Look!: Framework para Aplicaciones de Realidad Aumentada en Android

Proyecto de Sistemas Informáticos

2010/2011

Facultad de Informática

Universidad Complutense de Madrid

Sergio Bellón Alcarazo

Jorge Creixell Rojo

Ángel Serrano Laguna

Dirigido por Jorge J. Gómez Sanz

Nosotros, Sergio Bellón Alcarazo, Jorge Creixell Rojo y Ángel Serrano Laguna, creadores del presente documento y del proyecto de Sistemas Informáticos *Look!: Framework para Aplicaciones de Realidad Aumentada en Android*, autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Sergio Bellón Alcarazo
DNI: 71224978-G

Jorge Creixell Rojo
DNI: 54053166-S

Ángel Serrano Laguna
DNI: 06276034-R

Resumen

Se presenta el framework de aplicaciones de realidad aumentada *Look!*, desarrollado para el sistema operativo móvil Android. *Look!* pretende aunar en un sólo framework funcionalidades básicas requeridas en el desarrollo de aplicaciones de realidad aumentada.

El framework se valida con cuatro desarrollos prototípicos: una galería de imágenes en 3D, un mundo virtual, un juego interactivo en tres dimensiones y una aplicación para la creación de redes sociales con soporte para geolocalización.

Adicionalmente, se han escrito tutoriales que asistan al uso de este framework, y se ha documentado suficientemente su funcionamiento por si otros equipos quisieran continuar con este desarrollo.

Palabras clave realidad aumentada, android, localización en interiores, framework, arquitectura rest, wifi, navegación inercial

Abstract

This document introduces the application framework for augmented reality *Look!*, developed for the mobile operating system Android. *Look!* provides basic functionalities required in the development of augmented reality applications.

The framework is validated with four prototypic developments: a 3D image gallery, a virtual world, a 3D game, and a location-based social network.

Additionally, tutorials assisting the development of applications through this framework have been written, and it is sufficiently documented in case this work was continued.

Keywords augmented reality, android, indoor location, framework, rest architecture, wifi, inertial navigation

Índice general

1. Introducción **10**
- 1.1. Realidad Aumentada . . . 10
- 1.2. Objetivos . . . 11
- 1.3. Estructura del Documento . . . 12

2. Requisitos **13**
- 2.1. Requisitos Funcionales . . . 13
  - 2.1.1. Capas de Gráficos en 2D y 3D . . . 13
  - 2.1.2. Construcción de Entidades representables en Realidad Aumentada . . . 14
  - 2.1.3. Interacción con los Objetos Virtuales . . . 14
  - 2.1.4. Localización en Interiores de Edificios . . . 14
  - 2.1.5. Servicio de Persistencia de Datos . . . 14
- 2.2. Requisitos No Funcionales . . . 15
- 2.3. Restricciones . . . 15

3. Estado del Arte **16**
- 3.1. Frameworks para realidad aumentada . . . 16
  - 3.1.1. Layar Reality Browser . . . 17
  - 3.1.2. mixare . . . 18
  - 3.1.3. AndAR . . . 18
  - 3.1.4. Conclusiones . . . 19
- 3.2. Sistemas operativos . . . 19
  - 3.2.1. Android . . . 20
  - 3.2.2. iOS . . . 21
  - 3.2.3. Tecnología de escritorio más webcam . . . 21
  - 3.2.4. Tecnología escogida . . . 22
- 3.3. OpenGL ES . . . 22
  - 3.3.1. OpenGL ES 1.1 . . . 22
  - 3.3.2. OpenGL ES 2.0 . . . 23
  - 3.3.3. OpenGL ES 1.1 vs 2.0 . . . 23
- 3.4. Análisis de Servicios Web . . . 23
  - 3.4.1. Arquitecturas más comunes para Servicios Web . . . 24
  - 3.4.2. Compatibilidad con Android . . . 24
  - 3.4.3. Valoración . . . 26
- 3.5. Análisis de Proyectos de Localización . . . 26
  - 3.5.1. PlaceLab . . . 26
  - 3.5.2. Ekahau . . . 29

3.5.3. Proyecto Indoor Navigation System for Handheld Devices 30
3.5.4. Conclusiones . . . 32
3.6. Técnicas de Localización en Interiores . . . 32
3.6.1. Localización mediante etiquetas RFID . . . 33
3.6.2. Localización mediante marcas visibles . . . 33
3.6.3. Localización mediante el Cálculo del Punto Central a partir de nodos predefinidos . . . 34
3.6.4. Localización mediante Triangulación de la Señal . . . 35
3.6.5. Localización mediante Sistemas Inerciales . . . 36
3.6.6. Localización mediante Detección de Movimiento . . . 37
3.6.7. Localización mediante Mapas de Radio WiFi . . . 38
3.6.8. Localización mediante Mapas de Radio obtenidos Automáticamente . . . 39
3.7. Conclusiones . . . 40

4. Arquitectura Global **41**
4.1. Módulo de Localización . . . 41
4.2. Módulo de Datos . . . 42
4.3. Módulo de Realidad Aumentada . . . 42
4.4. Uniendo los módulos para la creación de aplicaciones . . . 43

5. Módulo de localización **45**
5.1. Orientación del móvil en Android . . . 45
5.1.1. *Pitch*, *azimuth* y *roll* . . . 45
5.1.2. Obteniendo la orientación: *DeviceOrientation* . . . 46
5.2. Subsistema de Navegación Inercial . . . 46
5.2.1. Diseño . . . 47
5.2.2. Arquitectura . . . 47
5.2.3. Implementación . . . 51
5.3. Subsistema de Localización por WiFi . . . 51
5.3.1. Requisitos . . . 52
5.3.2. Diseño . . . 53
5.3.3. Arquitectura . . . 57
5.3.4. Implementación . . . 59
5.3.5. Pruebas . . . 60
5.4. Integración de los subsistemas de Localización . . . 63

6. Módulo de datos **65**
6.1. *EntityData* como unidad básica de datos . . . 65
6.2. Conectando los datos con el módulo de realidad aumentada: Mundo y Entidades del mundo . . . 67
6.2.1. *World* . . . 67
6.2.2. *WorldEntity* . . . 67
6.2.3. Actualizando el mundo . . . 69
6.3. Obteniendo y almacenando datos: *DataHandler* . . . 69
6.3.1. Obtención de datos sin persistencia . . . 70
6.3.2. Obtención de datos con persistencia local . . . 71
6.3.3. Obtención de datos con persistencia remota . . . 74
6.4. Otras fuentes de datos: Archivos binarios . . . 84
6.4.1. Administrador de archivos . . . 85

6.4.2. Optimizaciones del administrador de ficheros 86

7. Módulo de Realidad Aumentada 87
7.1. Visión general del sistema de capas: LookAR 87
7.2. Capa 3D 88
7.2.1. Consideraciones con OpenGL ES 1.1 90
7.2.2. Proyección de elementos de coordenadas reales a coordenadas OpenGL 90
7.2.3. Situación de la cámara 90
7.2.4. Características OpenGL soportadas por *Look!* 91
7.2.5. Dibujado de entidades: *Entity3D* 91
7.2.6. ObjMesh3D: cargando mallas desde archivos .obj Wavefront 92
7.2.7. Creación de texturas: *TextureFactory* 93
7.2.8. Funcionalidades geométricas y colisiones 94
7.3. Integrando cámara y 3D 95
7.4. Capa 2D 97
7.4.1. Canvas vs Proyección Ortogonal en OpenGL 97
7.4.2. SurfaceView vs. View personalizada 97
7.4.3. Dibujado 2D 98
7.5. Animaciones 99
7.6. Interacción del usuario 100
7.6.1. Interacción por teclado 100
7.6.2. Interacción táctil 100
7.6.3. Interacciones de cámara 101
7.6.4. HUD en 2D 103
7.7. Capa HUD 103
7.8. Utilidades: LookARUtil 104

8. Construyendo aplicaciones con *Look!* 105
8.1. Planteando la aplicación 105
8.2. Codificando una aplicación básica 106
8.2.1. Creando la Activity principal 106
8.2.2. Definiendo los elementos de la aplicación: *EntityData* . . 107
8.2.3. Definiendo factorías de elementos 108
8.2.4. Añadiendo interacciones 111
8.2.5. Añadiendo un HUD a la aplicación 113
8.2.6. Configurando una base de datos 115
8.3. Preparando el Sistema de Localización 116
8.3.1. Definición de Nodos y Puntos de Acceso 117
8.3.2. Captura de Datos 118
8.3.3. Probando la Localización por Wifi 120
8.3.4. Ajuste de Parámetros del Sistema de Navegación Inercial 120
8.4. Integrando localización 120

9. Experimentación: Aplicaciones de ejemplo desarrolladas con *Look!* 122
9.1. Mundo 3D 122
9.1.1. Módulos incorporados 123
9.1.2. Definición de los EntityData 123
9.1.3. Factoría de WorldEntity 124

9.1.4. Interacciones . . . 124
9.1.5. Localización . . . 125
9.2. Galería Look! . . . 126
9.2.1. Módulos incorporados . . . 126
9.2.2. Definición de los EntityData . . . 126
9.2.3. Factoría de WorldEntity . . . 128
9.2.4. Interacciones . . . 129
9.3. Invaders 360 . . . 129
9.3.1. Módulos incorporados . . . 130
9.3.2. Definición de los EntityData . . . 130
9.3.3. Factoría de WorldEntity . . . 131
9.3.4. Interacciones . . . 132
9.4. Look! Social . . . 132
9.4.1. Descripción . . . 133
9.4.2. Módulos incorporados . . . 135
9.4.3. Definición de los EntityData . . . 135
9.4.4. Factoría de WorldEntity . . . 136
9.4.5. Interacciones . . . 136
9.4.6. Localización . . . 137
9.4.7. Datos remotos . . . 137

10. Apuntes finales **139**
10.1. Problemas encontrados durante el proyecto . . . 139
10.2. Trabajo futuro . . . 140
10.3. Conclusiones . . . 141

A. Sistema de Navegación Inercial mediante el Cálculo del Desplazamiento Relativo **145**
A.1. Introducción . . . 145
A.2. Descripción . . . 145
A.2.1. Fórmulas . . . 146
A.2.2. Eliminación del efecto de la gravedad . . . 146
A.2.3. Transformación de Coordenadas . . . 147
A.3. Implementación . . . 148
A.4. Resultados . . . 148
A.5. Precisión de acelerómetros . . . 149

B. Acceso a la API CoreWLAN de Mac OS X desde Java mediante JNI **150**
B.1. Introducción . . . 150
B.2. Pasos Necesarios . . . 151
B.2.1. Creación del fichero de cabecera . . . 151
B.2.2. Mezclando código C y Objective-C . . . 152
B.2.3. Accediendo a CoreWLAN . . . 153
B.2.4. Implementación de la Librería Nativa . . . 155
B.2.5. Resultado y Conclusión . . . 158

C. Exportando mallas .obj con Blender **159**
C.1. Introducción . . . 159
C.2. Exportando mallas . . . 159

Índice de figuras

1.1. Ejemplo de Realidad Aumentada 10

3.1. La aplicación de realidad aumentada *Layar* 17
3.2. Estrucutra de una aplicacion con *mixare* 18
3.3. Captura de *mixare* 19
3.4. Esquema del funcionamiento de *AndAR* 20
3.5. Mercado Android entre 2009 y 2010 21
3.6. PlaceLab Stumbler 27
3.7. PlaceLab Tracker 28
3.8. Mapa de Radio resultado del procesado 31
3.9. Ejecución de Indoor Navigation System for Handheld Devices . . 32
3.10. Fluctuación de la señal en redes WiFi 35

4.1. Capas del Desarrollo de Aplicaciones mediante Look 41
4.2. Diagrama de Componentes de Localización 42
4.3. Diagrama de Componentes del módulo de datos 43
4.4. Diagrama de Componentes de la Interfaz de Realidad Aumentada 44
4.5. Diagrama de componentes general para aplicaciones construidas con *Look!* 44

5.1. Definición de *pitch*, *azimuth* y *roll* 46
5.2. Clase *DeviceOrientation* 46
5.3. Diagrama de Flujo: Sistema de Navegacion Inercial 48
5.4. Diagrama de Clases INS 49
5.5. Diagrama de secuencia INS 50
5.6. Aplicación de prueba para el subsistema de Navegación Inercial . 52
5.7. Esquema de la localizacion por WiFi 54
5.8. Ejemplo de seleccion de Nodos 54
5.9. Red Neuronal Perceptron Multicapa 57
5.10. Diagrama de clases de la localizacion por WiFi 58
5.11. Diagrama de Secuencia Servicio Localizacion WiFi 59
5.12. Artefactos Generados por el sistema de Localizacion WiFi 61
5.13. Captura del programa de localizacion WiFi en funcionamiento . 62
5.14. Nodos definidos piso 62
5.15. Integración de los subsistemas de Localización 64

6.1. La clase EntityData 66
6.2. Diagrama de Clases del Acceso a Datos 66

6.3. Diagrama de clases de LookData, World, WorldEntity y EntityData, con sus relaciones. 68
6.4. Actualización del mundo 69
6.5. Relación de *DataHandler* con *EntityData* 70
6.6. Implementación de *BasicDataHandler* 70
6.7. Implementación de *DBDataHandler* 71
6.8. Entidad de tipo *user* usada en la aplicación Look! Social 73
6.9. Diagrama de secuencia para el acceso de datos 75
6.10. Diagrama de secuencia para el almacenamiento de datos 76
6.11. Implementación de *RemoteDataHandler* 77
6.12. Diagrama de secuencia de updateDB() 78
6.13. Arquitectura del módulo de datos en el servidor 82

7.1. Situación de capas de representación en *Look!* 88
7.2. Diagrama de clases de los módulos de Realidad Aumentada. . . 88
7.3. Relación de *GLSurfaceView* con *Renderer3D* 89
7.4. La clase *Entity3D* y sus componentes 92
7.5. Diagrama de secuencia del dibujado 3d 93
7.6. Clase TextureFactory 94
7.7. Contenido del paquete es.ucm.look.ar.math.geom 95
7.8. Contenido del paquete es.ucm.look.ar.math.collision 96
7.9. Diagrama de secuencia del dibujado en dos dimensiones 99
7.10. Apariencia de la aplicación en modo normal. Existen cuatro entidades (cuatro imágenes) en cada una de las esquinas 102
7.11. Apariencia del buffer táctil de la aplicación anterior. Pueden verse las cuatro áreas táctiles, cada una de un color distinto (distintos tonos de azul) 102
7.12. Apariencia del buffer táctil de la aplicación anterior. Pueden verse las cuatro áreas táctiles, cada una de un color distinto (distintos tonos de azul) 103

8.1. Captura de la aplicación tras añadir un elemento 108
8.2. Captura de la aplicación con la nueva factoría 110
8.3. La interfaz *TouchListener* 111
8.4. Captura de la aplicación tras pulsar uno de los elementos 112
8.5. La interfaz *CameraListener* 112
8.6. Captura de la aplicación mientras se enfoca directamente a uno de los elementos. Puede apreciarse como su representacion 3D ha cambiado de un cubo a un plano. 114
8.7. Captura de la aplicación con HUD. Podemos comprobar arriba a la izquierda como se añadió el botón. 115
8.8. Ejemplo de Definición de Nodos 117
8.9. CCNWifi: Detección automática de Puntos de Acceso 118
8.10. CCNWifi: Captura de Datos 119
8.11. CCNWifi: Probando la localización 119

9.1. Captura de Mundo 3D 123
9.2. Captura de Galería Look! 127
9.3. Captura de Galería Look!, con una imagen seleccionada 127
9.4. Captura de Invaders 360 130

9.5. Registro y *login* de usuario en *Look!Social* 133
9.6. Pantalla principal de *Look!Social* con menús 134
9.7. *Look!Social*: Realidad Aumentada 134

A.1. Accediendo al mando de la Wii 148

C.1. Creando una malla con Blender 160
C.2. Creando una malla con Blender 160
C.3. Exportando la malla a una archivo .obj 160

Capítulo 1

Introducción

En la actualidad, las tecnologías móviles, y en concreto el mercado de los *smartphones*, se encuentran en su mayor momento de expansión. Estos nuevos dispositivos traen nuevas formas de comunicación entre personas, así como nuevas formas de interacción entre usuario y máquina. La posibilidad de llevar un *smartphone* en tu bolsillo a todas partes conectado a Internet da la oportunidad de ofrecer nuevos tipos de servicios: servicios basados en la localización del usuario o en la información que proporcionan los sensores y otros elementos hardware del dispositivo. En concreto, dentro de este nuevo campo de aplicaciones, están empezando a popularizarse las aplicaciones de realidad aumentada. Este proyecto analiza la problemática del desarrollo de este tipo de aplicaciones y ofrece una solución a dichos problemas.

En este capítulo se ofrece una visión general del concepto de *Realidad Aumentada*, se describen los objetivos perseguidos en el desarrollo del proyecto *Look!* y se define la estructura del presente documento.

1.1. Realidad Aumentada

En el mundo de los dispositivos móviles se conoce por Realidad Aumentada (en inglés "augmented reality" ó AR) a la tecnología que permite la superposición, en tiempo real, de imágenes generadas por ordenador sobre imágenes del mundo real. Estos datos superpuestos pueden ser tanto información relati-

Figura 1.1: Ejemplo de Realidad Aumentada

va a elementos reales que están siendo visualizados, como datos independientes asociados a referentes físicos. En un entorno de este tipo, el usuario puede, además, interactuar con los objetos virtuales permitiendo, por ejemplo, alterarlos o añadir nuevos. En conclusión, se trata de incorporar información virtual al mundo real de una manera tal que el usuario pueda llegar a pensar que forma parte de su realidad cotidiana.

La Realidad Aumentada es una tecnología que está experimentando un auge en los últimos años, quizá debido al abaratamiento de los teléfonos inteligentes, y que está llamada a revolucionar nuestras vidas y la forma en la que interactuamos con la realidad. Por ello, es un campo abierto y para el que aún quedan infinidad de aplicaciones por descubrir. Algunos de los ámbitos en los que es posible el empleo de Realidad Aumentada son: proyectos educativos (aplicaciones en museos, parques temáticos ó exposiciones), medicina (asistencia al cirujano en tiempo real), entretenimiento (aplicación en nuevos tipos de juegos), simulación (de vuelos o trayectos terrestres) ó servicios de emergencias (evacuaciones de edificios)[12, 13, 14, 15, 16].

La implementación de un sistema de realidad aumentada conlleva la resolución de una serie de problemas de cierta complejidad de entre los que destaca la obtención de la localización del usuario, el dibujado de las distintas capas gráficas, la interacción con los objetos virtuales o el acceso a servicios remotos. Una vez resueltos estos problemas, la implementación de aplicaciones de realidad aumentada se vuelve más mecánica. Este proyecto ofrece una solución a dichos problemas.

1.2. Objetivos

El objetivo del proyecto es crear un framework de realidad aumentada que resuelva los problemas comunes encontrados en el desarrollo de aplicaciones de este tipo.

Además de otros, se abordarán como principales los siguientes problemas:

- **Localización en interiores**: El problema de localización exterior está resuelto en mayor o menor medida con la tecnología GPS. Sin embargo, la localización en interiores resulta imprescindible para el tipo de aplicaciones que se desea construir y que proporcionan información adicional dentro de edificios.

- **Representación gráfica**: El framework pretende aportar funcionalidad suficiente para permitir la representación gráfica de elementos tanto en dos dimensiones como tres dimensiones. El objetivo es que el usuario del framework no deba programar los mecanismos de dibujado, sino sólo proporcionar al sistema los datos que deben ser dibujados.

- **Interacción con los objetos virtuales**: El framework proveerá un sistema de interacción táctil con los objetos virtuales de forma que el programador de aplicaciones solo deba preocuparse de indicar el efecto producido por dicha interacción.

- **Acceso a Servicios Remotos**: Adicionalmente se pretende facilitar el acceso a servicios externos tales como bases de datos alojadas en servidores remotos.

Estos objetivos se validan con cuatro desarrollos experimentales:

- **Galería de Imágenes en 3D**: Galería de imágenes en tres dimensiones que permita interactuar con las mismas mediante realidad aumentada y que sirva además como experimentación sobre formas diferentes de visualizar imágenes, aprovechando las tres dimensiones, y ensayo con nuevas formas de organización de contenidos.
- **Red Social con soporte para Localización**: Se quiere con esta aplicación definir redes sociales que tengan sentido dentro de un edificio. En concreto, se quiere ensayar formas de comunicación donde la ubicación del individuo sea relevante, asociando información geográfica a documentos y definiendo filtros que impiden o autorizan el acceso a los mismos, así como utilizar la información de localización para encontrar usuarios.
- **Mundo Virtual**: Mundo de realidad virtual en el que los usuarios pueden desplazarse por medio de la realidad aumentada e interactuar con los objetos virtuales.
- **Juego interactivo**: Un juego en el que se muestren las capacidades gráficas y de interacción entre elementos, y que sirva para poner de manifiesto la introducción de lógica de cómputo compleja (trayectorias, colisiones, animaciones) entre los elementos de realidad aumentada.

1.3. Estructura del Documento

La presente memoria se estructura de la siguiente forma: En primer lugar se plantean los requisitos del software a desarrollar y se hace un repaso sobre el estado actual del arte. A continuación se presenta la arquitectura global del sistema y se describe en detalle cada uno de los módulos que la conforman. Seguidamente se presenta un tutorial para el desarrollo de aplicaciones mediante el framework y se describe el desarrollo de cuatro aplicaciones utilizadas para validar la funcionalidad implementada. Finalmente se incluyen algunos apuntes finales y las conclusiones del trabajo.

Capítulo 2

Requisitos

A la hora de desarrollar un framework de realidad aumentada es necesario especificar una serie de requisitos que sirvan de guía para las labores de diseño e implementación. En el presente capítulo se define en primer lugar la funcionalidad básica a implementar en forma de requisitos funcionales, y se explica en detalle cada uno de ellos. Seguidamente se definen una serie de requisitos no funcionales que aportan información extra a los requisitos funcionales. Finalmente se presenta una serie de restricciones adicionales que se deben cumplir.

2.1. Requisitos Funcionales

Se pretende desarrollar un framework que, con el objetivo de facilitar el desarrollo de aplicaciones de realidad aumentada, proporcione las siguientes funcionalidades:

- Dibujado de gráficos en dos y tres dimensiones.
- Construcción de Entidades representables en Realidad Aumentada.
- Interacción con los Objetos Virtuales.
- Localización en Interiores de Edificios.
- Servicio de Persistencia de Datos.

A continuación se describe en mayor detalle cada uno de los requisitos funcionales.

2.1.1. Capas de Gráficos en 2D y 3D

Para poder realizar aplicaciones atractivas y vistosas para el usuario, el framework definirá herramientas para el dibujado de elementos en dos y tres dimensiones.

Definirá objetos comunes en el dibujado, como textos y formas básicas, y ofrecerá herramientas para definir los estilos de dibujados, como colores y texturas.

También proveerá de funcionalidades geométricas, para facilitar labores comunes en el desarrollo de gráficos: puntos, vectores, matrices, planos y rayos, y todas las operaciones relacionadas.

2.1.2. Construcción de Entidades representables en Realidad Aumentada

A partir de una serie de datos característicos sin procesar, deberán poder construirse entidades complejas, de distintos tipos, que puedan ser representadas en realidad aumentada.

Así, la especificación de los datos que servirán como base para la construcción de elementos, deberá ser lo más general posible, para permitir la definición de un mayor número de elementos.

A partir de los datos del elemento y su tipo, podrán deducirse cosas comunes, cómo las representaciones gráficas para el elemento, y las interacciones permitidas.

2.1.3. Interacción con los Objetos Virtuales

El framework ofrecerá procesamiento de eventos táctiles (tocar, arrastrar y soltar) y eventos de cámara sobre los elementos de realidad aumentada, proporcionando interfaces para que el programador pueda implementar las respuestas adecuadas a dichos eventos.

2.1.4. Localización en Interiores de Edificios

Se debe proporcionar un sistema mixto de localización en interiores formado por los siguientes módulos:

- **Sistema Primario de Localización** por medio de señales Wifi que proporcione una localización a intervalos de tiempo definidos[1].
- **Sistema Secundario de Localización basado en Navegación Inercial** que complemente el sistema primario de localización proporcionando una posición en base al movimiento relativo del dispositivo en el espacio.
- **Integración** de los Sistemas de Localización Primario y Secundario.

Deberá ser posible la utilización de cada uno de los sistemas de forma separada y de ambos de manera combinada, siendo el programador de aplicaciones el encargado de decidir la funcionalidad necesaria para su aplicación.

2.1.5. Servicio de Persistencia de Datos

Se debe proporcionar un servicio de persistencia de Datos que proporcione dos tipos de almacenamiento.

- **Almacenamiento Local**, que almacene los datos de la aplicación durante distintas ejecuciones o que funcione como una caché de datos para agilizar las transacciones entre cliente y servidor.
- **Almacenamiento Global** donde se guarden los datos de todos los clientes de manera centralizada.

[1] Tasa de refresco definida en la sección 2.3.

El servicio de persistencia de datos permitirá el desarrollo de aplicaciones multiusuario por medio del acceso concurrente a datos comunes a todos los usuarios, de forma que compartan información y se permita la interacción entre ellos.

2.2. Requisitos No Funcionales

A continuación se definen los requisitos no funcionales del framework de realidad aumentada:

- Interacción con el usuario por medio de interfaces táctiles.
- Funcionamiento en dispositivos Android versión 2.2 y superior.
- No necesidad de Hardware externo al dispositivo.
- La persistencia deberá realizarse en Base de Datos.

Se debe garantizar un funcionamiento correcto en al menos los siguientes dispositivos:

- Google Nexus One
- HTC Desire HD
- HTC Tattoo

2.3. Restricciones

Las principales restricciones del proyecto son:

- **Precisión de la localización en interiores**: Precisión a nivel de habitación.
- **Tasa de refresco de datos de localización**: Actualizaciones a intervalos menores de 5 segundos para garantizar el funcionamiento en tiempo real.
- **Rendimiento**: Debe garantizarse fluidez en el dibujado de los gráficos sobre la realidad (mínimo 15 fps) en un dispositivo con 528 MHz y 256 MB de RAM.

Capítulo 3

Estado del Arte

Antes de empezar el desarrollo del framework de realidad aumentada se ha realizado un estudio sobre el estado del arte. Se pretende conocer un poco más de cerca cómo ha sido abordado el tema, en qué estado se encuentra en el momento de plantear el proyecto y cuáles son las tendencias, así como comprobar si es posible aprovechar alguna de las tecnologías existentes. En este capítulo se presentan las tecnologías analizadas y se resumen las conclusiones de dicho análisis.

En primer lugar se realiza un análisis de diferentes frameworks de realidad aumentada actuales. Una vez demostrado que no satisfacían los requisitos presentados en el capítulo anterior, decidimos comenzar al desarrollo de nuestro framework.

La siguiente decisión consistía en elegir el sistema operativo en el que desarrollar el framework. En la sección 3.2 se analizan los sistemas operativos móviles más relevantes en la actualidad.

