

PROYECTO DE FIN DE CARRERA

TÍTULO DEL PFC: Integración de técnicas de visión artificial y redes sociales en Android

TITULACIÓN: Ingeniería Técnica en Informática de Sistemas

AUTOR: Salvador Roig Mascarell

DIRECTOR: Vicente Luis Atienza Vanacloig

FECHA: 18 de julio de 2011

Contenidos

Introducción	6
Contexto	6
Objetivos	7
1. Conceptos básicos sobre Android	8
1.1. Estructura de una aplicación	8
1.2. Android Manifest	9
1.3. Ciclo de vida de una aplicación	10
1.4. Interfaz de usuario	12
1.5. Recursos en Android	17
2. Librerías de reconocimiento	19
3. API de Facebook	20
4. Creación de la aplicación	22
4.1. Diagrama de actividades	22
4.2. Requisitos	23
4.3. Desarrollo de la aplicación	23
4.3.1. Menu.java	23
4.3.2. Camera.java	24
4.3.3. SD.java	25
4.3.4. LocalPhoto.java	26
4.3.5. Edit.java	27
4.3.6. Tagging.java	28
4.3.7. FriendsArrayAdapter.java	32
4.3.8. FBFriends.java	33
4.3.9. FBAlbums.java	33
4.3.10. AlbumsArrayAdapter.java	33
4.3.11. FBGallery.java	33
4.3.12. PhotosAdapter.java	35
4.3.13. FBPhoto.java	35
4.3.14. Settings.java	35
4.3.15. Android Manifest.xml	35
4.3.16. Recursos	36
5. Publicidad en Android	37
6. Protección	39
7. Publicación en el Market	41

Conclusión	43
Limitaciones y problemas encontrados	43
Valoración personal	44
Líneas futuras	45
Anexos	47
I. Instalación y configuración del IDE	47
II. Instalación y configuración del SDK de Facebook	48
III. Importación de un proyecto en Eclipse	49
IV. Uso del emulador sin necesidad de Eclipse	49
Bibliografía	50

Introducción

Contexto

En los últimos años estamos viviendo un auge en el sector de la telefonía móvil, en especial en las ventas de los llamados teléfonos inteligentes. Como apunta un artículo de 'PC World' ⁽¹⁾, el segundo trimestre del año anterior crecieron las ventas un 64%, y se estima que en 2013 estos dispositivos abarquen casi el 30% de las ganancias del mercado de la telefonía móvil. La capacidad de estos dispositivos ha crecido vertiginosamente hasta el punto de que ya empiezan a salir al mercado los primeros dispositivos móviles con procesadores de doble núcleo.

Esto hace que cobre gran interés la explotación de estas capacidades para el uso multimedia de los terminales más allá del ocio. Cada vez surgen más aplicaciones sobre reconocimiento de imágenes, de caracteres o con características de realidad aumentada, haciendo uso de la cámara. Por otro lado, el uso de redes sociales se ha convertido en algo popular y de crecimiento continuado. Servicios que, además, han visto en parte su incremento gracias a su uso en dispositivos móviles.

En el mundo de los sistemas operativos, están surgiendo nuevas propuestas que se adapten las capacidades de los dispositivos móviles. Una de las plataformas que más crecimiento está experimentado es Android. Las razones para elegir Android como plataforma objetivo giran en torno a cuatro ventajas:

- *Programación en Java, o en C mediante librerías adicionales.* A diferencia del iPhone ⁽²⁾, cuya programación se realiza en Objective C (mayoritariamente usado para dicho móvil y Mac OS X), Android hace uso de lenguajes de amplia difusión, al alcance de cualquier desarrollador.
- *SDK multiplataforma.* Las herramientas de desarrollo se encuentran disponibles para Windows, Mac OS X y Linux y se ofrece la facilidad de integrarlas en el IDE de Eclipse, aunque es posible programar en cualquier entorno.
- *Código libre y política de publicación sin censura.* Mientras que Apple ejerce un estricto control sobre el contenido que se publica en su tienda de aplicaciones, Google proporciona más libertad a los desarrolladores y solo restringe aquellas aplicaciones que puedan comprometer la seguridad del dispositivo o incumplan patentes.
- *Plataforma con una tasa de crecimiento elevada* ⁽³⁾⁽⁴⁾. Según un estudio de la consultora NPD, Android se convirtió a mediados del año pasado en la plataforma líder de 'smartphones' en el mercado americano. Otro estudio más reciente desprende que, en la actualidad, es la plataforma más vendida a nivel mundial, desbancando a Nokia.

Objetivos

El interés del proyecto cubre varios frentes. En primer lugar, explorar el proceso de desarrollo de una aplicación móvil, que abarca desde la creación de la interfaz hasta la lógica interna del programa, centrándose en las capacidades multimedia del dispositivo. Se ha elegido la plataforma *Android* por las ventajas y posibilidades que ofrece, así como el auge que está experimentando y su proyección de futuro. En segundo lugar, adentrarse en las bases del campo de la visión artificial y el procesamiento de imágenes. Por último, conocer los mecanismos de creación de una aplicación social integrando la API proporcionada por Facebook, ya que es la plataforma líder a nivel mundial en redes sociales.

El propósito del proyecto consiste en desarrollar una aplicación fotográfica que aplique un algoritmo de detección de rostros para el procesamiento de la imagen y permita integrarse con la red social de Facebook. La idea consiste en realizar fotografías de gente y analizar las imágenes mediante una función que localice las caras de las personas. A continuación, el usuario tiene la posibilidad de interactuar con los rostros marcados, de forma que puede asociar la marca a un determinado contacto de Facebook. Por último, la imagen es enviada al servicio (haciendo uso de su API) con la información de etiquetado, junto a otras informaciones como el título de la misma. Por otro lado, se incorpora un álbum de las fotos alojadas en el servidor, ya sean propias o de los amigos asociados a la cuenta.

Es interesante remarcar las posibilidades de mejora de la aplicación. En un futuro, con mayores conocimientos sobre visión artificial, se podría crear una base de datos de usuarios de forma que, mediante algoritmos de reconocimiento facial, se pudiera contrastar la fotografía tomada con las imágenes de los usuarios ya almacenadas en memoria, y así realizar el proceso de etiquetado de forma totalmente automática. Por tanto, en vista a ello, la aplicación a realizar contendrá un método encargado de recortar los rostros identificados, almacenándolos en disco.

1. Conceptos básicos sobre Android

1.1. Estructura de una aplicación

Dentro de una aplicación de *Android* hay cuatro componentes principales: *Activity*, *Service*, *Content Provider* y *Broadcast Receiver*. Todas las aplicaciones de *Android* están formadas por algunos de estos elementos o combinaciones de ellos.

Activity: Las *Activities* (o Actividades) son el elemento constituyente de *Android* más común. Para implementarlas se utiliza una clase por cada Actividad, que extiende de la clase base *Activity*. Cada clase mostrará una interfaz de usuario, compuesta por *Views* (o Vistas). Cada vez que se cambie de Vista se cambiará de Actividad, como, por ejemplo, en una aplicación de mensajería en la que se tiene una Vista para mostrar la lista de contactos y otra Vista para escribir los mensajes. Cuando cambiamos de Vista, la anterior queda pausada y puesta dentro de una pila de historial para poder retornar en caso necesario. También se pueden eliminar las Vistas del historial en caso de que no se necesiten más. Para pasar de vista en vista, *Android* utiliza una clase especial llamada *Intent*.

Intent: Un *Intent* es un objeto mensaje que, en general, describe qué quiere hacer una aplicación. Las dos partes más importantes de un *Intent* son la acción que se quiere realizar y la información necesaria a proporcionar para poder realizarla, la cual se expresa en formato *URI*. Un ejemplo consistiría en ver la información de contacto de una persona, la cual se podría obtener mediante un *Intent* con la acción *ver* y la *URI* que representa a esa persona. Existe una clase relacionada llamada *IntentFilter*, que es una descripción de qué *Intents* puede gestionar una Actividad. Mediante los *IntentFilters* el sistema puede resolver *Intents*, buscando cuáles posee cada actividad y escogiendo aquel que mejor se ajuste a sus necesidades. El proceso de resolver *Intents* se realiza en tiempo real, ofreciendo dos beneficios:

- Las actividades pueden reutilizar funcionalidades de otros componentes simplemente haciendo peticiones mediante un *Intent*.
- Las actividades pueden ser reemplazadas por nuevas actividades con *IntentFilters* equivalentes.

Service: Un Servicio es un trozo de código que se ejecuta en segundo plano durante un tiempo indefinido sin necesidad de interfaz gráfica, de manera que el usuario puede seguir realizando otras tareas. En caso de que haya múltiples servicios a la vez, se les puede indicar diferentes prioridades según las necesidades.

Content Provider: En *Android*, las aplicaciones pueden guardar su información en ficheros, BBDD, etc. Pero, en caso de que se quiera compartir dicha información con otras aplicaciones, se necesita un *Content Provider*. Un *Content Provider* es una clase que implementa un conjunto estándar de métodos, que permiten a otras aplicaciones guardar y obtener la información que maneja dicho *Content Provider*.

Broadcast Receiver: Este tipo de componentes se utiliza para recibir y reaccionar ante ciertas notificaciones broadcast. Carecen de interfaz gráfica y pueden reaccionar ante eventos como cambio de zonas horarias, llamadas, nivel de batería... Todos los receivers heredan de la clase base `BroadcastReceiver`.

1.2. Android Manifest

En *Android* existe un archivo *XML* llamado *AndroidManifest* que, aunque no forme parte del código principal de la aplicación, es necesario para su correcto funcionamiento. Este archivo es el fichero de control que le dice al sistema qué tiene que hacer con todos los componentes anteriormente mencionados que conforman la aplicación.

Al igual que otros ficheros *XML* de *Android*, el fichero *AndroidManifest* incluye una declaración de espacio de nombres. Esta declaración permite usar una variedad de atributos estándares de *Android*. Los manifiestos incluyen un elemento `<application>`, que define todas las *Activities* que hay disponibles en el *package* definido. Todo *package* presentado al usuario como aplicación de alto nivel necesitará incluir como mínimo un componente `<activity>` que soporte la acción *MAIN* y la categoría *LAUNCHER*, indicando que se trata de la actividad principal que se inicia al ejecutar la aplicación.

Es importante destacar que las aplicaciones de *Android* no tienen permisos asociados a no ser que se configuren previamente en el *AndroidManifest*. De esta manera se evitan modificaciones no deseadas en la información del dispositivo. Por lo tanto, para poder utilizar las funcionalidades protegidas, se deben incluir los *tags* `<uses-permission>` en el *AndroidManifest*, declarando los permisos necesarios.

1.3. Ciclo de vida de una aplicación ⁽⁵⁾

Cada aplicación de *Android* corre en su propio proceso. Éste es creado por la aplicación cuando se ejecuta, y permanece hasta que la aplicación deja de trabajar o el sistema necesita memoria para otras aplicaciones. Una característica fundamental de *Android* es que el ciclo de vida de una aplicación no está controlado por la misma aplicación sino que lo determina el sistema a partir de una combinación de estados (por ejemplo, qué aplicaciones están funcionando, qué prioridad tienen para el usuario o cuánta memoria queda disponible en el sistema). De esta manera, *Android* sitúa cada proceso en una jerarquía de “importancia” basada en los estados comentados. Existen diferentes procesos de acuerdo a esta jerarquía:

1. Un proceso en primer plano es aquel requerido para lo que el usuario está actualmente haciendo. Se considera en primer plano si:

- Esta ejecutándose una Actividad perteneciente a la pantalla con la que el usuario está interactuando.
- Está ejecutando un *BroadcastReceiver*.
- Esta ejecutándose un servicio.

