Look!: Framework para Aplicaciones de Realidad Aumentada en Android

Diseño de aplicaciones con Look!

Sergio Bellón Alcarazo

Jorge Creixell Rojo

Ángel Serrano Laguna

En este tutorial se proponen los pasos fundamentales para la creación de aplicaciones utilizando el framework de realidad aumentada *Look!*. Primero se plantearán los pasos necesarios para definir la aplicación a desarrollar, y después los pasos en la codificación de una aplicación sencilla.

1. Planteando la aplicación

Antes de lanzarnos a programar nuestra aplicación en *Look!*, debemos plantearnos qué necesitamos hacer exactamente, y qué módulos vamos a utilizar para ello. No es lo mismo crear una galería de imágenes en tres dimensiones, que una aplicación que nos sitúe con un círculo a otros usuarios en el espacio.

Antes de empezar a programar, y con la funcionalidad de la aplicación de realidad aumentada detallada, el programador debería hacerse las siguientes preguntas:

1. Qué **tipos de entidades** van a ser representadas en la realidad aumentada.
2. Qué **características** definen estas entidades.
3. **De dónde se obtienen esas entidades y sus características**. Podría accederse a un **servidor remoto**, si los datos cambian de manera dinámica; o albergarse de manera **local**, si no son susceptibles de cambios, o los datos no son compartidos.
4. **Cómo se representarán gráficamente**. Pudiendo ser dos y/o tres dimensiones.
5. Qué **interacciones** serán permitidas para cada entidad:
   - Efectos al pulsar
   - Efectos al arrastrar
   - Efectos al soltar
   - Efectos al enfocar con la cámara
6. Si queremos que aparezca de fondo la imagen obtenida por **cámara**.
7. **Si es necesario localizar al usuario** para obtener la funcionalidad buscada.
8. Y de ser así, qué **tipo de localización** sería la adecuada: **Relativa**, con el sistema inercial; O **absoluta**, con el sistema de localización WiFi.
9. Qué **sistema de referencia** se utiliza para situar a los elementos y al usuario.
10. Dónde está el **origen de coordenadas** del mundo.
11. **Si se necesitan añadir elementos extras de interfaz** (botones, menús, cajas de texto, *Activity*s secundarias, etc.) para completar la funcionalidad requerida.

Una vez tomadas las decisiones, puede comenzar el desarrollo. En las siguientes secciones se resuelve el *cómo* abordar estos puntos con *Look!*.

2. Codificando una aplicación básica

En esta sección se explican los pasos a seguir en la codificación de una aplicación sencilla con *Look!*.

Se da por supuesto que el lector tiene instalado el SDK de Android en su entorno de programación, que sabe añadir bibliotecas externas (en este caso, *Look!*) a nuevos proyectos Android, y que sabe instalar y ejecutar estos proyectos en dispositivos físicos o en el emulador Android.

2.1. Creando la Activity principal

Look! ofrece en la *Activity LookAR* el módulo completo de realidad aumentada. El procedimiento habitual será extender esta clase y realizar las inicializaciones pertinentes en el método *onCreate* (el de creación de la actividad). El constructor de *LookAR* tiene el siguiente aspecto:

```
public LookAR(boolean usesCamera, boolean uses3D,
        boolean uses2D, boolean usesHud,
        float maxDist, boolean fullscreen)
```

Como puede observarse, sus parámetros de configuración son mayoritariamente booleanos que definen si la capa correspondiente de realidad aumentada debe ser añadida.

Tiene dos parámetros más: *maxDist*, que define la máxima distancia (medida en el sistema de coordenadas fijado por el programador) a la que una entidad será visible; y *fullscreen*, que define si la *Activity* es a pantalla completa, eliminando la barra de tareas de Android.

Normalmente, el código de iniciación de las *Activity* herederas seguirán el siguiente esquema:

```
public class MyARActivty extends LookAR {

        public MyArActivity(){
                super( true, true, true, true,
                        100.0f, true );
        }

        @Override
        protected void onCreate(Bundle savedInstanceState){
                super.onCreate(svaedInstanceState);
                // Init everything
        }


}
```

En esta caso, hemos definido una actividad que muestra todas las capas de realidad aumentada, incluida la camera, a pantalla completa y que muestra elementos a una distancia máxima de 100 unidades.

