event.values[0];
     accel_Y = event.values[1];

     // Mostramos por pantalla la aceleración en el eje Z.
     accel_Z_text.setText(Float.toString(accel_Z));
 break;
 }

 // Estaremos girando en el eje Z cuando la velocidad sea
 // mayor que menos 5 (estemos girando hacia
```

```
        // la derecha) y el giro registrado en los otros ejes sea
        // menor que el registrado en el eje Z.
        if(gyr_Z < -5 && Math.abs(gyr_Z) > Math.abs(gyr_X) &&
           Math.abs(gyr_Z) > Math.abs(gyr_Y)){
            girando = true;
        }

        // Estaremos moviendo el dispositivo en el eje Z cuando
        // la aceleración sea mayor de 5 y la
        // aceleración en el resto de los ejes sea
        // menor que la aceleraci n en el eje Z.
        if(accel_Z > 5 && accel_Z > accel_X && accel_Z >accel_Y){
            hitting = true;
        }

        // Si estamos girando, reproducimos el sonido y volvemos a poner la variable
        // girando a false.
        if (girando) {
            reproductorMaracas.start();
            Toast toast = Toast.makeText(getApplicationContext(),
                R.string.repr_maracas, Toast.LENGTH_SHORT);
            toast.show();
            girando = false;
        }