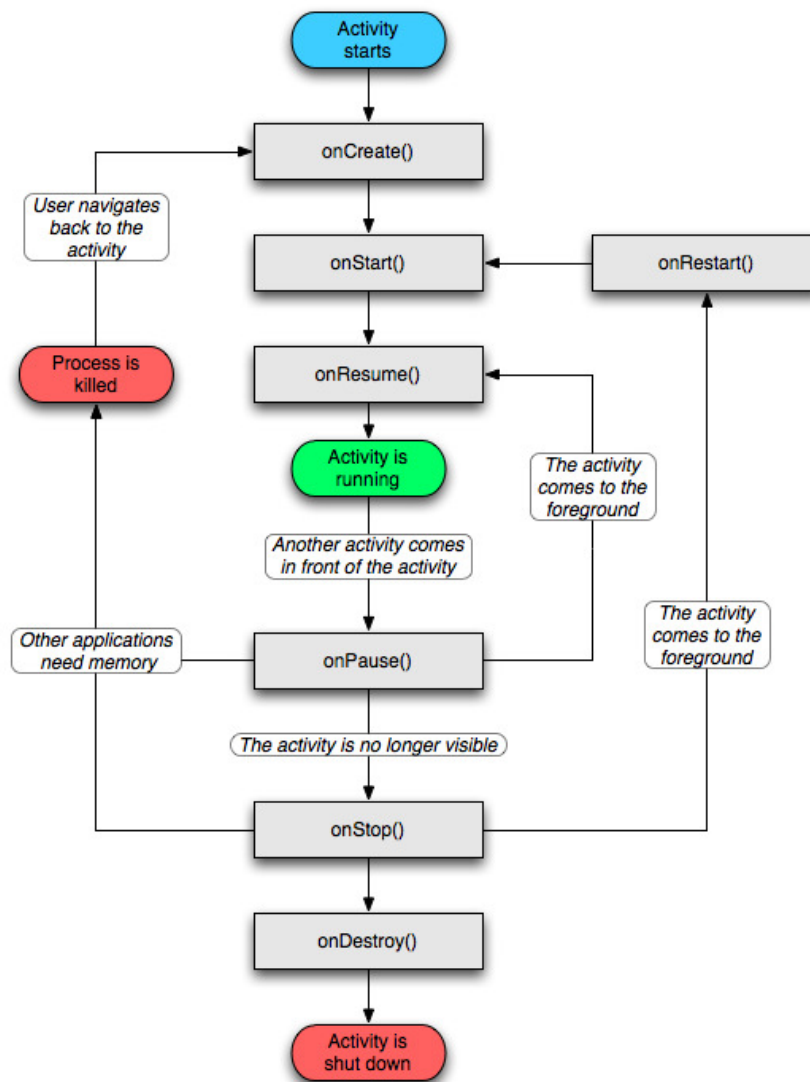
2. Un proceso visible es aquel que contiene una Actividad que es visible al usuario mediante la pantalla, pero no en primer plano (está pausada). Este proceso solo se eliminará en caso de que sea necesario para mantener en ejecución los procesos en primer plano.

3. Un proceso de servicio es aquel que contiene un servicio que ha sido inicializado. No son directamente visibles al usuario y el sistema los mantendrá a no ser que no pueda servir los dos anteriores.

4. Un proceso en *background* es aquel que acoge una actividad que no es actualmente visible al usuario. Mientras estos procesos implementen bien su propio ciclo de vida, el sistema puede eliminarlos para dar memoria a cualquiera de los 3 servicios anteriores.

5. Un proceso vacío es aquel que no contiene ningún componente activo de ninguna aplicación. La única razón para mantener dicho proceso es para mejorar sus inicializaciones posteriores a modo de caché.

Para comprender mejor el ciclo de vida de una aplicación de *Android*, en la siguiente figura se muestra el diagrama de flujo de dicho ciclo, mencionando también los métodos que se llaman durante el transcurso del mismo:



Como se puede observar, una Actividad tiene esencialmente tres estados:

Reanudado (*resumed*): A veces referido como *running* (en ejecución). La actividad se encuentra en primer plano y tiene el foco del usuario.

Pausado (*paused*): Es otra actividad la que se encuentra en primer plano y posee el foco, pero ésta sigue siendo visible. Es decir, la otra actividad se visualiza por encima de la actual sin llegar a ocupar la totalidad de la pantalla o mostrándose parcialmente transparente. Una actividad pausada se encuentra completamente viva (sigue en memoria y mantiene toda la información de estado) pero puede ser eliminada por el sistema en situaciones de muy poca memoria libre.

Detenido (*stopped*): Esta actividad se encuentra completamente oculta por otra (se encuentra en segundo plano). La actividad sigue viva pero no es visible al usuario y el sistema puede eliminarla cuando se necesite memoria para otras actividades.

1.4. Interfaz de usuario

A la hora de diseñar la UI, *Android* permite hacerlo tanto por código *Java* como por ficheros en lenguaje *XML*. La primera manera puede llegar a ser muy compleja y confusa, mientras que la segunda añade los beneficios y la potencia del uso del lenguaje *XML*, además de seguir un modelo de programación en el que se separa la interfaz de usuario de la lógica del programa.

Android define un gran número de elementos personalizados, cada uno representando una subclase de *View*. Para diseñar una interfaz, se anidan los diferentes elementos y se guardan en un fichero *XML* dentro del directorio *res/layout* de la aplicación. Cada fichero describe una sola pantalla, pero esta puede estar compuesta tanto por un elemento simple como por un conjunto de ellos.

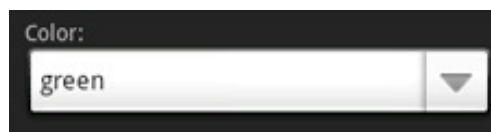
Si la unidad básica funcional de una aplicación de *Android* son las actividades, la unidad básica de la UI son las vistas (*views*) y los grupos de vistas (*viewgroups*):

Views: Las *views* son objetos de la clase base *android.view.View*. Es básicamente una estructura de datos que almacena la presentación de un área de la pantalla del dispositivo, controlando así diferentes parámetros como pueden ser las proporciones, *scrolling*, etc. Mediante la clase *View* podemos implementar subclases llamadas *widgets* que se encargan de dibujar elementos interactivos en la pantalla. Los elementos más comunes son:

Button: Representa un botón, el cual realiza una acción cuando es pulsado. Mediante la propiedad *android:text* asignamos un texto. Además de esta propiedad, podríamos utilizar muchas otras como el color de fondo (*android:background*), estilo de fuente (*android:typeface*), color de fuente (*android:textcolor*), tamaño de fuente (*android:textSize*), etc.



TextView: El control *TextView* es uno de los clásicos en la programación de GUIs, las etiquetas de texto, y se utiliza para mostrar un determinado texto al usuario. Al igual que en el caso de los botones, el texto del control se establece mediante la propiedad *android:text*. El resto de propiedades son las mismas que las de los botones.

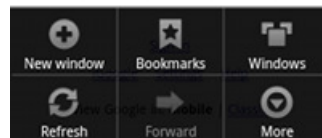


EditText: El control *EditText* es el componente de edición de texto que proporciona la plataforma *Android*. Permite la introducción y edición de texto por parte del usuario, por lo que la propiedad más interesante a establecer, además de su posición, tamaño y formato, es el texto a mostrar (`android:text`).



ImageView: El control *ImageView* permite mostrar imágenes en la aplicación. La propiedad más interesante es `android:src`, que permite indicar la imagen a mostrar. Lo habitual será indicar como origen de la imagen el identificador de un recurso de nuestra carpeta `'/res/drawable'`. Además de esta propiedad, existen algunas otras como las destinadas a establecer el tamaño máximo que puede ocupar la imagen, `android:maxWidth` y `android:maxHeight`.

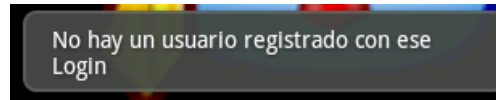
Options Menu: Presenta una interfaz para gestionar los elementos de un menú. Por defecto, cada *Activity* soporta un menú de opciones, que se activa al pulsar la tecla MENU. Se puede añadir elementos a este menú y gestionar los clicks en ellos.



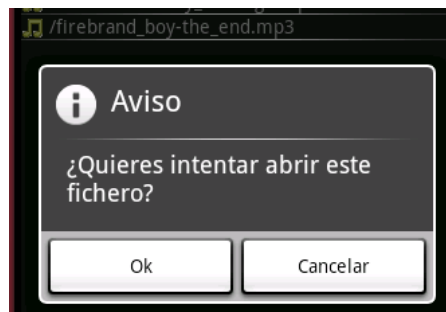
Context Menu: Similar al menú, salvo que se despliega al pulsar sobre algún elemento de la pantalla.



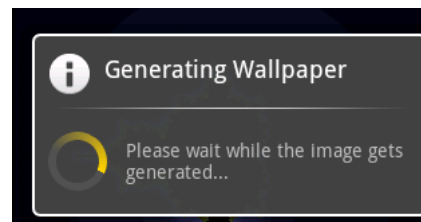
Toast: Es una vista que contiene un pequeño mensaje que se muestra brevemente al usuario. Cuando la vista aparece, se muestra en una vista flotante sobre la aplicación y nunca recibe el foco de acción.



AlertDialog: Al contrario que el *Toast*, este *widget* sí que recibe el foco de acción y se puede interactuar con él.



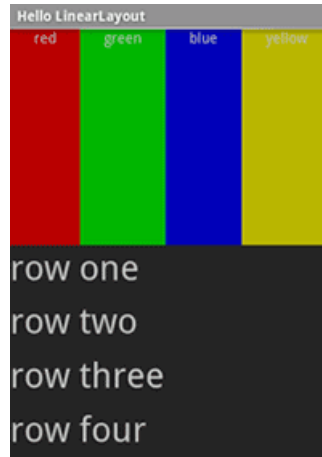
ProgressDialog: Una vista que muestra un indicador de progreso con un mensaje opcional.



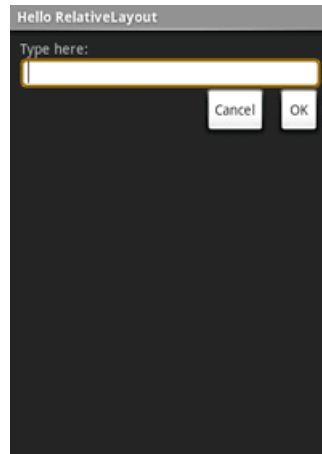
Viewgroups: Un *ViewGroup* es un tipo especial de *View* cuya función es contener y controlar un conjunto de *Views* y/o *ViewGroups*. Cada *ViewGroup* tiene la responsabilidad de medir el espacio disponible y asignárselo a las *Views* contenidas. La manera más corriente de asignar espacio es adaptarse a las dimensiones del contenido de la *View* o ser tan grande como lo permita el *ViewGroup* contenedor. Los *ViewGroups* más utilizados son los siguientes:

LinearLayout: Este *ViewGroup* alinea todos los elementos en una única dirección (vertical u horizontal) y los posiciona, uno detrás de otro, formando una sola columna o fila. *LinearLayout* permite asignar *padding* (o relleno) entre los diferentes elementos, así como especificar su peso en la pantalla, entre otras características.

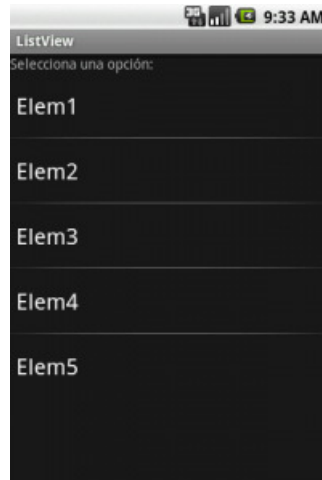
El caso del peso es interesante debido a que, según el tipo pantalla que estemos mostrando, habrá elementos que nos interesará que se muestren más que otros. Mediante el peso, podemos expandir los elementos hasta que rellenen el espacio sobrante en la pantalla.



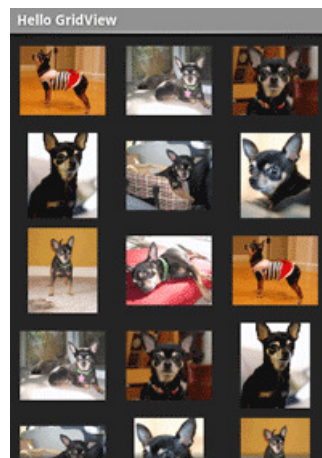
RelativeLayout: Permite a los nodos hijo especificar su posición relativa de dos formas: uno respecto a otro (definida por el ID), o respecto al nodo padre. Si se usa *XML* para especificar esta disposición, se precisa del elemento de referencia antes de referirlo.



ListView: Es una colección ordenada de elementos seleccionables. En caso de existir más opciones de las que se pueden mostrar, se podrá hacer *scroll* sobre la lista para acceder al resto de elementos.



GridView: Se trata de otro *ViewGroup* importante, utilizado en esta aplicación. El control *GridView* de *Android* presenta al usuario un conjunto de opciones seleccionables distribuidas de forma tabular, o dicho de otra forma, divididas en filas y columnas.



Dada la naturaleza del control sus propiedades más importantes son:

- `android:numColumns`, indica el número de columnas de la tabla, o "auto_fit" si queremos que sea calculado por el propio sistema operativo a partir de las siguientes propiedades.
- `android:columnWidth`, indica el ancho de las columnas de la tabla.

- `android:horizontalSpacing`, indica el espacio horizontal entre celdas.
- `android:verticalSpacing`, indica el espacio vertical entre celdas.
- `android:stretchMode`, indica qué hacer con el espacio horizontal sobrante. Si se establece al valor “`columnWidth`” este espacio será absorbido a partes iguales por las columnas de la tabla. Si por el contrario se establece a “`spacingWidth`” será absorbido a partes iguales por los espacios entre celdas.