Recuerda que para que pueda mostrarse la cámara, ha de añadirse el siguiente permiso al manifiesto de la aplicación:

```
<uses-permission
        android:name="android.permission.CAMERA" />
```

Con este paso ya podríamos lanzar la *Activity* y entrar en la realidad aumentada. Pero aún no veríamos nada, aparte de la cámara de fondo, puesto que no hemos definido ningún elemento para que sea mostrado.

2.2. Definiendo los elementos de la aplicación: *EntityData*

Para mostrar entidades en la realidad aumentada, debemos añadir elementos *EntityData* que puedan ser procesados y convertidos en *WorldEntity*.

Modifiquemos el método *onCreate* de **MyARActivity**, para añadir un elemento:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        // Create the element data
        EntityData data = new EntityData( );
        data.setLocation(10, 0, 0);

        // Add the data to the data handler
        LookData.getInstance().getDataHandler()
                .addEntity(data);

        // Updates the recent added data
        LookData.getInstance().updateData();

}
```

Primero creamos un elemento vacío, y lo situamos en la posición (10, 0, 0). Después, lo añadimos al módulo de datos, y finalmente actualizamos para que se muestren los cambios realizados. Si ahora ejecutamos, veremos que, tras orientar la cámara en la dirección adecuada, aparecerá un cubo con un cuadro de texto, que indica el identificador de la entidad y su tipo, en este caso null, puesto que no se definió la propiedad (Figura 1).

Esta apariencia es creada de manera automática por la factoría *WorldEntityFactory*, que es la que *Look!* instancia por defecto.

Sin embargo, en nuestras aplicaciones querremos definir nuestras propias apariencias. Para ello deberemos implementar nuestras propias factorías *WorldEntityFactory*.

2.3. Definiendo factorías de elementos

Una factoría de elementos, heredera de *WorldEntityFactory*, debe sobreescribir un único método: *createWorldEntity(EntityData data)*.

Supongamos que nuestros elementos tienen una propiedad llamada **name**, que indica el nombre del elemento. Y una propiedad **color**, que indica su color. Queremos que, en su representación, aparezca, en dos dimensiones, un texto con

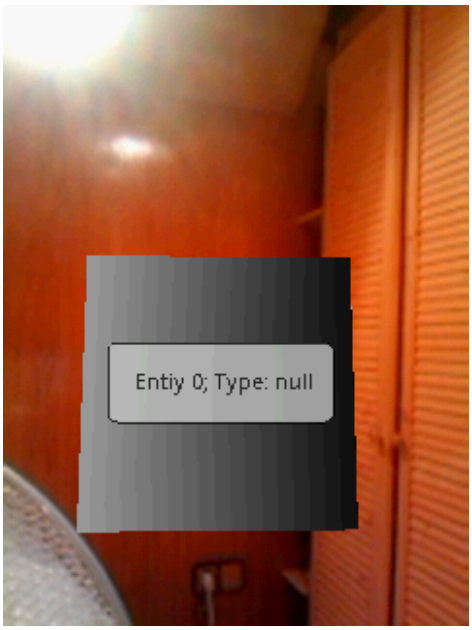

Figura 1: Captura de la aplicación tras añadir un elemento

el nombre del elemento, y en tres dimensiones, un cubo del color indicado por la propiedad.

El código para la factoría sería el siguiente:

```
public class MyWorldEntityFactory
                        extends WorldEntityFactory {

        public static final
                String NAME = "name";

        public static final
                String COLOR = "color";

        @Override
        public WorldEntity
                createWorldEntity(EntityData data) {
                        WorldEntity we =
                                new WorldEntity( data );
                        we.setDrawable2D(
                        new Text2D( data.
                                getPropertyValue(NAME)));

                Entity3D drawable3d
                        = new Entity3D( new Cube( ) );
                String color = data.
                        getPropertyValue(COLOR);
                if (color.equals("red")){
```

```
                        drawable3d.
                                setMaterial(new
                                Color4(1.0f, 0.0f, 0.0f));
                }
                else if (color.equals("green"))
                        drawable3d.
                                setMaterial(new
                                Color4(0.0f, 1.0f, 0.0f));
                //...