Dentro del desarrollo en sí, se plantearon diferentes caminos para acometer la parte gráfica. El dibujado en dos dimensiones está resuelto de manera satisfactoria para nuestras necesidades por Android. Sin embargo, la aproximación al dibujado en tres dimensiones planteaba dos soluciones bien diferenciadas sobre Android, basadas en dos versiones de OpenGL ES. La problemática de la elección de versión se trata en la sección 3.3.

También es necesario estudiar la integración de la aplicación con sistemas remotos, para lo cual, Android plantea varias soluciones. En la sección 3.4 se estudian las distintas alternativas disponibles.

Finalmente, una aplicación de realidad aumentada necesita conocer la localización del usuario en interiores. En la sección 3.5 se estudian distintas alternativas tecnológicas existentes, y en la sección 3.6 diversas técnicas de localización en interiores.

Tras el análisis de estos problemas, la sección 3.7 recoge las conclusiones de todo lo presentado.

3.1. Frameworks para realidad aumentada

Puesto que el espectro de aplicaciones abarcando las funcionalidades ofrecidas por *Look!* resulta bastante amplio, se han escogido las más representativas y las

que más funcionalidades de *Look!* implementan. Dichas funcionalidades son las siguientes:

- Sistema de localización en interiores.
- Capa de representación gráfica en 2 dimensiones.
- Capa de representación gráfica en 3 dimensiones.
- Interacción con objetos virtuales.
- Integración con servicios de persistencia remotos.

En la tabla 3.1 se compara la funcionalidad de cada uno de de los sistemas analizados, comprobando si poseen las siguientes características: integración con cámara, gráficos 2D y 3D; integración y creación de servicios de persistencia remotos; localización en exteriores e interiores; soporte para sistemas de navegación inercial e interacción con objetos virtuales.

3.1.1. Layar Reality Browser

Layar [1] es una navegador de realidad aumentada, desarrollado para plataformas móviles como Android o iPhone (ver figura 3.1). Tiene una licencia privativa por lo que no se dispone de acceso al código fuente.

Está basado en un sistema de capas que funcionan sobre el navegador de realidad aumentada base, y que el usuario puede decidir si mostrar o no. Cada una de estas capas es desarrollada independientemente por compañías, personas a título personal o programadores independientes, y representan mundos de realidad aumentada paralelos y disjuntos.

Figura 3.1: La aplicación de realidad aumentada *Layar*

[1] http://www.layar.com/

Características Sus características principales son:

- Localización basada en GPS.
- Capas en dos dimensiones
- Capas en tres dimensiones
- Estructura de cliente-servidor, permitiendo la descarga de datos de las capas definidas por los usuarios en tiempo real.
- Promoción de las capas: las capas definidas por el usuario pueden ser puestas a disposición de la comunidad de manera centralizada.

3.1.2. mixare

mixare (mix Augmented Reality Engine)[2] es un framework de código abierto para realidad aumentada, publicada bajo la licencia GPLv3 [3]. mixare está disponible para sistemas Android y para iPhone.

Este framework permite construir aplicaciones completas y proporciona funciones para asociar puntos (coordenadas) y texto. Es decir, su funcionalidad se resume a permitir asociar texto a localizaciones (ver figura 3.3).

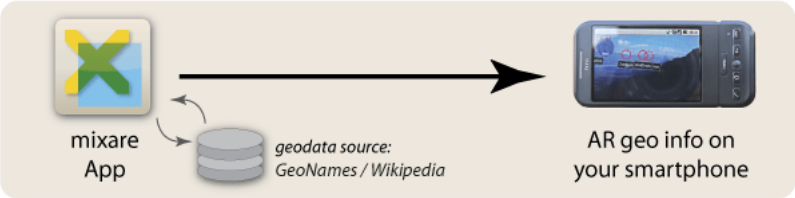

Figura 3.2: Estrucutra de una aplicacion con *mixare*

mixare ofrece capacidades de representación en dos dimensiones limitadas a cajas de texto e imágenes, localización por GPS, y acceso a datos por conexión de red.

3.1.3. AndAR

AndAR es una aplicación que permite la representación de objetos tridimensionales sobre marcas físicas en el mundo real.

Consiste en un visor de elementos 3D, que son situados sobre marcas predeterminadas. Estos elementos se verán afectados por la orientación y el punto de vista del usuario.

[2] http://www.mixare.org/
[3] http://www.gnu.org/licenses/gpl-3.0.html

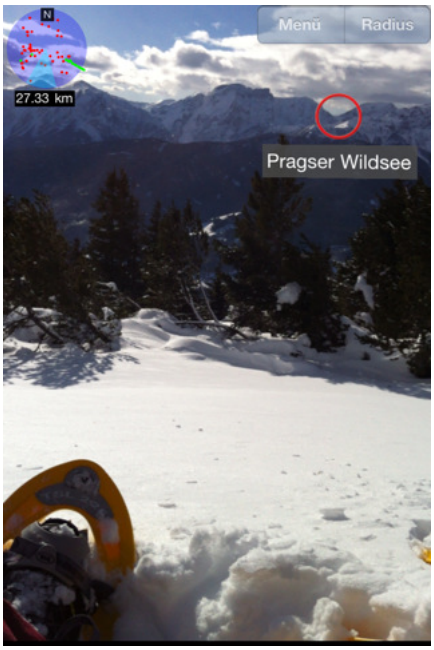

Figura 3.3: Captura de *mixare*

3.1.4. Conclusiones

Se observa en la tabla 3.1 que AndAR y mixare ofrecen una funcionalidad muy pobre para nuestras necesidades. Layar es la opción más completa, pero no dispone de localización en interiores ni permite extender dicha funcionalidad por lo que resulta inadecuado para nuestro proyecto. Así, optamos por desarrollar el framework desde cero.

3.2. Sistemas operativos

Para el desarrollo de software de realidad aumentada, son necesarias tecnologías que proporcionen las funcionalidades adecuadas. Se requiere tecnologías con capacidad de localización, capacidad para mostrar imágenes desde una cámara y APIs gráficas en dos y tres dimensiones. Además, es necesario disponer de tecnologías de creación de interfaces de usuario amigables y que provean elementos comunes y suficientemente conocidos (como ventanas, botones, cajas de texto, menús, listas, etc.).

Se pueden catalogar los *smartphones* actuales en función de su sistema operativo. Los *smartphones* de altas prestaciones se encuentran dominados por dos sistemas operativos mayoritarios:

- **iOS**: Se encuentra en los *smartphones* y tabletas fabricados por Apple (*iPhone, IPad*). De naturaleza cerrada, sus aplicaciones deben de ser aprobadas por Apple y están sometidas a un conjunto de normas más o menos estricto.

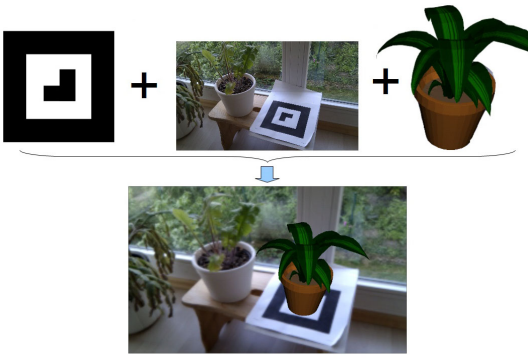

Figura 3.4: Esquema del funcionamiento de *AndAR*

Aplicación	**Look!**	**Layar**	**mixare**	**AndAR**
Integración con cámara	X	X	X	X
2D	X	X	X	
3D	X	X		X
Integración con Servicios Remotos	X	X	X	
Creación de Servicios Remotos	X	X		
Localización en interiores	X			
Localización por GPS		X	X	
Sistema de Navegación Inercial	X			
Interacción con Objetos Virtuales	X	X		

Cuadro 3.1: Tabla comparativa de las características de *Look!* con otras aplicaciones realidad aumentada

- **Android**: De naturaleza más abierta, basado en software libre y desarrollado por Google pero disponible en *smartphones* de diversos fabricantes *(*HTC, Motorola, Sony Ericsson, Samsung, etc.)

En concreto, como muestra la figura 3.5, el crecimiento de Android ha sido exponencial en los últimos años. Esto es relevante debido a que desarrollar para dicho sistema supone hoy en día llegar a millones de usuarios de dispositivos móviles de diferentes fabricantes en todo el mundo, y da una muestra de que esta tendencia se encuentra lejos de revertirse en un futuro cercano. Esto lo convierte en una plataforma muy atractiva para el desarrollo de nuevas aplicaciones móviles.

En las siguientes subsecciones se explican los principales sistemas operativos consideradas para la implementación de *Look!*, sus características, ventajas y desventajas, y finalmente se expone la decisión tomada.

3.2.1. Android

Android es un sistema operativo de código abierto basado en *Linux* diseñado para dispositivos móviles, como smartphones o tablets, desarrollado por Google.

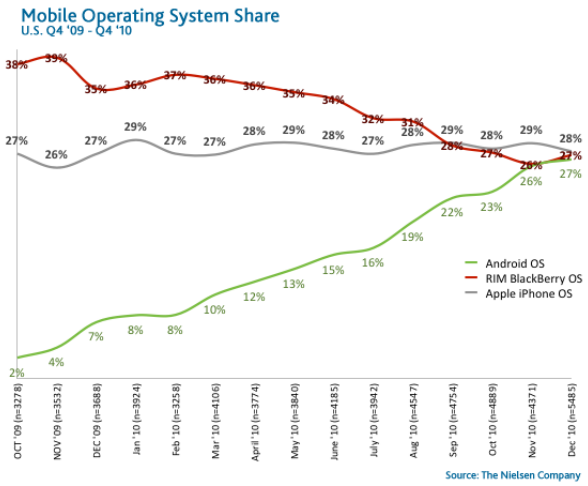

Figura 3.5: Mercado Android entre 2009 y 2010

Apareció en 2005, y es un sistema relativamente joven e inmaduro. Su núcleo está programado en C y C++, y puede ser modificado y compilado para crear sistemas personalizados.

Aunque puede ejecutarse código C directamente sobre el núcleo de la plataforma a través del NDK (Native Development Kit), la mayoría de aplicaciones son programadas a través del Android SDK, que utiliza el lenguaje Java como base.

3.2.2. iOS

iOS es un sistema operativo móvil cerrado de Apple utilizado por el iPhone, el iPod Touch y el iPad.

Es un sistema maduro que permite la programación de aplicaciones en Objective-C, un lenguaje de programación orientado a objetos creado como superconjunto de C, pero que implementa un modelo de objetos parecido al de Smalltlak.

Para el desarrollo, se utiliza el **iOS SDK**, que puede ser obtenido de manera gratuita tras registrarse como desarrollador de la plataforma.

El desarrollo para iOS debe realizarse desde el sistema operativo Mac.

3.2.3. Tecnología de escritorio más webcam

Por la complejidad asociada a los dispositivos móviles, se plantea la posibilidad de desarrollar el framework a modo prueba de concepto para equipos de escritorio, simulando la realidad aumentada con ayuda de la webcam, con un lenguaje orientado a objetos, como C++ o Java.

Esta vía aporta una capacidad de cómputo mayor, así como infinidad de bibliotecas con funcionalidades inexistentes en los sistemas móviles.

Su clara desventaja es que sólo sería una prueba de concepto, puesto que la realidad aumentada adquiere su mayor sentido en dispositivos móviles.

3.2.4. Tecnología escogida

La tecnología escogida es **Android**, por las siguientes razones:

- **Resultados reales**: Aunque la prueba de concepto en una plataforma de escritorio supondría menos límites técnicos y de procesamiento, el desarrollo directo en la plataforma móvil arrojará unos resultados más reales de los problemas y soluciones que pueden lograrse.
- **Java**: Los miembros del equipo tienen amplios conocimientos de Java, pero no de Objective-C, lo que ahorrará tiempo en el aprendizaje de la plataforma.
- **Sistema abierto**: Android es un framework de código abierto, lo que permitirá todo tipo de experimentación necesaria con elementos no ofrecidos a priori por el SDK.
- **Plataforma en expansión**: Aunque iOS lleva más tiempo en el mercado, Android está sufriendo un crecimiento exponencial en la actualidad, lo que amplia enormemente el mercado para el framework.
- **Acceso a dispositivos de prueba**: Los miembros del equipo cuentan con dispositivos de prueba Android, pero no de iOS.

Al momento del inicio del proyecto, Android se encuentra en su versión 2.2. Ésta será la versión del SDK utilizada en el desarrollo de *Look!*, no descartando utilizar una versión más moderna si apareciera.

3.3. OpenGL ES

OpenGL[4] es una API gráfica, multilenguaje y multiplataforma, diseñada para la producción de gráficos 2D y 3D.

OpenGL ES[5] es una versión para plataformas portables de OpenGL. Consiste en una versión reducida de la versión *OpenGL* de escritorio, y está pensada para funcionar en sistemas embebidos: consolas portátiles, dispositivos móviles, accesorios electrónicos o vehículos.

Android implementa OpenGL ES en sus dos versiones, la 1.1 y la 2.0, pero de diferentes maneras. A continuación, una breve descripción para cada una de ellas.

3.3.1. OpenGL ES 1.1

OpenGL ES 1.1 está definida a partir de la especificación 1.5 de Open GL e intenta explotar la aceleración hardware de los dispositivos. Además, es compatible hacia atrás con la versión 1.0.

Android provee la clase *GLSurfaceView*, una clase completamente preparada para trabajar con esta versión de Open GL ES y que se maneja como cualquier otra *View* de Android. Su programación es sencilla y puede ser realizada desde el SDK.

[4] http://www.opengl.org/
[5] http://www.khronos.org/opengles/

3.3.2. OpenGL ES 2.0

OpenGL Es 2.0 está definida con base en la especificación 2.0 de Open GL. Utiliza una filosofía totalmente distinta a su versión predecesora, basado en pipeline 3D programable y shaders definidos por el OpenGL ES Shading Language. Además, no es compatible hacia atrás.

Al inicio del proyecto, en octubre de 2010, la utilización de esta versión de OpenGL estaba supeditada al uso del NDK (Native Development Kit). El NDK permite ejecutar directamente programas en C o C++ de forma nativa en Android, puesto que el sistema operativo está construido sobre una base Linux. Así, sólo era posible programar la versión 2.0 en C o C++ a través del NDK.

En las últimas versiones de Android, esto ha cambiado, y en la actualidad es posible programar con Open GL ES 2.0 desde el SDK de Android.

3.3.3. OpenGL ES 1.1 vs 2.0

Aunque la versión 2.0 ofrece mayor rendimiento y mayor funcionalidad, en *Look!* se ha decidido utilizar la versión 1.1 por lo siguientes motivos:

- **Compatibilidad de dispositivos**: La versión 2.0 necesita una aceleración hardware con la que no cuentan todos los dispositivos actuales. La versión 1.1, sin embargo, es soportada por todos los dispositivos Android.
- **Sencillez**: Cuando se inició el proyecto, el uso de la versión 2.0 implicaba programación con el NDK, lo que conlleva mayor complejidad en el código y la programación, al tener que combinar dos lenguajes distintos. La versión 1.1 puede ser programada directamente desde el SDK.
- **Rendimiento adecuado**: Aún siendo la versión 2.0 la de mayor rendimiento, la mayoría de aplicaciones existentes al inicio del proyecto, funcionando con 1.1, mostraban una capacidades gráficas similares o superiores a las buscadas en *Look!*. Así que fue considerado que el equilibrio entre complejidad y resultado era el adecuado.

3.4. Análisis de Servicios Web

Una de las decisiones más importantes que se deben tomar a la hora de diseñar un sistema distribuido es la elección de la tecnología de comunicación entre cliente y servidor, ya que marcará las limitaciones del framework para el acceso a servicios remotos. Por este motivo nos decantamos por un **Servicio Web**[6], donde nos conectamos usando el protocolo HTTP, permitiéndonos elegir el puerto para establecer la conexión. Un servicio web es adaptable a otros tipos de cliente (Web, Windows, IOS, ...), haciendo posible una futura ampliación del framework a otros entornos.

A continuación presentamos un estudio sobre las arquitecturas más utilizadas para dar soporte a servicios web y su compatibilidad con *Android*. Tras valorar esta comparativa, nos decidimos por RESTful.

[6]El consorcio W3C define los Servicios Web como sistemas software diseñados para soportar una interacción interoperable maquina a maquina sobre una red. Los Servicios Web suelen ser APIs Web que pueden ser accedidas dentro de una red (principalmente Internet) y son ejecutados en el sistema que los aloja.

3.4.1. Arquitecturas más comunes para Servicios Web

- ***Remote Procedure Calls*** (RPC, Llamadas a Procedimientos Remotos): Los Servicios Web basados en RPC presentan una interfaz de llamada a procedimientos y funciones distribuidas, lo cual es familiar a muchos desarrolladores. Típicamente, la unidad básica de este tipo de servicios es la operación WSDL (WSDL es un descriptor del Servicio Web, es decir, el homologo del IDL para COM). Las primeras herramientas para Servicios Web estaban centradas en esta visión. Algunos lo llaman la primera generación de Servicios Web. Esta es la razón por la que este estilo está muy extendido. Sin embargo, ha sido algunas veces criticado por no ser débilmente acoplado, ya que suele ser implementado por medio del mapeo de servicios directamente a funciones específicas del lenguaje o llamadas a métodos.

- ***Arquitectura Orientada a Servicios*** (Service-oriented Architecture, SOA). Los Servicios Web pueden también ser implementados siguiendo los conceptos de la arquitectura SOA, donde la unidad básica de comunicación es el mensaje, más que la operación. Esto es típicamente referenciado como servicios orientados a mensajes. Los Servicios Web basados en SOA son soportados por la mayor parte de desarrolladores de software y analistas. Al contrario que los Servicios Web basados en RPC, este estilo es débilmente acoplado, lo cual es preferible ya que se centra en el "contrato" proporcionado por el documento WSDL, más que en los detalles de implementación subyacentes.

- ***REST*** (REpresentation State Transfer). Los Servicios Web basados en REST intentan emular al protocolo HTTP o protocolos similares mediante la restricción de establecer la interfaz a un conjunto conocido de operaciones estándar (por ejemplo GET, PUT,...). Por tanto, este estilo se centra más en interactuar con recursos con estado, que con mensajes y operaciones.

  Aunque REST no es un estándar, está basado en estándares: HTTP, URL, representación de los recursos mediante formatos estándar (XML/HTML/GIF/JPEG/...), tipos MIME (text/xml, text/html...).

3.4.2. Compatibilidad con Android

Analizamos cada una de estas tres arquitecturas evaluando su implementación en el cliente (Android).

El protocolo elegido para el cliente también servirá para diseñar el servidor, ya que este es más adaptable y puede ser desarrollado en cualquier tecnología.

XML-RPC

XML-RPC es un protocolo de llamada a procedimiento remoto que usa XML para codificar los datos y HTTP como protocolo de transmisión de mensajes. Destaca por su simplicidad, a pesar de que la mayoría de los protocolos basados

en *RPC* son muy complejos de utilizar.

Los datos en este protocolo son enviados como "text/xml" y hay librería para casi todos los lenguajes, incluido Android (android-xmlrpc[7]) que simplifica el uso de esta tecnología encapsulando las clases entidad en un *XML* para su envío o recepción.

La desventaja de este protocolo es que a pesar de su sencillez, comunicar tipos u objetos definidos por el usuario se vuelve innecesariamente complejo[8].

SOAP

SOAP es un protocolo basado en *SOA*, por el cual la comunicación se basa en el envío de mensajes entre el cliente y el servidor. Este a pesar de estar muy extendido entre los Servicios Web, Android lo desaconseja ya que es muy pesado y complejo de implementar.

Hay una librería para Android (ksoap2-android)[9] que nos permite crear los "objetos SOAP" y procesar su envío. Aunque en nuestras pruebas hemos comprobado que no está del todo estable y esto provoca muchos fallos.

RESTful (HTTP)

REST proporciona un protocolo cliente/servidor sin estado: cada mensaje HTTP contiene toda la información necesaria para comprender la petición. Como resultado, ni el cliente ni el servidor necesitan recordar ningún estado de las comunicaciones entre mensajes.

Un Servicio Web **RESTful** (también llamado RESTful web API) es un simple servicio web implementado usando HTTP y los principios de REST. Estos son una colección de recursos, con estos aspectos definidos:

- La URI base para el servicio web, tal como http://example.com/resources/
- El tipo de datos soportado por el servicio web. Este es a menudo JSON, XML o YAML pero podemos usar cualquier otro MIME[10] (Internet media type).
- El conjunto de operaciones soportadas por el servicio web usando los métodos HTTP (POST, GET, PUT o DELETE).

A diferencia de los servicios web basados en SOAP, no hay ningún estándar oficial para servicios web RESTful aunque si nos aconsejan sobre su implementación. También hay una librería que implementa un servicio web basado en REST, Restlet[11] la cual tiene también versión para Android.

[7] Información sobre la librería para Android: http://code.google.com/p/android-xmlrpc/

[8] Requiere implementar el mecanismo de serialización en XML, ó el uso de librerías externas.

[9] Más información en la web oficial: http://code.google.com/p/ksoap2-android/

[10] Multipurpose Internet Mail Extensions o MIME son una serie de convenciones o especificaciones dirigidas al intercambio a través de Internet de todo tipo de archivos (texto, audio, vídeo, etc.) de forma transparente para el usuario

[11] Más información en la web oficial: http://www.restlet.org/

La gran ventaja de usar RESTful es su acogida por parte de Android, quien ha publicado un esquema para su implementación. Aunque la librería Restlet ayude a crear el servicio web, vemos preferible definir los métodos de la arquitectura en nuestro framework, ajustándose a los requisitos necesarios, tal como lo especifica Android.

3.4.3. Valoración

Del análisis anterior, tanto SOAP como RESTful resuelven la misma funcionalidad. Aunque nos decantamos por RESTful por tener una mayor acogida y documentación sobre su uso en Android.

XML-RPC podría haber sido la alternativa a REST, precisamente no hemos escogido este protocolo para no limitar el Servicio Web. Con RESTful en caso de ampliar el framework para enviar al servidor ficheros u objetos complejos no tendríamos problema.

3.5. Análisis de Proyectos de Localización

Uno de los componentes principales del framework de realidad aumentada es la localización de un dispositivo móvil dentro de un edificio. Esta funcionalidad permite un sinfín de aplicaciones, y es una parte esencial en nuestro desarrollo. Sin embargo, encontrar la solución a este problema no es en absoluto trivial, ya que dentro de un edificio no es posible utilizar sistemas GPS, y los requerimientos en cuanto a precisión son mayores.

A la hora de desarrollar un sistema de localización en interiores de edificios para nuestro framework, decidimos realizar en primer lugar un estudio de de proyectos existentes de localización desarrollados por diversas empresas y universidades con el fin de analizar su adecuación a nuestro proyecto y decidir si merece la pena hacer un desarrollo desde cero o reutilizar alguno de los proyectos ya existentes.

En las siguientes secciones se describen los diferentes proyectos que se han analizado: *PlaceLab*, *Ekahau* y *Indoor Navigation System for Handheld Devices*. Finalmente se enumeran las conclusiones del análisis.

3.5.1. PlaceLab

PlaceLab[12] es un proyecto Open Source desarrollado por Intel Research con el objetivo de crear un sistema de geolocalización basado en la técnica conocida como 'Cálculo del punto central a partir de nodos predefinidos' que se explica en la sección 3.6.3. La documentación del proyecto reza que dicha técnica es válida tanto para exteriores como para interiores de edificios.

Para ello se vale de fuentes de ondas de radiofrecuencia que pueden ser de tres tipos distintos:

- Puntos de acceso WiFi.
- Dispositivos Bluetooth.

[12] `http://ils.intel-research.net/place-lab`

- Antenas de radio GSM.

De esta forma, la información recibida en un momento dado se contrasta con bases de nodos fuente de ondas, conocidos como balizas, y sus coordenadas. Se trata de buscar qué ondas se reciben en cada momento, dónde se encuentran sus fuentes y a partir de esta información calcular el punto central de todas ellas.

En las siguientes subsecciones se describen los principales módulos que conforman la aplicación, nuestra experiencia utilizando la herramienta y las conclusiones finales de nuestro análisis.

Módulo Stumbler

La localización mediante esta técnica se basa en contrastar las ondas detectadas en un momento dado con bases de datos previamente elaboradas de nodos fuente y coordenadas asociadas. Es necesario elaborar en primer lugar dichas bases de datos.

En el caso de PlaceLab, se utilizan bases de datos ya existentes y de acceso público provenientes de páginas de *War Driving*[13], y también utilizan su propia base de datos. Para ello, se utiliza una base de datos local como caché que se sincroniza con dichas bases de datos de forma periódica.

Además, el software provee una aplicación conocida como *Stumbler* que, conectada a un GPS, es capaz de capturar nuevos nodos y añadirlos a la base de datos caché para su posterior sincronización.

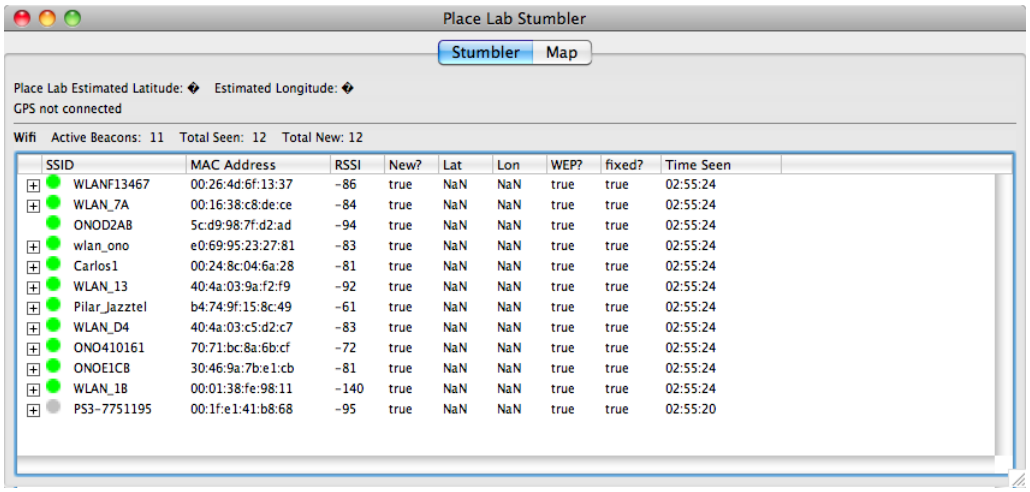

Figura 3.6: PlaceLab Stumbler

Módulo Tracker

El tracker es la parte encargada de estimar una localización del dispositivo en base a las balizas detectadas en un momento dado. Existen distintos métodos para hacerlo y además dicha funcionalidad es extensible, permitiendo añadir métodos propios a los ya existentes.

Por defecto, Placelab incluye los siguientes métodos:

[13]Por ejemplo `http://wigle.net/`.

- **Centroid**: Consiste en la búsqueda del punto central de las balizas detectadas.
- **Filtro de partículas**: Se utiliza un filtro de partículas para modelar el movimiento del dispositivo y aplicar modelos estadísticos para predecir la posición en base a dicha información.