Adapter Views: Para poder interactuar entre el código de la aplicación y los elementos de la UI se precisa de una subclase llamada *AdapterView*. Estos objetos, una vez instanciados en el código, pueden realizar dos tipos de tareas:

- Rellenar la pantalla con datos.
- Atender a las selecciones de los usuarios.

Además de proveer de datos a los controles visuales, el adaptador también será responsable de generar a partir de estos datos las vistas específicas que se mostrarán dentro del control de selección. Por ejemplo, si cada elemento de una lista estuviera formado a su vez por una imagen y varias etiquetas, el responsable de generar y establecer el contenido de todos estos “sub-elementos” a partir de los datos será el propio adaptador.

1.5. Recursos en Android

Los recursos en *Android* son ficheros externos (no de código) que son usados y compilados dentro de la aplicación. *Android* soporta diferentes tipos de recursos como *XML*, *PNG* o *JPEG*. Los ficheros *XML* tienen diferente formato dependiendo de qué describan.

Los recursos se exteriorizan respecto al código. Los ficheros *XML* se compilan dentro un archivo binario por razones de eficiencia y los *strings* se comprimen en una forma más eficiente de almacenamiento. Es por esta razón que hay diferentes tipos de recursos en *Android*. Los tipos de recursos más habituales son los siguientes:

- **Drawable:** Imágenes en formato *PNG*, *JPG* o *GIF* que se compilan como bitmaps (mapas de bits).
- **Layout:** Ficheros *XML* que definen la estructura de una vista de pantalla.
- **Menu:** Ficheros *XML* que definen los menús de la aplicación, tales como menús de opciones, menús contextuales...
- **Values:** Ficheros *XML* que definen varios tipos de recursos como strings, arrays, colores...

El valor de un atributo o recurso puede ser una referencia a otro recurso. Esto se utiliza frecuentemente en ficheros *layout* para referenciar strings (para que puedan ser traducidos) o imágenes (que pertenecen a otro fichero).

Para hacer uso de un recurso desde el código, tan solo es necesario conocer el tipo de recurso al que se desea acceder y el ID asignado a dicho recurso. Se utiliza la siguiente sintaxis:

```
R.resource_type.resource_name
```

2. Librerías de reconocimiento

Android `FaceDetector` consiste en una librería básica de reconocimiento de caras incorporada en *Android*. Al parecer, la clase utiliza como patrón de reconocimiento los ojos de una cara; no utiliza ningún patrón adicional como las dimensiones de la cabeza o la detección de la nariz o la boca.

Para un correcto funcionamiento, el algoritmo de detección de caras se ha de ejecutar sobre una imagen con unas condiciones de luz adecuadas. Esto implica el uso de flash en condiciones bajas de luz, como escenas nocturnas. El uso de gafas no supone un impedimento a la hora de detectar los ojos de cada rostro. Sin embargo, la posición de la cara de una persona es determinante. El algoritmo es incapaz de encontrar a una persona cuando se encuentra ladeada, ya que no obtiene la información de ambos ojos y no es capaz de calcular su punto medio. Además, el algoritmo utilizado es ineficaz a la hora de posicionar rostros cuya línea de unión entre los ojos posee cierto grado de inclinación.

La clase *FaceDetector* identifica las caras de la gente dado un objeto gráfico tipo *Bitmap*. Se debe configurar al tamaño de la imagen a analizar y es necesario establecer un límite máximo de rostros a detectar. El número máximo que se puede solicitar es 64. Es interesante destacar que, en caso de que el número solicitado sea mayor al de rostros presentes en la imagen, no supondrá un decremento de velocidad. También, cuanto mayor es el tamaño de la imagen más sencillo es detectar la posición de las caras.

El método encargado de realizar la detección es `findFaces`. Éste recibe como parámetros el objeto *Bitmap* y un array de objetos *Face* (que se debe encontrar inicializado con un tamaño igual al número de rostros permitidos). La imagen debe estar en formato RGB565 (5 bits para representar el color rojo, 6 para el verde y otros 5 para el azul). El objeto *Face* contiene toda la información sobre la localización de la cara. Éste posee tres funciones interesantes: `eyesDistance` devuelve la distancia entre los ojos, `getMidPoint` devuelve las posiciones x e y del punto medio entre éstos, y `pose` retorna la pose de una cara, es decir, las rotaciones o ángulos alrededor de los ejes X, Y y Z.

3. API de Facebook ⁽⁶⁾

En el núcleo de *Facebook* se encuentra el grafo social, que contiene a las personas y las conexiones que éstas tienen con todo lo que les importa. La Graph API presenta una vista del grafo social de *Facebook*, representando los objetos del grafo (personas, fotos...) y las conexiones existentes entre ellos (amistades, contenido compartido...).

Cada objeto tiene un identificador (ID) único. Se puede acceder a las propiedades de un objeto mediante la petición: <https://graph.facebook.com/ID>. Alternativamente, se puede acceder a las personas y páginas que poseen nombre de usuario utilizando éste en lugar de su ID. Todas las respuestas del servidor se reciben en forma de objetos *JSON*.

Los objetos que componen la plataforma de *Facebook* son:

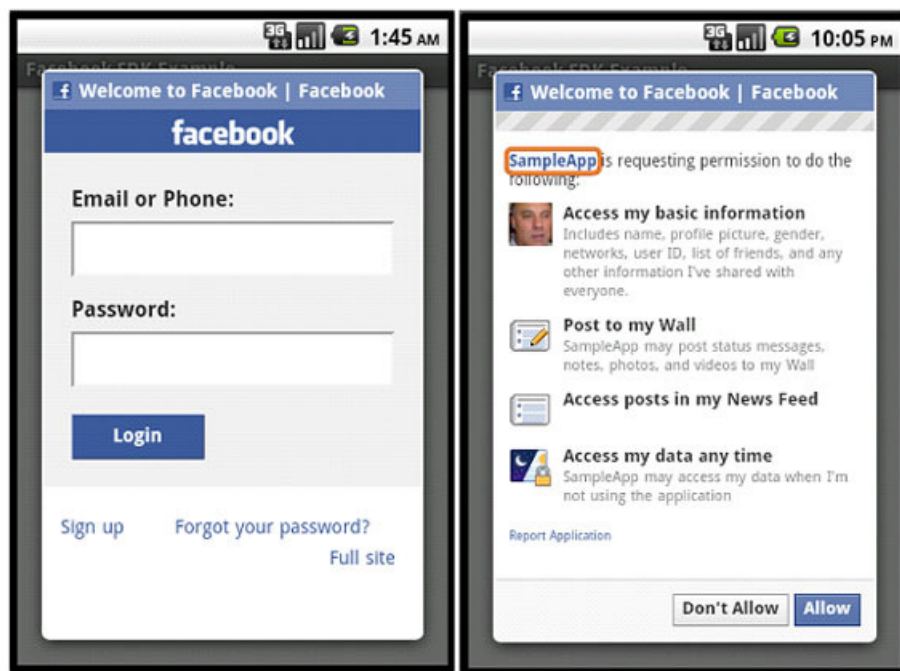
- Usuarios
- Páginas
- Eventos
- Grupos
- Aplicaciones
- Mensajes de estado
- Fotos
- Álbumes
- Fotos de perfil
- Vídeos
- Notas
- Check-ins

Todos los objetos del grafo social de *Facebook* están conectados entre sí mediante relaciones. A estas relaciones se les denomina conexiones. Se pueden examinar las relaciones entre objetos usando la URL: https://graph.facebook.com/ID/CONNECTION_TYPE. Las conexiones soportadas para usuarios y páginas incluyen:

- Amigos
- Fuentes de noticias
- Fuentes del perfil (muro)
- 'Gustos'
- Películas

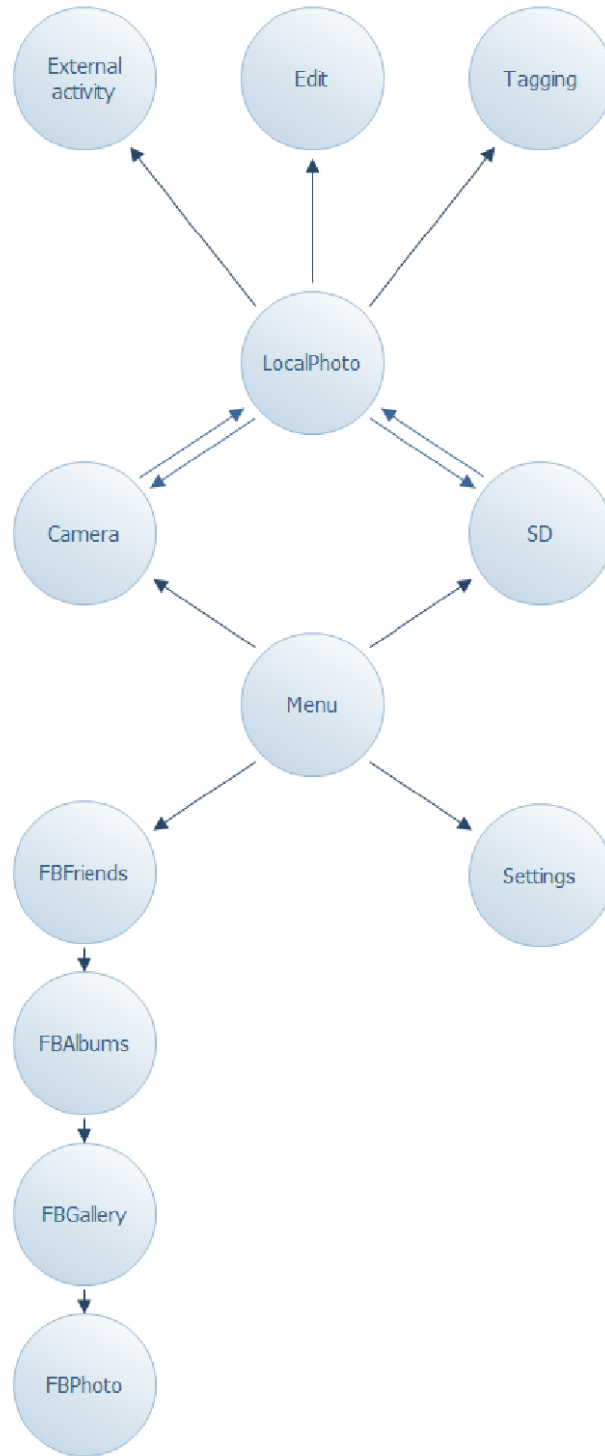
- Música
- Libros
- Notas
- Permisos
- Fotos etiquetadas
- Álbumes
- Vídeos etiquetados
- Vídeos subidos
- Eventos
- Grupos
- Check-ins

Para realizar cierto tipo de operaciones es necesario poseer permisos. De forma previa, se deberá obtener un *access token* (ID de sesión) para identificar al usuario. Para ello, la aplicación pide al usuario sus datos mediante un cuadro de diálogo de autenticación. Esto permite al usuario conceder a la aplicación permiso para acceder a su información.



4. Creación de la aplicación

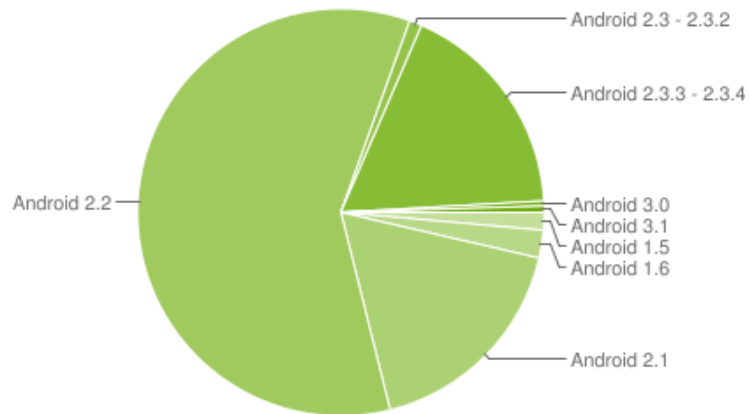
4.1. Diagrama de actividades



4.2. Requisitos

La aplicación se ha adaptado de forma que se pueda ejecutar sobre dispositivos con distinta resolución, tamaño de pantalla y densidad de píxel. Esto se hace utilizando recursos (gráficos) específicos para las distintas densidades de pantalla, evitando el uso de interfaces absolutas y utilizando unidades relativas para las dimensiones de los elementos de la UI.

Por otro lado, se ha utilizado el *SDK* de *Android* para la versión 2.1 del sistema operativo. Las razones de esto han sido básicamente dos: la tasa de usuarios de las versiones anteriores es muy baja y la gran mayoría ya dispone de la versión utilizada, como se observa en la gráfica de más abajo, y esto, a su vez, asegura que la aplicación se ejecute en dispositivos con el hardware necesario para funcionar de forma eficiente.