                we.setDrawable3D(drawable3d);
                return we;
        }

}
```

Como vemos, primer creamos un *WorldEntity* que recibe como atributo el elemento con los datos. Después definimos para su representación en dos dimensiones un texto, y para su representación en tres dimensiones, un *Entity3D* que contiene un cubo, y al que se la asigna un color, atendiendo al valor de la propiedad.

Ahora debemos comunicarle a *LookData* qué factoría debe utilizar para la creación de entidades. Así que añadimos el siguiente código en el *onCreate* de la aplicación:

```
LookData.getInstance()
                .setWorldEntityFactory(
                        new MyWorldEntityFactory());

// Create the element data
EntityData data = new EntityData();
data.setLocation(10, 0, 0);
data.setPropertyValue(MyWorldEntityFactory.NAME,
        ''Element 1'');
data.setPropertyValue(MyWorldEntityFactory.COLOR,
        ''green'');

EntityData data1 = new EntityData();
data1.setLocation(10, 0, 5);
data1.setPropertyValue(MyWorldEntityFactory.NAME,
        ''Element 2'');
data1.setPropertyValue(MyWorldEntityFactory.COLOR,
        ''red'');

// Add the data to the data handler
LookData.getInstance().getDataHandler().addEntity(data);
LookData.getInstance().getDataHandler().addEntity(data1);

// Updates the recent added data
LookData.getInstance().updateData();
```

Asignamos la factoría y añadimos dos entidades, una roja y otra verde.

El resultado de la ejecución de esta aplicación es el mostrado por la figura 2.

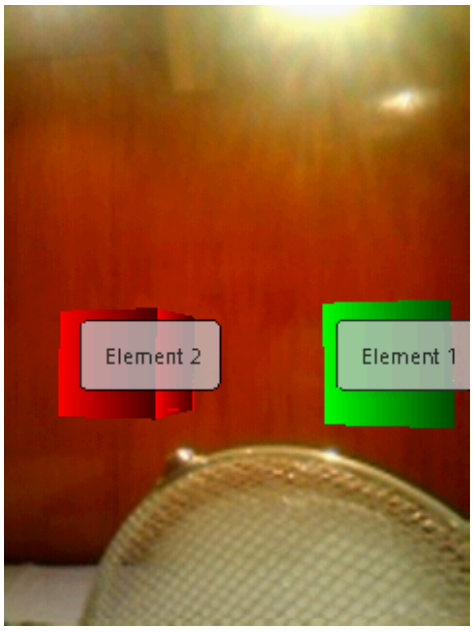

Figura 2: Captura de la aplicación con la nueva factoría

2.4. Añadiendo interacciones

Cada una de las entidades permiten que le sean añadidos una serie de *listeners* que respondan a los distintos eventos que puedan surgir durante la ejecución de la aplicación.

2.4.1. Añadiendo procesado de eventos táctiles: TouchListener

Cuando una entidad reciba un evento táctil, pasará ese evento a todos los *listeners* de su lista de *TouchListener*. Cómo puede verse en la figura 3, la interfaz define métodos para los tres tipos de eventos táctiles principales. Cada uno de los métodos recibe como parámetro la entidad que recibió el evento y las coordenadas de pantalla dónde sucedió.

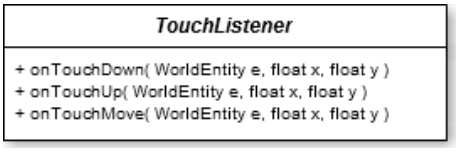

Figura 3: La interfaz *TouchListener*

Supongamos que, siguiendo el ejemplo desarrollado hasta ahora, queremos que cada vez que se toque uno de los elementos, éste se desplaza una unidad en

el eje coordenado *y*.

Para lograrlo, implementaríamos el siguiente *TouchListener*:

```
public class MyTouchListener implements TouchListener {

        @Override
        public boolean onTouchDown(WorldEntity e,
                float x, float y) {
                Point3 p = e.getLocation();
                LookData.getInstance().getDataHandler()
                        .updatePosition(e.getData(),
                                p.x, p.y - 1, p.z);
                return true;
        }

        @Override
        public boolean onTouchUp(WorldEntity e,
                float x, float y) {
                return false;
        }

        @Override
        public boolean onTouchMove(WorldEntity e,
                float x, float y) {
                return false;
        }
```

Sólo nos quedaría añadir este *listener* a la lista de *TouchListener* de cada una de las entidades. Para ello, añadimos la siguiente línea al código de creación de entidades de *MyWorldEntityFactory*:

```
we.addTouchListener(new MyTouchListener( ));
```

La figura 4 muestra una captura de la aplicación, tras tocar uno de los elementos. Puede apreciarse como el *TouchListener* ejecutó su lógica y trasladó la entidad.

2.4.2. Añadiendo eventos de cámara *CameraListener*

Look! considera un evento de cámara cuándo el usuario enfoca (o deja de enfocar) directamente una entidad con el centro de su dispositivo. Cómo muestra la figura 5, *CameraListener* cuenta con dos eventos: uno para cuándo el usuario enfoca la entidad, y otro para cuándo deja de enfocarla.

Es decir, si colocamos el dispostivo tal que una entidad ocupe el centro de la pantalla, se ejecutará el método *onCameraEntered*. Cuándo, tras mover el dispositivo, la entidad abandone la zona central, se lanzará el método *onCameraExited*. Ambos reciben como parámetro el objeto que recibió del evento.

Supongamos que, en el ejemplo seguido hasta ahora, queremos que al enfocar directamente una entidad, cambie su malla de un cubo a un plano cuadrado, y que recupere su estado original cuándo el usuario deja de de enfocar. El código necesario sería el siguiente:

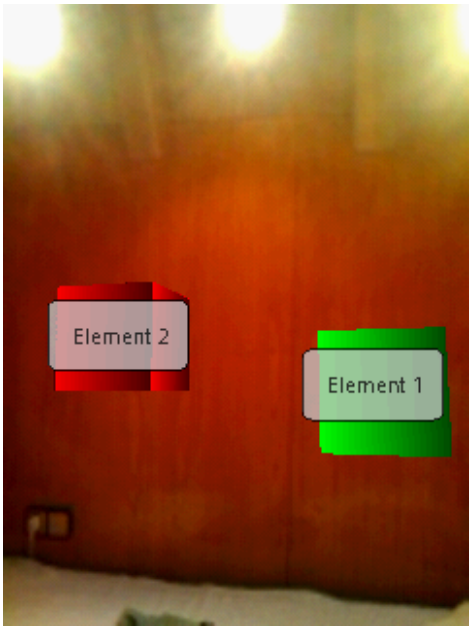

Figura 4: Captura de la aplicación tras pulsar uno de los elementos

Figura 5: La interfaz *CameraListener*

```
public class MyCameraListener
        implements CameraListener {

        private static SquarePrimitive square
                = new SquarePrimitive();

        private Drawable3D oldDrawable;

        @Override
        public void onCameraEntered(WorldEntity entity) {
                oldDrawable = entity.getDrawable3D();
                entity.setDrawable3D(square);

        }

        @Override
        public void onCameraExited(WorldEntity entity) {
                entity.setDrawable3D(oldDrawable);
        }

}
```

Ahora deberíamos añadir el *listener* a la lista de *CameraListener*. Para ello, añadimos el siguiente código a la factoría de elementos:

```
we.addCameraListener(new MyCameraListener( ));
```

La figura 6 muestra una captura de la aplicación, mientras el usuario enfoca directamente a una de las entidades. Puede apreciarse como el *CameraListener* ejecutó su lógica y cambió la malla 3D del elemento.

2.5. Añadiendo un HUD a la aplicación

Look! permite añadir vistas Android a modo de HUD. Estas vistas podrían facilitar la interacción con el usuario. Por ejemplo, si quisiéramos añadir un texto para dar cierta información al usuario, podríamos utilizar una TextView, con el texto requerido, alojada en la capa de HUD.

Supongamos que ahora queremos añadir a nuestra aplicación de ejemplo, un botón en la parte superior. De momento solo queremos que aparezca, ya le añadiremos funcionalidad después.

Lo primero que deberíamos hacer sería asegurarnos de que la capa HUD está configurada a *true*, en la llamada al constructor de *LookAR* desde *MyARActivity*. Después, definimos una vista con un layout XML de Android, que contenga lo que buscamos para nuestro HUD (en este caso, un único botón):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
```