La herramienta *Tracker* utiliza alguno de estos métodos para la estimación de la posición en cada instante. Para ello en primer lugar captura la información de las ondas de radio en cada instante y procesa la posición en base al algoritmo seleccionado.

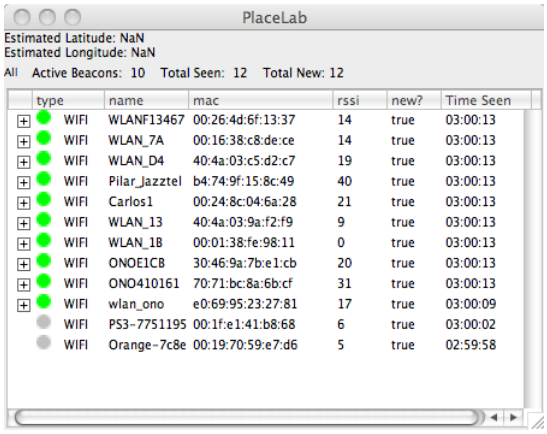

Figura 3.7: PlaceLab Tracker

Análisis

A la hora de realizar el análisis de esta herramienta, nos encontramos con algunas dificultades debido a que en el momento de realizar las pruebas el proyecto llevaba ya algunos años discontinuado. Las principales dificultades que encontramos fueron la siguientes:

- **Bases de datos**: Desde la aplicación nos resultó imposible cargar las bases de datos de puntos wifi a la caché local. La base de datos propia de placelab se encontraba fuera de línea, y la conexión con la base de datos de wigle.net se basaba en una API antigua. Por ello, para realizar las pruebas tuvimos que editar manualmente la base de datos cache (HSQLDB) introduciendo manualmente las coordenadas y la información de los puntos de acceso a utilizar.
- **Librerías**: Por limitaciones de hardware disponible, las pruebas se realizaron en una máquina con Mac OS X utilizando únicamente puntos de acceso WiFi. Para ello, debido a la antigüedad del software, fue necesario portar el código encargado de acceso al hardware a librerías nativas que utilizaran la nueva API de Mac OS X. Se dedica el anexo B para describir este proceso con mayor detalle, ya que no resultó en absoluto trivial.

- **Hardware necesario**: Para la función de captura de nodos (Stumbler) se requería hardware GPS conectado al puerto de serie, por lo que no fue posible utilizar dicha funcionalidad.

Una vez sorteados los problemas iniciales, se hicieron diversas pruebas con hasta 4 routers colocados a lo largo de un edificio para comprobar la precisión que es capaz de obtener y la adecuación a nuestro proyecto.

Es necesario destacar que mediante este sistema de localización la posición de los routers es relevante ya que la posición resultante se infiere a través de las posiciones de cada uno de los routers definidos.

Sin embargo, ninguno de los algoritmos consiguió resultados satisfactorios, ya que la precisión obtenida fue muy baja. El algoritmo de calculo de la posición central a los nodos es demasiado simple y no tiene en cuenta la intensidad de señal recibida por cada red. Por otra parte, sistemas como el filtro de partículas puede resultar útil en vehículos, donde tiene sentido obtener una predicción de la trayectoria, pero no resulta adecuado para personas en interiores.

Por otra parte, existe una limitación añadida debido a que este software no está diseñado para trabajar en tres dimensiones sino en coordenadas bidimensionales, por lo que surge el problema de la detección de la planta dentro de un edificio. Existe un proyecto basado en placelab que amplía las características del software añadiendo soporte para plantas de edificio.

Sin embargo, sus conclusiones no se presentaban demasiado halagüeñas, y llegado a este punto, decidimos optar por otras alternativas para la localización.

Creemos que este sistema puede resultar útil para obtener una localización aproximada en exteriores en situaciones en las que se carece de un sistema de posicionamiento GPS, pero el punto de vista desde el que aborda el problema lo hace inadecuado para las necesidades de precisión de nuestro proyecto, en torno a unos pocos metros. De todas formas, el análisis de este proyecto nos proporcionó una valiosa experiencia que utilizamos posteriormente para diseñar sistemas de localización más adaptados a nuestras necesidades.

3.5.2. Ekahau

Se trata de una empresa que ofrece soluciones avanzadas de localización en interiores mediante sistemas wifi. En un primer momento tratamos de analizar su sistema de localización para comprobar si se adecuaba al nuestro proyecto y si era posible implantarlo evitándonos tener que diseñar un sistema de localización desde cero. Para ello nos pusimos en contacto con la empresa para tratar de obtener una demo gratuita del producto para analizarla más en detalle, ya que la información disponible era bastante escasa. Sin embargo, tardamos mucho tiempo en obtener respuesta y su sistema nos pareció poco aplicable tal y como estaba planteado el proyecto, además de tener un coste elevado, por lo que finalemente decidimos descartarlo en favor de un desarrollo propio acorde a nuestras necesidades.

3.5.3. Proyecto Indoor Navigation System for Handheld Devices

Este proyecto se basa en el trabajo *Application of Channel Modeling for Indoor Localization Using TOA and RSS* [14]realizado por Ahmad Hatami para el *Worscester Polytechnic Institute* de Massachusetts, y trata de proporcionar un prototipo utilizando las ideas que se desarrollan en dicha investigación.

Estas ideas son principalmente la creación automática de un mapa de radio mediante procesado de mapas por ray tracing[15], la detección de movimiento[16] y la combinación de geolozalización por ondas wifi y movimiento inercial mediante técnicas estadísticas. El proyecto en sí está formado por dos componentes principales:

- Una serie de librerías para Matlab para el procesado de mapas y obtención de los mapas de radio.
- Una aplicación para Android que implementa la localización en función de los mapas de radio, y la detección de pasos.

En las siguientes subsecciones se describen en mayor detalle cada uno de los módulos y se presentan las conclusiones del análisis realizado sobre este proyecto.

Módulo de Creación de mapas de radio mediante ray tracing para Matlab

Consiste en un conjunto de librerías para Matlab que implementa los algoritmos para el procesado de mapas y el modelado de la propagación de la señal wifi de diferentes puntos de acceso dentro de un edificio.

Para ello, es necesario introducir los siguientes parámetros de entrada:

- Información del Punto de Acceso:
  - Coordenadas (X,Y).
  - Potencia del punto de acceso, medida a un metro de distancia.
  - Dirección MAC.
- Una lista de muros, definidos por sus coordenadas (X1,Y1), (X2,Y2) y el nivel de atenuación que producen en la señal.

El resultado del procesado es un mapa de radio como el que se muestra a continuación, resultado de procesar un mapa de prueba creado para la ocasión.

Módulo de Inferencia de la posición

La inferencia de la posición se encuentra implementada en una aplicación Android que consta de los siguientes módulos:

- Captura de datos wifi y contraste con los mapas de radio.

[14]`http://www.wpi.edu/Pubs/ETD/Available/etd-053106-160141/`.
[15]Explicado en la sección 3.6.8
[16]Explicado en la sección 3.6.6

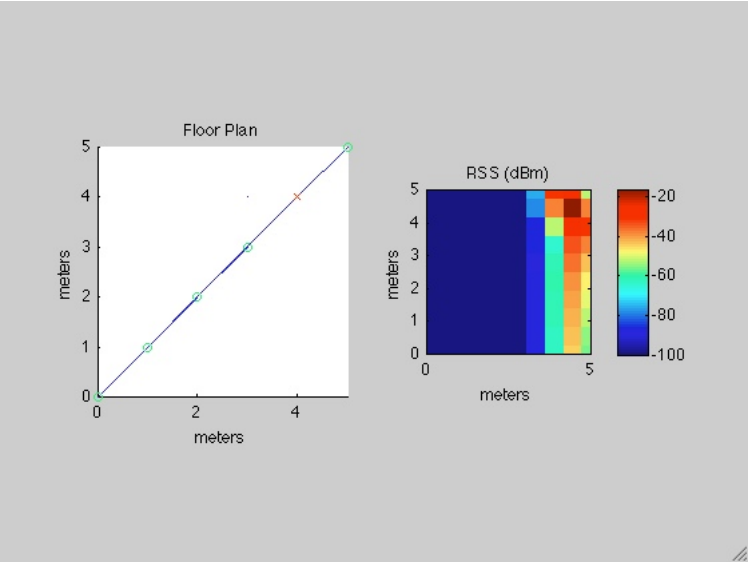

Figura 3.8: Mapa de Radio resultado del procesado

- Sistema de navegación inercial basado en la detección de movimiento.
- Combinación de los sistemas anteriores mediante técnicas estadísticas para la obtención de la posición final.

La aplicación se encarga de mostrar la posición del usuario en el mapa, además de algunas otras funcionalidades tales como el cálculo de la ruta entre dos puntos del mapa. En la figura 3.9 se aprecia este software funcionando sobre un dispositivo Android y mostrando en un mapa (simplemente un cuadrado en rojo) la posición del usuario y su orientación.

Análisis

Tuvimos bastantes dificultades para la implantación de este sistema, ya que no se trata de un proyecto abierto cuyo código fuente puede descargarse, sino de un documento en el que parte del código se encuentra disponible en un anexo, aunque hay partes incompletas. Una vez que conseguimos poner en marcha la funcionalidad principal, realizamos algunas pruebas.

Sin embargo, nos encontramos con algunas limitaciones para poder poner a prueba el sistema en situaciones reales:

- No disponibilidad de mapas de calidad a escala.
- Desconocimiento de la localización exacta de los puntos de acceso.

Finalmente decidimos utilizar las lecciones aprendidas en el estudio de este sistema para el diseño de un sistema de localización más adaptado a las necesidades de nuestro proyecto. Algunas partes, tales como la detección de pasos han sido aprovechadas en buena medida. Otras partes, como la elaboración de los mapas de radio, han sido rediseñadas. No obstante, el estudio de este proyecto nos aporto información muy valiosa, y sin ello nos habría resultado muy difícil desarrollar nuestro propio sistema de localización.

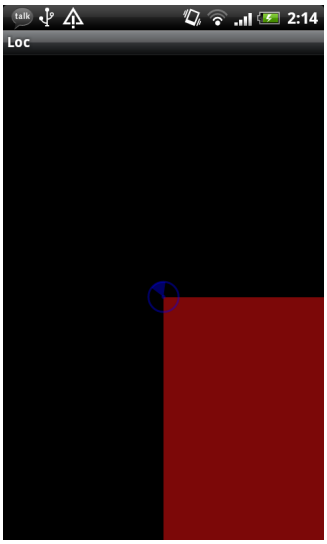

Figura 3.9: Ejecución de Indoor Navigation System for Handheld Devices

3.5.4. Conclusiones

Después de realizar el análisis de los sistemas de localización anteriormente descritos, se llegó a la conclusión de que ninguno de ellos se ajustaba a las necesidades de nuestro proyecto y se decidió hacer un desarrollo desde cero partiendo de la experiencia adquirida en dicho análisis.

De entre las lecciones aprendidas, destacamos las siguientes:

- Los sistemas de localización por Wifi basados en mapas de radio ofrecen resultados aceptables sin necesidad de hardware externo.
- La necesidad de combinar más de un sistema de localización: un sistema primario de localización por radiofrecuencia y un sistema secundario de navegación inercial.

3.6. Técnicas de Localización en Interiores

Una vez descartada la reutilización de proyectos de localización existentes y tomada la decisión de iniciar un desarrollo desde cero más adecuado a nuestras necesidades, en primer lugar se realizó un exhaustivo análisis de las posibles técnicas que pueden emplearse para la resolución de este problema.

Los requerimientos principales que debe tener nuestro sistema de localización son en primer lugar precisión y versatilidad [17], y en segundo lugar facilidad de implantación y aprovechamiento del hardware existente en dispositivos móviles actuales.

De entre los métodos analizados, se realizó una selección de los más prometedores para un estudio en mayor profundidad, realización de pruebas e implementación de prototipos. Finalmente se optó por una combinación de las técnicas

[17] En el sentido de que el usuario no se tenga que preocupar del sistema que subyace para la obtención de la localización.

descritas en las secciones 3.6.6 (localización mediante detección de movimiento) y 3.6.7 (localización mediante mapas de Radio Wifi).

En esta sección se describen las posibles técnicas análizadas, sus ventajas e inconvenientes.

3.6.1. Localización mediante etiquetas RFID

Este fue uno de los primeros planteamientos que se hizo, y consiste en la colocación a lo largo de todo el edificio de etiquetas RFID. Cada una de las etiquetas posee un identificador único que puede ser leído por radiofrecuencia, de forma que al ser detectadas por un lector RFID puede inferirse la posición a partir de una base de datos de identificadores y posiciones.

Este sistema fue descartado por diversos motivos, entre otros la necesidad de un hardware específico, su escasa aplicabilidad en edificios grandes debido al número de etiquetas necesarias y su dificultad de implantación.

Ventajas:

- Sencillez.

Inconvenientes:

- Necesidad de un gran número de etiquetas para que sea preciso, debido a su escaso alcance.
- Dificultad de implantación y poca escalibilidad.
- Necesidad de hardware externo al dispositivo y coste asociado.

3.6.2. Localización mediante marcas visibles

Este es otro de los métodos analizados y rápidamente descartados. Consiste en la utilización de marcas visibles en las paredes que puedan ser detectadas mediante una cámara y reconocidas por un sistema de visión. Cada marca equivale a una posición específica almacenada en una base de datos.

Aunque este método puede tener aplicaciones útiles, por ejemplo en museos, no cumple con nuestros requerimientos de localización ya que se pretende disponer de una localización pasiva en la que el usuario no tenga que preocuparse de en ningún momento de la forma en que la posición es inferida.

Ventajas:

- Al ser marcas localizadas se consigue gran precisión.

Inconvenientes:

- Localización no pasiva y no continua.
- Reconocimiento de marcas no trivial.
- Deja de funcionar si se interponen obstáculos entre la cámara y las marcas.

3.6.3. Localización mediante el Cálculo del Punto Central a partir de nodos predefinidos

Consiste en la creación de una base de datos de puntos predefinidos, que pueden ser antenas de telefonía, puntos de acceso wifi, dispositivos bluetooth, etc. Estos nodos tienen asociadas en la base de datos sus coordenadas reales. De esta forma, el dispositivo móvil comprueba periódicamente cuántos de estos puntos es capaz de detectar, y a partir de los nodos detectados infiere su posición mediante el cálculo del punto central a todos ellos, también conocido como *Centroid* ó *Centro Geométrico*.

Este método no debe confundirse con el método de inferencia de la posición mediante la fuerza de la señal recibida para cada punto de acceso, que se comenta en los apartados 3.6.7 y 3.6.8. La diferencia fundamental radica en que mediante este método no es necesaria la elaboración de mapas de radio, aunque la precisión disminuye de forma drástica.

Se pueden aplicar diversas mejoras sobre esta tecnología, tales como el uso de información estadística acerca del movimiento del dispositivo para calcular la posición en base a la probabilidad de que este se encuentre en un punto determinado.

Existen diversos proyectos que utilizan esta tecnología en conjunción con bases de datos de puntos de acceso obtenidos por "war driving"que permiten localizar un móvil con una precisión aceptable sin la necesidad de usar GPS. El ejemplo más claro es *Placelab*[18], que se analiza en la sección 3.5.1. Sin embargo, esta tecnología no ofreció los resultados que esperabamos en interiores de edificios por varios motivos.

En primer lugar, el hecho de que el mecanismo de inferencia se base sólamente en la detección o no detección de un determinado punto de acceso limita en gran medida la precisión que es posible obtener. Esta puede ser suficiente para localizar un móvil en exteriores de manera aproximada, pero no para obtener un posición en el interior de un edificio.

Además, cambios en la detección de los puntos de acceso, que pueden fluctuar mucho dentro de un edificio para una misma posición (especialmente en redes wifi), producen grandes cambios en la predicción. Este problema se ilustra en la figura 3.10, en la cual se aprecian los cambios producidos en la señal de diferentes puntos de acceso Wifi a lo largo del tiempo para un mismo punto.

Por tanto esta tecnología se mostró inadecuada para las necesidades de nuestro proyecto.

Ventajas:

- No requiere hardware externo.
- Precisión aceptable en exteriores.

Inconvenientes:

- Precisión insuficiente para interiores. Si el alcance a cada emisor es muy grande la precisión es muy baja incluso en exteriores.
- Inestabilidad.
- Necesita reconocimiento previo de los nodos.

[18] `http://ils.intel-research.net/place-lab`

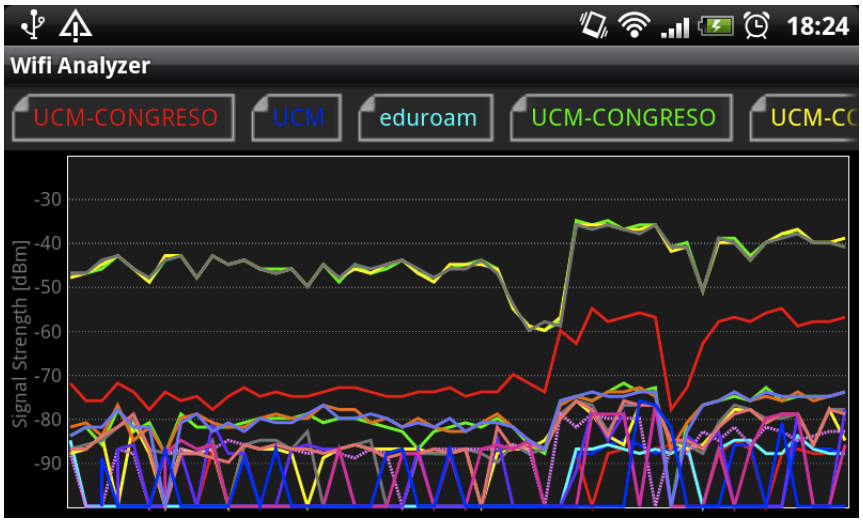

Figura 3.10: Fluctuación de la señal en redes WiFi

3.6.4. Localización mediante Triangulación de la Señal

Existen dos métodos de triangulación para sistemas basados en ondas de radio:

- **Triangulación por tiempo de llegada (TOA)**: consiste en calcular el tiempo que tarda una onda desde que sale del dispositivo fuente hasta que llega al dispositivo destino. En base a dicha información, y utilizando al menos tres nodos fuente, se infiere la posición en que se encuentra el dispositivo móvil.

- **Triangulación por intensidad de señal (RSS)**: consiste en calcular la fuerza de la señal que llega de al menos tres dispositivos fuente distintos. En base a dicha información, se infiere la posición del dispositivo

Ninguno de estos métodos es aplicable a las ondas wifi, ya que dichas ondas rebotan en el interior, de un edificio, produciendo reflexiones y refracciones que alteran el tiempo de llegada y la intensidad de la señal. Además surge el problema de la propagación multitrayecto, ya que las ondas pueden llegar al dispositivo destino por multiples caminos, produciendo tiempos de llegada y atenuaciones distintos.

Es posible aplicar estos métodos a las redes de telefonía, pero debido a la lejanía que hay entre las diferentes antenas de telefonía, la precisión que se conseguiría sería muy baja y en cualquier caso insuficiente para la localización en interiores.

Se han estudiado aplicaciones de estos métodos para ondas de baja frecuencia. [19] Sin embargo, esto hace necesaria la disponibilidad de hardware externo

[19] En el documento *Application of Channel Modeling for Indoor Localization Using TOA and RSS*, Worcester Polytechnic Institute (`http://www.wpi.edu/`) se realiza un estudio sobre este tipo de sistemas y se consiguen resultados aceptables, del orden de 4 metros de precisión, mediante la combinación de técnicas de Ray Tracing y el uso de sistemas de baja frecuencia de 500 MHz.

específico, y resulta poco versátil e inadecuado para los fines que persigue nuestro proyecto, por lo que este método fue finalmente descartado.

Ventajas:

- Puede conseguir buenas precisiones.

Inconvenientes:

- No aplicable con el hardware disponible.
- Implantación costosa.

3.6.5. Localización mediante Sistemas Inerciales

Este método consiste en la utilización de sistemas inerciales tales como acelerómetros[20] y giroscopios[21] para el cálculo del movimiento relativo de un dispositivo móvil en el espacio.

Para ello, se procesan todos los eventos de aceleración detectados por el sensor en cada uno de los ejes cartesianos (X, Y, Z). Es necesario eliminar el efecto de la gravedad en cada uno de los ejes para obtener las aceleraciones absolutas en cada instante. Para ello, se utiliza un filtro que se encarga de calcular la fuerza de la gravedad en cada eje y restarla a la aceleración detectada.

Una vez hecho esto, se integra la aceleración respecto al tiempo para obtener la velocidad, y se vuelve a integrar la velocidad con respecto al tiempo para obtener el desplazamiento. Este desplazamiento resultante es el desplazamiento relativo en ese instante en cada uno de los ejes.

Mediante el cálculo de los desplazamientos relativos, teóricamente, es posible obtener la posición de un móvil en cada instante. Sin embargo, en función de la precisión del acelerómetro se producirán mayores o menores errores entre cómputo y cómputo que determinarán el tiempo durante el cual el cálculo es fiable. A partir de cierto tiempo es necesario resetear la posición para evitar grandes desfases.

Hemos realizado diversos prototipos de sistemas de navegación inercial, tanto para el mando de la consola Nintendo Wii como para móviles Android, y en todos los casos la precisión del acelerómetro resultó ser muy baja y los errores acumulados se volvieron inasumibles, ya que estos errores descompensaban las fuerzas de aceleración y deceleración volviendo el sistema rápidamente inestable.[22]

Sin embargo, creemos que con acelerómetros de mayor precisión[23] es posible obtener resultados aceptables mediante esta técnica, siempre y cuando sea combinada con otras técnicas de localización para poder resetear la posición cada cierto tiempo y evitar grandes errores acumulativos.[24]

Ventajas:

[20] Instrumento destinado a medir aceleraciones.

[21] Intrumento utilizado para medir la orientación.

[22] Estos prototipos se especifican en mayor detalle en el Anexo A.

[23] Se puede encontrar más información acerca de la precisión de los acelerómetros en la sección A.5.

[24] Se trata de utilizar una posición obtenida mediante otras técnicas de localización como base, y entre intervalos de actualización utilizar el sistema inercial para calcular la posición relativa en base a dicho punto.

- Puede ser un buen complemento a otras técnicas de localización.

Inconvenientes:

- Necesidad de hardware de gran precisión.
- Error acumulativo.
- Insuficiente como único método de localización.

3.6.6. Localización mediante Detección de Movimiento

Esta técnica consiste en detectar, mediante el uso de un acelerómetro, cuándo la persona que porta el dispositivo está caminando, para posteriormente aplicar una velocidad de paso estándar en la dirección correcta. [25]

Para ello, se mantiene una lista con las últimas N mediciones del acelerómetro, y se calcula su desvación estándar. Se trata de buscar una desviación estándar por encima de cierto umbral para detectar movimiento continuado. Si la persona está en movimiento se producirán variaciones contínuas en la aceleración que aumentarán la desviación estándar.

Una vez se ha detectado si la persona está o no caminando, se utiliza la brújula del dispositivo para calcular la dirección de la persona con respecto al Norte magnético de la Tierra, y se aplica una velocidad de paso estándar en dicha dirección.

Esta técnica no debe confundirse con los sistemas inerciales descritos en la sección 3.6.5. La diferencia radica en que la primera técnica trata de obtener los valores exactos de desplazamiento relativo por medio de la aceleración, mientras que está técnica trata de aproximar el desplazamiento por medio de una velocidad estándar cuando se detecta que la persona está en movimiento.

Hemos desarrollado diversos prototipos. y esta técnica ha resultado en la práctica comportarse mucho mejor que la anteriormente descrita, detectando con bastante precisión cuándo una persona camina. A pesar de ello, sigue habiendo error acumulado tanto por la velocidad de paso aplicada, que no deja de ser una aproximación, como por los falsos positivos que pudieran darse al detectar el movimiento de la persona, por lo que sigue siendo necesario un método de localización auxiliar que permita resetear la posición cada cierto tiempo.

Este método ha sido implementado e incluido en el proyecto en conjunción con otros ya que ha sido el sistema de navegación inercial que mejor se ha comportado en la práctica. En la sección 5.2 se describe en mayor detalle la implementación llevada a cabo de este sistema.

Ventajas:

- Efectivo como método auxiliar de localización.
- Aporta precisión extra.
- Hardware necesario disponible en teléfonos móviles actuales.

Inconvenientes:

[25] Con velocidad de paso se hace referencia a una velocidad estándar a la que camina un ser humano, estimada en 5 Km / hora. La dirección se calcula por medio de un giroscopio, que puede utilizarse para emular el funcionamiento de una brújula.

- Error acumulativo, requiere ser reseteado periódicamente.
- Insuficiente como único método de localización.

3.6.7. Localización mediante Mapas de Radio WiFi

Este método consiste en la obtención, a partir de un proceso de captura de datos, de un mapa de radio (radiomap) de las ondas wifi de diferentes puntos de acceso en diferentes nodos previamente definidos. Este mapa de radio es utilizado posteriormente para inferir la posición del dispositivo mediante distintos tipos de algoritmos.

El primer paso consiste en obtener una lista de los puntos wifi que se utilizarán en la creación del mapa de radio. Estos pueden ser definidos por SSID o por dirección MAC de los mismos.

Una vez obtenida la lista de puntos wifi, se define una serie de nodos y se llevan a cabo capturas de datos en cada nodo. En estas capturas se guarda información de los niveles de señal de cada uno de los puntos de acceso en ese punto. La información obtenida en las capturas conforma el mapa de radio. [26]

Obtenido el mapa de radio, la posición del dispositivo se infiere mediante la captura de datos y búsqueda del nodo con mayor similitud. Esto puede hacerse con distintos tipos de algoritmos.

Mediante este método se han obtenido los mejores resultados, llegando a conseguir con bastante éxito localización a nivel de habitación. En posteriores capítulos se estudia con mayor detalle la precisión que es posible conseguirse con este método de localización variando distintos parámetros.

Ventajas:

- El mecanismo de entrenamiento asegura una buena precisión.
- Hardware necesario disponible en teléfonos móviles actuales.

Inconvenientes:

- Se requiere captura previa de datos.
- Los datos obtenidos para un dispositivo pueden no ser válidos para otro.
- Cambios en la estructura del edificio, de mobiliario, etc, pueden invalidar los datos de entrenamiento.

A continuación de presentan los algoritmos utilizados en este proyecto para la inferencia de la posición en base a los mapas de radio.

Closest Neighbor

Consiste en la búsqueda del nodo con máxima similitud a los datos obtenidos en cada medición. Para ello se calcula la distancia ecuclídea de cada nodo con respecto a los datos obtenidos, y se devuelve el nodo cuya distancia euclídea es menor.

Este algoritmo ha resultado comportarse realmente bien en la inferencia de la posición a pesar de su simpleza.

[26] También conocido como *fingerprint*.

Redes Neuronales

También se han empleado redes neuronales para la inferencia de la posición con respecto a los datos obtenidos. Se trata de utilizar la red para un problema de reconocimiento de patrones. Para ello en primer lugar es necesario de definir el tipo y la arquitectura de la red. En las pruebas se ha utilizado una red neuronal Perceptrón Multicapa, con propagación hacia atrás y con diversas arquitecturas y número de neuronas. Se ha utilizado una codificación en código Gray para las entradas y una salida para cada uno de los puntos de acceso empleados en la captura de datos, de forma que la salida para cada nodo se encuentra entre 0 y 1 en función de su similitud a los datos de entrada.