4.3. Desarrollo de la aplicación

El desarrollo de este apartado se ha estructurado por actividades, de forma que se irá profundizando en la metodología empleada por la aplicación y revisando nuevos conceptos sobre *Android* y sobre el *API* de *Facebook*, así como de procesamiento de imágenes y reconocimiento de caras.

4.3.1. Menu.java

Se trata de la actividad principal que se inicia al ejecutar la aplicación. Consiste en un menú que da acceso a las distintas funciones de ésta, además de iniciar sesión en *Facebook* (solicitando las credenciales necesarias) en caso de que no se haya hecho previamente.

En *Android*, toda clase ejecutable debe heredar de la clase base *Activity*. El método principal de la clase (aquel que sirve de punto de inicio) es `onCreate`.

Esta clase posee una serie de atributos relacionados con el inicio de sesión en *Facebook*. El primero de ellos es un objeto *Facebook*, que contiene el identificador asociado a la aplicación (proporcionado al darla de alta en la web para desarrolladores, como se detalla en el anexo II). El resto de atributos son identificadores en forma de *string* que utilizaremos más adelante.

Al principio del método principal, hay que asociar la actividad actual con el fichero *XML* que contiene la interfaz gráfica. Esto se hace mediante el método `setContentView`. A continuación, como es el caso, podemos referenciar los distintos objetos que componen la interfaz para asignarles una determinada funcionalidad, mediante el método `findViewById`. Para ello, dichos objetos deben poseer un ID en el *XML*.

En nuestro caso, para manejar los clicks del usuario, crearemos un objeto *Listener* y lo asociaremos a cada botón mediante el método `setOnClickListener`. El método `onClick` del objeto *Listener* contendrá el código a ejecutar cada vez que se realice un click sobre el botón. Para lanzar una nueva actividad, crearemos un objeto *Intent* especificando el contexto en el que se crea (la actividad actual) y la clase a ejecutar. Por último, lanzaremos la nueva actividad mediante el método `startActivity`, pasando el *Intent* como argumento.

Además, como se ha comentado, la clase realiza un inicio de sesión en Facebook al ejecutarse. Para ello, en primer lugar, se comprueba si existe una sesión previa válida. En caso negativo se ejecuta el método `login`. Éste a su vez llama al método `authorize` para autenticarse y otorgar los permisos necesarios para la aplicación (como puede ser la visualización o la publicación de fotos). Dicho método recibe un array de *string* con la lista de permisos y un objeto *Listener* encargado de manejar la respuesta (ya sea satisfactoria o errónea).

Si la operación es exitosa, el método `saveCredentials` se encargará de guardar el ID de sesión y la fecha de expiración (en nuestro caso, nula). Esto se realiza mediante el objeto *SharedPreferences* ⁽⁷⁾⁽⁸⁾, que almacena datos privados permanentes en forma de pares clave-valor. Se trata de un tipo específico de almacenamiento para preferencias de usuario, de forma alternativa al uso de ficheros o bases de datos. Los datos se mantienen más allá de la sesión del usuario (aunque la aplicación vea terminada su ejecución). En caso de fallar la operación, se mostrará un mensaje de texto al usuario.

4.3.2. Camera.java

Camera.java es una sencilla actividad que se encarga de ejecutar la aplicación nativa de la cámara, obtener la imagen resultante y enviarla a la siguiente actividad, encargada de mostrar la fotografía al usuario.

Esta clase, así como el resto de las que conforman la aplicación, incluye un método denominado `onConfigurationChanged`. Android, por defecto, destruye y reinicia la actividad actual ante un cambio en la configuración (por ejemplo, apertura del teclado o cambio de orientación de la pantalla). Con dicho método se modifica este comportamiento. Para ello, primero es necesario añadir el siguiente atributo, en la etiqueta correspondiente a la actividad actual, dentro del Android Manifest:

```
android:configChanges="keyboard|keyboardHidden|orientation"
```

Ante los cambios de configuración especificados, la actividad seguirá en ejecución y llamará al método mencionado, ejecutando el código contenido.

Para ejecutar la aplicación nativa de la cámara haremos uso del objeto *Intent* de una forma distinta a la utilizada hasta ahora. En primer lugar, es necesario indicar a la aplicación sobre qué archivo va a almacenar la imagen resultante. Para ello, primero crearemos un objeto *File* especificando la ruta del directorio donde almacenar la imagen, y luego crearemos dicho directorio en caso de no existir en la tarjeta SD. En segundo lugar, crearemos un fichero con extensión `.jpg` sobre ese directorio y obtendremos la ruta en formato *URI*. Por último, crearemos un objeto *Intent* especificando la acción que va a realizar (en este caso `MediaStore.ACTION_IMAGE_CAPTURE`, indicando la captura de imágenes) y añadiremos al objeto la ruta. Para lanzar la nueva actividad utilizaremos el método `startActivityForResult`. Éste se diferencia de `startActivity` en que permite a la sub-actividad transmitir de vuelta determinada información al terminar su ejecución.

Con el método anterior entra en juego una nueva función, `onActivityResult`, que se ejecuta al terminar la sub-actividad y permite obtener la información relacionada. Esta función tiene como parámetros de entrada un código de petición (utilizado con el método anterior para identificar la sub-actividad), un código de resultado (respuesta de estado) y la información resultante.

Una vez realizada la fotografía, la actividad actual lanzará la sub-actividad encargada de mostrarla, pasándole como parámetro la ruta de la imagen mediante la función `setData`, que se ejecuta sobre el objeto *Intent*.

4.3.3. SD.java

Actúa de forma similar a la clase anterior. Se ejecuta la galería fotográfica por defecto del dispositivo y, una vez seleccionada una imagen, se obtiene de vuelta su ruta.

Utilizaremos un objeto *Intent* con la acción `Intent.ACTION_GET_CONTENT`, que permite al usuario seleccionar datos de un tipo determinado. Mediante el método `setType` indicaremos el tipo *MIME*.

En nuestro caso, `image/*` indica que se trata de imágenes en cualquier formato.

En el método `onActivityResult`, utilizaremos la función `getData` del objeto *Intent* para obtener la ruta de la imagen en formato *URI* (ésta es devuelta por la galería fotográfica).

4.3.4. LocalPhoto.java

La clase actual se encarga de mostrar la imagen seleccionada anteriormente y ofrecer una serie de operaciones mediante una interfaz de menú. Las operaciones son: elección de una nueva imagen (regresa a la actividad anterior), edición (lanza `Edit.java`), compartición (ejecutando la aplicación externa correspondiente) y etiquetado (lanza `Tagging.java`).

El método `onCreate` comienza obteniendo el objeto *Intent* enviado desde la actividad anterior mediante la función `getIntent`. A continuación, obtenemos el *URI* asociado que contiene la ruta de la imagen a mostrar.

Después, haremos una serie de operaciones básicas previas a la visualización de la imagen. En primer lugar, referenciamos el *XML* de la *UI* mediante el método `findViewById`. A continuación, mediante el objeto *BitmapFactory.Options* estableceremos la codificación a utilizar en la imagen (RGB565). Por último, llamaremos a las funciones `scaleFactor` y `createBitmap`, encargados de escalar la imagen a las dimensiones utilizadas por el API de Facebook (además de permitir mejorar el rendimiento) y de asociarla a la interfaz de usuario.

El método `scaleFactor` establece como tamaño máximo de borde 2048px (que es el máximo permitido por Facebook a la hora de subir la imagen al servidor). Para reducir la imagen utilizaremos un factor de escala en función de la diferencia de tamaño entre la imagen original y la imagen de destino. Este número se almacenará en un atributo del objeto *BitmapFactory.Options* para escalar la imagen a la hora de decodificarla.

En primer lugar, con el atributo `inJustDecodeBounds` del objeto *BitmapFactory.Options* especificamos que sólo queremos obtener las dimensiones de la imagen. Con el método `openFileDescriptor`, abriremos el fichero en modo lectura a partir de la *URI* proporcionada, obteniendo un objeto de tipo *FileDescriptor*. A partir de este objeto podemos decodificar la imagen (debido al atributo `inJustDecodeBounds` sólo obtendremos sus dimensiones) con el método `decodeFileDescriptor`. Por último, mediante un sencillo algoritmo, obtendremos el factor de escala. Este algoritmo obtiene el valor del borde más largo y lo compara con el valor máximo permitido. En caso de ser mayor, se divide su valor entre el máximo y se redondea hacia arriba, obteniendo el factor de escala (debe ser un entero).

El método `createBitmap` aprovecha el *FileDescriptor* previamente abierto para decodificar la imagen (esta vez en su totalidad), almacenándola en un objeto de tipo *Bitmap*. A continuación, se encarga de cerrar el fichero puesto que no se va a volver a utilizar, y asocia la imagen a la interfaz mediante el método `setImageBitmap`.

Además, cabe mencionar el uso del método `onDestroy`, que se ejecuta al final del ciclo de vida de la aplicación (esto es, cuando se detiene mediante la llamada `finish` o cuando el proceso es destruido). Este método se encarga de llamar a otro, `clearBitmap`, el cual se encarga de borrar todas las referencias en memoria a la imagen y de utilizar la función `recycle` para liberar la memoria asociada al bitmap.

Por último, el código incluye la creación de un menú de opciones ⁽⁹⁾. Éste se muestra en la parte inferior de la pantalla al pulsar la tecla MENU. Cuando se crea por primera vez se llama al método `onCreateOptionsMenu`, donde deberemos instanciar el fichero XML correspondiente (en nuestro caso, `localphoto_menu.xml`), que contiene los elementos que forman parte de la interfaz del menú. El método `onOptionsItemSelected` incluye las acciones a realizar para cada una de las opciones. La opción de compartir (share) incluye elementos que no hemos visto. Hace uso de un objeto `Intent` con la acción `ACTION_SEND`, utilizada para enviar información, y mediante la función `Intent.createChooser` muestra un menú emergente con las aplicaciones instaladas capaces de realizar dicha acción.

4.3.5. Edit.java

Esta actividad dispone de un menú de opciones para aplicar efectos sobre la imagen, guardarla o revertirla al original. Para regresar a la actividad anterior sin aplicar ningún cambio se utiliza la tecla `BACK` del dispositivo.

La opción de efectos básicos (*basic_effects*) incluye un submenú en el *XML* con la lista de efectos a aplicar (en nuestro caso, blanco y negro). El efecto 'blanco y negro' se basa en el método `grayscaleEffect`:

En un principio se implementó de forma poco eficiente, recorriendo el mapa de bits mediante dos bucles anidados. Para cada píxel se obtenía su información de color (mediante la función `getPixel` de *Bitmap*), se aplicaba un algoritmo de conversión a escala de grises ($R,G,B = R*0.3 + G*0.59 + B*0.11$) y se volcaba sobre el píxel correspondiente de la nueva imagen (función `setPixel`).

La implementación actual realiza una transformación de la imagen mediante el uso de una matriz de colores (*ColorMatrix*). Con la función `setSaturation` ponemos la saturación a 0, es decir, en escala de

grises y asociamos la matriz al objeto `ColorMatrixColorFilter`, que actúa como un filtro de color. Este filtro lo asociamos a un objeto `Paint`, que almacena la información de dibujado. Por último, creamos un nuevo `Bitmap` con las mismas dimensiones del original, lo asociamos a un objeto `Canvas`, que implementa las llamadas sobre el dibujado de la imagen, y realizamos dicho dibujado a partir del `Bitmap` original y del objeto `Paint` mediante la función `drawBitmap`. Después, para liberar la referencia del objeto `Canvas` al `Bitmap`, pondremos su valor a `null` y haremos una llamada al `Garbage Collector`.

La opción de revertir los efectos (`reject_changes`) simplemente borra la imagen actual mediante la función `clearBitmap`, mencionada en actividades anteriores, y ejecuta de nuevo las funciones `scaleFactor` y `createBitmap` para redecodificar la imagen original.

La opción de guardado (`accept_changes`) comprueba, en primer lugar, que se hayan realizado cambios, en cuyo caso ejecuta el método `saveFile`. Éste se encarga de crear un nuevo fichero para almacenar la imagen modificada y de obtener su ruta en formato `URI`. Para volcar la información del `Bitmap` al fichero, crearemos un objeto `FileOutputStream` asociado al archivo, y transferiremos los datos con el método `compress` de `Bitmap`. Este método acepta como argumentos el formato de la imagen a codificar y la calidad de ésta (inversa al nivel de compresión), y devuelve un `booleano` sobre el resultado de la operación. Tras esto, cerraremos el `FileOutputStream`. Una vez almacenada, se escribirá sobre un `Intent` la nueva ruta y se devolverá la ejecución (junto con el resultado) a la actividad anterior.