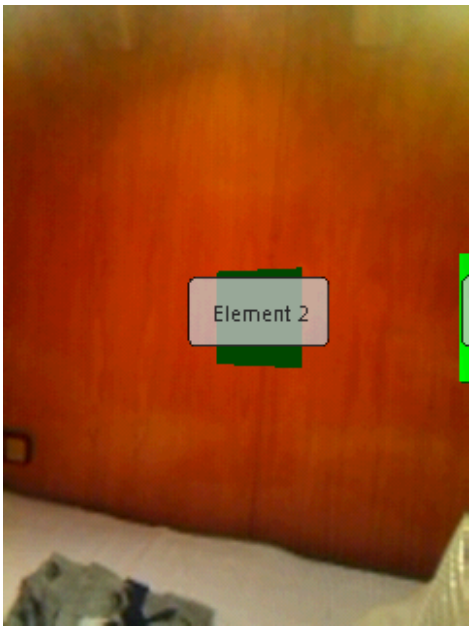

Figura 6: Captura de la aplicación mientras se enfoca directamente a uno de los elementos. Puede apreciarse como su representacion 3D ha cambiado de un cubo a un plano.

```
    >
    <Button android:text="Button"
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    </Button>
</LinearLayout>
```

Supongamos que este archivo es nombrado como "main.xml", y está guardado en la carpeta de recursos layout de Android. Deberíamos añadir el siguiente código en MyARActivity para que esta vista apareciera como HUD:

```
@Override
public void onCreate(Bundle savedInstanceState) {
        // ...
        ViewGroup v = this.getHudContainer();
        v.addView(LookARUtil.getView(R.layout.main, null));
}
```

Con *getHudContainer* obtenemos el contenedor del HUD, al que podemos añadirle vistas. En este caso, utilizamos la función de *LookARUtil* para crear una vista a partir de un archivo de recurso, y la añadimos.

El resultado de la ejecución es el mostrado por la figura 7.

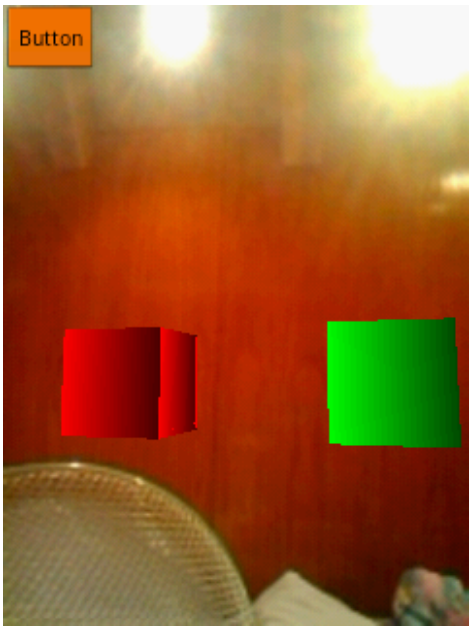

Figura 7: Captura de la aplicación con HUD. Podemos comprobar arriba a la izquierda como se añadió el botón.

2.6. Configurando una base de datos

Cuando queremos que los elementos que han sido añadidos durante una ejecución, sean recordados en subsecuentes ejecuciones, debemos utilizar una base de datos.

Look! ofrece la creación de una base de datos, a través de unos pocos pasos.

2.6.1. Extendiendo el ContenProvider

La clase *LookSQLContentProvider* representa una base de datos genérica, que el usuario debe extender para utilizarla. La extensión es sencilla, sólo necesitamos definir el nombre de la base de datos, y la autoridad.

```
public class MyContentProvider
        extends LookSQLContentProvider {

        public MyContentProvider() {
                super(''mydatabase.db'',
                ''es.ucm.myaractivity.contentprovider'');
        }

}
```

En este caso, el nombre para la base de datos será *mydatabase*, y la autoridad, el nombre del paquete dónde se encuentra alojada la clase.

Ahora, para que la base de datos sea accesible desde la aplicación, debemos definir el *content provider*.

Deberíamos añadir la siguiente línea a AndroidManifest.xml:

```
<provider android:name=
"es.ucm.myaractivity.contentprovider.MyContentProvider"
        android:authorities=
"es.ucm.myaractivity.contentprovider">
</provider>
```

En **name**, ponemos la ruta hasta la clase, y en **authorities**, la autoridad definida en el constructor.

2.6.2. Incorporando al DBDataHandler

Por defecto, el módulo de datos *LookData*, instancia como DataHandler un BasicDataHandler, que está basado en una lista Java, que contiene todos los datos.