Antes de proceder a la localización es necesario entrenar la red neuronal con las datos obtenidos para cada nodo. Una vez alcanzado un error en el entrenamiento por debajo de cierto umbral, la red queda entrenada y lsta para recibir nuevos datos de entrada.

Los resultados con este método dependen en gran medida de los parámetros de la red tales como la arquitectura y el número de neuronas utilizadas, y ha permitido obtener resultados aceptables aunque creemos que aún mejorables con un estudio más detallado de diferentes tipos de arquitectura para resolver este problema.

3.6.8. Localización mediante Mapas de Radio obtenidos Automáticamente

Esta técnica consiste en la obtención del mapa de radio mediante el procesado de mapas en lugar de en la captura de datos, ya que dicho proceso puede llegar a ser muy costoso.

Para ello, se emplean técnicas basadas en ray tracing para el modelado del comportamiento de las ondas en el interior del edificio, sus reflexiones y refracciones. Para ello, se carga un mapa a escala del edificio y se fijan los puntos de acceso y el nivel de atenuación que producen los muros. Con estos datos, se lanzan rayos en todas direcciones para cada uno de los puntos de acceso y se trata de calcular de manera aproximada cómo se comportarán las ondas. Con esta información se crean los mapas de radio para ser utilizados en la inferencia de la posición mediante alguno de los algoritmos descritos anteriormente.

Diversas investigaciones, como la llevada a cabo por el *Worcester Polytechnic Institute de Massachusetts* [27], arrojan unos resultados prometedores mediante esta técnica. Sin embargo, nosotros decidimos descartarla debido a que es necesaria información acerca de la atenuación de muros, localización exacta de los puntos de acceso y mapas de calidad a escala.

Ventajas:

- Tiene las ventajas de las técnicas basadas en mapas de radio, pero elimina la necesidad de realizar capturas de datos previas, que pueden ser muy costosas.

Inconvenientes:

[27] `http://www.wpi.edu/`.

- Al realizarse un modelado, los datos obtenidos son aproximados y pueden ser incorrectos en determinadas zonas del edificio.
- Requiere información de la atenuación de los muros, colocación exacta de los puntos de acceso y mapas a escala de calidad.

3.7. Conclusiones

Ninguno de los framework de realidad aumentada analizados satisface nuestros requisitos, bien porque su funcionalidad es demasiado simple, ó bien porque son cerrados y presentan limitaciones: no disponibilidad del código fuente ni posibilidad de extensión. Por tanto se ha tomado la decisión de hacer un desarrollo desde cero para cubrir el vacío existente de frameworks de realidad aumentada de código abierto y con capacidad de extensión.

Se ha seleccionado el sistema operativo *Android* para el desarrollo debido a que es un sistema operativo móvil abierto y en expansión, que nos permite trabajar en Java y para el que tenemos acceso a dispositivos.

Se ha decidido utilizar OpenGL ES 1.1 debido a su mayor compatibilidad con dispositivos, sencillez y rendimiento suficiente para nuestras necesidades.

A la hora de incorporar un servicio web al framework, se ha elegido RESTful porque es la arquitectura recomendada por Google para su integración en Android y está mejor documentada que el resto de tecnologías.

Ninguno de los proyectos de localización en interiores analizados nos permite su reutilización debido a que no cubren nuestras necesidades de precisión y capacidad de integración en Android. De entre las técnicas de localización analizadas se ha seleccionado una técnica híbrida basada en las siguientes: *Localización mediante Detección de Movimiento* (sección 3.6.6) y *Localización mediante Mapas de Radio WiFi* (sección 3.6.7).

Capítulo 4

Arquitectura Global

La arquitectura global proporciona una visión general de cómo está construido el framework, qué módulos lo conforman y cómo se encuentran organizados dichos módulos.

La construcción de aplicaciones mediante el framework se puede dividir en dos capas: la capa del framework y la capa de aplicación, que se construye sobre la del framework (figura 4.1).

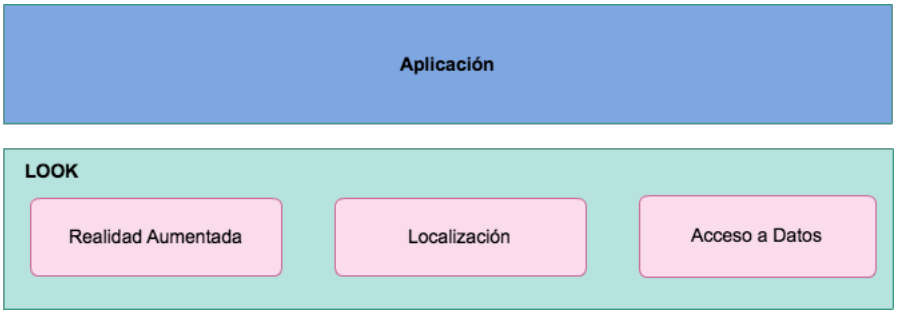

Figura 4.1: Capas del Desarrollo de Aplicaciones mediante Look

La capa del framework está formada por los siguientes módulos: *Módulo de Localización*, *Módulo de Datos* y *Módulo de Realidad Aumentada*. Cada uno de ellos se describe en este capítulo de manera general en su sección correspondiente [1]. Finalmente se dedica una sección a la integración de los módulos para la creación de aplicaciones.

4.1. Módulo de Localización

El módulo de localización es el encargado de proporcionar la funcionalidad relativa al sistema de localización. Está formado por los siguientes componentes (Figura 4.2) :

- **Localización Wifi**: Encargado de proporcionar un sistema de localización primario mediante wifi.

[1] Cada uno de los módulos se describe más detalle en su capítulo correspondiente.

- **Navegación Inercial**: Proporciona un sistema secundario de localización mediante un sistema de navegación inercial, con el fin de aumentar la precisión del sistema de localización primario, o de utilizarlo como sistema de localización independiente.
- **Módulo de localización**: Encargado de integrar los sistemas de localización primario y secundario.

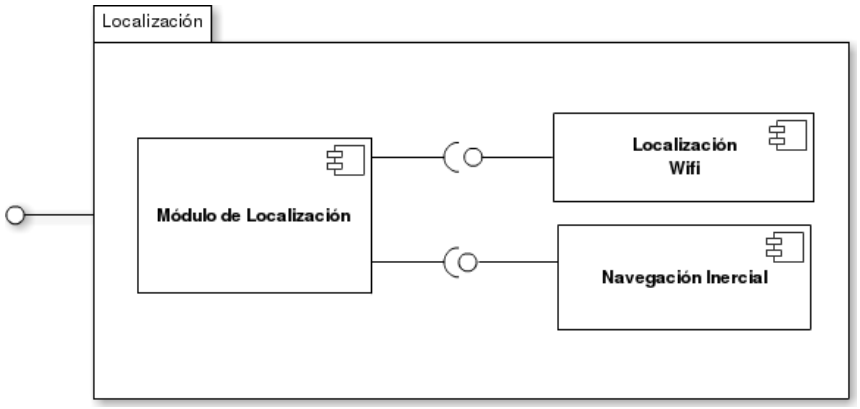

Figura 4.2: Diagrama de Componentes de Localización

La salida obtenida desde el módulo de localización es **la posición del dispositivo**, con respecto al sistema de referencia definido (se verá con más detalle en el capítulo 5). Esta localización será recogida y procesada por el Módulo de Datos.

4.2. Módulo de Datos

El Módulo de Datos es el núcleo de todas las aplicación construidas con *Look!*. Tiene dos funcionalidades muy diferenciadas: la primera, obtiene los datos de una serie de recursos definidos, locales o remotos; la segunda, procesa los datos obtenidos y provee de elementos representables por el Módulo de Realidad Aumentada.

En la figura 4.3 se muestra una visión conjunta del Módulo de Datos con los proveedores de contenido. Puede configurarse, a través de interfaces comunes, de dónde proceden los datos. Pueden estar definidos de manera programática, obtenerse de archivos de recursos (por ejemplo, en formato XML), de una base de datos local o de una base de datos remota.

El Módulo de Datos puede tener acoplado el Módulo de Localización, como una fuente de obtención de datos más.

El Módulo de Datos y los distintos proveedores de datos se tratarán con mayor detalle en el capítulo 6.

4.3. Módulo de Realidad Aumentada

El módulo de Realidad Aumentada contiene elementos con capacidades de representación gráfica que serán utilizados para la creación de la interfaz de

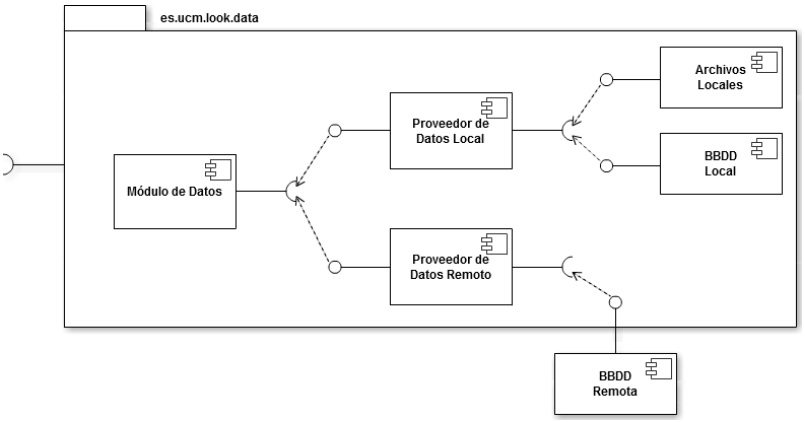

Figura 4.3: Diagrama de Componentes del módulo de datos

usuario.

Existe un módulo base al que, opcionalmente, pueden ser añadidos cada uno de los módulos de representación individuales que aparecen en la figura 4.4:

- **Cámara**: Módulo que ofrece la visión de la cámara.
- **Visualización 3D**: Módulo para la visualización de elementos animados en tres dimensiones.
- **Visualización 2D**: Módulo para la visualización de elementos animados en dos dimensiones.
- **HUD**: Módulo para albergar elementos adicionales a la interfaz de usuario, como botones, cajas de texto o otras *View* ofrecidas por Android.

El módulo de realidad aumentada debe ser conectado a un módulo de datos que provea de los elementos (*WorldEntity*) que han de ser representados de manera gráfica por la aplicación.

Se hará una descripción en detalle de todos los módulos de capas en el capítulo 7.

4.4. Uniendo los módulos para la creación de aplicaciones

Cada uno de los módulos presentados hasta ahora representa una funcionalidad aislada. La combinación de los diferentes módulos, dependientes de la aplicación a desarrollar, darán lugar a las aplicaciones.

En las figura 4.5 se muestra la arquitectura global (en términos de componentes) de las aplicaciones creadas con *Look!*.

Como base, siempre tendremos el **módulo de datos**. Éste módulo utilizará el **módulo de Localización** para actualizar la posición del usuario, con un tasa de refresco que puede ser configurada.

Por último, el **módulo de realidad aumentada**, accederá al módulo de datos, y representará los elementos obtenidos.

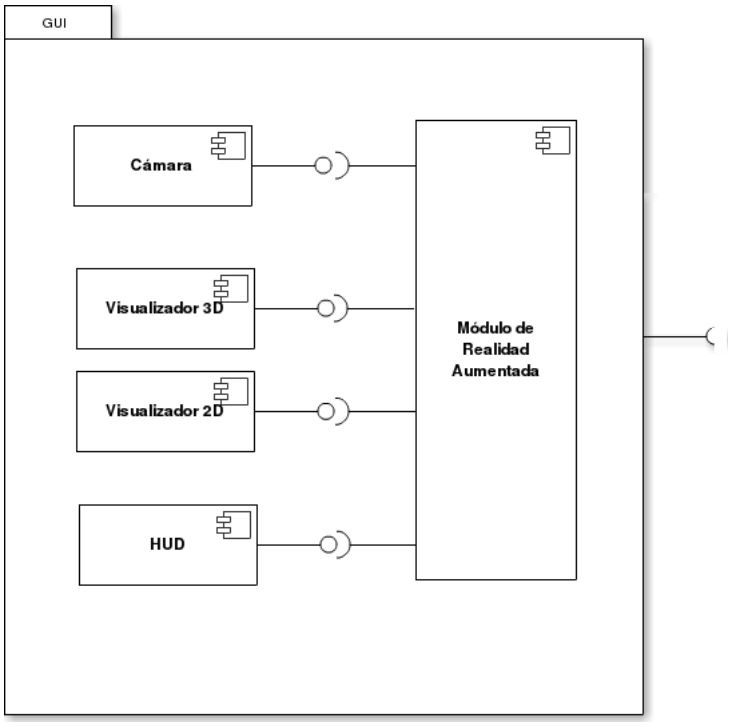

Figura 4.4: Diagrama de Componentes de la Interfaz de Realidad Aumentada

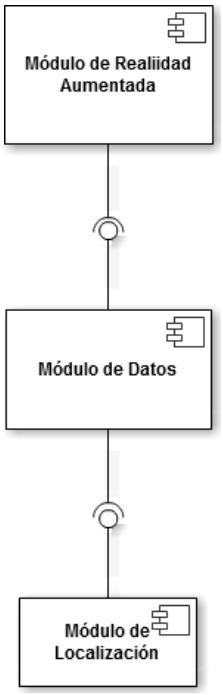

Figura 4.5: Diagrama de componentes general para aplicaciones construidas con *Look!*

Capítulo 5

Módulo de localización

Uno de los pilares básicos del framework que hemos desarrollado es la capacidad de obtener la localización de un dispositivo en el interior de un edificio. Dada la dificultad del problema, se ha invertido gran esfuerzo y tiempo en la investigación de posibles técnicas para llevar a cabo esta tarea. Una vez adquirida cierta experiencia, y descartada la reutilización de sistemas de localización ya existentes, hemos procedido a diseñar un sistema propio de localización de acuerdo a nuestras necesidades. Este sistema se basa en la combinación de una técnica primaria de localización mediante wifi (explicada en la sección 3.6.7) y una técnica secundaria de localización mediante un sistema de navegación inercial (explicado en la sección 3.6.6). Asimismo, para el funcionamiento de dichos sistemas, es necesario conocer la orientación del dispositivo.

En esta sección se explica en primer lugar cómo se obtiene la orientación de un dispositivo en Android. Seguidamente se describen los subsistemas de localización mediante Navegación Inercial y mediante Wifi que se han implementado en el framework *Look!*. Por último, se detalla cómo se integran ambos subsistemas en un único API de localización accesible de manera global.

5.1. Orientación del móvil en Android

Para lograr aplicaciones en realidad aumentada creíbles es necesario conocer la orientación del dispositivo móvil para poder colocar los elementos de una manera lógica. En esta sección se explica cómo trata Android la orientación en sus dispositivos, los pasos esenciales para comprender su funcionamiento y su aplicación.

5.1.1. *Pitch*, *azimuth* y *roll*

Android define tres rotaciones para el dispositivo: *pitch*, *azimuth* y *roll*, que se definen como las rotaciones en los ejes x, y y z respecto a la posición de reposo, como muestra la figura.

La posición de reposo del dispositivo se establece con el mismo formando un ángulo perpendicular respecto a la fuerza de gravedad, y señalando con su parte superior al norte.

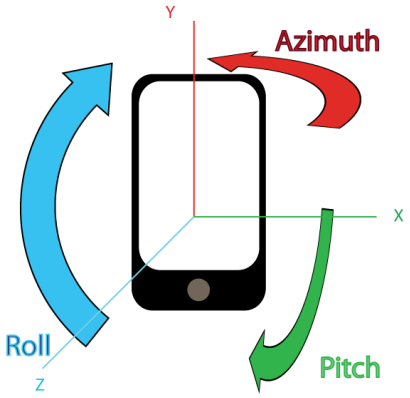

Figura 5.1: Definición de *pitch*, *azimuth* y *roll*

5.1.2. Obteniendo la orientación: *DeviceOrientation*

La clase *DeviceOrientation* implementa el patrón de diseño *Singleton* y proporciona en todo momento la actual orientación, definida por los tres parámetros presentados en la sección anterior, del dispositivo.

Para su cálculo, utiliza dos sensores incorporados en todos los dispositivos Android: el acelerómetro y el medidor de campo magnético. Android proporciona métodos que, a partir de estos datos, calculan la matriz de rotación del dispositivo, a partir de la cuál, se pueden obtener los tres parámetros buscados.

DeviceOrientation ofrece al framework estos tres parámetros y la matriz de rotación, que puede ser aplicada en algunos casos concretos de transformaciones 3D en *Open GL*.

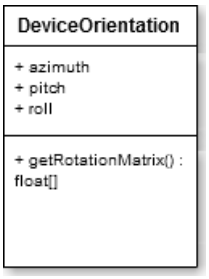

Figura 5.2: Clase *DeviceOrientation*

5.2. Subsistema de Navegación Inercial

En esta sección se decribe el subsistema de navegación inercial que se utiliza en el módulo de localización. El sistema desarrollado es fruto de diversos experimentos realizados en los que se trata de utilizar acelerómetros para el desarrollo

de un sistema de posicionamiento en interiores. Su objetivo es estimar el movimiento relativo del dispositivo en base a una posición de partida. Este trabajo está basado en el sistema de navegación inercial descrito en el proyecto *Indoor Navegation System for Handheld Devices.*

5.2.1. Diseño

La idea consiste en detectar cuándo el usuario está caminando y aplicar una velocidad estándar de paso en la dirección adecuada. Para ello se necesita obtener información de diversos sensores del dispositivo:

- Acelerómetro para detectar aceleraciones y movimiento.
- Giroscopio para detectar la dirección.

Para detectar cuándo el usuario está o no en movimiento, en primer lugar se guarda una cola con los últimos N eventos de aceleración recibidos por el sensor (acelerómetro). Estos eventos se reciben a una tasa de muestreo previamente fijada. Cada vez que se añade un nuevo evento de aceleración a la cola, en primer lugar se comprueba si esta está llena y en caso afirmativo se desecha el elemento más antiguo.

Una vez hecho esto, se calcula la desviación estándar sobre la magnitud total de aceleración de todos los elementos de la cola y se comprueba si esta cae por encima de un determinado umbral. Para calcular la magnitud total de cada evento de aceleración se calcula la raíz de la suma de los cuadrados de cada uno de los ejes de coordenadas (X,Y,Z):

$$M = \sqrt{\sum_{i=1}^{N} v[i]^2}$$

Si la desviación estándar está por encima del umbral significa que el dispositivo está en movimiento, en caso contrario es que el dispositivo no está en movimiento.

De manera análoga, a intervalos de tiempo fijados se actualiza la información de la dirección del dispositivo a partir de los sensores disponibles. En caso de que el dispositivo se encuentre en movimiento, se aplica una velocidad estándar de paso en la dirección correspondiente proporcional al tiempo transcurrido desde el último evento de aceleración.

5.2.2. Arquitectura

En esta sección se modela la arquitectura y el comportamiento del sistema para su posterior implementación.

Diagrama de Clases

A continuación se describen las principales clases que conforman el subsistema de navegación inercial, representadas en la figura 5.4:

- La clase *LocationProvider* es la encargada de centralizar la funcionalidad del subsistema de navegación inercial, registrando los listener para los distintos tipos de eventos y procesándolos según sea necesario.

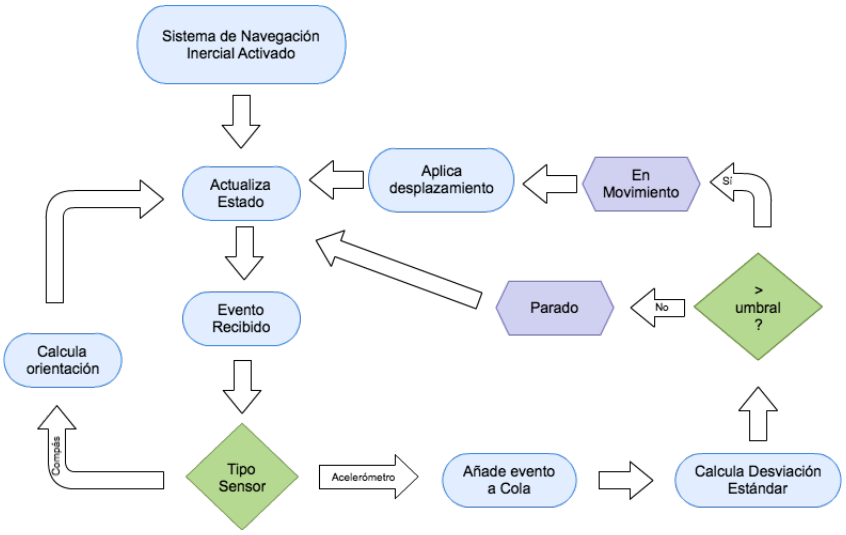

Figura 5.3: Diagrama de Flujo: Sistema de Navegacion Inercial

- *DeviceSensor* se encarga de aplicar el algoritmo de detección de movimiento.
- *Positioning* es la clase que mantiene la información de la posición, el desplazamiento relativo y la detección/no detección de movimiento.
- *InertialNavigationSystem* se encarga de realizar los cálculos relativos al desplazamiento producido.
- *Motion* representa la información de cada uno de los desplazamientos relativos.

Diagrama de Secuencia

A continuación se describe el comportamiento del sistema, representado en la figura 5.5, desde que se recibe un nuevo evento de aceleración hasta que el usuario lee la nueva posición.

1. La aplicación crea un nuevo *LocationProvider*, que registra listeners sobre los sensores y permanece a la espera de nuevos eventos. Además, inicializa el sistema de navegación inercial.
2. Se recibe un nuevo evento de aceleración y se notifica tanto a *Positioning* como a *DeviceSensor*.
3. *DeviceSensor* aplica el algoritmo de detección de movimiento. Si se detecta movimiento *Positioning* envía un mensaje a *InertialNavigationSystem* para que procese el nuevo evento desplazamiento relativo producido.
4. La aplicación llama a process para calcular la nueva posición en base a los desplazamientos relativos y obtiene la posición final.

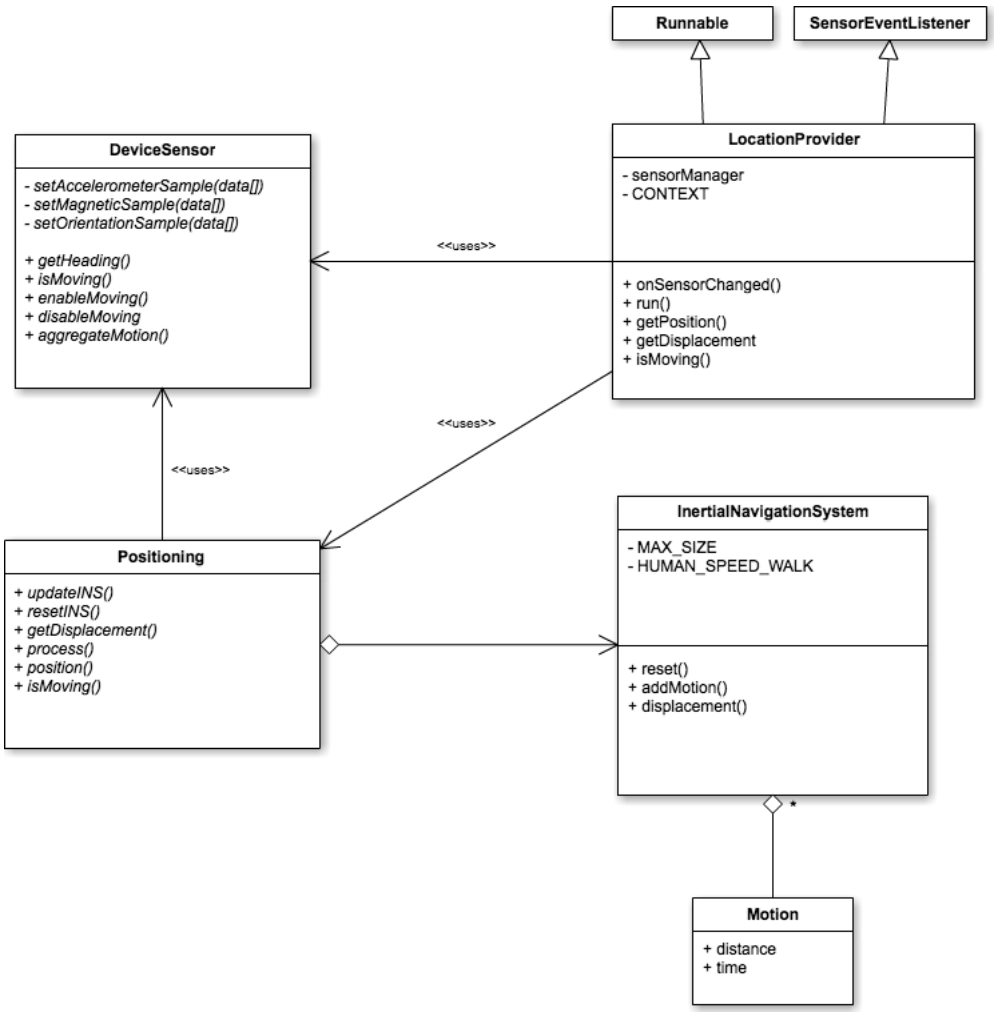

Figura 5.4: Diagrama de Clases INS

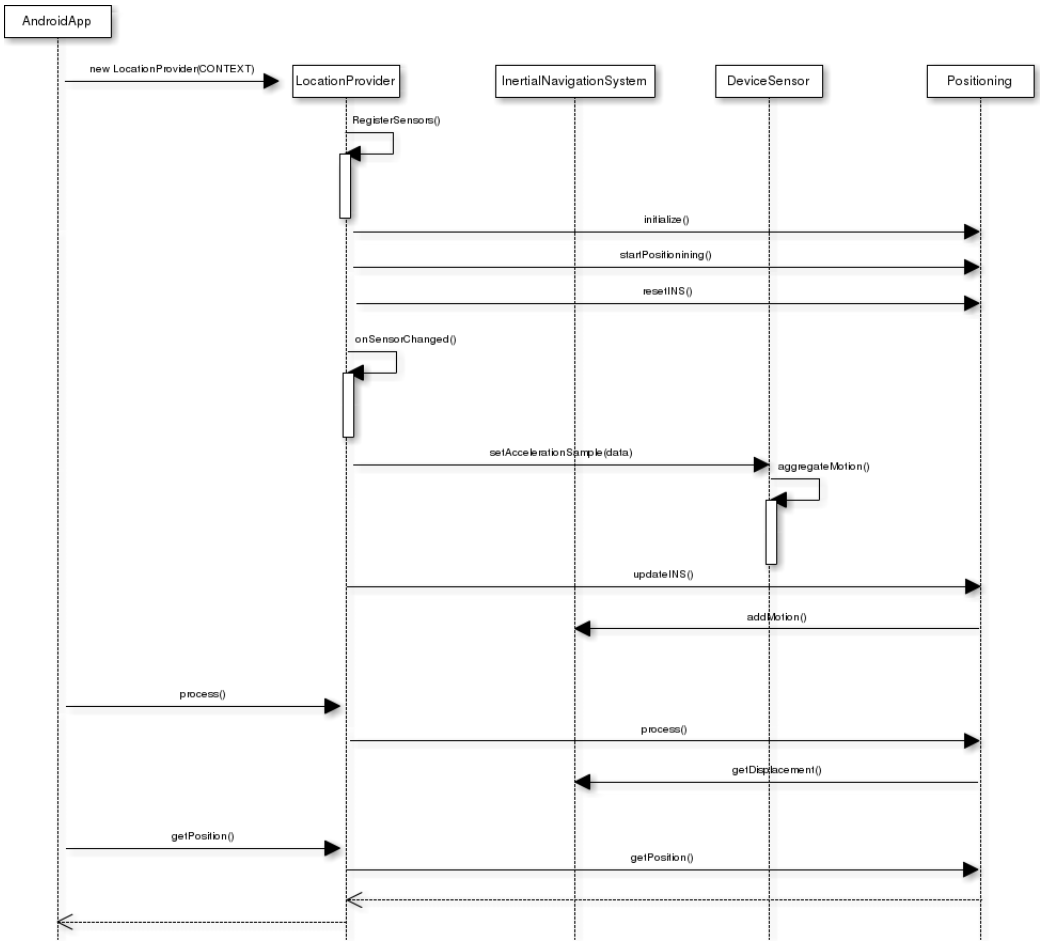

Figura 5.5: Diagrama de secuencia INS

5.2.3. Implementación

La implementación se ha realizado para un sistemas Android en su versión 2.2 o posterior. Se han dijado los siguientes parámetros de funcionamiento:

- HUMAN SPEED WALK: 5 Km / hora.
- Frecuencia de muestreo: SENSOR DELAY FASTEST.