4.3.6. Tagging.java

Esta actividad es la encargada de realizar el etiquetado de las personas presentes en la imagen y de subirla al servidor de Facebook ⁽⁶⁾.

En primer lugar, la actividad solicita al servidor la lista de contactos, de forma que ya se encuentre cargada cuando se realice el etiquetado. Al tratarse de una lista dinámica, para poder cargar los ítems que la componen hemos de instanciar la vista `XML` mediante el objeto `LayoutInflater` y la función `inflate`. Además, asociaremos a la lista un objeto `OnItemClickListener`, que permitirá obtener con cada click la posición del ítem dentro de ésta y así asociar su nombre e identificador a la etiqueta correspondiente. Una vez asociados, realizaremos un recorte (la implementación se detallará más adelante) del rostro pertinente para almacenarlo en la memoria SD y así poder realizar en un futuro una base de datos de contactos.

Para realizar la carga dinámica de la lista, hemos de utilizar un adaptador que se encargue de manejar los datos en forma de array de la lista (o `ListView` ⁽¹⁰⁾). Dicho adaptador es una instancia de la clase `FriendsArrayAdapter.java`, que comentaremos más adelante. A continuación, recuperaremos los datos

de sesión (mediante el método `restoreCredentials`) y realizaremos la solicitud. Debido a la implementación de la API de Facebook, es necesario realizar el proceso de petición en dos fases: una para solicitar la información del propio usuario actual y otra para solicitar la de su lista de contactos. Para evitar el bloqueo de la interfaz de usuario, la solicitud se realiza sobre un hilo especial del SDK de Facebook, el objeto `AsyncFacebookRunner`. Dicha petición es ejecutada mediante el método `request`, que acepta como parámetros la ruta URL al recurso ('me' solicita la información personal del usuario actual), una cadena de consulta o *query string* (en nuestro caso, pidiendo los campos de nombre e ID) y un objeto `Listener` para manejar la respuesta del servidor. Una vez terminada la solicitud, se ejecutan los métodos que implementa el `Listener` en función del tipo de respuesta recibida (satisfactoria o no). En caso de error, mostraremos un mensaje al usuario, y en caso de éxito procesaremos los datos obtenidos. La respuesta del servidor se encuentra en formato *JSON (JavaScript Object Notation)* por lo que es necesario importar sus librerías. Para procesar la respuesta, la encapsularemos en un objeto *JSON* y extraeremos los campos string de nombre e ID. Después, los guardaremos como atributos de un nuevo objeto `Friend` y lo añadiremos al `ArrayList` de contactos. Seguiremos el mismo proceso para los contactos del usuario, con la diferencia de que la respuesta recibida contiene un array de datos (*JSONArray*). Debemos recorrer dicho array extrayendo en cada iteración el objeto *JSON* referente a cada contacto. Finalmente, ejecutaremos la función `notifyDataSetChanged` para actualizar la vista que contiene el listado de contactos.

A continuación, la actividad decodifica la imagen a etiquetar para mostrársela al usuario y ejecuta un algoritmo automático para detectar la posición de los rostros de la gente (función `getFaces`).

Previo al algoritmo obtendremos las dimensiones de la imagen y de la pantalla así como la relación de aspecto de cada uno (ésta se calcula dividiendo el ancho por el alto). Las dimensiones de la pantalla se obtienen con los métodos `getWidth` y `getHeight` del objeto `Display`. Las de la imagen se consiguen a partir de los atributos `outWidth` y `outHeight` del `BitmapFactory.Options` asociado.

El método `getFaces` se basa en el uso del objeto `FaceDetector` de la API de *Android*. Este posee los métodos para encontrar las caras de las personas en una imagen dada (utiliza un algoritmo sencillo basado en la detección de los ojos), y tiene en sus atributos las dimensiones de la imagen a analizar, así como el número máximo de caras que se pueden detectar. Se creará un array de objetos `Face` (contienen la localización del rostro) al tamaño del máximo permitido y llamaremos al método `findFaces` que inicializará el array y devolverá el número de caras detectadas. Después, recorreremos dicho array y para cada cara obtendremos el punto medio entre los ojos. Este valor equivaldrá a las coordenadas que tendrá la etiqueta.

Aquí surge un problema: estas coordenadas son calculadas sobre la imagen original, la cual puede estar escalada para adaptarse a las dimensiones de la pantalla. Por tanto, es necesario transformar

dichas coordenadas a valores reales sobre la pantalla a la hora de situar el marco que rodeará la cabeza de una persona. De esto se encarga el método `getScrCoords`. Después de varias deliberaciones, se llegó a la conclusión de que a la hora de adaptar la imagen a las dimensiones de la pantalla intervenía el factor de la relación de aspecto. Una imagen con una relación de aspecto distinta a la del dispositivo presentará bordes negros en alguno de sus lados al realizar su escalado, por lo que se deberá tener en cuenta cuando se calculen las coordenadas correspondientes.

Se observa que, independientemente de la orientación del dispositivo, cuando la relación de aspecto de la imagen con respecto a la de la pantalla es mayor, se ajusta el ancho, cuando es menor se ajusta el alto y cuando es igual se ajustan ambos. Cuando son iguales se ejecuta un algoritmo sencillo. Primero se calcula el factor de escala (dividiendo el ancho de la imagen entre el de la pantalla, aunque serviría igualmente el alto) y después calculamos las coordenadas X e Y en la pantalla dividiendo las coordenadas de la imagen por el factor de escala. Si las proporciones son distintas, el algoritmo se complica para aquella dimensión en la que se visualizan bordes negros. Tomemos como ejemplo el caso en el que se ajusta el ancho mientras que el alto posee bordes negros. El factor de escala se calculará dividiendo el ancho de la imagen entre el de la pantalla. La coordenada X se calculan sencillamente de forma similar a la antes mencionada. Sin embargo, la coordenada Y se calcula siguiendo la siguiente fórmula:

```
scrY=(imgY/scaleFactor)+((scrHeight-(imgHeight/scaleFactor))/2);
```

La primera parte del algoritmo realiza el escalado de la dimensión Y, mientras que la segunda corrige el desplazamiento causado por la aparición de bordes negros. Este desplazamiento corresponde a la longitud de un solo borde negro, el superior. Por tanto, al alto de la pantalla restaremos el alto de la imagen escalada, obteniendo el espacio sobrante, y éste lo dividiremos entre 2.

Una vez calculadas las coordenadas, se añade la etiqueta mediante el método `addTag`. En primer lugar, instanciamos un objeto *Tag* que contendrá el nombre e ID del contacto y las coordenadas sobre la imagen, las cuales asignaremos. Crearemos un *ImageView*, que es un elemento de la interfaz que permite mostrar una imagen (este contendrá el marco que rodeará las caras), y lo asociaremos a la creación de un menú contextual (que comentaremos más adelante) para poder asociar a la etiqueta un contacto de la lista o poder eliminarla. Por último, añadiremos los objetos a sus respectivos arrays y a la interfaz actual.

Tras la detección de rostros, el método `onCreate` asigna a la imagen dos *Listeners*. El primero de ellos se ejecuta al tocar sobre la imagen, capturando las coordenadas del evento y transformándolas a coordenadas sobre ésta, de forma que se envíen posteriormente al servidor. El algoritmo trabaja de forma similar al anterior. Podemos ver, para el caso en que se ajusta el ancho, que se sigue el siguiente cálculo para la coordenada Y:

```
imgY=(scrY-((scrHeight-(imgHeight/scaleFactor))/2))*scaleFactor;
```

La parte subrayada corresponde al cálculo de la longitud que ocupa el borde negro sobre la pantalla. Éste se resta a la coordenada Y de la pantalla, corrigiendo el error de desplazamiento y obteniendo la coordenada Y correspondiente a la imagen (escalada). Multiplicando por el factor de escala obtenemos dicha coordenada sobre las dimensiones originales de la imagen.

El segundo *Listener* se ejecuta tras una pulsación larga sobre la imagen y añade una etiqueta con los valores de posición previamente calculados.

Como se ha mencionado anteriormente, las etiquetas tienen asociadas un menú contextual. La implementación de este menú se encuentra en los métodos `onCreateContextMenu` y `onContextItemSelected`.

El primero de ellos se ejecuta al crear el menú por primera vez. En dicho método, obtenemos el elemento visual para el que se abre el menú, es decir, el objeto *ImageView* de la etiqueta. A continuación, buscamos su posición dentro del *ArrayList* y a partir de ésta obtenemos el objeto *Tag* correspondiente. En función de si ya tiene asignado un nombre de contacto o no, se mostrará en la cabecera de título del menú.

El método `onContextItemSelected` contiene la implementación para cada una de las opciones del menú. La opción de asignar un contacto crea y ejecuta un *AlertDialog* ⁽¹¹⁾ o cuadro de diálogo, que es una ventana emergente para mostrar información o requerir acciones al usuario, y cuya implementación viene dada en el método `onCreateDialog`. Mediante la función `setView` establecemos el contenido del cuadro de diálogo, en nuestro caso la lista de contactos previamente cargada. La opción de eliminar etiqueta elimina la vista de la jerarquía de elementos del *XML* y también, junto al objeto *Tag*, del *ArrayList*.

Como se mencionó al principio del apartado, cuando se selecciona un contacto de la lista anterior, se realiza un recorte de la imagen correspondiente al rostro detectado. De esto se encarga la función `cropImage`. Para ello, crea un *Bitmap* con las dimensiones de la etiqueta (132px) e instancia un objeto *Canvas*, asociándolo a dicho *Bitmap*. El método `drawBitmap` utiliza dos rectángulos para definir la región de origen a copiar y la región de destino sobre la que dibujar. El primer rectángulo viene definido por las coordenadas de la etiqueta, mientras que el segundo viene definido por las dimensiones de ésta. A continuación, el método creará un fichero con el ID y nombre de contacto como nombre, y volcará el nuevo *Bitmap* sobre éste. Por último, liberaremos los recursos para dejar espacio libre en memoria.

Además, la actividad dispone de un menú de opciones. La primera de ellas permite añadir un pie de foto a la imagen. Ejecuta un *AlertDialog* cuya interfaz viene definida por el fichero *caption_dialog.xml*. Primero se obtiene la referencia al elemento *EditText*, que consiste en una caja de texto sobre la que puede teclear el usuario. Después se definen dos botones de acción que implementan un *OnClickListener*. El botón de confirmación captura el texto tecleado en la caja de texto y lo guarda en una variable global, mientras que el otro cierra el cuadro de diálogo mediante la función `cancel`.

La segunda de las opciones del menú es la que permite subir la imagen, con la información de etiquetado, al servidor. Para ello, realizaremos una petición de tipo *POST* con la URL *'me/photos'*. En caso de no existir ningún álbum para la aplicación, Facebook lo crea automáticamente. La petición tendrá como parámetros (*query string*) el pie de foto (cuyo identificador es *'message'*) y la imagen en forma de array de bytes (cuyo identificador es *'picture'*). Para convertir el objeto *Bitmap* en un array de bytes, en primer lugar utilizaremos la función `compress` sobre un *ByteArrayOutputStream*. A partir de este objeto podemos obtener su tipo primitivo mediante la función `toByteArray`.

Una vez realizado el envío de la imagen, enviaremos la información de etiquetado. De esto se encarga el *Listener* asociado a la petición anterior. En la respuesta obtendremos el ID de la fotografía creada y con éste realizaremos la petición de creación de *tags* o etiquetas. Para ello, recorreremos el array de etiquetas y, para cada una de ellas, realizaremos una petición al servidor con la URL *'num_id/tags'*, teniendo como parámetros el ID del usuario a etiquetar (con identificador *'to'*) y las coordenadas del *tag*. Por la forma de trabajar de *Facebook* las coordenadas requieren ser convertidas a porcentajes. Esto se realiza multiplicando la coordenada por 100 y dividiendo el resultado por el ancho o por el alto, según corresponda.

Por último, cabe mencionar la implementación del método `onDestroy`, que libera el *Bitmap* correspondiente a la imagen y borra el contenido de los *ArrayList*.

4.3.7. FriendsArrayAdapter.java

La clase se compone de una serie de atributos, el constructor de la clase y el método `getView`, que se encarga de actualizar el contenido del *ListView*.

El método obtiene el objeto *View* (cada uno de los ítems de la lista) a actualizar. Si se encuentra previamente creado (debido a que *Android* recicla los *Views* que han quedado fuera del área visual del scroll) no hará falta inicializarlo. En caso contrario, deberemos instanciar el fichero *XML* que lo define (viene dado por uno de los atributos de la clase). En cualquier caso, a partir de su posición, obtendremos el contacto correspondiente dentro del *ArrayList* y mostraremos su nombre como texto del ítem.

4.3.8. FBFriends.java

Esta actividad se encarga de mostrar al usuario la lista de contactos, permitiendo seleccionarlos para acceder a sus álbumes de fotos.