        // Si estamos golpeando y no estamos girando, reproducimos
        // el sonido y ponemos la variable hitting a false.
        if (hitting && gyr_X < 0.5 && gyr_Y < 0.5 && gyr_Z < 0.5){
            reproductor.start();
            Toast toast = Toast.makeText(getApplicationContext(),
                R.string.repr_tambor, Toast.LENGTH_SHORT);
            toast.show();
            hitting = false;
        }
    }
```

Como vemos, mostramos también con *Toast* el sonido que se está reproduciendo en cada momento. Además, mostramos por pantalla con *TextView* la aceleración y velocidad angular del eje Z en cada momento.

En el método `onDestroy()` tenemos que liberar los reproductores, en el método `onPause()` pausarlos en el caso de que estén reproduciendo algún sonido y en el método `onResume()` volver a obtener con `registerListener()` los sensores.

El código al completo se encuentra en el enlace a github presente en la sección 7.

La ejecución de la aplicación se puede ver en las siguientes dos fotografías:

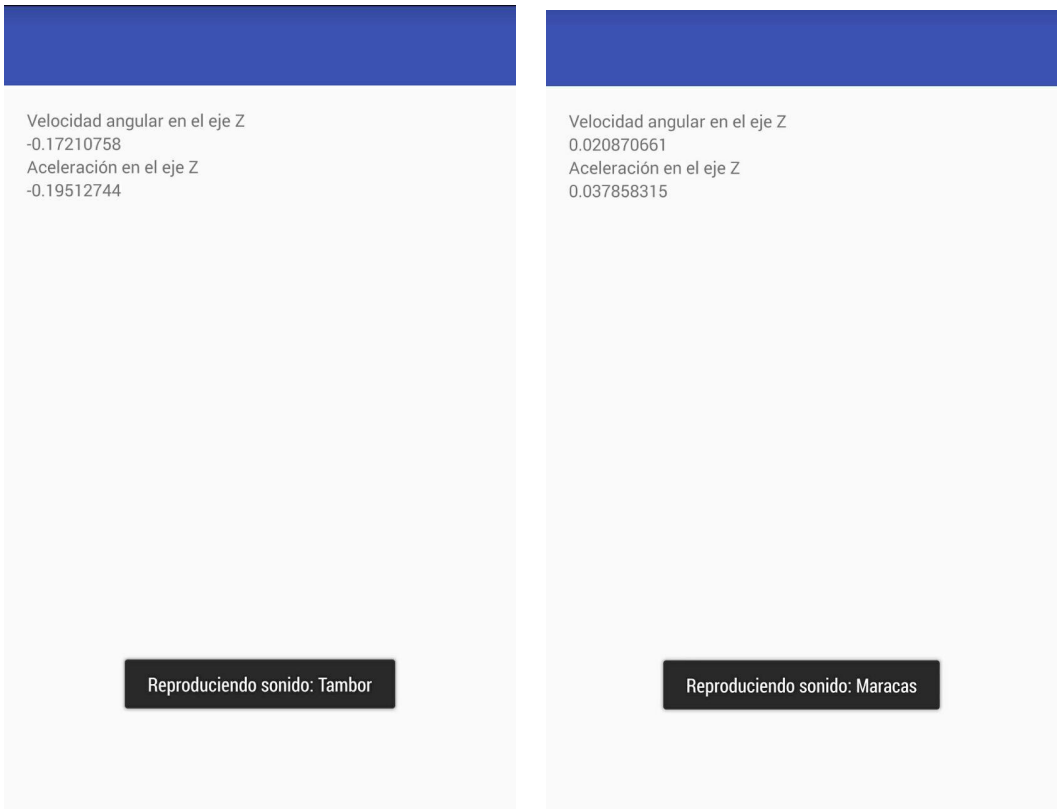

Figura 8: Pantalla inicial junto con mensajes de sonido

5. appSorpresa

La **appSorpresa** es una aplicación que servirá como punto de control. Está dividida en dos partes, una que será la que realizará el administrador y consistirá en escribir un texto que la aplicación codificará y en enviar este texto codificado mediante NFC a los jugadores, la otra parte de la aplicación, que lo único que podrán hacer será recibir el texto codificado e ir hasta el dispositivo NFC controlado por el administrador que será el que decodifique de nuevo este texto, permitiendo a los jugadores seguir la prueba.

En la actividad *MainActivity* nos encontramos un EditText donde tendremos que introducir la contraseña (NPI1415) correctamente para poder avanzar a la siguiente actividad. Esto lo haremos usando el botón Entrar para acceder como *Admin* o bien con el botón Omitir para entrar como *Player*. Para poder manejar la pulsación de estos botones, la clase implementará la interfaz *View.OnClickListener* y llevar a cabo las operaciones necesarias en el método `OnClick`. Estas acciones son generar una actividad con la clase correspondiente, es decir, *Admin* o *Player*. Para saber si nos hemos registrado correctamente, añadiremos al *Intent* que activa la actividad un booleano, así evitaremos que dos jugadores puedan acceder al texto codificado sin la contraseña.
Nos centraremos inicialmente en la *Activity Admin*. En esta actividad hay un Edit-

Text donde introduciremos el texto a codificar.
Para codificarlo, incluimos un nuevo botón de la manera explicada anteriormente que llamará a la función `code()` que recibe la cadena de este EditText y muestra por pantalla el texto codificado. El método `code()` simplemente codifica en base 64 el mensaje y lo devuelve.

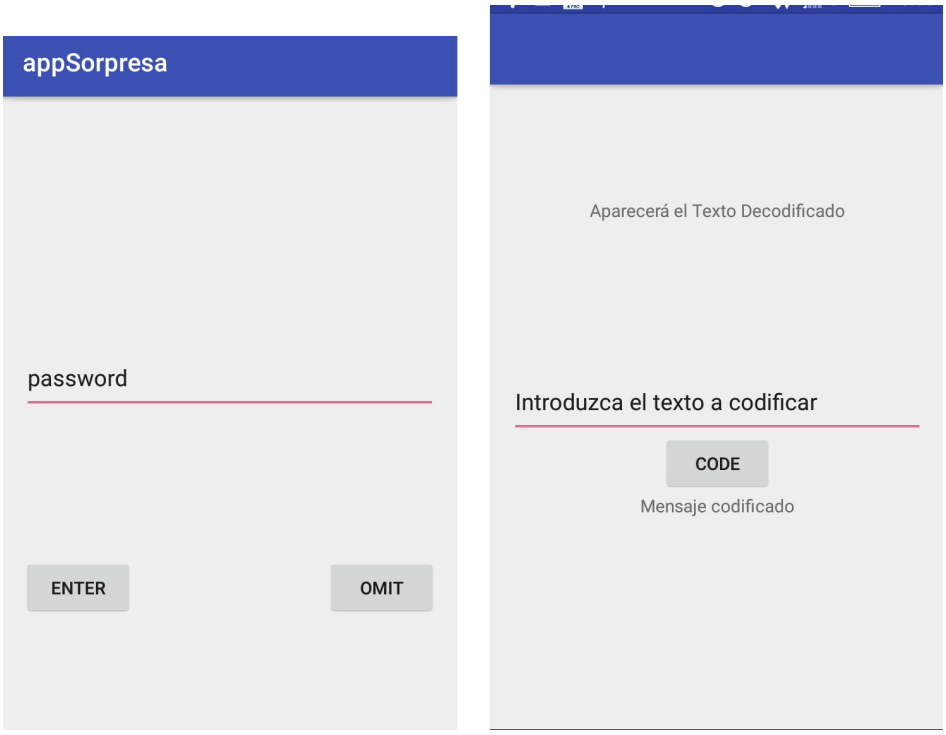

Figura 9: Pantallas iniciales

Ahora viene la parte importante de la aplicación, y es el envío y recepción de datos mediante NFC. Queremos que el paquete NFC llegue tanto a la actividad *Admin* como a *Player*, luego tendremos que añadir en el *Manifest* el `intent-filter` correspondiente al NFC:

```
<intent-filter>
    <action android:name="android.nfc.action.NDEF_DISCOVERED" />
    <category android:name="android.intent.category.DEFAULT" />
    <data android:mimeType="text/plain" />
</intent-filter>
```

Para mejorar la experiencia del usuario y que no se abra una nueva aplicación cada vez que reciba un NFC, añadiremos también al *Manifest* android:launchMode="singleTask
. También es necesario añadir la solicitud de permisos para usar el NFC con ¡uses-permission android:name=."ndroid.permission.NFC/
. Pasamos ahora a explicar cómo enviar y recibir mensajes.

Nuestras clases *Player* y *Admin* tendrán métodos similares para ello, ya que únicamente difieren en qué texto se envía y qué se hace con el texto recibido. Estas clases deben implementar las interfaces *NfcAdapter.CreateNdefMessageCallback*

y NfcAdapter.OnNdefPushCompleteCallback. El mensaje NFC se hará usando el método createNdefMessage(NfcEvent event). Nótese que lo enviamos como texto plano (*text/plain*). Se enviará gracias al método onNdefPushComplete() aunque en éste no tendremos que hacer nada distinto de lo que aparece en la documentación. Acercando dos dispositivos con tecnología NFC y pulsando en la pantalla sobre la actividad, se enviará el mensaje escogido.

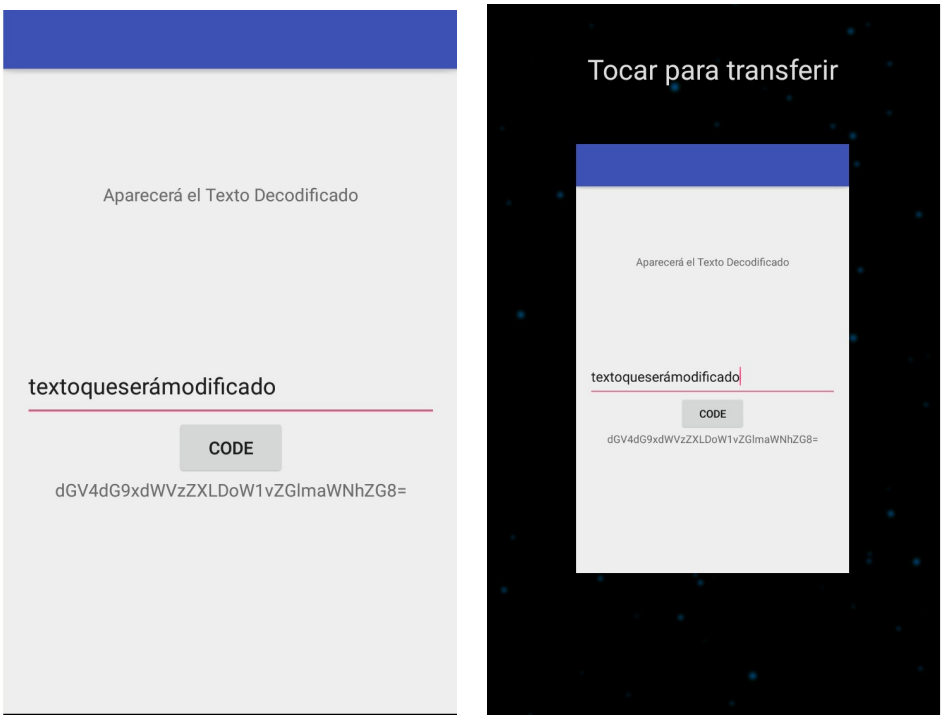

Figura 10: Enviando mensaje codificado

Ahora toca recibir el mensaje. Como habíamos puesto que estas dos *Activity* estaban esperando un paquete NFC (pues así podremos tratarlo y recibir el mensaje esperado), éstas se activarán con el paquete, y pasarán por el método OnResume(). Aquí, cogemos el *Intent* que habremos "puesto" previamente con onNewIntent() y comprobamos que lo que ha provocado la aparición de este *Intent* es efectivamente que se ha recibido un NFC mediante su *Action*. En la clase *Admin* debemos comprobar que se ha introducido la contraseña, mientras que en la *Player*, que esta no se ha introducido. Una vez llegado a este punto, obtenemos el texto incluido en el mensaje y procedemos a manipularlo. En la clase *Player* simplemente mostramos el texto codificado en un *TextView*, en cambio en la clase *Admin* lo decodificamos, y es esto lo que mostramos por pantalla.