El acceso a los sensores se realiza a través de la clase *SensorManager* del API de Android.

```
mSensorManager = (SensorManager) CONTEXT
        .getSystemService(Context.SENSOR_SERVICE);
Sensor asensor = mSensorManager
        .getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
Sensor msensor = mSensorManager
        .getDefaultSensor(Sensor.TYPE_MAGNETIC_FIELD);
Sensor osensor = mSensorManager
        .getDefaultSensor(Sensor.TYPE_ORIENTATION);

mSensorManager.registerListener(this, asensor,
        SensorManager.SENSOR_DELAY_FASTEST);
mSensorManager.registerListener(this, msensor,
        SensorManager.SENSOR_DELAY_FASTEST);
mSensorManager.registerListener(this, osensor,
        SensorManager.SENSOR_DELAY_FASTEST);
```

A pesar de que a partir versiones de la API de Android posteriores a la 1.5 se provee acceso directo al giroscopio, y en caso de no estar disponible se emula de manera automática mediante el acelerómetro, al estar basado en un proyecto desarrollado para la versión 1.5 calculamos la orientación a partir de los sensores de orientación y campo magnético de forma manual. A partir de la versión 2.3 de Android el API de acceso a los sensores ha mejorado considerablemente, realizando de manera automática funciones tales como la eliminación del efecto de la gravedad de la aceleración.

Para probar este subsistema, lo hemos integrado con una aplicación desarrollada mediante nuestro framework que utiliza la información de navegación inercial para simular un mundo virtual. Al movernos en el mundo real, nos movemos también en el mundo virtual. Los sensores de orientación permiten además saber hacia donde se está mirando, de forma que la experiencia de simulación es completa.

En la figura 5.6 se muestra una captura de dicha aplicación. En ella, se observa el mundo 3D y objetos con los que es posible interactuar. Cuando la persona que porta el dispositivo gira sobre si misma, también cambia la perspectiva en el mundo virtual, y si empieza a andar se detectará dicha situación y se avanzará en el mundo virtual.

5.3. Subsistema de Localización por WiFi

El sistema de localización mediante Wifi que se ha desarrollado consta de varias fases: una fase de planificación, una fase de captura de datos y una fase

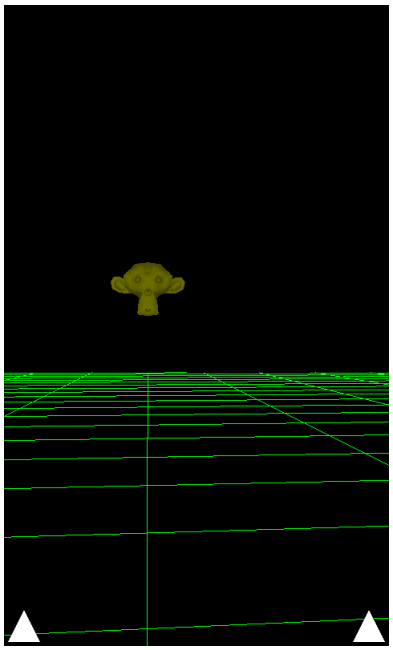

Figura 5.6: Aplicación de prueba para el subsistema de Navegación Inercial

de localización.

En esta sección se describen los requisitos, el diseño y la arquitectura de dicho sistema de localización y se explica en detalle cada una de las fases que lo componen.

5.3.1. Requisitos

Funcionales

- Se debe proporcionar un servicio autónomo que provea los siguientes métodos:
  - **Start**: Arrancar el servicio de localización.
  - **Stop**: Parar el servicio.
  - **getPosicion**: Acceso a la posición en coordenadas (X,Y,Z), correspondiente Z a la planta del edificio.
- Se deber implementar un sistema de logs. Debe ser posible seleccionar un archivo de log como fuente de datos para la realización de pruebas.

No Funcionales

- Precisiones a nivel de habitación pueden resultar suficientes para algunos tipos de aplicaciones.
- Tiempo de actualización de la posición no superior a 10 segundos.
- Hardware disponible en smartphones modernos.

5.3.2. Diseño

Se ha buscado un diseño que maximice la precisión, aunque ello significe sacrificar la facilidad de implantación. Para ello, el sistema consta de los siguientes módulos:

- Un primer módulo de **planificación** en el cual, dado el mapa de un edificio, se seleccionan nodos en posiciones relevantes, se enumeran y se etiquetan. El sistema tratará posteriormente de aproximar la posición actual al nodo más cercano de la lista de nodos definidos. Además, es necesario decidir qué puntos de acceso se utilizarán en la localización. Deben ser puntos de acceso estables, a ser posible propios, ya que la desconexión de alguno de ellos puede producir que el sistema de localización deje de funcionar.

- Un módulo de **captura de datos**, en el cual para cada uno de los nodos definidos, se registra la fuerza de la señal recibida de cada uno de los puntos de acceso. El resultado es un mapa de la cobertura wifi para cada nodo, conocido como mapa de radio o *fingerprint*. Dicha información se almacena en un fichero para su uso posterior. Para eliminar los efectos del ruido y estabilizar la señal recibida, se muestrea durante varios segundos y se compactan los datos.

- Un módulo de **localización**. En este módulo, el dispositivo realiza escaneos de redes wifi a intervalos de tiempo definidos y contrasta la información detectada con la almacenada en los mapas de radio. La inferencia del nodo más factible se lleva a cabo utilizando diversos algortitmos que se describen más adelante. Para eliminar los efectos del ruido y estabilizar la señal recibida, al igual que en la fase de captura de datos, se muestrea durante varios segundos y se compactan los datos antes de hacer la comparación. El resultado corresponde a la información del nodo más parecido a la posición actual.

En las siguientes subsecciones se describe en mayor detalle cada uno de los módulos.

En la figura 5.7 se representa la interacción entre los módulos desde que desde que inicia la planificación hasta que se infiere una posición.

Planificación

En primer lugar se seleccionan los nodos que se utilizarán en el sistema de localización. Conviene que los nodos se encuentren en habitaciones separadas, ya que la atenuación que provocan los muros permite diferenciar suficientemente las señales de forma que el sistema de localización se comporte de manera óptima.

Para cada uno de los nodos elegidos se debe recopilar la siguiente información: [1]

- **Coordenadas (X,Y,Z)**, donde Z corresponde al número de planta.

- **Número de Nodo**. Todos los nodos deben estar enumerados para su fácil identificación.

[1] La información de los nodos debe recopilarse sobre un mapa y ser incluida en el fichero *Lugares.txt* de forma manual.

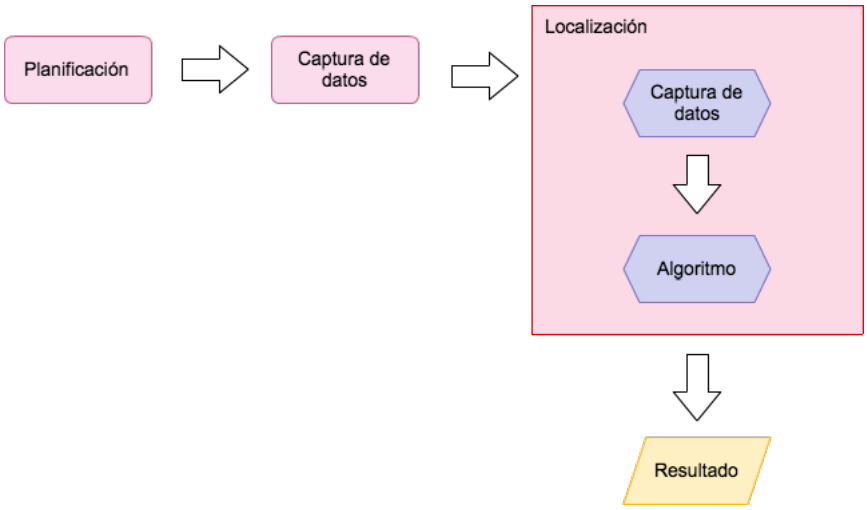

Figura 5.7: Esquema de la localizacion por WiFi

- **Etiqueta** describiendo a qué posición hace referencia el nodo dentro del edificio.

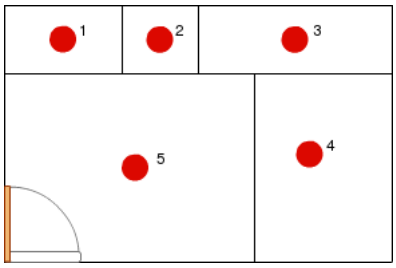

Figura 5.8: Ejemplo de seleccion de Nodos

También es necesario fijar los puntos de acceso que se utilizarán en la localización mediante una lista de direcciones MAC de los mismos. Se provee detección automática de puntos de acceso en un edificio si no se desea restringir los puntos de acceso a utilizar.

Toda esta información puede recopilarse de forma manual y debe ser incluida en los ficheros APs.txt y Lugares.txt dentro de la tarjeta SD para ser posteriormente utilizados por el resto módulos. A continuación se muestran ejemplos de cada uno de los ficheros que es necesario generar:

APs.txt

Lista de direcciones MAC de los puntos de acceso a utilizar.

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

Lugares.txt

Id, Planta, Coordenada X, Coordenada Y y Nombre respectivamente.

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

Captura de datos

En esta fase se muestrea la información wifi en cada uno de los nodos seleccionados a fin de elaborar un mapa de radio. Se pueden variar los siguientes parámetros de muestreo:

- **Tiempo de muestreo**: Indica el tiempo durante el cual se toman muestras en cada nodo. Esto se hace para evitar problemas derivados de la fluctuación de la señal y el ruido. Se intenta estabilizar la información capturado varias muestras y compactándolas en una sola.
- **Número de muestras**. Es posible capturar más de una muestra en cada nodo, de forma que al hacerse la comparación en la fase de localización existan mayores posibilidades de acierto.
- **Tasa de refresco**. Intervalo de tiempo entre cada muestra.

La compactación de muestras se realiza de la siguiente manera:

1. Se guarda la información de todos los puntos de acceso detectados en el total de muestras.
2. Si algún punto de acceso aparece más de una vez, se calcula la media de sus valores de fuerza de señal detectados.

Localización

Esta fase trata de obtener una posición aproximada a partir de los mapas de radio obtenidos en la fase de captura. Para ello, se capturan datos mediante el escaneo de redes wifi y se contrasta la información con la almacenada para cada nodo.

Los parámetros que se pueden fijar son los mismos que en la fase de entrenamiento.

- **Tiempo de muestreo**: Indica el tiempo durante el cual se toman muestras en cada nodo. Esto se hace para evitar problemas derivados de la fluctuación de la señal y el ruido. Se intenta estabilizar la información capturado varias muestras y compactándolas en una sola.
- **Número de muestras**. Es posible capturar más de una muestra en cada nodo, de forma que al hacerse la comparación en la fase de localización existan mayores posibilidades de acierto.
- **Tasa de refresco**. Intervalo de tiempo entre cada muestra.

Para obtener resultados consistentes, la localización se debe llevar a cabo con los mismos valores en los parámetros que los utilizados en la fase de captura.

Una vez compactados los datos del escaneo, se procesan mediante alguno de los algoritmos desarrollados para buscar el nodo con mayor similitud en base a los valores de fuerza de señal detectados. A continuación se describen dichos algoritmos.

Algoritmo Closest Neighbor

Consiste en la búsqueda del nodo más parecido de un conjunto de entrenamiento. Para ello se calcula la distancia euclídea entre los nodos detectados y los nodos entrenados y se selecciona aquel cuya distancia euclídea es menor.

La fórmula para calcular la distancia euclídea es la siguiente:

$$D = \sqrt{\sum_{i=1}^{N}(a_i - q_i)}$$

donde D es la distancia obtenida, a_i es la fuerza de la señal del punto de acceso i de el conjunto de entrenamiento y q_i es la fuerza de la señal del punto de acceso i de el conjunto detectado. [2]

De esta forma, el nodo seleccionado se elige de la siguiente forma:

$$NodoSel = \text{mín}(\sqrt{\sum_{i=1}^{N}(a_i - q_i)})$$

Las coordenadas del nodo seleccionado se devuelven como coordenadas actuales del dispositivo.

Redes Neuronales

Consiste en utilizar una red neuronal, entrenada mediante el conjunto de datos obtenidos en la fase de captura, para obtener el grado de similitud del conjunto de datos detectado con cada uno de los nodos del conjunto de entrenamiento. Se trata de un problema de reconocimiento de patrones. Se ha empleado una red neuronal Perceptrón Multicapa con las siguientes características:

- **Capa de entrada**: Se codifica la información de la señal de cada punto de acceso en números binarios de 7 bits. [3] Posteriormente se codifica la información binaria en código Gray (también de 7 bits) para evitar que pequeñas variaciones en la señal produzcan variaciones grandes en la codificación de la información. Por tanto habrá N * 7 neuronas de entrada en la red neuronal, donde N es el número de puntos de acceso fijados.
- **Capa Oculta**: Se ha utilizado una única capa oculta con un número de neuronas igual a 50 por limitaciones del hardware que se comentan en la sección de implementación.

[2] Los conjuntos de entrenamiento y detectado contienen el mismo número puntos de acceso gracias a que los puntos de acceso a utilizar están prefijados en la fase de planificación y almacenados en el fichero correspondiente. Los puntos de acceso no detectados en el análisis se añaden automáticamente con valor de atenuación -100dB, que corresponde a una fuerza de señal igual a 0.

[3] La información de señal está en el intervalo [-100, 0] dB. Tomando el valor absoluto, 7 bits son suficientes para representar toda la información de la señal

- **Capa de Salida**: Hay una neurona de salida por cada uno de los nodos. El valor de salida se encuentra en el intervalo [0,1] y corresponde al nivel de similitud entre los datos de entrada y los datos entrenados. Un valor de 0 equivale a ninguna similitud, y un valor de 1 similitud total.
- **Entrenamiento** mediante algoritmo de propagación hacia atrás.
- **Función de transferencia**: Sigmoid.
- **Tasa de aprendizaje**: Se ha fijado en 0.5. Valores demasiado altos pueden hacer que el algoritmo de entrenamiento se atasque en un mínimo local. Valores demasiado bajos pueden hacer muy lento el proceso.
- **Condición de parada**: Error en entrenamiento inferior a 0.01.

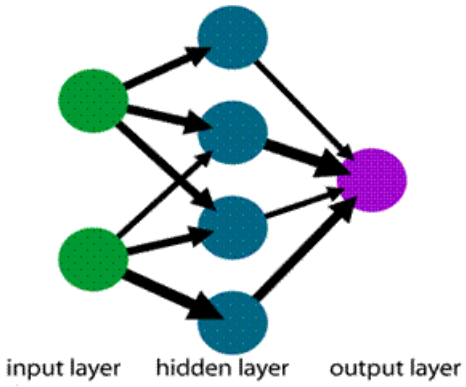

Figura 5.9: Red Neuronal Perceptron Multicapa

Para entrenar la red se crea un conjunto de entrenamiento en el que los datos de entrada son los datos obtenidos en la fase de captura y la salida esperada es 1 para la neurona que representa el nodo actual y 0 para el resto.

Una vez entrenada la red, esta puede utilizarse para la localización. Para ello, se crea un conjunto de entrada a partir de la información detectada y se introduce en la red. Para cada neurona de salida, se obtiene un valor en el intervalo [0,1] correspondiente al nivel de similitud de la información con dicho nodo.

Finalmente, se selecciona el nodo cuya similitud es mayor y se comprueba si está por encima de un umbral definido. En caso afirmativo, se devuelven las coordenadas de dicho nodo como localización actual del dispositivo.

5.3.3. Arquitectura

En esta sección se modela la arquitectura y el comportamiento del sistema para su posterior implementación.

Diagrama de Clases

A continuación se describen las principales clases que conforman el subsistema localización por WiFi representadas en la figura 5.10:

- *WifiLocation* corresponde al servicio de localización en sí. Provee métodos para arrancar y parar el servicio, y un método para acceder a la información de la posición actual.
- *Lugar* es la clase que contiene la información de un nodo.
- *Lugares* proporciona una abstracción para el acceso a la información de los nodos contenida en el fichero de descripción de los nodos. A partir de un número identificador devuelve información tal como las coordenadas del nodo y su etiqueta.
- *NodoWifi* contiene la información de un punto de acceso.
- *DeviceReader* proporciona acceso al contenido de un fichero almacenado en la tarjeta SD.
- *DeviceWriter* proporciona funciones de escritura de ficheros en la tarjeta SD.
- *DateUtils* proporciona funciones de acceso a la hora del sistema.

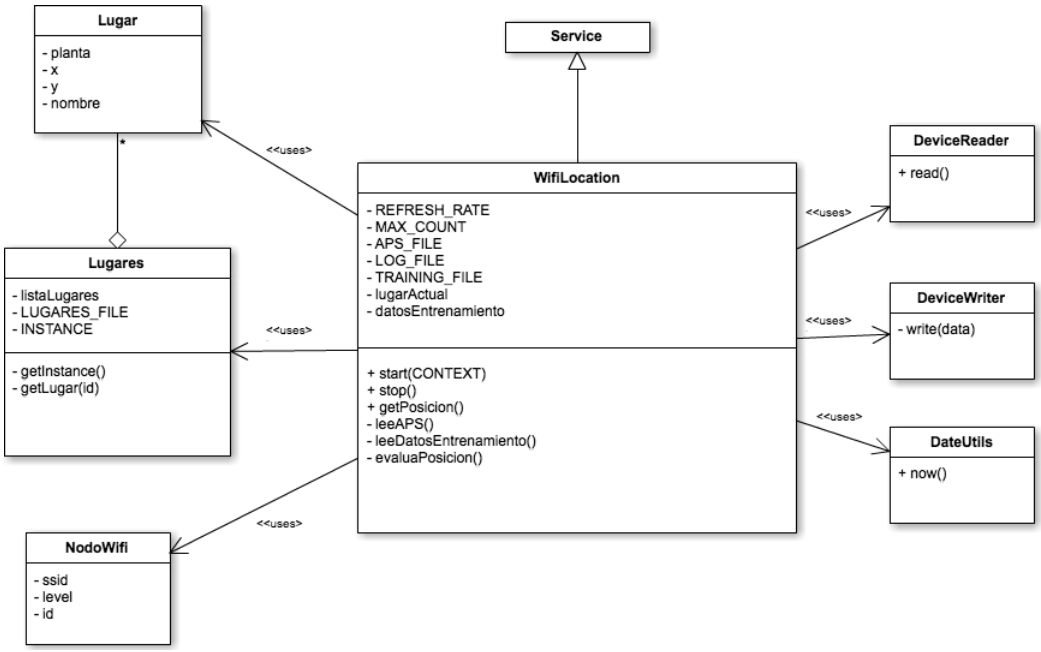

Figura 5.10: Diagrama de clases de la localizacion por WiFi

Diagrama de Secuencia

A continuación se describe el comportamiento del servicio de localización, representado en la figura 5.11, desde que se inicia el servicio hasta que se lee la posición.

1. El objeto cliente llama al método *start()* para arrancar el servicio de localización. El servicio registra un listener en el API *WifiManager* de Android y empieza a recibir eventos de escaneo de redes.

2. Se realizan varios escaneos de redes y se van compactando los datos tantas veces como esté definido en el parámetro *MAX_COUNT*, que indica el número de escaneos a realizar por cada procesamiento de datos.

3. Una vez hecho esto, se llama el método *evaluaPosicion()*, encargado de preprocesar los datos ejecutar el algoritmo de inferencia de la posición correspondiente. Se actualiza la información de la posición que el resultado de la ejecución del algoritmo.

4. Este proceso se repite hasta que el cliente llama al método *stop()*. El cliente puede acceder a los datos de localización en cualquier momento a través del método *getPosicion()*.

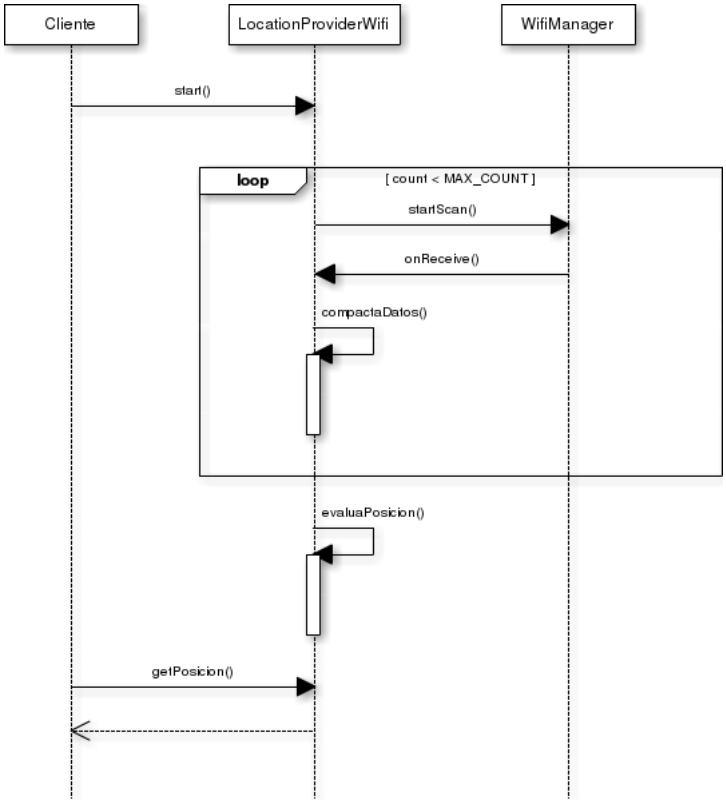

Figura 5.11: Diagrama de Secuencia Servicio Localizacion WiFi

5.3.4. Implementación

Para la implementación, se ha desarrollado una aplicación para Android que implementa todas las fases descritas en este capítulo. La aplicación se compone de los siguientes módulos:

- **Detección de puntos de acceso**: Se encarga de la detección automática de los puntos de acceso y la escritura del fichero de puntos de acceso. Esto elimina la necesidad de que el usuario tenga que confeccionar esta información a mano. Es posible añadir restricciones tales como la detección de puntos de acceso con determinado SSID.
- **Captura de datos**: Se encarga de la captura de los datos en cada uno de los nodos y en la creación del fichero que contiene el mapa de radio. El usuario sólo tiene que especificar el número de identificador del nodo correspondiente, situarse en ese punto y comenzar el escaneo.
- **Entrenamiento**: (Sólo para redes neuronales). Se encarga de crear la red neuronal, construir el conjunto de entrenamiento y ejecutar el algoritmo de entrenamiento sobre esta. El resultado de la ejecución es una red neuronal entrenada, que se almacena en un fichero para ser cargada por el módulo de localización. Además crea otros ficheros necesarios para la ejecución del algoritmo de localización por red neuronal.
- **Localización**: Se encarga de ejecutar el servicio de localización, mostrando los datos del nodo actual en pantalla. Para ello utiliza los ficheros creados en las etapas anteriores y el algoritmo correspondiente.

Artefactos Generados

En la figura 5.12 se representa un esquema de los artefactos que participan y se generan en cada una de las fases del subsistema de localización por Wifi:

- **APs.txt**: Direcciones MAC de los puntos de acceso que se utilizarán en la localización.
- **Entrenamiento.txt**: Mapa de radio generado por el módulo de captura de datos.
- **nndata.txt**: Red neuronal entrenada serializada para su uso en el módulo de localización.
- **PostProccess.txt**: Entradas y salidas de la red neuronal postprocesadas para su uso en el módulo de entrenamiento.
- **Nodos.txt**: Información de los nodos necesaria para el algoritmo de localización (Nodos que participan y su orden).
- **Lugares.txt**: Información de los nodos: id, coordenadas y etiqueta.

5.3.5. Pruebas

Para comprobar la precisión del sistema de localización, se han realizado pruebas utilizando los dos algoritmos en un piso de 90 metros cuadrados, y registrado el porcentaje de acierto en diferentes nodos.

Se han definido 4 nodos correspondientes a distintas habitaciones separados por un espacio de entre 1 y 3 metros. Los nodos son los siguientes:

- **ID: 1**: Habitación

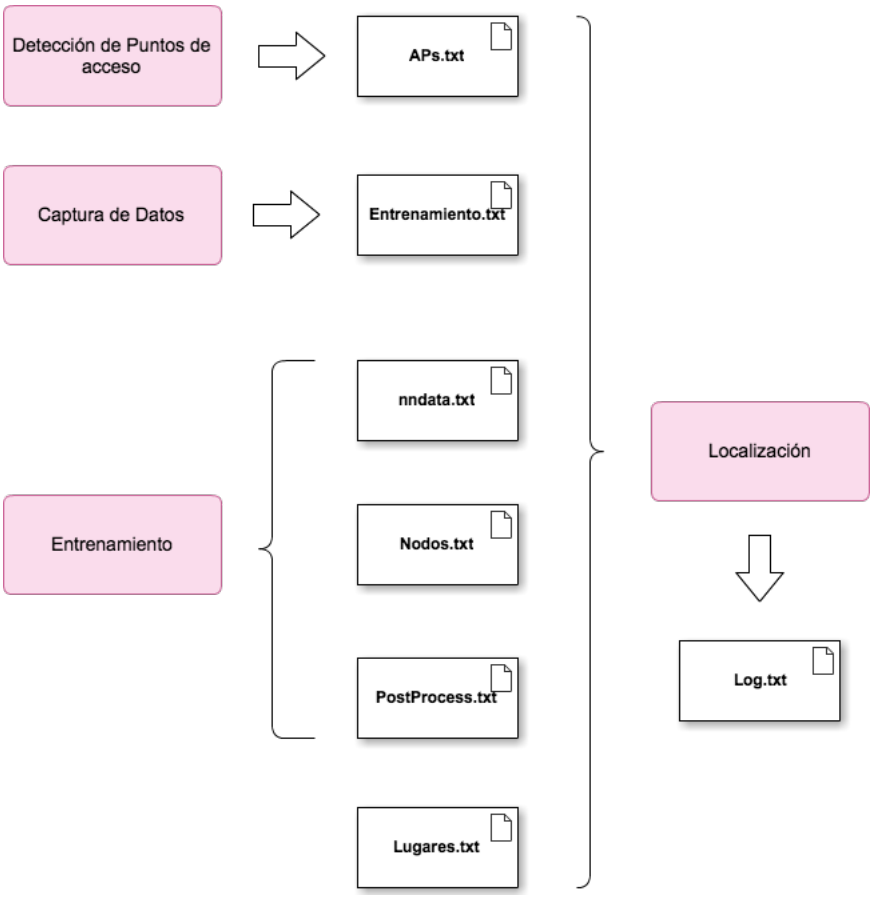

Figura 5.12: Artefactos Generados por el sistema de Localizacion WiFi

Figura 5.13: Captura del programa de localizacion WiFi en funcionamiento

- **ID: 2**: Baño.
- **ID: 3**: Cocina.
- **ID: 4**: Salón.