La metodología es muy similar a la seguida en las anteriores clases. La diferencia a destacar se encuentra en la implementación del objeto `OnItemClickListener`. En este caso, al seleccionar un contacto, obtendremos su nombre e identificador y se pasarán a la clase *FBAlbums.java*.

4.3.9. FBAlbums.java

Esta clase, de manera similar a las anteriores, muestra una lista con los álbumes de los contactos.

El formato de la petición es el siguiente: la ruta *URL* contiene el identificador del contacto deseado seguido de '/albums', y se utiliza el parámetro 'limit' con valor a 0, indicando que deseamos la lista completa de álbumes.

El manejador de la respuesta añade en primer lugar, al principio de la lista, un álbum con ID igual al del contacto. Este álbum se corresponde con el que almacena las fotos en las que el contacto se encuentra etiquetado. Una vez hecho esto, procesaremos la respuesta de la forma explicada anteriormente salvo por la adición de un dato adicional, 'count', que indica el número de fotografías que contiene el álbum. Este dato se mostrará como texto junto al título.

Por último, al realizar un click sobre un álbum ejecutaremos la siguiente actividad, pasando sus atributos.

4.3.10. AlbumsArrayAdapter.java

Esta clase es similar a *FriendsArrayAdapter.java*, salvo por la concatenación en el texto del ítem del número de fotografías contenidas.

4.3.11. FBGallery.java

La actividad actual consiste en una galería de las fotos que componen el álbum seleccionado. Éstas se cargan de forma secuencial, dinámicamente, y se muestran mediante una interfaz en forma de rejilla (vista tipo *GridView*⁽¹²⁾).

En primer lugar, se hace una pequeña comprobación sobre si el álbum posee contenido o no, en cuyo caso se muestra un mensaje al usuario y se regresa a la actividad anterior.

Debido a que *Android* calcula de forma ineficiente el número y el ancho de las columnas de la interfaz tipo *GridView*, definiremos estas características mediante código. En primer lugar, estableceremos un ancho de columna inicial de 200px (tras diversas pruebas se llegó a la conclusión de que era el ancho máximo ideal para mantener una calidad apreciable de las miniaturas). Después, obteniendo el ancho del dispositivo, calcularemos el número de columnas a encajar mediante una división y un redondeo hacia infinito. Por último se recalcula el ancho de columna, que será menor o igual al valor inicial.

A continuación, definimos el método `onItemClick`, que ejecuta una nueva actividad pasando como parámetro la *URL* de la imagen seleccionada, y asociamos el adaptador (*PhotosAdapter.java*) encargado de manejar los elementos de la cuadrícula o rejilla. Tras esto, se realizará la solicitud al servidor mediante la *URL* 'album_ID/photos' y el parámetro 'limit' igual a 0.

En el método encargado de atender la respuesta, capturaremos el hilo actual (recordemos que se trataba de un hilo propio del SDK de Facebook) para poder interrumpirlo en caso de que abortemos la actividad antes de tiempo, de forma que se detenga la carga de las imágenes. La respuesta consiste en un *JSONArray*, donde cada imagen posee un ID ('id'), una *URL* a la imagen original ('source') y un conjunto de *URLs* de la misma imagen en distintos tamaños ('images').

En función de si la imagen se encuentra a alta resolución (superior a 720px) o no, dicho conjunto contiene cuatro o cinco direcciones (la de la imagen a alta resolución). En cualquier caso, la miniatura de mayor calidad (que es la que tomamos) siempre se encuentra en la tercera posición comenzando por la cola.

El siguiente paso consiste en obtener cada una de las imágenes, decodificarlas y actualizar la interfaz, añadiendo además el objeto *Photo* a un *ArrayList* para almacenar los datos de las imágenes descargadas. Estas operaciones se repiten en cada iteración del bucle, de forma que se actualiza la galería de forma secuencial y sin bloquear la UI, ya que se realizan dentro de un hilo. La obtención de las imágenes se realizará mediante la creación de un objeto *URL*, que contendrá la dirección de la información a descargar, y obteniendo a partir de éste un *stream* de entrada en forma de bytes (mediante el método `getContent`). Una vez hecho esto, se decodificará mediante la función `decodeStream` de *BitmapFactory*.

Por último, el método `onConfigurationChanged` se encarga de recalcular las propiedades de la rejilla cuando se produce un cambio de orientación, mientras que el método `onDestroy` interrumpe el hilo encargado de procesar la respuesta del servidor y cargar la galería (lo hace activando un flag en sus

atributos, el cual habrá que comprobar mediante la función `isInterrupted`).

4.3.12. PhotosAdapter.java ⁽¹²⁾

Esta clase se encarga de manejar las imágenes que componen la rejilla. El método `getView` obtendrá el objeto actual a actualizar. En caso de que no se encontrara previamente creado, en lugar de instanciarlo a partir de un fichero XML, inicializaremos mediante código sus propiedades. Éstas son sus dimensiones, el relleno alrededor de la imagen y la forma de escalado.

4.3.13. FBPhoto.java

Esta actividad se encarga de mostrar al usuario la fotografía seleccionada, tal y como se ha visto en las primeras actividades. La diferencia radica en la forma de obtener la imagen. Esta operación se realiza, como hemos visto, mediante un stream de entrada obtenido a partir del objeto `URL`. En este caso, utilizaremos la dirección de la imagen original (a tamaño completo), que se recibe de la actividad anterior.

4.3.14. Settings.java

La clase actual se trata de un tipo especial de actividad, denominada *PreferenceActivity* ⁽¹³⁾. Esta es una implementación especial para las actividades encargadas de almacenar los ajustes y preferencias del usuario. Presentan una lista de opciones cuyos ajustes se guardan automáticamente mediante *SharedPreferences*.

Nuestra interfaz presenta dos botones de conexión al servidor (para el inicio y cerrado de sesión), los cuales se referencian del *XML* mediante la función `findPreference` (mediante su correspondiente ID). Al inicio de la actividad se comprueba la existencia de una sesión previa abierta, de forma que, en caso positivo, se deshabilita el botón de inicio (inhabilitando los clicks y mostrando el texto en gris) y, en caso negativo, se deshabilita el de cerrado de sesión.

El botón de inicio de sesión ejecuta el método `login`, que ya se ha comentado anteriormente. El otro botón ejecuta el método `logout`, que requiere la ejecución de un hilo adicional. Este hilo llamará a la función `logout` del *SDK* de *Facebook*, implementando un *Listener* encargado de habilitar y deshabilitar los botones pertinentes de la interfaz.

4.3.15. Android Manifest.xml ⁽¹⁴⁾

El archivo de la aplicación actual especifica la versión objetivo, los permisos necesarios y la definición

de las distintas actividades que la componen.

Mediante la etiqueta `uses-permission` especificamos los permisos requeridos, en nuestro caso, acceso externo (tarjeta SD) de escritura, para el almacenamiento de las imágenes tomadas o manipuladas, y de acceso a Internet, para la conexión al servidor de *Facebook*.

La etiqueta `uses-sdk` permite definir la versión del S.O. mínima requerida y la versión objetivo.

Por último, en la definición de las actividades podemos asignar el tipo de orientación permitido, un título, o el estilo de la interfaz (si se muestra la barra de título, si se presenta en blanco o en negro...).

4.3.16. Recursos

Estos se organizan por carpetas dentro del directorio `/res`.

La carpeta `'xml'` contiene el archivo con la interfaz de las preferencias de usuario.

`'values'` contiene un archivo con todos los textos de la aplicación. De esta forma, se puede localizar ⁽¹⁵⁾ la aplicación a otros idiomas, creando una jerarquía de lenguajes mediante sus respectivos sufijos (`values-en`, `values-es`, `values-fr`...).

`'menu'` contiene las definiciones de los menús de opciones y contextuales.

`'layout'` incluye el diseño de la interfaz de usuario de las actividades.

La carpeta base `'drawable'` incluye una serie de ficheros *XML* (uno por cada botón del menú) para definir distintas imágenes en función del estado de los botones (pulsado, enfocado...) ⁽¹⁶⁾. El resto de carpetas contienen los recursos gráficos de la aplicación, esto es, las imágenes del icono, los botones del menú o el logo de la aplicación, y se organizan conforme a la densidad de pantalla de los distintos dispositivos.

Las aplicaciones en *Android* no trabajan directamente en términos de resolución de pantalla. En lugar de eso, trabajan con los conceptos de tamaño de pantalla (pulgadas en la diagonal) y densidad de pantalla. Ésta última define la cantidad de píxeles respecto a un área de la pantalla y se mide en puntos por pulgada. Para no perder definición en la calidad de los recursos, se deben proporcionar con distintas dimensiones en función de la densidad de pantalla ya que, de otra forma, se pixelarían al reescalar. Además, se debe evitar el uso de interfaces absolutas y se recomienda utilizar unidades relativas, en lugar de píxeles, en el código de la aplicación.

5. Publicidad en Android ⁽¹⁷⁾

Lo habitual a la hora de publicar una aplicación es implementar dos versiones: una de pago y otra con publicidad. La fuente de ingresos que proporciona la opción por publicidad es baja, si bien la idea consiste en limitar esta versión con características básicas, tentando al usuario de comprar en un futuro la versión más completa.

Lo primero que debemos hacer es registrarnos en *Admob* y rellenar nuestros datos bancarios, por ejemplo indicando una cuenta de *Paypal*. Para recibir la publicidad de *Admob* en nuestra aplicación, debemos darla de alta en la opción correspondiente.



Nombre	Tipo	Estado	Anuncios de autopublicidad	Ingreso	Solicitudes	eCPM	Porcentaje de relleno	RPM
Voice Message			No se completó el inventario.	\$0.16	1,445	\$0.11	99.45%	\$0.11
Trivial time			Inactivo	\$0.12	4,388	\$0.17	15.91%	\$0.03
Record my life			No se completó el inventario.	\$0.02	7,994	\$0.00	98.92%	\$0.00
pichunter thumbs			Inactivo	\$0.00	579	\$0.00	97.06%	\$0.00
trivial8			Inactivo	\$0.00	627	\$0.00	13.40%	\$0.00

Pulsando el botón “Agregar Aplicación”, aparecerá una selección del tipo de dispositivo, ya que esta red da soporte a todo tipo de sistemas. Al seleccionar *Android*, deberemos incluir los datos básicos de nuestra aplicación.



ID de editor: [redacted] | [Obtener código del editor](#)

[Filtros de anuncios](#) [Configuración de la aplicación](#) [Anuncios de autopublicidad](#)

Configuración de la aplicación

Estilo de anuncio:

- Usar los colores establecidos en el código de cliente
- Usar los colores establecidos a continuación:

Vista previa: 

Color del texto:

Color de fondo:

Actualización automática:

- Usar la frecuencia de actualización establecida en el código de cliente
- Sin actualizar
- Frecuencia de actualización: segundos (12 - 120 segundos)

Modo de prueba:

- Usar configuración de modo de prueba establecida en el código de cliente
- Desactivar modo de prueba para todas las solicitudes

Google AdSense:

- Use Google AdSense para mejorar la velocidad de completación. He leído y acepto los [Términos y condiciones](#) de Google AdSense.
- No use Google AdSense.

El código que se encuentra en rojo es el que deberemos incluir en nuestro desarrollo para que *Admob* identifique nuestra aplicación y pueda servir y registrar anuncios.

Una vez que tenemos el código de la aplicación, debemos descargar la librería de *Admob*, que viene comprimida en formato .zip, pulsando el botón que aparecerá al seleccionar la opción “Obtener código del editor” de la pantalla anterior.

Lo primero que debemos hacer es almacenar el fichero .jar en una carpeta y añadir ésta a nuestro proyecto como una librería externa.

Ahora, en el fichero *AndroidManifest.xml* de la aplicación, debemos registrar el código de la aplicación añadiendo una línea en el apartado *<Application>*.

```
<meta-data android:value="axxxxxxxxxxxxxxxxxxa" android:name="ADMOB_PUBLISHER_ID" />
```

Además, añadiremos los permisos necesarios para que pueda conectarse a Internet y, si lo deseamos, los permisos de localización para que los anuncios sean acordes con el público objetivo.