Sin embargo, para manejar la base de datos, necesitamos utilizar un *DBDataHandler*, y así debemos configurarlo en el *onCreate* de la aplicación, con el siguiente código:

```
LookData.getInstance()
        .setDataHandler(new DBDataHandler(this));
```

El *DBDataHandler* se encarga de cargar el content provider definido en el manifiesto de la aplicación, y de dar acceso a la base de datos con sus métodos, de manera transparente.

Los elementos que se añadan a través del método *addEntity* del *DBDataHandler* serán guardados en la base de datos, y serán accesibles en siguientes ejecuciones.

3. Preparando el Sistema de Localización

La localización por Wifi conlleva una serie de pasos previos de preparación para que esta funcione correctamente. Para ello, junto al framework *Look!* se provee la aplicación *CNWifi* que implementa de manera sencilla todos los pasos necesarios para preparar el entorno de localización.

Asimismo, el sistema de navegación inercial requiere el ajuste de una serie de parámetros para un funcionamiento adecuado.

En esta sección se describe cómo utilizar el programa *CNWifi* para llevar a cabo la configuración necesaria y empezar a utilizar el sistema de localización por Wifi en las aplicaciones desarrolladas con el framework, y cómo ajustar los parámetros de navegación inercial.

3.1. Definición de Nodos y Puntos de Acceso

3.1.1. Definición de Nodos

En primer lugar es necesario definir una serie de nodos en el mapa del edificio sobre el que se quiere utilizar la localización. Se recomienda una distancia óptima entre nodos de entre 5 y 10 metros en función del número de puntos de acceso a utilizar. [1] Estos nodos se etiquetan con un identificador único y

[1] Se pueden definir varios nodos para una misma localización, de forma que la captura de datos se realice en diferentes puntos de, por ejemplo, una misma habitación. Todos los nodos

escriben en un fichero de nombre *Lugares.txt* con el siguiente formato:

[Id, Planta, Coordenada X, Coordenada Y, Nombre] [2]

En la figura 8 se muestra un ejemplo de selección y etiquetado de nodos. El fichero resultante se muestra a continuación:

```
1 5 14 5 Habitacion
2 5 8 4 banno
3 5 4 4 Cocina
4 5 2 1 Salon
```

El fichero resultante debe incluirse en la raiz de la tarjeta SD del dispositivo.

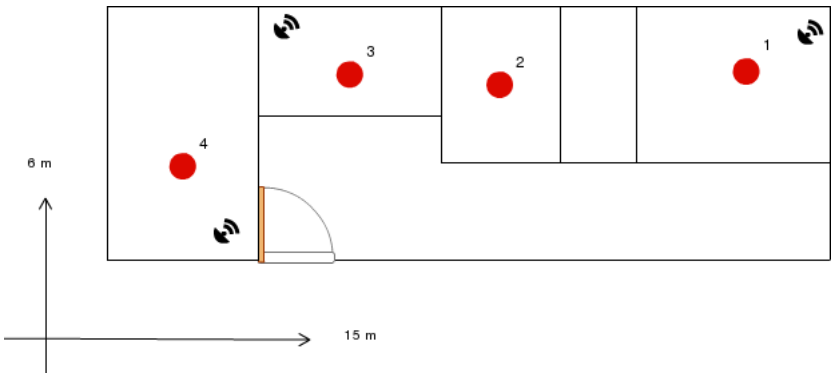

Figura 8: Ejemplo de Definición de Nodos

3.1.2. Definición de Puntos de Acceso

El siguiente paso consiste en la definición de los puntos de acceso a utilizar en el sistema de localización. Es importante tener en cuenta que aunque un mayor número de puntos de acceso aumentará la precisión, los puntos de acceso deben de ser estables ya que si uno de ellos deja de funcionar afectará al sistema de localización.

Esto paso puede llevarse a cabo de dos formas:

Definición Manual

Si se desea restringir los puntos de acceso a utilizar, deberán definirse las direcciones MAC de los puntos de acceso elegidos en el fichero APs.txt. A continuación se muestra un ejemplo del contenido de este fichero:

```
00:11:f5:a1:47:ac
00:1a:2b:5b:ff:28
70:71:bc:8a:6b:cf
```

tendrían las mismas coordenadas aunque diferente id. Esto permite aumentar la precisión al utilizarse un mayor número de muestras en la inferencia de la posición.