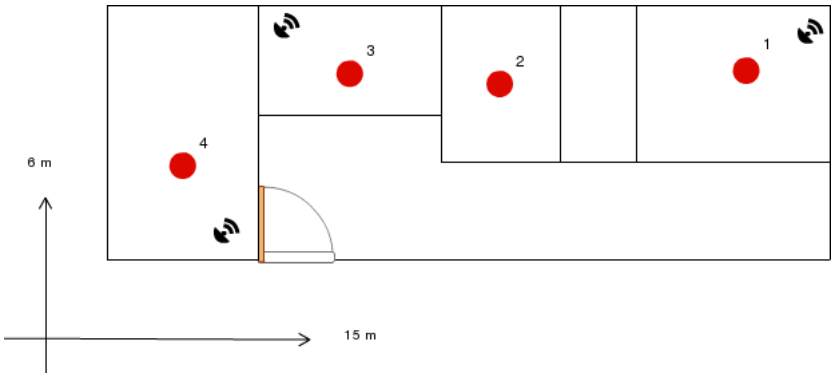

Figura 5.14: Nodos definidos piso

Se han utilizado 3 puntos de acceso propios colocados de forma estratégica a lo largo del piso. Se ha realizado el entrenamiento y se ha ejecutado el algoritmo de localización 10 veces en cada nodo. El resultado se presenta en porcentaje de acierto en la siguiente tabla:

Algoritmo	**% Habitación**	**% Baño**	**% Cocina**	**% Salón**
Closest Neighbor	100 %	0 %	100 %	100 %
Red Neuronal	90 %	30 %	30 %	30 %

[4]

Se puede comprobar en los resultados que el algoritmo *Closest Neighbor* se comporta mucho mejor que la red neuronal a la hora de aproximar los nodos, al menos con la configuración utilizada en la red neuronal. Esto puede ser debido a una arquitectura inadecuada de la red neuronal, pero limitaciones del tamaño de la pila de programa de Android nos ha impedido utilizar redes de mayor complejidad. Creemos que un mayor estudio de la aplicación de redes neuronales a este problema permitiría mejorar en gran medida los resultados obtenidos con este método.

5.4. Integración de los subsistemas de Localización

Los subsistemas de localización se integran a través de la clase *LocationManager*, que proporciona una interfaz global para el acceso al sistema de localización. Permite utilizar cada uno de los subsistemas de forma separada ó ambos de forma conjunta de forma transparente para el programador de aplicaciones. En la figura 5.15 se muestra en mayor nivel de detalle cómo se integran dichos subsistemas.

[4]Mediante el algoritmo Closest Neighbor, en todas las ocasiones se detectó el nodo número 2 como el número 3. Esto indica que la distancia entre los nodos 2 y 3 es demasiado pequeña y el nodo 3 solapa al nodo 2 en la predicción.

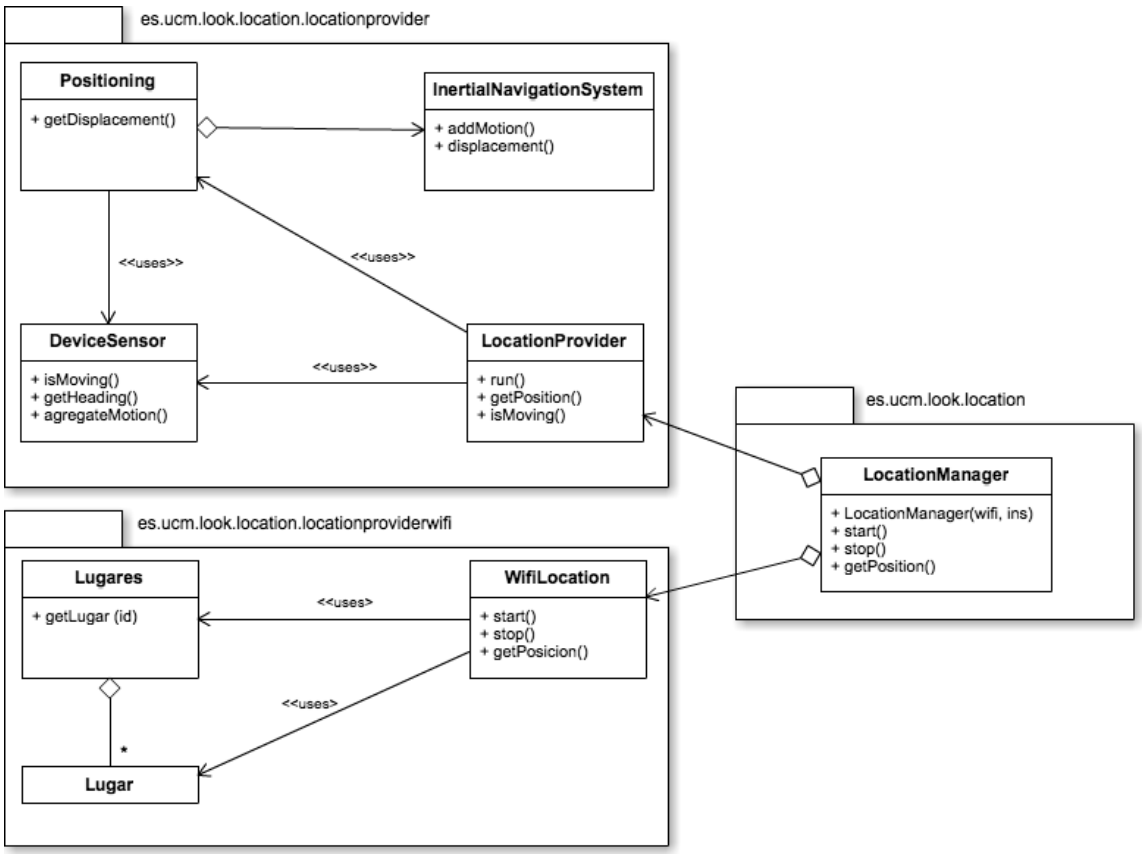

Figura 5.15: Integración de los subsistemas de Localización

Capítulo 6

Módulo de datos

En esta sección se detalla qué datos son los mínimos indispensables para definir elementos en la realidad aumentada, y cómo están estructurados en *Look!*.

Estos datos se representan con una unidad básica de datos (*EntityData*),que detallamos a continuación. Además, para formar el "Mundo" en la Realidad Aumentada se define la *WorldEntity*, que se construye a partir de un *EntityData*.

Los datos "*EntityData*"pueden estar almacenados en bases de datos locales o remotas, o definidos de manera programática. De ello se encarga la interfaz *DataHandler*, la cual accede a los datos de una manera transparente a la aplicación. También comentamos como se representan los datos, el tratamiento de la persistencia y las implementaciones llevadas a cabo para ello.

6.1. *EntityData* como unidad básica de datos

Antes de plantear la estructuración de los datos, conviene establecer dos definiciones que serán utilizadas a lo largo del texto: Consideramos un **dato** como una propiedad asociada a un valor determinado. Un **elemento**, se corresponderá a un conjunto de datos que representan una entidad de un tipo determinado.

Para permitir el mayor número de aplicaciones desarrollables por el framework, el modelo de datos debe ser general y flexible, permitiendo la definición de todo tipo de elementos.

La aproximación básica sugiere que, para permitir la mayor flexibilidad posible, los datos de un elemento podrían estar representados por una tabla (implementada con un *Map* de *Java*), que guardara propiedades asociadas a sus respectivos valores.

Sin embargo, existen tres propiedades especiales, compartidas por todos los tipos de elementos, que se destacan sobre las demás:

- **Identificador**: Un identificador único que permita distinguir los elementos entre sí.
- **Localización**: En otros contextos no tendría relevancia, pero en Realidad Aumentada, todo los elementos, salvo excepciones, están asociados a una

posición en el espacio.

- **Tipo**: La clase de entidad a la que pertenece. Este tipo agrupará elementos, y facilitará funciones de filtrado de elementos.

En base a todo lo dicho, en *Look!*, la unidad básica de datos está representada por la clase *EntityData* (Figura 6.1). Sus atributos principales son un identificador único, el tipo, la localización, y una tabla de propiedades. Ofrece métodos para la consulta y modificación de todos ellos.

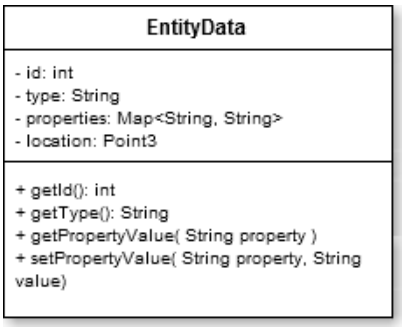

Figura 6.1: La clase EntityData

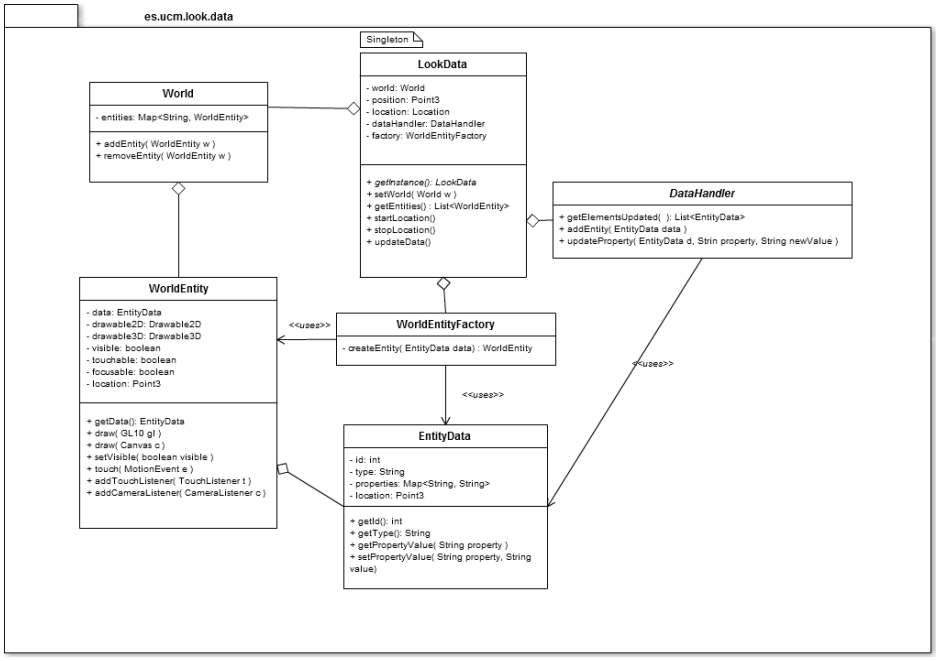

Figura 6.2: Diagrama de Clases del Acceso a Datos

6.2. Conectando los datos con el módulo de realidad aumentada: Mundo y Entidades del mundo

Una vez hemos obtenido los datos *EntityData* de nuestra aplicación, debemos convertirlos en entidades reconocibles por el módulo de realidad aumentada, que se verá con detalle en el capítulo 7.

Un *WorldEntity* es una entidad basada en un *EntityData*, y es representable por el módulo de realidad aumentada.

El Mundo, reflejado en la clase *World*, representa el contenedor general y único de todos los entidades que pueden aparecer.

6.2.1. *World*

El mundo *World* ejerce como contenedor, a todos los efectos, de entidades *WorldEntity*. Contiene métodos *thread-safe* que permiten añadir, consultar y eliminar entidades del mundo.

Como elemento adicional, el mundo contiene una **posición**. Esta posición representa la localización actual del usuario, dentro de ese mundo.

Al igual que *WorldEntity*, *World* está pensado como una clase base que debe ser extendida para aportar mayor funcionalidad.

LookData contiene un Mundo, y será lo que procese el módulo de realidad aumentada. En la figura 6.3 puede verse un diagrama de clases en dónde se relacionan todas estas clases.

6.2.2. *WorldEntity*

Cada uno de los elementos contenidos por el mundo es una entidad *WorldEntity*. Estos elementos están construidos a partir de un *EntityData* a través de una factoría *WorldEntityFactory*, que se verá con más detalle después.

Contiene atributos relacionados con su representación gráfica e interacciones permitidas, y que son iniciados a partir de los atributos esenciales anteriormente definidos:

- **Representación 2D**: representación gráfica que se le da a la entidad en la capa 2D. Se verá con mayor detalle en la sección 7.4.
- **Representación 3D**: representación gráfica que se le da a la entidad en la capa 3D. Se verá con mayor detalle en la sección 7.2.
- ***Touch Listeners***: lista de *listeners* que responden y actúan ante eventos táctiles sobre el elemento.
- ***Camera Listeners***: lista de *listeners* que responden y actúan cuando el usuario apunta directamente (con el centro de la pantalla) a la entidad.

También cuenta con otros atributos sencillos que sirven para configurar el comportamiento general de la entidad dentro del mundo:

- **Visibilidad**: Si el elemento es visible.

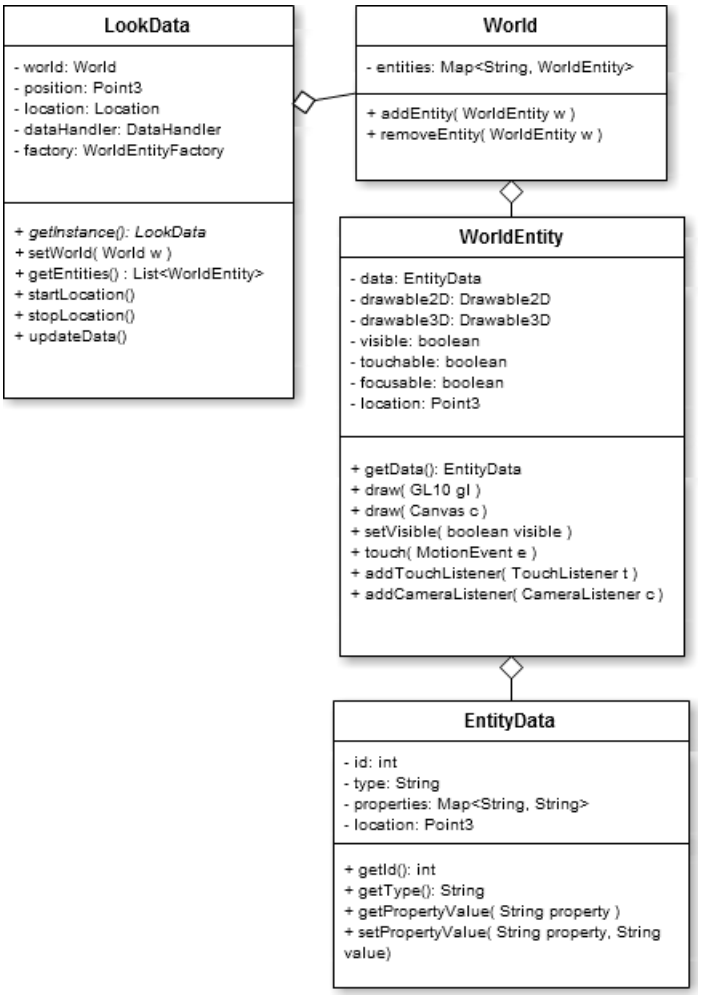

Figura 6.3: Diagrama de clases de LookData, World, WorldEntity y EntityData, con sus relaciones.

- **Habilitado**: Si recibe eventos táctiles o de cámara.
- **Táctil**: Si puede recibir eventos táctiles.
- **Enfocable**: Si puede recibir eventos de cámara.

La clase *WorldEntity* provee los métodos necesarios para la adecuada manipulación de la entidad. Puede y debe ser extendida y personalizada para lograr comportamientos más funcionales.

En el capítulo 7 se verá con mas detalle las interacciones y las representaciones en dos y tres dimensiones.

6.2.3. Actualizando el mundo

Cuando el método *LookData.getInstance().update()* es invocado, los *EntityData* de la aplicación son convertidos en *WorldEntity*, a través de una factoría (Figura 6.4), y se produce la actualización del mundo.

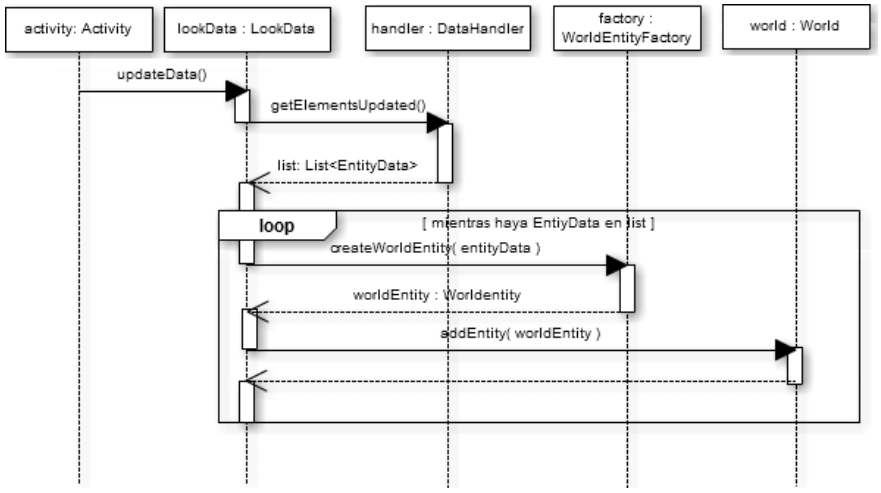

Figura 6.4: Actualización del mundo

La factoría de elementos *WorldEntityFactory* tiene un único método, *createWorldEntity(EntityData data)*, que se encarga de convertir los datos puros en elementos representables por la realidad aumentada.

Esta clase debe ser extendida para crear *WorldEntity* con apariencias personalizadas.

6.3. Obteniendo y almacenando datos: *DataHandler*

Una vez definida la estructura básica del modelo de datos, conviene preguntarse de dónde se obtienen esos datos que construyen *EntityData*, y cómo y dónde se guardarían nuevos datos añadidos durante la ejecución de la aplicación, en caso de que se requiriera persistencia de datos.

La interfaz *DataHandler* ofrece métodos generales para la obtención, modificación y almacenamiento de datos. Está basado en dos interfaces separadas: *DataGetter*, que agrupa toda la funcionalidad de obtención de datos, y *DataSetter*, que contiene métodos para la adición y modificación de datos ya existentes (Figura 6.5).

Corresponde a cada implementación concreta definir de dónde se obtienen los datos, y dónde se almacenan las modificaciones.

Look! ofrece implementaciones básicas para tres tipos de acceso a datos: datos definidos de manera programática sin persistencia, datos almacenados en una

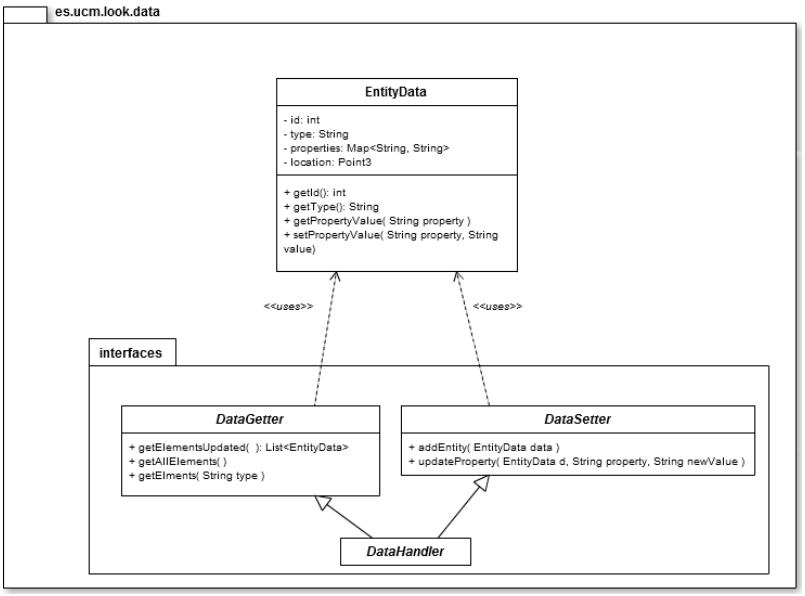

Figura 6.5: Relación de *DataHandler* con *EntityData*

base de datos local con persistencia y datos almacenados en una base de datos remota con persistencia.

A continuación, se detalla cada una de ellas.

6.3.1. Obtención de datos sin persistencia

La clase ***BasicDataHandler*** implementa *DataHandler* y ofrece un administrador de datos básico. Almacena en una lista los *EntityData* para la aplicación (Figura: 6.6).

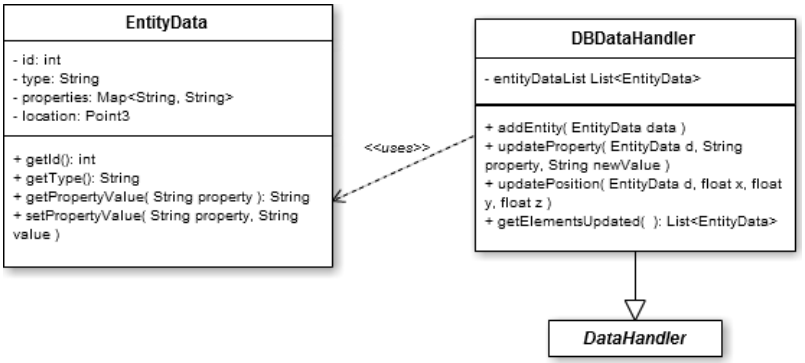

Figura 6.6: Implementación de *BasicDataHandler*

Su utilización es sencilla: Tras su instanciación, al inicio de la ejecución, se le añadirán de manera programática los *EntityData* necesarios para la aplicación.

También podrían añadirse, a través de él, nuevos *EntityData* durante la ejecución de la aplicación, así como manipular propiedades de los ya existentes.

Cuándo la aplicación finaliza, los elementos creados son eliminados.

Se explicará su uso con mayor detalle en el capítulo 8.

6.3.2. Obtención de datos con persistencia local

Para implementar una persistencia de datos en nuestras aplicaciones, necesitamos una estructura que almacene los datos de la aplicación cuándo ésta no esté en ejecución. Una estructura a través de la cuál se pueda acceder a los datos persistentes entre ejecuciones, y que permita las operaciones básicas de consulta y modificación.

La estructura elegida para lograr esta persistencia es una base de datos, puesto que Android permite la creación bases de datos *SQLite*. Además, nuestra estructura de datos se ajusta bien a esta representación.

Para el acceso a los datos extendemos la interfaz *DataHandler* a la clase ***DBDataHandler*** que es la encargada de administrar los datos para la persistencia de una manera transparente a la aplicación (Figura: 6.7).

A continuación describimos la estructura de la base de datos y su implementación mediante un *Content Provider*[1].

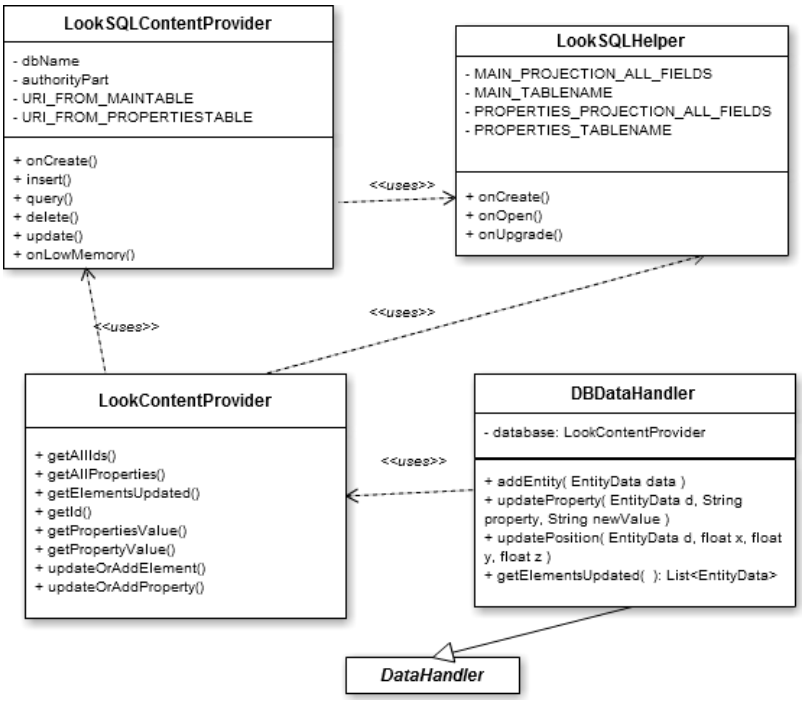

Figura 6.7: Implementación de *DBDataHandler*

[1]Un *Content Provider* es un proveedor de contenido en Android, permite compartir datos entre aplicaciones usando una base de datos SQLite

Estructura de la base de datos

En la base de datos se almacenan elementos *EntityData* en forma de tablas. La base de datos está formada por dos tablas: La tabla **Main**, donde se almacena un registro por cada entidad, con su posición y última fecha de actualización; y la tabla **Properties**, dónde se guarda cada una de sus propiedades en un registro. El tipo de la entidad se guardará como una propiedad más.

Describimos a continuación cada una de las tablas:

Main

Compuesta por 5 campos:

- **id** Identificador entero único para cada entidad, clave primaria de la tabla. En la base de datos del servidor esta clave es además auto incremental. Así nos aseguramos que no se repita en dos entidades.
- **x** Coordenada X con la posición del objeto en el mundo.
- **y** Coordenada Y con la posición del objeto en el mundo.
- **z** Coordenada Z con la posición del objeto en el mundo.
- **last update** Fecha y hora con la última actualización de esta entidad en la base de datos.