Los anuncios se muestran en las actividades y suelen colocarse arriba o abajo. Para ello hay que incluir el código correspondiente en el *Layout* de la *Activity*.

```
<com.admob.android.ads.AdView  
android:id="@+id/ad"  
android:layout_width="fill_parent"  
android:layout_height="wrap_content"  
>
```

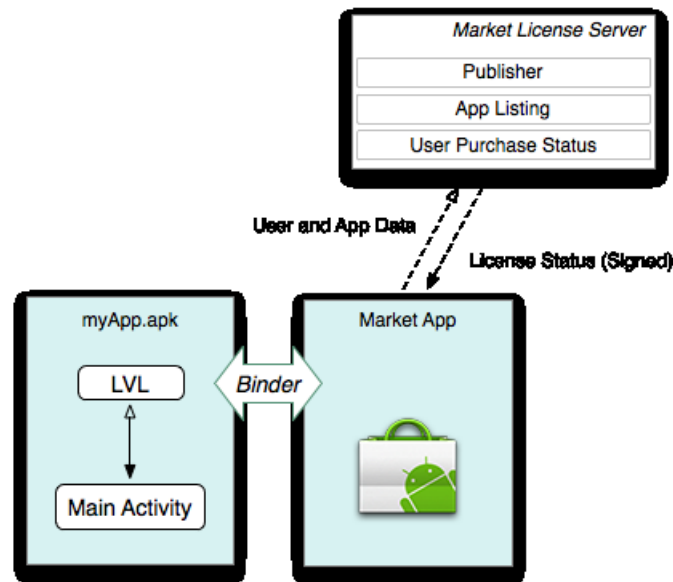
Una vez publicada la aplicación en el *Market*, y con los anuncios funcionando, podemos ver informes por cada aplicación, siendo interesante el porcentaje de relleno, el número de solicitudes recibidas y por supuesto, el número de clics, que es por lo que vamos a recibir nuestros ingresos.

Además, es interesante saber que *Admob* tiene muchas más opciones, como los anuncios de autopublicidad (para hacer publicidad de tus otras aplicaciones).

Los clics de usuarios españoles tienen poco valor para *Admob*, siendo los más valorados los clics de los americanos debido a las redes de publicidad. Por tanto, dependiendo de quién haga clic, recibiremos más o menos ingresos.

6. Protección

Para evitar la piratería, Google implementa un sistema de licencias basado en la nube. Con este sistema, las aplicaciones pueden realizar una petición al servidor de la tienda de aplicaciones (*Android Market*) para obtener el estado de la licencia asociada al usuario actual y actuar en consecuencia, permitiendo o impidiendo su uso.



El servicio de licencias es un medio seguro para controlar el acceso a nuestras aplicaciones. Cuando una aplicación comprueba el estado de una licencia, el servidor firma la respuesta utilizando un par de claves (pública/privada) asociadas a la cuenta del desarrollador. Nuestra aplicación almacena la clave pública en el archivo .apk compilado y la utiliza para verificar la validez de la licencia.

El servidor considera licenciado a un usuario si éste ha sido registrado como comprador de una determinada aplicación o si la aplicación se encuentra disponible de forma gratuita. Para identificar al usuario y determinar el estado de la licencia, el servidor necesita información sobre la aplicación en cuestión y el usuario (la aplicación y el cliente de *Android Market* trabajan conjuntamente para ensamblar la información y pasársela al servidor).

Para facilitar la adición de este servicio a nuestra aplicación, el *SDK* de *Android* proporciona la librería de verificación de licencias (*LVL*), que maneja las comunicaciones entre el servicio de licencias y el cliente de *Android Market*. Con la *LVL*, nuestra aplicación puede determinar el estado de la licencia para el usuario actual llamando, simplemente, a un método verificador e implementando una retrollamada que reciba la respuesta de estado.

Por otro lado, para asegurar la seguridad de nuestra aplicación, es importante ofuscar el código de ésta. Ofuscar debidamente nuestro código hace más difícil a los usuarios malintencionados decompilar el código binario, modificarlo (por ejemplo, eliminando la verificación de la licencia) y recompilarlo nuevamente.

Hay disponibles varios programas de ofuscación de código para la plataforma *Android*, entre ellos *ProGuard* ⁽¹⁸⁾, que además ofrece características para la optimización del código. El uso de *ProGuard* u otros programas similares de ofuscación es altamente recomendado para aquellas aplicaciones que utilicen el servicio de licencias.

La herramienta *ProGuard* reduce, optimiza y ofusca el código de la aplicación a base de eliminar código inútil o de renombrar clases, campos y métodos por nombres confusos. Esto da como resultado un fichero .apk más pequeño que es más difícil de descifrar mediante ingeniería inversa. Debido a esto, es importante hacer uso de ella cuando la aplicación utilice características sensibles a la seguridad, como es el caso del licenciamiento de aplicaciones.

ProGuard se encuentra integrado en el sistema de *Android*, por lo que no es necesario invocarlo manualmente. *ProGuard* se ejecuta únicamente cuando se desarrolla la aplicación en modo *release* (de publicación), de manera que no hay que tratar con código ofuscado cuando se desarrolla en modo *debug* (de depuración).

7. Publicación en el Market

De forma previa al proceso de publicación, es necesario firmar nuestra aplicación. El sistema de *Android* requiere que todas las aplicaciones instaladas estén firmadas digitalmente con un certificado cuya clave privada es mantenida por el desarrollador de la aplicación. El sistema utiliza dicho certificado para identificar al autor de una aplicación y establecer relaciones de confianza entre aplicaciones. Este certificado no necesita ser firmado por una autoridad de certificación: está permitido para las aplicaciones de *Android* utilizar certificados firmados por uno mismo.

Android no instalará o ejecutará aplicaciones que no estén firmadas debidamente. Esto se aplica a cualquier lugar en el que sea ejecutado el sistema operativo, ya sea en un dispositivo real o sobre el emulador proporcionado. Por esta razón, se debe firmar la aplicación antes de publicarla o depurarla.

Las herramientas del *SDK* de *Android* proporcionan asistencia a la hora de firmar las aplicaciones en modo de depuración. Cuando la aplicación se encuentra lista para su publicación, el desarrollador debe compilarla en modo *release* y firmar el archivo .apk con su clave privada.

Con *Eclipse* y el plugin *ADT*, se puede utilizar el asistente de exportación (*Export Wizard*) para exportar un fichero .apk firmado. El asistente interactúa con las herramientas de firmado, lo que permite firmar el paquete usando una interfaz gráfica en lugar de realizar manualmente los procedimientos de compilación, firmado y alineación.

Para crear un ejecutable .apk firmado y alineado en *Eclipse*, hay que seleccionar el proyecto y pulsar sobre la opción 'Export' del menú 'File'. A continuación, desplegaremos la carpeta *Android* y seleccionaremos la opción 'Export Android Application'. Se ejecutará el asistente de exportación, que nos guiará a través del proceso de firmado de la aplicación, incluyendo los pasos necesarios para seleccionar una clave privada con la que firmar el paquete. Finalmente, la aplicación será compilada, firmada, alineada y preparada para su distribución.

Una vez hecho esto, procederemos con el proceso de publicación. Primero, es necesario registrarse con el servicio utilizando una cuenta de *Google* y aceptando los términos de uso (para lo que habrá que abonar un pago de 25 dólares). Una vez registrado, podemos subir nuestra aplicación al servicio. Hecho esto, los usuarios pueden ver nuestra aplicación en la tienda de aplicaciones *Android Market*, descargarla y calificarla.

Para subir nuestra aplicación, nos dirigiremos a la dirección web de la consola de desarrolladores (<http://market.android.com/publish/Home>). Mediante el botón 'Upload application' iniciaremos el

proceso de subida.

La opción 'Screenshots' nos permitirá añadir un par de imágenes para mostrar al usuario cuando vea la descripción de nuestra aplicación. Éstas deberán tener un tamaño de 320x480 o 480x854 píxeles y deberán estar codificadas en formato PNG de 24 bits o JPG.

A continuación, asignaremos un título y una descripción a nuestra aplicación, así como el tipo de aplicación, su categoría y el precio de venta. Para añadir una aplicación de pago deberemos asociar una cuenta de algún servicio de pagos online, como Paypal o Google Checkout. Todos estos datos podremos incorporarlos en distintos idiomas, en función de la localización del usuario.

Por último, podremos indicar nuestra información de contacto y marcar una serie de opciones de publicación. La primera de estas consiste en la protección de copia. Este mecanismo obligará a la aplicación a instalarse en la memoria *ROM* del dispositivo e impedirá que se pueda realizar una copia de seguridad de ésta en la tarjeta SD. Como inconveniente, se incrementará el volumen de memoria necesario que requerirá la aplicación para instalarse en el dispositivo.

La siguiente opción, 'Locations', nos permitirá elegir los países en los que deseamos que se publique nuestra aplicación. Esto nos permitirá subir diversas versiones de nuestra aplicación destinándolas a ubicaciones distintas.

Una vez subida nuestra aplicación, se nos mostrarán sus estadísticas en la consola de desarrolladores. Podremos visualizar los comentarios realizados por los usuarios de nuestra aplicación, la puntuación asignada (de 0 a 5 estrellas), el número de descargas realizadas, el número de descargas efectivas (esto es, aquellas cuyos usuarios mantienen la instalación de la aplicación) y los errores que hayan detectado y reportado nuestros usuarios.

Conclusión

Limitaciones y problemas encontrados

A lo largo del desarrollo de la aplicación encontré una serie de trabas que tuvieron que ser solventadas.

Una de las más destacables es la limitación de memoria destinada a las aplicaciones. Éstas disponen únicamente de un total de 16MB para cargar sus recursos. Esto obligaba a realizar una gestión de las imágenes decodificadas una vez utilizadas, ya que se encuentran a alta resolución y ocupan en memoria un tamaño del orden de varios MB. Una sobrecarga en la RAM provocaba un *OutOfMemoryError*, obligando a la actividad a abortar su ejecución. Por ello se optó por hacer uso del método *recycle* una vez ya no fueran necesarias, liberando así los recursos de la memoria. Este método se ejecuta cada vez que se sustituye una imagen por otra, cuando se cambia de actividad o cuando se destruye la actividad actual. Además, aprovechando la limitación de tamaño por parte del servidor de *Facebook* se optó por realizar un reescalado de las imágenes en su decodificación.

Otra de las trabas fue la limitación a la hora de pasar ciertos tipos de archivo entre actividades. En nuestro caso, los objetos *Bitmap* o *FileDescriptor* (que contiene la información sobre el fichero abierto) no podían ser utilizados como argumento, por lo que finalmente se obtuvo por enviar la ruta al fichero en formato *URI*.

Como se ha comentado anteriormente, a la hora de aplicar efectos sobre las imágenes usando métodos rudimentarios añadía una gran carga sobre el sistema, puesto que éstas poseen una alta resolución. Por ello, hubo que documentarse y realizar el procesado de las imágenes mediante el uso de matrices. Otro de los problemas encontrados fue el de realizar la conversión de coordenadas sobre la pantalla (al realizar una pulsación sobre ésta) a coordenadas reales sobre la imagen (puesto que en la mayoría de casos se encuentra escalada).

Por último, otro de los grandes obstáculos fue el de implementar la galería fotográfica (por una parte, solventando las deficiencias de *Android* a la hora de calcular las propiedades de la cuadrícula y, por otra, evitando el bloqueo de la interfaz de usuario realizando una carga dinámica mediante hilos).

Como limitaciones encontramos que, para que el algoritmo de reconocimiento de caras funcione correctamente, éstas deben estar más o menos enfrentadas a la cámara y con los ojos alineados horizontalmente. También, por la forma en que *Facebook* sirve la información, el servidor entrega el listado de amigos ordenado por ID y sirve las imágenes junto con datos adicionales innecesarios.

Valoración personal

Con el desarrollo de este proyecto pienso que se han cumplido los intereses iniciales de profundizar en la plataforma *Android*, adentrarse en el ámbito del procesamiento de imágenes y la visión artificial y conocer las posibilidades que otorga una red social.

En primer lugar, se han cubierto todos los conocimientos básicos del sistema operativo, tanto en la implementación lógica como en el diseño de la interfaz. Además, se han explorado temas más avanzados como la gestión de memoria, la gestión del ciclo de vida de una actividad, el procesado de imágenes o el uso de hilos.

En cuanto al tratamiento de imágenes, se decidió incorporar el uso de efectos (no contemplado en un principio) por un tema de interés personal. Se ha trabajado con éstas de diversas formas: calculando las dimensiones apropiadas y escalándolas al tamaño deseado, recortándolas, modificando sus píxeles mediante el uso de matrices y desarrollando un algoritmo matemático que permitiera transformar las coordenadas en la pantalla a coordenadas reales sobre la imagen. Además se ha trabajado con las librerías de reconocimiento de caras incorporadas en *Android* y realizado una serie de pruebas para conocer su funcionamiento y sus limitaciones.

En tercer y último lugar, he aprendido sobre el uso de la *API de Facebook*, aplicable no sólo a *Android* sino a cualquier plataforma. Se han revisado las tareas de inicio y cerrado de sesión, así como tareas de interacción con la plataforma, ya sea solicitando información o subiendo nueva al servidor.

En definitiva, este proyecto ha contribuido a conocer diversas tecnologías con mucha presencia en la actualidad y a saber integrarlas y unificarlas para desarrollar una utilidad práctica.