[2]Si se desea integrar el sistema de localización mediante Wifi con el sistema de Navegación Inercial, las coordenadas deberán estar expresadas en metros.

El fichero resultante debe incluirse en la raiz de la tarjeta SD del dispositivo.

Detección automática
Tal y como se muestra en la figura 9, *CNWifi* proporciona un método de detección automática de puntos de acceso y creación del fichero correspondiente.

Figura 9: CCNWifi: Detección automática de Puntos de Acceso

3.2. Captura de Datos

El siguiente paso es la captura de datos en cada uno de los nodos definidos para la elaboración del mapa de radio. Para ello, *CNWifi* proporciona un sistema automático de captura de datos, como se aprecia en la figura 10.

Para llevar a cabo la captura de datos, el usuario deberá colocarse en cada uno de los nodos definidos en el paso 3.1.1, indicar su id y pulsar en el botón *Entrenar*. Cuando haya terminado la captura de todos los nodos, el usuario deberá pulsar el botón *Guardar* para que los datos sean procesados y se genere el archivo de mapa de radio.

En este punto el sistema está listo para empezar a localizar el dispositivo desde cualquier aplicación.

3.3. Probando la Localización por Wifi

CNWifi proporciona un módulo para probar la localización, mostrando la información de la ubicación actual en tiempo real. Este módulo puede verse en la figura 11 y permite asegurarnos de que la selección de nodos es la adecuada y que el sistema de localización funciona correctamente.

Figura 10: CCNWifi: Captura de Datos

Figura 11: CCNWifi: Probando la localización

3.4. Ajuste de Parámetros del Sistema de Navegación Inercial

3.4.1. Factor de Orientación

El sistema de navegación inercial tiene en cuenta la orientación del dispositivo para calcular el desplazamiento en las coordenadas (X,Y). Si estamos trabajando sobre un mapa cuyo eje Y no está alineado con el Norte, es necesario especificar el factor de corrección mediante el parámetro **NORTH** de la clase *Mapa* contenida en el paquete *es.ucm.look.locationProvider.map*.

Esta información se utiliza para mapear las coordenadas reales a coordenadas del mapa de manera automática y transparente para el programador de aplicaciones.

3.4.2. Escala

Si se desea hacer un dibujado de la posición del usuario sobre una imagen, es posible ajustar este parámetro y utilizar las funciones proporcionadas en la clase *Mapa* del paquete *es.ucm.look.locationProvider.map* para convertir las coordenadas reales a coordenadas de la imagen.

Para ello es necesario ajustar el parámetro **SCALE** contenido en dicha clase para indicar la correspondencia entre píxeles y metros.

4. Integrando localización

A la hora de integrar localización en nuestra aplicación, en primer lugar se debe indicar si se utilizará el subsistema de Localización por Wifi, el subsistema de Navegación Inercial o ambos de manera combinada. Para ello se debe inicializar la clase *LocationManager*[3] con los parámetros correspondientes a cada uno de ellos:

```
public LocationManager(
        Context context, boolean wifi, boolean ins);
```

Una vez hecho esto, se puede iniciar y detener la localización mediante los métodos *start()* y *stop()*. Esta clase se encarga de inicializar y parar los servicios de localización necesarios, actualizar los datos de la posición, combinarlos si es necesario y proporcionar acceso a los mismos desde la aplicación.

En el siguiente ejemplo se muestra como se utilizaría el sistema de navegación por Wifi desde una aplicación desarrollada mediante el framework:

```
LocationManager location = new LocationManager(
        this, true, false);
location.start();
...
Point3 posicion = location.getPosition();

//Para actualizar el mundo con la nueva posicion:
LookData.getInstance().getWorld().setLocation(posicion);
...
```

[3]LocationManager es la interfaz con el sistema de localización.

```
location.stop();
```

NOTA: Para que todo funcione correctamente, se deben añadir los siguientes permisos al manifiesto de la aplicación:

```
<uses-permission android:name=
        "android.permission.CHANGE_WIFI_STATE" />
<uses-permission android:name=
        "android.permission.ACCESS_WIFI_STATE" />
<uses-permission android:name=
        "android.permission.WRITE_EXTERNAL_STORAGE" />
```