	Campo	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Extra
☐	id	int(11)			No	*None*	AUTO_INCREMENT
☐	x	double			No	0	
☐	y	double			No	0	
☐	z	double			No	0	
☐	last_update	datetime			Sí	*NULL*	

Tabla Properties

Describe cada propiedad mediante los siguientes campos:

- **id** Identificador de la entidad, el registrado en la tabla *Main*.
- **property** Nombre de la propiedad, es variable para cada tipo distinto de entidad.
- **value** Valor de la propiedad descrita en *type*.

	Campo	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado
☐	id	int(11)			No	*None*
☐	property	varchar(15)	latin1_spanish_ci		No	*None*
☐	value	varchar(100)	latin1_spanish_ci		Sí	*NULL*

Nota: No puede haber dos propiedades del mismo tipo (campo *property*) para la misma *id* (en la implementación *id* y *type* son claves primarias).

Dependencias

Cuando se guarda un elemento se crea un registro en la tabla *Main* con una *id* única. Esta misma *id* es usada para almacenar cada una de sus propiedades en la tabla *Properties*.

La propiedad **type** define el tipo de una entidad. Esta propiedad obligatoria será definida por cada nueva "clase de entidad". El campo *property* será *type* y en value irá definido el tipo.

Si las coordenadas del objeto no se indican estas serán inciadas a (0, 0, 0).
También el campo *last update* será inicializado con la fecha y hora del actual[2] con el siguiente formato "YYYY-MM-DD hh:mm:ss", cada vez que es actualizada una instancia en la base de datos, sea la posición, sea una de sus propiedades, este dato es actualizado con la nueva fecha y hora de la modificación.

Ejemplo de una entidad tipo *user*:

id	property	value
113	user	jmartin
113	type	user
113	password	****
113	name	Juan Martín
113	info	En el lab 12
113	email	juanmartin@dominio.com

Figura 6.8: Entidad de tipo *user* usada en la aplicación Look! Social

Implementación de la base de datos

Para el acceso al contenido de la base de datos *SQLite* hemos definido nuestro *Content Provider* denominado ***LookContentProvider***, este se encarga de configurar y actualizar una base de datos *SQLite*.
Un *Content Provider* es un objeto de la clase "ContentProvider" de Android el cual permite almacenar datos de un determinado tipo (en la base de datos) para que puedan ser accedidos desde cualquier aplicación.
La elección de usar un *Content Provider* nos permite que el contenido de una base de datos sea compartido por dos o más aplicaciones distintas en el dispositivo. Así podríamos comunicar dos aplicaciones desarrolladas con el framework.

LookContentProvider realiza la lógica entre las *EntityData* y la base de datos *SQLite*, para ello dispone de dos clases más para ayudarnos a generar, modificar o consultar la base de datos (LookSQLContentProvider y LookSQLHelper). En

[2]Si la base de datos es remota, en el caso de querer dar acceso a usuarios internacionales el campo *last update* almacena la hora local del servidor, por lo que el cliente tendría que calcular la última actualización respecto a su zona horaria.

la figura 6.7 se representa la estructura del *Content Provider*.

LookSQLContentProvider

En esta clase están definidos los métodos Query(), Insert(), Update() y Delete() con las operaciones básicas de la base de datos *SQLite*.

El acceso a los datos se realiza mediante URIs (recursos), cada elemento tiene asociada una URI única que lo representa. Cada una de las tablas de la base de datos se representa como un elemento, por lo que definimos las siguientes URIs:

- *Main* URI_FROM_MAINTABLE = "content://APLICACIÓN/lookMain/"
- *Properties* URI_FROM_PROPERTIESTABLE = "content://APLICACIÓN/lookProperties/"

"APLICACIÓN" se refiere al nombre de la aplicación a desarrollar. Una aplicación con una base de datos nueva tiene que definir un nombre para esta único, al igual que las URIs únicas para el acceso a sus elementos. Se explica en detalle en el capítulo 8.

LookSQLHelper

En esta clase se definen las constantes con los campos y las tablas de la base de datos. Además se encarga de crear, abrir o actualizar la base de datos con los siguientes métodos:

- onCreate(): Se llama la primera vez que ejecutamos la aplicación para crear la base de datos.
- onUpgrade(): Cuando actualizamos la base de datos se llama este método. Actualmente está vacío.
- onOpen(): Se ejecuta cuando cargamos la aplicación. Actualmente está vacío.

6.3.3. Obtención de datos con persistencia remota

Para tratar los datos de las aplicaciones con acceso a servicios remotos, debemos proveer de comunicación con un servidor al framework. En este apartado explicamos la gestión de la persistencia sobre la estructura de datos con acceso a un servicio web.

Ahora las *EntityData* son almacenadas tanto en el servidor (datos globales) como en el cliente (datos locales). En el servidor se guardan en una base de datos *MySQL*, aquí estarán los datos de todos los clientes y serán actualizados por estos con consultas enviadas a través del servicio web. Gracias a esto conseguimos reducir las transacciones con el servidor accediendo al él solamente para añadir / modificar un elemento o para actualizar la base de datos de un cliente con los datos globales. Para el acceso a los datos del servidor se encarga el servicio web detallado en el siguiente apartado.

Una vez que el Servicio Web recoge los datos, estos son almacenados en el cliente en una base de datos *SQLite* con una estructura similar a la *MySQL* del servidor, esta representa una pequeña copia con los datos que utilizaremos del servidor. La estructura de las bases de datos es la misma detallada anteriormente en el apartado 6.3.2.

La implementación de la base de datos *SQLite* se configura con *LookContentProvider* tal como se detalló para la persistencia local. Solo que ahora esta restringido a la consulta de datos (*DataGuetter*), para modificaciones o actualizaciones se accede directamente al servicio web.

El acceso a los datos está simplificado con estas dos interfaces:

- **DataGetter:** (Lectura de datos) El acceso a los datos (*EntityData*) estará restringido al *Content Provider*, el cual es actualizado por medio del Servicio Web de una manera transparente. Está detallado en "Content Provider".

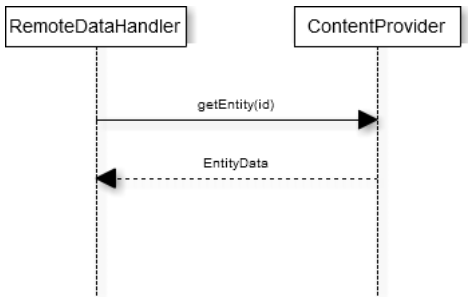

Figura 6.9: Diagrama de secuencia para el acceso de datos

- **DataSetter:** (Modificación/Actualización de datos) Si queremos enviar nuevas entidades o actualizaciones de estas se accederá directamente al Servicio Web que está conectado con el servidor. Lo detallamos en "Servicio Web".

Al igual que para el acceso a los datos sin persistencia o con persistencia local, implementamos *DataHandler* para crear la clase ***RemoteDataHandler***, encargada de administrar los datos para la persistencia con el servicio web de una manera transparente a la aplicación (Figura: 6.11).

Servicio Web

Para añadir o modificar entidades la aplicación se utiliza el servicio web, el cual envía los datos al servidor además se encarga de actualizar la base de datos local del *Content Provider* para mantener la coherencia.

Para esto, está conectado directamente con el *Content Provider* y cada vez que el dato enviado es procesado por el servidor, si se almacenó correctamente en los datos globales, se actualiza también en la base de datos del cliente (*SQLite*).

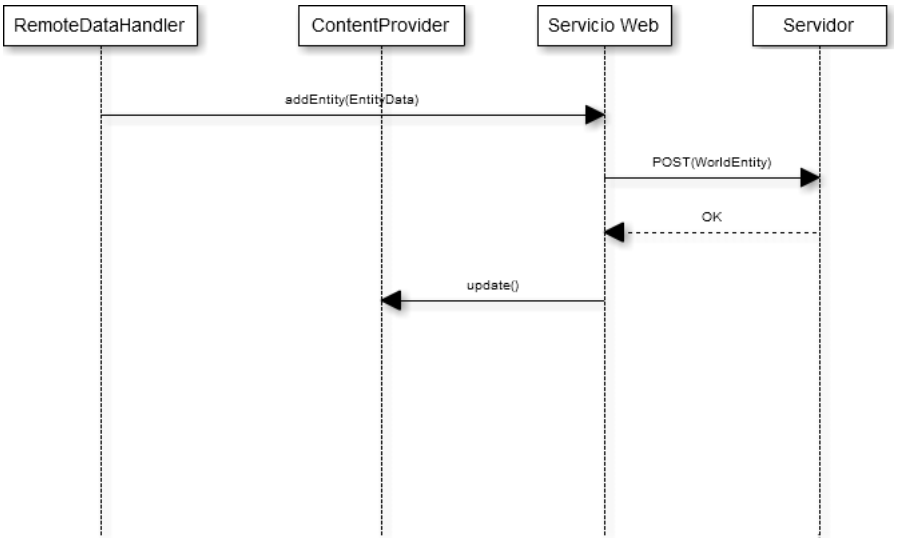

Figura 6.10: Diagrama de secuencia para el almacenamiento de datos

A continuación describimos los métodos accesibles en el Servicio Web, representados en la interfaz llamada **DataSetter**.

- `addElement(EntityData)`

  Envía una nueva entidad al servidor para que sea almacenada en la base de datos global. También la crea en la base de datos local para que esté sincronizada. El servidor devuelve el *id* único de la nueva entidad y la copia en local se almacenaría con este *id*.

  La *EntityData* también contiene la posición de la entidad x, y, z y un *Map* con las propiedades que tendía el elemento, en caso de no estar vacío serán insertadas en la tabla *properties*. Además contiene el tipo representado por *type*, este es insertado como una propiedad de esta entidad con el campo *"property = type"*.

- `updateElementPosition(id, x, y, z)`

  Actualiza con la posición recibida (x, y, z) la entidad *id*. Si hay algún error el servidor le enviará una respuesta para informarle. No es necesario informar al *Content Provider* ya que este dato solamente es usado para modificar nuestra posición, la cual procede del *Location Provider*[3] .

- `updateOrAddProperty(id, property, value)`

  Para tratar una propiedad de una entidad solamente es necesario acceder a este método. Si en la entidad con el *id* especificado no existe esta propiedad, es insertada por el servidor. En caso de existir previamente será modificada por la nueva. Al igual que los métodos anteriores la base de datos local también será actualizada con los nuevos datos.

[3]Nos informa de nuestra posición, sistema de localización vía Wifi e inercial

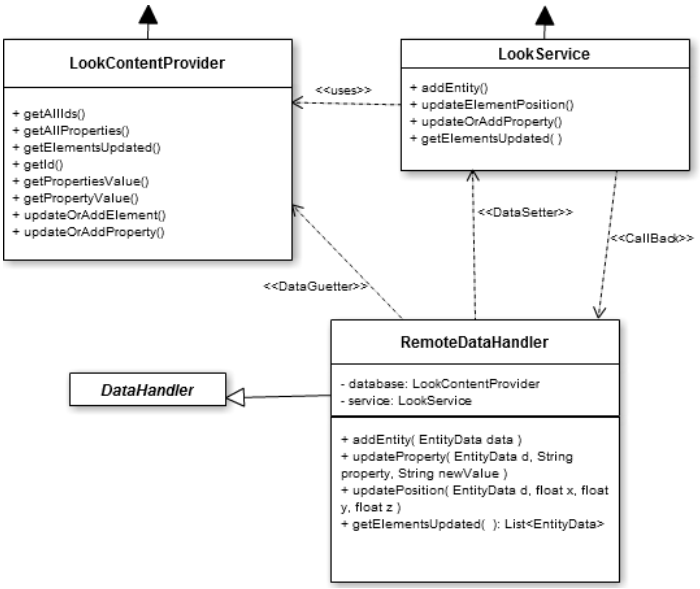

Figura 6.11: Implementación de *RemoteDataHandler*

- `deleteElement(id)`

  Eliminamos una entidad (*EntityData*) con la identificación *id*. Será eliminada de la tabla *Main* y de *Properties* (todas sus propiedades) tanto en el servidor como en la base de datos local.

- `updateDB(x, y, z, radius, types)`

  Actualizamos las entidades con los tipos representados en la lista *types* en el cliente (base de datos local). Estos deben estar dentro del radio indicado respecto a la posición enviada.

  Ese radio será calculado respecto a las coordenadas x e y, la coordenada z tiene que ser igual en la entidad. Es decir, el elemento tiene que estar entre ((x - radius/2) AND (x + radios/2)) AND ((y - radius/2) AND (y + radius/2))

  Además comprobamos que fecha de la última actualización de cada entidad con la última actualización por el cliente, así evitamos descargar al cliente los datos sin modificaciones.

  El servidor en este caso solamente realiza una petición para que devuelva estos datos, por lo que la base de datos global no será modificada.

  Este método es el encargado de sincronizar la base de datos del cliente con las actualizaciones de los demás usuarios, cada aplicación puede llamar a este método o automáticamente por medio de un *thread* o manualmente (con un botón de actualizar). Al comienzo de cada aplicación (la primera vez que se ejecuta) llamamos a este método sin comprobar fechas ya que la base de datos *SQLite* estará vacía.

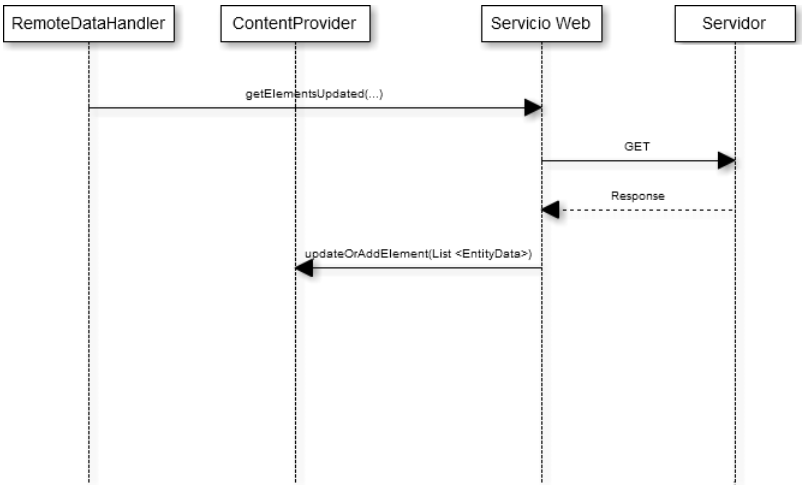

Figura 6.12: Diagrama de secuencia de updateDB()

- `doLogin(username, password)`

  Este método es especial, trabaja con las propiedades *username* y *password* de los elementos en la base de datos. Lo hemos implementado porque creemos que casi cualquier aplicación tendrá un método para registrarse, así nos evitamos realizar la lógica para cada aplicación.

  Es necesario que uno de los elementos de la aplicación contenga la propiedad *username* y *password*.

NOTA: Eliminar una propiedad no existe como método, en el estado actual del framework consideramos que una vez creada no tiene sentido eliminarla, ya que el número de propiedades consideradas es fijo.

Conexión con el Content Provider

Para mantener actualizado la base de datos local cada vez que enviamos o recibimos datos del servidor, el servicio web internamente se conecta con el *Content Provider* para sincronizarlo.

Para ello el *Content Provider* dispone de los tres métodos siguientes:

- `updateOrAddElement(id, x, y, z)`

  Si no existe el elemento con este *id*, este es creado guardándose en él las coordenadas (x, y, z). En caso de existir será actualizada su posición.

- `updateOrAddProperty(id, property, value)`

  Si en la entidad con el *id* especificado no existe esta propiedad, es insertada. En caso contrario será modificada con el nuevo valor.

- `deleteElement(id)`

  Eliminamos la entidad *id*.

Content Provider

A continuación describimos los métodos accesibles en el *Content Provider*, al igual que en el servicio web, se dispone de una interfaz llamada **DataGetter** que define los siguientes métodos:

- `getEntities(x, y, z, radius, types)`

  Obtenemos en una lista de *EntityData* con todas las entidades de los tipos incluidos en la *List* "types". Estos deben estar dentro del radio respecto a la posición indicada.

  Ese radio será calculado respecto a las coordenadas x e y, la coordenada z tiene que ser igual en la entidad. El elemento tiene que estar entre ((x - radius/2) AND (x + radios/2)) AND ((y - radius/2) AND (y + radius/2))

- `getId(type, value)`

  Devuelve el *id* único para la entidad que tenga una propiedad cuyos campos sean "Property = type" y "Value = value". Si hay más de una entidad con esa misma propiedad devuelve la primera de ellas (la primera se corresponde con el elemento más antiguo). En caso de que no exista en la base de datos ningún elemento con esta propiedad devolverá "-1".

- `String getPropertyValue(id, propertyName)`

  Devuelve la propiedad *propertyName* de la entidad *id*.

- `getPropertiesValue(id, propertiesName);`

  Similar al anterior solo que ahora devuelve una "colección de Propiedades" de la entidad *id*. En *propertiesName* irá una lista con los nombres de cada propiedad que queremos recibir.

  La colección es un Map(propiedad, valor).

- `getAllProperties(id);`

  Obtenemos todas las propiedades de la entidad *id* en un *Map* (propiedad, valor).

- `getAllIds(typeProperty)`

  Devolvemos una lista con los *ids* de los elementos cuya propiedad tipo se corresponda con "Property = TYPE" y "Value = typeProperty"

Implementación del Servicio Web

El servicio web ha sido desarrollado tanto en el servidor como en el cliente. En los siguientes apartados hablaremos sobre su implementación.

Integración en el servidor

Para el servidor se ha implementado un **servicio web RESTful** con soporte a persistencia. Este tipo de servicio da soporte a las siguientes operaciones que pueden ser realizadas contra la tabla de una base de datos (operaciones *CRUD*):

create, *read*, *update* y *delete*. El servidor no procesa ningún tipo de lógica sobre los datos que contiene. Se limita a servir consultas y procesar modificaciones a través de las operaciones anteriores.

El servicio web ha sido creado con NetBeans 6.9. Para ello en NetBeans hemos establecido una aplicación web utilizando el servidor GlassFish 3.1 y Java EE 6; a ésta le añadimos una unidad de persistencia con una base de datos MySql 5.1 externa donde se almacenan todos los datos.

Las clases generadas en el servidor se dividen en los siguientes paquetes:

- ***look***: contiene las clases con la representación de cada una de las tablas de la base de datos. **Main**, **Properties** y **PropertiesPK** (para identificar la clave primaria de Properties). Se generan una serie de Queries para poder realizar búsquedas sobre las mismas y las anotaciones de relación necesarias para vincularse.
- ***service***: contiene los servicios web generados y el servicio de persistencia que tiene el acceso al entityManager, que da acceso a la unidad de persistencia. Se crearán dos servicios web por cada entidad, para trabajar con el listado de entidades y con una entidad individual, y estarán anotadas con el soporte de jax-ws necesario para que los servicios sean publicados correctamente.
- ***converter***: contiene un conversor para el servicio web, que es quien hace uso del mimo. Los conversores están anotados con jaxb y dan el soporte necesario para la serialización de ojetos a xml ó json y viceversa. De este modo, cuando se solicita un dato se obtiene la entidad que lo representa y esta es convertida a xml — json para su devolución. La conversión de xml — json a objeto se realiza en las modificaciones e inserciones, puesto que lo que recibe el servicio web es un xml — json que debe ser convertido a entidad para persistirlo.

Acceso a los datos del Servidor

Las entidades, al igual que en un Content Provider son identificadas mediante URIs. Una URI es necesaria para poder acceder al recurso en el servidor, y estos se identifican con la clave primaria de las entidades.

Hay dos tipos de URIs, cada entidad debe tener su propia URI con una apropiada representación. Además, la colección de recursos es otro recurso.

La *URI Base* de nuestro framework es:

```
"http://serverip/LookServer/resources/"
```

Donde *serverip* es la dirección *IP* del servidor, junto con el puerto para la conexión, */LookServer* la dirección con la aplicación web y */resources* donde se almacenan los recursos.

Esta se corresponde con la URL de nuestro servidor y es configurable por el framework con el método *setServerURL(String URL)* de la clase *ConfigNet*.

Representación de la *URI* para la consulta de las entidades:

- entidad **main**: Colección de recursos de todas las entidades de *main*

  `"http://serverip/LookServer/resources/mains/"`

  Entidad 100 de *main*

  `"http://serverip/LookServer/resources/mains/100"`

- entidad **properties**: Colección de recursos de todas las entidades de *properties*

  `"http://serverip/LookServer/resources/propertiess/"`

  Entidad 103,name de *properties*

  `"http://serverip/LookServer/resources/propertiess/103,name"`

Para almacenar objetos en la base de datos se debe proveer la URI de la colección de recursos, en nuestro caso:

Entidad *main*

`"http://serverip/LookServer/resources/mains/"`

Entidad *properties*

`"http://serverip/LookServer/resources/propertiess/"`

Integración en el Cliente

Para el desarrollo de *RESTful* en el cliente implementamos un servicio en Android. El servicio se encarga de procesar la comunicación entre cliente-servidor, este se ejecutará de forma automática en segundo plano y sin depender de ninguna *activity*.

A grandes rasgos, según la figura 6.13 podemos añanir / actualizar elementos del mundo o hacer peticiones de estos.

Para añadir / actualizar un elemento se hará una llamada al servicio, el cual está conectado con el servidor mediante "REST Method", el servidor al recibir el elemento enviará una respuesta que recoge el servicio, procesa y devuelve al *ServiceManager* que devolverá como *CallBack*[4] a la actividad o clase que envió la llamada.

En caso de solicitar un elemento, se pedirá al *ContentProvider* siendo devuelto directamente por este. El contenido del *ContentProvider* será actualizado por el servicio para respetar la coherencia con el servidor.

[4]Un CallBack se refiere a una respuesta a la solicitud enviada por una activity / clase

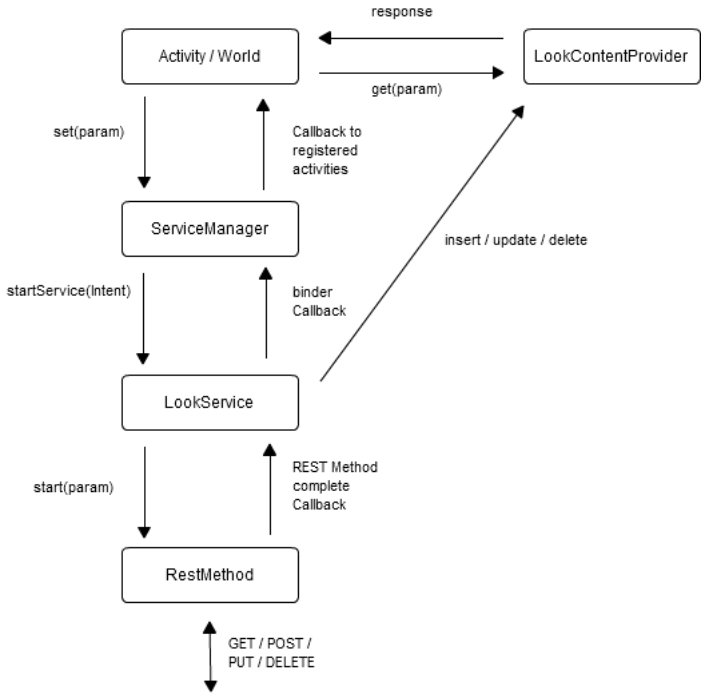

Figura 6.13: Arquitectura del módulo de datos en el servidor

En el próximo capítulo entraremos en detalle con la persistencia de la estructura de datos.

A continuación detallamos cada una de las partes de la arquitectura de acceso a datos en el cliente:

ServiceManager

Clase Singleton asíncrona a la que se accede en primer lugar. Esta comienza el servicio si este no estaba activo, establecer su conexión y lo para en caso de estar inactivo.

Además tiene una lista de *CallBacks* y se encarga de manejar las respuestas a la *activity* o clase que generó la llamada al servicio.

Para capturar una *CallBack* es necesario crear un *Handler*[5] dentro de de la clase donde será llamado el servicio e informar al servicio sobre el *Handler* activo.

Código para crear el Handler que captura la CallBack devuelta por ServiceManager:

```
Handler mHandler = new Handler() {
 public void handleMessage(Message msg) {
  switch (msg.what) {
```

[5]Un Handler es una clase de Android que se encarga de enviar o recibir mensajes

```
            //seleccionamos la CallBack a capturar,
            //puede haber varias distintas.
            case ACTION_CALLBACK_1:

              //procesamos la CallBack
              processCallBack();

              break;
              default:
            super.handleMessage(msg);
            }
    }
};
```

Código para informar al servicio sobre el *Handler* activo:

```
LookData.getInstance().getServiceManager()
                    .setHandler(mHandler);
```

LookService

Esta clase representa el Servicio de Android encargado de pedir los datos al servidor y procesar las respuestas recibidas por este.

Cuando recibe la solicitud por parte del *ServiceManager*, comienza el correspondiente *Rest Method* (doPost, doPut, doGet, doDelete).

Además recoge la respuesta, la procesa e invoca en el *ServiceManager* la nueva *CallBack*. Para que las respuestas se envíen en orden por el *ServiceManager* se guardan en una cola donde se irán desapilando según se vayan procesando.

Ese necesario para que los métodos del servicio sean accesibles desde cualquier aplicación fuera del framework, hay que definir los métodos de este mediante una interfaz AIDL[6]. Nuestro servicio dispone de dos, una para las peticiones con los métodos de acceso al servicio web (DataSetter) y otra con los métodos para cada tipo de *CallBack* (IRemoteServiceCallBack).

Cuando hacemos una llamada al servicio web se inicializa en un nuevo thread, para no interrumpir la aplicación. Además si se desea se puede mostrar un mensaje en pantalla mientras se espera la respuesta con las clases de Android *ProgressDialog* o *ProgressBar*

Para activar el servicio es necesario incluir en el *manifest* de cada aplicación este código:

```
<service android:enabled="true"
android:name=".data.restful.LookService"></service>
```

[6](Android Interface Definition Language) http://developer.android.com/guide/developing/tools/aidl.html

Rest Method

Esta clase contiene los métodos de las operaciones básicas para el envío y respuesta de las entidades mediante *HTTP*:

- doGet(url) Elimina un recurso (dirección url).
- doPost(url, json) Inserta el recurso json en la url indicada.
- doPut(url, json) Actualiza un recurso con el archivo *json* enviado
- doDelete(url) Elimina el recurso.

Las respuestas de los métodos son decodificadas y encapsuladas en un nuevo objeto *json*

Transmisión de objetos entre el cliente-servidor

Los datos transmitidos por la red desde el servidor a la base de datos del cliente (o viceversa) pueden representarse tanto en *XML* como en objetos *JSON*[7], aunque en nuestro framework hemos usado solamente *JSON* por comodidad, ya que Android implementa las librerías para su tratamiento.

Un objeto *JSON* es una colección de pares nombre/valor. El formato de los *JSON* usados para tratar las tablas del servidor es el siguiente:

```
//formato JSON de un registro de la tabla properties
properties:
{
        "propertiesPK":{"id":"x","property":"xxx"},
        "value":"xxx"
}

//formato JSON de un registro de la tabla main
main:
{
        "id":"x", "x":"xx", "y":"xx", "z":"xx"
}
```

6.4. Otras fuentes de datos: Archivos binarios

Los archivos usados por las aplicaciones como objetos 3D o las imágenes no son almacenadas en la base de datos, en estas solamente tendremos el nombre de estos o su ruta relativa a una URL base.