Líneas futuras

Las mejoras a aplicar sobre la aplicación básicamente cubren dos frentes. Por una parte, hemos comentado anteriormente las deficiencias de *Facebook* a la hora de servir la información. La lista de contactos se devuelve ordenada por el número de ID que, a su vez, depende de la fecha de registro, y la información que se solicita al servidor a menudo viene acompañada de atributos innecesarios que ralentizan la respuesta. Esto es debido a que, por defecto, no se pueden especificar los datos que se requieren de un objeto dado. Sin embargo, existe una *API* avanzada denominada *Facebook Query Language*, basada en el lenguaje *SQL*, que permite utilizar características avanzadas tales como realizar múltiples peticiones en una sola llamada o solicitar datos específicos. Esta *API* también permite obtener la foto de perfil para cada uno de los contactos, de forma que se puede mejorar la interfaz a la hora de mostrar el *ListView*.

Por otro lado, la aplicación ofrece posibilidades de mejora en la línea del reconocimiento facial. Una de ellas consistiría en utilizar un algoritmo más robusto mediante el uso de librerías externas o mediante el desarrollo de un algoritmo propio, basándose en otros patrones como la detección de la nariz o de la boca. La segunda de ellas consistiría en extender las posibilidades de la aplicación mediante la inclusión de una base de datos de contactos. La idea consistiría en utilizar un algoritmo capaz de realizar comparaciones de los rostros detectados contra la base de datos. Esta base de datos se iría incrementando a medida que la gente se fuera etiquetando, de forma que el algoritmo sería capaz, por sí mismo, de identificar a las personas presentes y subir la imagen al servidor automáticamente. En esta línea, recientemente se han portado las librerías de *OpenCV* a la plataforma *Android*. Para hacer uso de estas librerías, basadas en el lenguaje C, es necesario instalar la herramienta *NDK* para *Android*, que permite programar porciones de la aplicación usando código nativo en C o C++. *OpenCV* dispone entre sus ejemplos de un par de aplicaciones para la detección de caras y el reconocimiento facial, *FaceDetection* ⁽¹⁹⁾ y *FaceRecognition* ⁽²⁰⁾. Para adelantar parte de la tarea, la aplicación actual se encarga de recortar las imágenes una vez son etiquetadas y almacena los recortes en un directorio de la tarjeta SD con el nombre e ID del contacto asociado.

Anexos

Tutoriales

Anexo I. Instalación y configuración del IDE ⁽²¹⁾⁽²²⁾

Las herramientas de desarrollo de *Android* permiten una fácil integración con *Eclipse*. Para empezar a desarrollar, necesitaremos tener instalado el JDK (*Java Development Kit*), que permite crear y ejecutar aplicaciones en Java mediante la máquina virtual JRE, y luego instalar Eclipse Classic y el *SDK* de *Android*, junto al plugin ADT (*Android Development Tools*) para Eclipse.

Descargaremos el *SDK Starter Package* de <http://developer.android.com/sdk/> y descomprimiremos su contenido en el directorio de nuestra preferencia. Habrá que asegurarse de que las carpetas vacías 'platforms' y 'add-ons' se creen en la descompresión puesto que, en caso contrario, no se instalarán las herramientas. A continuación, ejecutaremos el archivo SDK Setup.exe (*Android SDK Manager*) para instalar las diversas plataformas (versiones del S.O.) sobre las que desarrollar nuestras aplicaciones. En caso de problemas de conexión con el repositorio, utilizaremos el protocolo *HTTP* en lugar de *HTTPS*, marcando la opción correspondiente en el apartado *Settings* del *Manager*.

Para instalar el plugin ADT haremos uso de la herramienta de actualización remota de Eclipse. Abriremos la opción 'Install new software...' del menú 'Help' y pulsaremos sobre el botón 'Add'. Entonces introduciremos la *URL* del repositorio (<https://dl-ssl.google.com/android/eclipse/>) y aceptaremos. Por último, marcaremos todas las descargas y reiniciaremos el IDE.

Para la emulación de aplicaciones a través de una máquina virtual, seleccionaremos la opción 'Android SDK and AVD Manager' del menú 'Window'. Aquí crearemos y configuraremos los dispositivos sobre los que testear nuestras aplicaciones. Podremos asignar un nombre a la máquina, la versión del S.O. sobre la que se ejecuta, la cantidad de memoria que dispone o la resolución que posee.

También es posible ejecutar las aplicaciones sobre un dispositivo real, conectándolo mediante *USB*. Para ello será necesario tener instalados los drivers proporcionados por el fabricante y activar la opción de *debugging* en el teléfono. Esto se hace accediendo a los ajustes, al apartado de aplicaciones y activando la opción 'Depuración USB' del apartado 'Desarrollo'.

Anexo II. Instalación y configuración del SDK de Facebook ⁽²³⁾⁽²⁴⁾

Para comenzar el desarrollo con el *SDK* de *Facebook* es necesario instalar Git (el cliente de control de versiones utilizado por el *SDK*) y después clonar la última versión del kit de desarrollo desde Github. Para ello introduciremos la siguiente orden en la consola de comandos de Windows:

```
git clone git://github.com/facebook/facebook-android-sdk.git
```

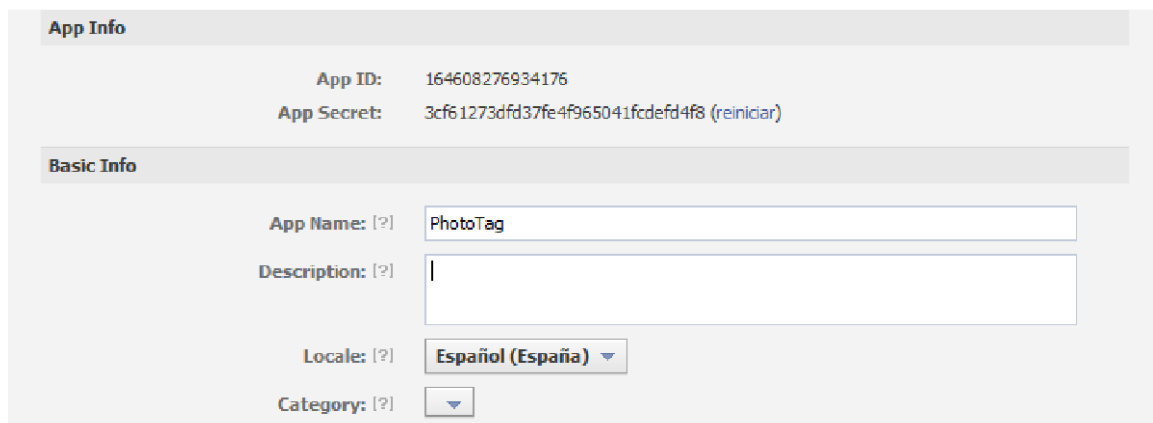
A continuación, importaremos el *SDK* como un proyecto existente de Eclipse y añadiremos una referencia hacia él desde nuestro propio proyecto. Esto último se realiza abriendo las propiedades de nuestro proyecto y seleccionando el botón 'Add' del apartado 'Android'. Para garantizar el correcto funcionamiento de la aplicación, deberemos modificar el Android Manifest añadiendo el permiso de conexión a Internet.

Para registrar la aplicación en Facebook es necesario exportar la firma de ésta mediante la herramienta 'keytool'. Habrá que asegurarse de que hemos añadido el directorio de instalación de Java al path del sistema, instalar *OpenSSL* y agregarlo también a la variable de entorno. Ejecutaremos la instrucción:

```
keytool -exportcert -alias androiddebugkey -keystore ~/.android/debug.keystore | openssl sha1 -binary | openssl base64
```

donde ~ equivale a la ruta en la que se encuentra el *SDK* de *Android*. El string generado lo guardaremos para más adelante.

A continuación, daremos de alta la aplicación en la web para desarrolladores de *Facebook* (<https://developers.facebook.com/>).



The screenshot shows the Facebook Developer console interface. It is divided into two sections: 'App Info' and 'Basic Info'.
Under 'App Info', the 'App ID' is 164608276934176 and the 'App Secret' is 3cf61273dfd37fe4f965041fcdefd4f8, with a '(reiniciar)' link next to it.
Under 'Basic Info', there are four fields:
- 'App Name: [?]' with the value 'PhotoTag' entered in a text box.
- 'Description: [?]' with an empty text box.
- 'Locale: [?]' with a dropdown menu set to 'Español (España)'.
- 'Category: [?]' with a dropdown menu showing a downward arrow.

Una vez hecho, pegaremos la clave generada en la sección 'Mobile'. En esa misma sección figura el ID de la aplicación, que será el que utilizemos en nuestro código para identificar nuestra aplicación ante el servidor.

App Info	
App ID:	164608276934176
App Secret:	3cf61273dfd37fe4f965041fcdefd4f8 (reiniciar)

iOS de Apple	
iOS Bundle ID: [?]	<input type="text"/>
Required: iOS Bundle ID for iOS native app	
ID de iTunes App Store:	<input type="text"/>

Android	
Clave hash:	<input "="" type="text" value="BkP135D+IjHs4aGksTDI9PLhrTg="/>

Anexo III. Importación de un proyecto en Eclipse

Eclipse, al iniciarse por primera vez, solicita al usuario indicar una ruta en la que guardar sus proyectos. En esta ruta se creará el directorio 'workspace', que contendrá los subdirectorios con los distintos desarrollos. El archivo del proyecto actual, 'PhotoTag.zip', deberá ser descomprimido en dicho directorio.

A continuación, nos dirigiremos al menú 'File' de *Eclipse* y seleccionaremos la opción 'Import...'. Aparecerá una ventana emergente en la que seleccionaremos la opción 'Existing Projects into Workspace'. Después, indicaremos la ruta al directorio mencionado, elegiremos el proyecto a importar y aceptaremos.

Anexo IV. Uso del emulador sin necesidad de Eclipse

En primer lugar, deberemos tener previamente creada una máquina virtual. Ésta se encuentra en la carpeta 'C:/Users/~/.android', junto a la memoria SD virtual y otros ajustes de configuración.

Para ejecutar el emulador, iremos desde la consola de comandos al directorio <C:/Users/~/.android-sdk-windows/tools> y ejecutaremos:

```
emulator -avd <maquina_virtual>
```

Bibliografía

- (1) <http://www.idg.es/pcworldtech/Crece-la-venta-de-smartphones-un-64-por-ciento-con/doc99050-actualidad.htm>
- (2) <http://syncstaff.es/en-la-mente-del-programador-iphone-vs-android>
- (3) <http://www.engadget.com/2010/08/04/npd-android-is-now-top-selling-os-in-american-smartphones/>
- (4) <http://www.canalys.com/pr/2011/r2011013.html>
- (5) <http://developer.android.com/guide/topics/fundamentals/activities.html>
- (6) <https://developers.facebook.com/docs/reference/api/>
- (7) <http://developer.android.com/guide/topics/data/data-storage.html>
- (8) <http://www.sgoliver.net/blog/?p=1731>
- (9) <http://developer.android.com/guide/topics/ui/menus.html>
- (10) <http://developer.android.com/resources/tutorials/views/hello-listview.html>
- (11) <http://developer.android.com/guide/topics/ui/dialogs.html>
- (12) <http://developer.android.com/resources/tutorials/views/hello-gridview.html>
- (13) <http://developer.android.com/reference/android/preference/PreferenceActivity.html>
- (14) <http://developer.android.com/guide/topics/manifest/manifest-intro.html>
- (15) <http://developer.android.com/resources/tutorials/localization/index.html>
- (16) <http://developer.android.com/guide/topics/resources/drawable-resource.html#StateList>
- (17) <http://www.androidizados.com/desarrollo/2011/04/26/admob-para-desarrolladores-ingresos-mediante-publicidad/>
- (18) <http://developer.android.com/guide/developing/tools/proguard.html>
- (19) <http://opencv.willowgarage.com/wiki/FaceDetection>
- (20) <http://opencv.willowgarage.com/wiki/FaceRecognition>
- (21) <http://www.londatiga.net/it/how-to-setup-android-application-development-on-eclipse/>
- (22) <http://www.androidsis.com/programacion-android-i-entorno/>

(23) <https://developers.facebook.com/docs/guides/mobile/#android>

(24) <http://integratingstuff.com/2010/10/14/integrating-facebook-into-an-android-application/>

Otras referencias:

Gramlich, Nicolas: *andbook! Android Programming*. 2009. 62 p. <http://andbook.anddev.org/>

Ortiz, C. Enrique: *Introduction to Facebook APIs*. 2010. 26 p.

<http://www.ibm.com/developerworks/library/x-androidfacebookapi/index.html>

Luypaert, Steffen: *Adding Facebook integration to an Android Application*. 2010.

<http://integratingstuff.com/2010/10/14/integrating-facebook-into-an-android-application/>