La dificultad lógica encontrada ha sido el uso de NFC y la forma de incluir un mensaje, pues es de gran importancia al crear el paquete NFC que es de texto plano. Otro punto a tener en cuenta es que debemos indicar que si al dispositivo le llega un paquete NFC, es preciso incluir en el *Manifest* que lo estamos esperando.

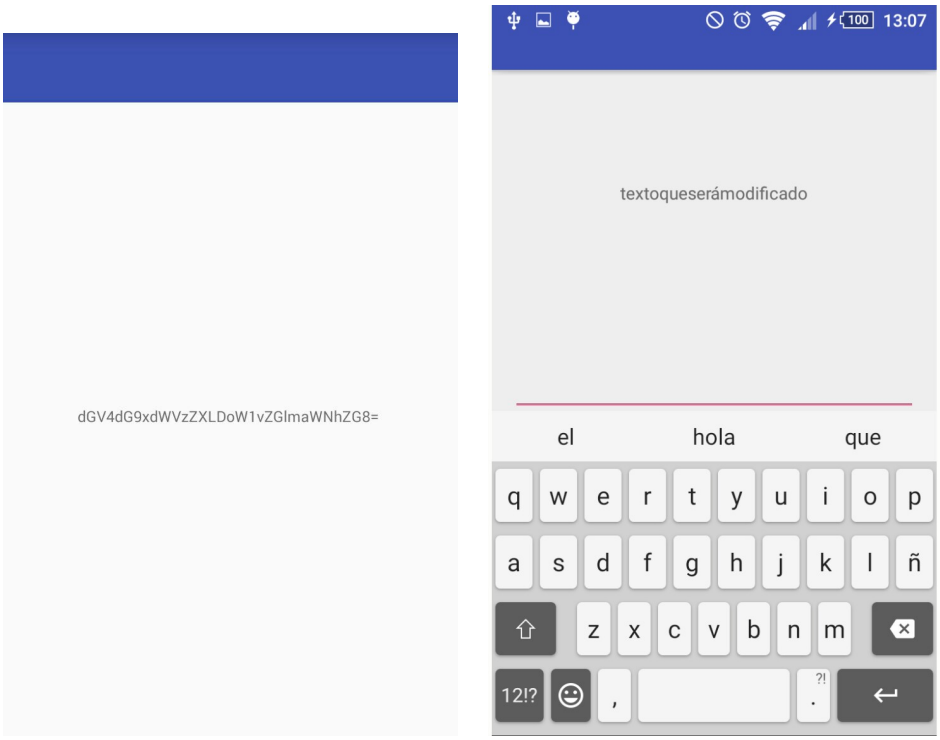

Figura 11: Recibiendo mensaje codificado

6. Bibliografía

1. appFotoVoz:
   a) Material facilitado por la profesora Zoraida Callejas.
   b) `http://developer.android.com/intl/es/guide/topics/sensors/sensors_overview.html`
   c) `http://developer.android.com/intl/es/guide/topics/sensors/sensors_motion.html`
   d) `http://developer.android.com/intl/es/reference/android/hardware/SensorManager.html`
2. appGPSQR:
   a) `http://expocodetech.com/usar-google-maps-en-aplicaciones-android-mapa/`
   b) `http://developer.android.com/intl/es/training/location/retrieve-current.html`
   c) `http://www.hablemosdeandroid.com/p/como-hacer-un-lector-de-codigos-con.html`
   d) `https://github.com/zxing/zxing`
3. appGestosFoto:

a) http://developer.android.com/intl/es/guide/topics/media/camera.html

b) http://mobiledevtuts.com/android/android-sdk-how-to-make-an-automatic-snap

c) https://nuevos-paradigmas-de-interaccion.wikispaces.com/Detecci%C3%B3n+de+patrones+en+Android+-+Gesture+Builder

4. appMovimientoSonido:

   a) https://nuevos-paradigmas-de-interaccion.wikispaces.com/Reproducci%C3%B3n+de+audio+sobre+Android

   b) http://developer.android.com/intl/es/guide/topics/sensors/sensors_motion.html

5. appSorpresa:

   a) http://developer.android.com/intl/es/guide/topics/connectivity/nfc/index.html

   b) http://developer.android.com/intl/es/guide/topics/connectivity/nfc/nfc.html

   c) http://android-er.blogspot.com.es/2014/04/communication-between-android-us
   html

   d) http://www.iteramos.com/pregunta/48153/en-base-64-codificar-y-decodificar-

7. Enlace al código

El código de todas las aplicaciones se encuentra disponible en el siguiente enlace a git: https://github.com/AnabelGRios/NPI-Android