Para optimizar el uso de la red, disponemos de un "administrador de archivos" representado en la clase *LookFilesManager*.

Cuando solicitemos un archivo con la ruta descrita en la base de datos primero comprobamos si existe en el sistema de ficheros del dispositivo (carpeta

[7]Formato de intercambio de objetos alternativo a XML, http://www.json.org/

local dentro del dispositivo), en caso contrario el administrador de archivos se conecta a la *URL* con la dirección del recurso y lo almacenará en la carpeta local.

Para acceder a los archivos remotos se realiza una conexión mediante uno de estos protocolos 6.4.1 a una *URL*, usando URLConnection y serán descargados uno a uno a una carpeta local, tratando esta como una cache de archivos.

En caso de que una entidad sea modificada o eliminada del servidor, cuando actualicemos el cliente comprobaremos si esta incluye la ruta de objetos (archivos) en sus propiedades y de ser así, si el archivo está almacenado en la carpeta local será eliminado; y no volverá a estar en el dispositivo hasta que no sea solicitado de nuevo.

6.4.1. Administrador de archivos

El administrador de archivos usa la clase implementada por Android URLConnection para obtener un archivo desde una *URL* sin necesidad de conocer de antemano su tipo o longitud.

La URL se configura desde la aplicación con el método *setFilesURL(String URL)* de la clase ConfigNet.

Esta se puede configurar con cualquiera de estos protocolos:

- **File** Los recursos pueden ser cargados desde el sistema local de archivos usando *URIs*. Estos solamente pueden ser leídos.
- **FTP** También es aceptado el protocolo *FTP*, las conexiones pueden ser usadas para entrada o salida de datos pero no ambas a la vez. Por defecto, las conexiones *FTP* serán hechas usando *anonymous* como nombre de usuario y dejando el campo de contraseña vacío. Para especificar esto hay que seguir la siguiente estructura en la URL: ftp://username:password@host/path.
- **HTTP y HTTPS** Se puede hacer referencia mediante las subclases HttpURLConnection y HttpsURLConnection. También es aceptado al crear una URL con el prefijo "http://" o "https://".
- **Jar** Hace referencia a la subclase *JarURLConnection*.

Mediante el siguiente ejemplo se muestra como descargaríamos una imagen a nuestro dispositivo:

```
    URL url = new URL(MyURL);
    URLConnection urlConnection = url.openConnection();
    InputStream in = new BufferedInputStream(
                                urlConnection.getInputStream());

    leerImagen(readStream(in));

}
```

Esta clase soporta un *timeout* para la conexión y otro la lectura del archivo, si las capturamos evitamos un error un fallo en la conexión. Además con

IOException nos aseguramos que no ha habido una incoherencia en los datos e intentamos descargar un archivo no existente.

6.4.2. Optimizaciones del administrador de ficheros

- **Almacenamiento en la tarjeta SD** La carpeta */Look* mencionada antes la creamos directamente en la tarjeta SD:

  ```
  public static final String directorySD = Environment
  .getExternalStorageDirectory() + "/Look/";
  ```

- **Compresión de imágenes** Mediante el siguiente código guardamos las imágenes en la SD comprimidas, para optimizar más aún el espacio.

  ```
  bitmap.compress(Bitmap.CompressFormat.JPEG, 90, out);
          //90% de calidad respecto a la original
  ```

- **Organización por tipos de archivo** La carpeta raíz usada es */Look*, a partir de esta creamos para cada tipo de archivo una diferente, para asegurarnos que tenemos los datos organizados además de permitir nombres iguales para distintos tipos de archivo (por si extendiésemos el framework a algo como archivos tipo foto y tipo avatar). La carpeta se incluye en la ruta guardada por la base de datos y en la URL remota.

  Las URL de los archivos también están estructuradas, por ejemplo, las imágenes en Look! Social están almacenadas en:

  ```
  public static final String imageURL =
          "http://www.lookapp.es/look/images/";
  ```

NOTA: Actualmente nuestro framework dispone de dos tipos de archivo, objetos 3D e imágenes. Aunque por ahora solamente tratamos de esta manera a las imágenes. Los objetos 3D al ser muy pocos están incluidos en el *.apk*.

Capítulo 7

Módulo de Realidad Aumentada

En este capítulo se detalla el sistema de capas de representación de realidad aumentada utilizado en *Look!*. En primer lugar se da una visión global del módulo completo, con todas las capas que lo conforman y su distribución. A continuación, se entra en detalle en cada una de las capas.

Primero, la capa de visualización 3D, con algunos fundamentos y consideraciones de OpenGL, el sistema de entidades 3D seguido para la creación de elementos, procesado de texturas y mallas en formato *.obj, y la integración con la cámara.

Después, la capa de dibujado 2D, cómo se dibujan los elementos en ella, y cómo se calculan las proyecciones de posiciones en tres dimensiones a coordenadas de pantalla en dos dimensiones. Se introduce el sistema de animaciones y las interacciones de usuario.

Finalmente, se especifica la Capa HUD y su funcionamiento.

7.1. Visión general del sistema de capas: LookAR

Look! define cuatro capas, de abajo a arriba: cámara, 3D, 2D y HUD. La inclusión o no de cada una de estas capas es configurable para todas las aplicaciones.

Todo el sistema de capas se encuentra implementado en la *Acitivity LookAR*. Para Android una *Actitvity* es un proceso con una interfaz gráfica. Similar a una ventana en un sistema de escritorio.

Así, en términos Android, *LookAR* es una actividad preconstruída que contiene todas las capas que hayan sido configuradas en su creación, superpuestas en el orden que muestra la figura 7.1. En su creación, también pueden ser configurados otros parámetros, como si se desea que la actividad se muestre en pantalla completa, o la máxima distancia a la que los objetos serán visibles para el usuario.

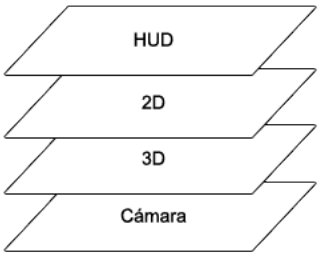

Figura 7.1: Situación de capas de representación en *Look!*

Estas capas son creadas vacías, y deberán ser provistas de elementos a ser representados, ya sea desde el módulo de datos, a elementos gráficos añadidos de manera programática, por ejemplo en las capas de HUD.

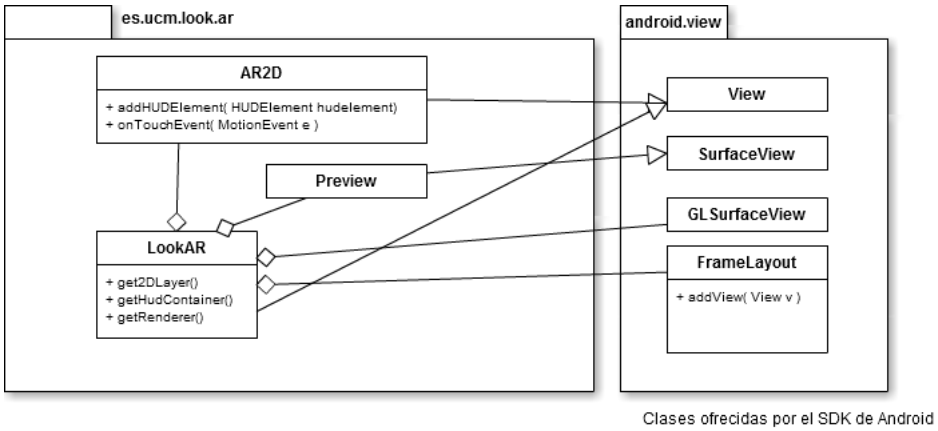

Figura 7.2: Diagrama de clases de los módulos de Realidad Aumentada.

Como puede apreciarse en la figura 7.2, las capas de realidad aumentada tienen como base vistas de Android. *LookAR* es el módulo completo que agrega cada una de las capas. *Preview* se corresponde con el módulo de cámara, el visualizador 3D se implementa mediante una *GLSurfaceView*, el visualizador 2D, AR2D, es una *View* personalizada, el módulo contenedor del HUD se corresponde con un *FrameLayout*.

7.2. Capa 3D

La capa 3D es la encargada del renderizado de la representación 3D de las entidades. Como se explicó en el apartado de tecnologías, *Look!* utiliza OpenGL ES 1.1 para renderizado en tres dimensiones, a través de la vista *GLSurfaceView*

de Android.

GLSurfaceView necesita un *Renderer*. *Renderer* es una interfaz de Android que define, entre otros, el método de dibujado que ha de ejecutarse en la vista. En *Look!*, esta interfaz es implementada por la clase *Renderer3D*.

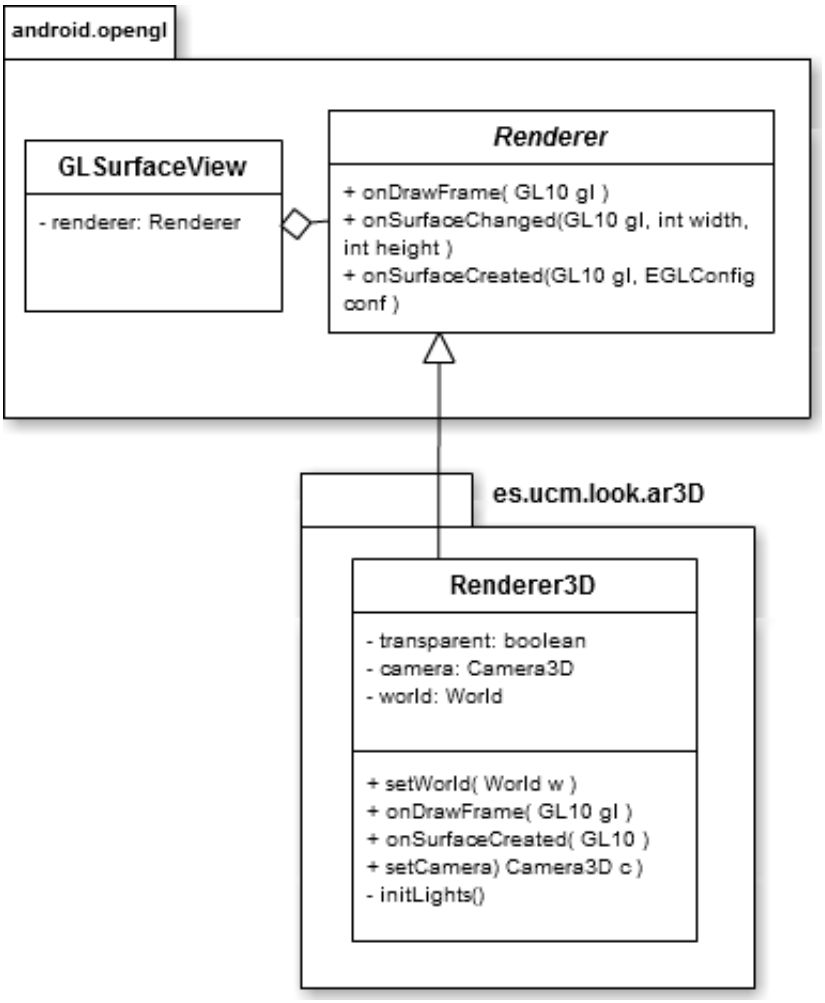

Figura 7.3: Relación de *GLSurfaceView* con *Renderer3D*

Tanto el dibujado 2D como el 3D se hace recorriendo la lista de *WorldEntity* proporcionada por un contenedor *World*. Como se detalló en secciones anteriores, las entidades *WorldEntity* contienen un *Drawable3D*. *Drawable3D* es una interfaz que define un único método *draw(GL10 gl)*. Éste método es utilizado en la renderización 3D de la entidad.

Una WorldEntity no tiene por qué definir una representación 3D (o 2D) concreta. En caso de no tenerla, se ignora y se pasa a la siguiente entidad.

7.2.1. Consideraciones con OpenGL ES 1.1

Antes de pasar a la jerarquía de clases utilizada, unas breves anotaciones resultado de la experimentación con esta versión de OpenGL ES.

OpenGL ES 1.1 está basada en la versión 1.5 de la versión de escritorio de OpenGL, sin embargo, existen diferencias fundamentales en cómo se realizan algunas de las tareas en cada una de las versiones:

- **Dibujado de primitivas**: Por cuestiones de eficiencia, OpenGL ES 1.1 carece de las primitivas *glBegin()* y *glEnd()* para el dibujado de primitivas. El dibujado se hace a partir de buffers que previamente alberguen los datos a dibujar (vértices, normales y coordenadas de texturas). Esta disposición dificulta la modificación de vértices concretos en tiempo de ejecución.
- **Clases de primitivas**: OpenGL ES 1.1 sólo permite el dibujado de puntos, líneas y triángulos. No de cuadriláteros y polígonos, como en su versión de escritorio.
- **Modo selección**: En la versión de escritorio, existe un modo de renderizado (llamado de selección), que permite extraer el objeto asociado a uno o varios píxeles de la pantalla. En este modo, cada píxel guarda información sobre el objeto del que forma parte, haciendo muy sencillo a posterori la identificación de los elementos asociados a un píxel cualquiera. OpenGL ES 1.1 carece de este modo, por lo que se necesitan métodos adicionales para la identificación de objetos pulsados.

7.2.2. Proyección de elementos de coordenadas reales a coordenadas OpenGL

Look! utiliza la proyección perspectiva en su representación 3D. Esta proyección, computa la distancia de los objetos hasta la cámara, y los escala acorde a ella, provocando sensación de profundidad.

Para la colocación de los objetos, se han tomado **escalas relativas** medidas en metros. Así, si un objeto se encuentra en la posición *(1, 5, 2)*, se encuentra en el punto situado a un metro al norte, cinco metros de altura y dos metros hacia el este.

Esta situación es definida en relación al origen de coordenadas del mundo, que es definido por el usuario, y que debe ser situado antes de la colocación de cualquier elemento. Este origen no está contenido de manera explícita en ningún módulo de *Look!*. Sólo es un punto orientativo que el programador debe utilizar cuándo localice elementos.

7.2.3. Situación de la cámara

El framework provee en el paquete *es.ucm.look.ar.ar3D.core.camera* dos clases que implementan las funciones de cámara necesarias en la realidad aumentada.

- ***Camera3D***: Una cámara estática. Ofrece funciones para cambiar su posición, así cómo las operaciones *pitch, rool* y *yaw*.

- ***OrientedCamera***: Derivada de la anterior, utiliza el proveedor de orientación *DeviceOrientation* para reajustar de manera dinámica su orientación acorde a la posición actual del dispositivo móvil.

7.2.4. Características OpenGL soportadas por *Look!*

A día de hoy, *Look!* implementa las siguientes características 3D:

- **Iluminación**: Puede configurarse si los elementos están iluminados (haciendo uso de iluminación por vértice, basada en normales) o no (*shadeless*), dónde el objeto se dibuja con el material seleccionado, sin verse afectado por la iluminación.

- **Texturas**: Pueden aplicarse texturas a aquellos elementos que tengan definidos coordenadas de textura. Estas texturas son creadas de manera automática y transparente para el programador, a partir de una dirección de fichero o un identificador de recurso, que es asignado a una representación 3D de una entidad. Las texturas se generan automáticamente con la la factoría *TextureFactory*.

- **Transformaciones afines**: Los elementos pueden definir traslaciones, rotaciones y escalas propias, que serán tenidas en cuenta durante el renderizado. La clase *Matrix3* guarda la matriz de transformación del elemento, y provee de métodos comunes de transformaciones, eliminando la necesidad de hacer llamadas explícitas a las funciones de transformación de OpenGL (glTranslate, glRotate, glScale).

En las siguientes secciones, se detallan estas características.

7.2.5. Dibujado de entidades: *Entity3D*

La clase *Entity3D* implementa *Drawable3D* y ofrece las funcionalidades básicas enumeradas en el apartado anterior.

Algunos de sus atributos:

- **Matriz de transformación**: La clase *Matrix4* representa una matriz de 16x16, utilizada para guardar las transformaciones afines (traslaciones, rotaciones y escalaciones) de la entidad 3D. Esta clase contiene métodos para generar estas transformaciones.

- **Material**: Se considera como material el color que será utilizado cuándo las caras de esta entidad sean renderizadas. *Color4* es una clase que contiene las cuatro componentes de color: rojo, verde, azul y transparencia.

- **Mesh3D**: De forma recursiva, Mesh3D es un *Drawable3D*. Contiene los vértices y las normales de una malla. Tras ser establecidos las transformaciones, el color y la textura, Entity3D procede a dibujar esta malla.

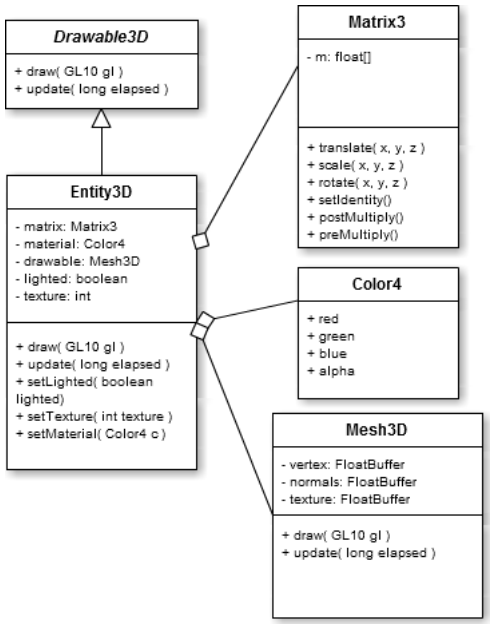

Figura 7.4: La clase *Entity3D* y sus componentes

7.2.6. ObjMesh3D: cargando mallas desde archivos .obj Wavefront

ObjMesh3D es una clase heredada de **Mesh3D**, la clase que define una malla de manera general. **ObjMesh3D** representa un tipo de malla que se construye a partir de un archivo OBJ.

OBJ es un formato de definición de mallas 3D, desarrollado originalmente por *Wavefront Technologies*[1]. Define los objetos 3D en forma de texto, de una manera sencilla y fácilmente procesable.

Un archivo *obj* tiene el siguiente aspecto:

```
v 1.000000 1.000000 0.000000
v 1.000000 -1.000000 0.000000
v -1.000000 -1.000000 0.000000
v -1.000000 1.000000 0.000000
vt 0.000000 0.000000
vt 1.000000 0.000000
vt 1.000000 1.000000
vt 0.000000 1.000000
vn 0.000000 0.000000 1.000000
f 1/1/1 4/2/1 3/3/1
f 1/1/1 3/3/1 2/4/1
```

Éste archivo está definiendo un cuadrado de lado 2, centrado en el origen. Dependiendo del prefijo que se encuentre al inicio de cada línea, los números subsiguientes representarán cosas distintas:

[1] http://www.wvfront.com/

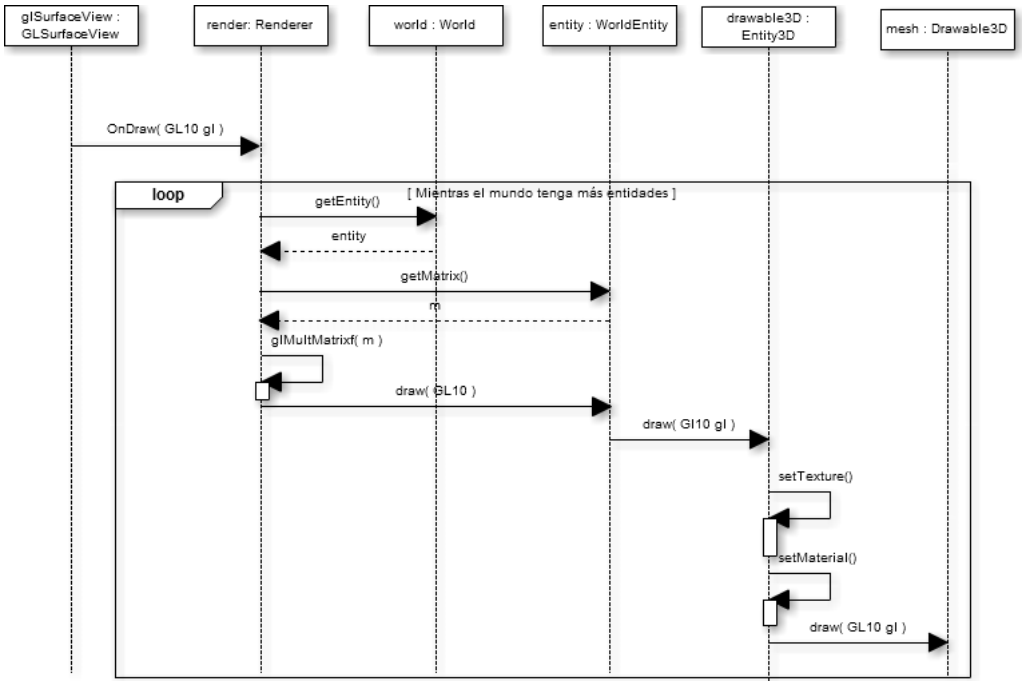

Figura 7.5: Diagrama de secuencia del dibujado 3d

- **v**: representa un vértice, formado por sus tres coordenadas x, y y z.
- **vt**: representa las coordenadas x e y de una textura.
- **vn**: representa un vector normal.
- **f**: representa una cara de la malla. Cada uno de las triplas que lo siguen representa un vértice de la cara. Éste vértice está formado por tres componentes: la primera es el índice del vértice; la segunda, el índice de la textura; y la tercera, el índice de la normal. Siendo el índice el número de orden con el que apareció en el archivo.

Esta clase de archivos pueden ser generados por la mayoría de programas de modelado 3D, como *3D Max*, *Blender*, o *Maya*. El único requisito a tener en cuenta en la exportación de *.obj es que las caras se exporten en forma de triángulos.

En la construcción de un objeto **ObjMesh3D**, se definirá el archivo OBJ que debe ser cargado en la malla, y automáticamente, la clase *MeshObjParser* procesará este archivo y creará una malla renderizable por *Look!*.

7.2.7. Creación de texturas: *TextureFactory*

Para el rellenado de primitivas, OpenGL utiliza dos cosas: colores y texturas. Para la generación de texturas, se utiliza la clase *TextureFactory*. Esta clase, que implementa el patrón *Singleton*, ejerce como caché de identificadores de texturas OpenGL.

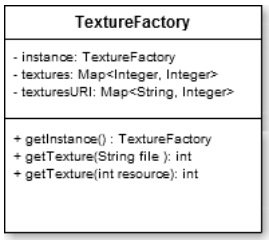

Figura 7.6: Clase TextureFactory

Fundamentalmente, ofrece dos métodos para la creación de texturas:

- **Textura a partir de un recurso local**: Esté método recibe un recurso contenido en la aplicación, cuyo identificador queda definido en los archivos autogenerados *R.drawable*, y devuelve la textura OpenGL asociada.
- **Textura a partir de ruta**: Recibe la ruta completa de un archivo de imagen y crea la textura OpenGL.

Nota: Para evitar problemas de compatibilidad entre dispositivos, las dimensiones de las imágenes utilizadas en la creación de texturas deben ser **potencias de dos**.

7.2.8. Funcionalidades geométricas y colisiones

En algunas aplicaciones, puede resultar beneficioso incorporar características como colisiones, o cálculos de intersección entre elementos u objetos.

Cómo característica adicional, en el paquete *es.ucm.look.ar.math* se ofrecen algunas funcionalidades relacionadas con el ámbito de la geometría y la física de colisiones.

El paquete *es.ucm.look.ar.math.geom* ofrece clases para realizar operaciones geométricas comunes en el espacio tridimensional: operaciones vectoriales, operaciones matriciales, intersecciones de rayos e intersecciones de esferas.

El paquete *es.ucm.look.ar.math.collision* (Figura 7.8) ofrece un esqueleto básico para definición de *armaduras* en elementos, y métodos para decisión de intersecciones con rayos.

Armature define una interfaz general que está asociada a objetos en tres dimensiones (como *Entity3D*). Tiene métodos para decidir si un punto está contenido dentro de la armadura, en que punto interseca un rayo, o si el rayo interseca.

Como clase base se ofrece *SphericalArmature*. Representa una armadura esférica que recubre un objeto en tres dimensiones, y define los métodos exigidos por la interfaz.

Nota: En la carga de mallas desde archivos *.obj se crean también parámetros para la definición de una armadura esférica para el objeto cargado, calculando el punto central de la malla y el radio mínimo que contiene a toda la figura.

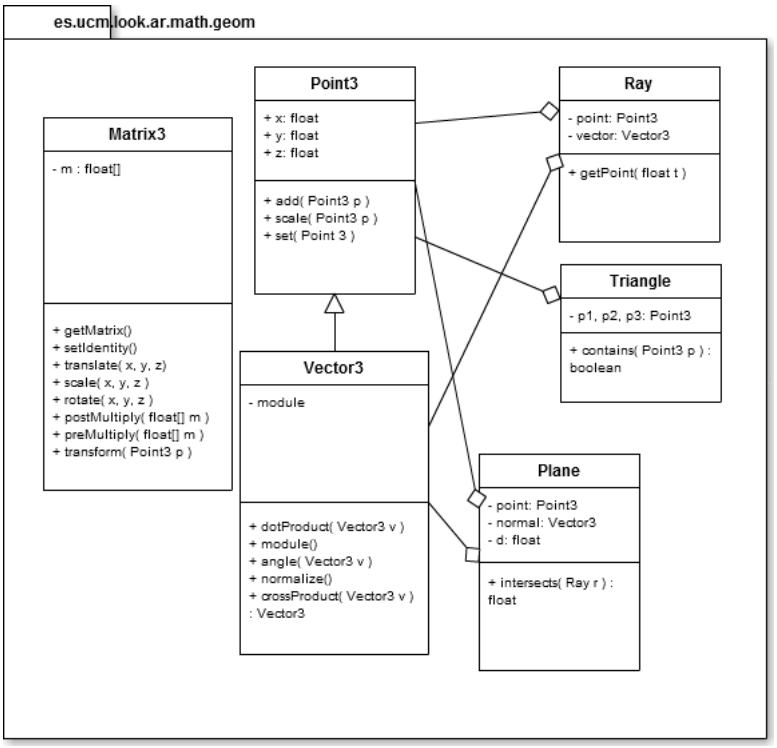

Figura 7.7: Contenido del paquete es.ucm.look.ar.math.geom

7.3. Integrando cámara y 3D

En *Look!* la capa de cámara esta representada por la clase *Preview*.

La capa está preparada para adaptarse a todo tipo de pantallas y autoconfigurarse sin necesidad de parámetros adicionales, así como para reaccionar ante los cambios de orientación del dispositivo.

Para que la capa pueda funcionar adecuadamente, ha de añadirse la siguiente línea de código al manifiesto de la aplicación (*AndroidManifest.xml*):

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

Para utilizar determinadas funciones ofrecidas por el sistema operativo, cómo el acceso a servicios de red, o la cámara, Android define una serie de permisos que el usuario debe conceder (o no) cuándo instala la aplicación.

Con esta línea incluida en el manifiesto, si el usuario accede a ello durante la instalación, la aplicación tendrá acceso a la cámara, y la capa *Preview* funcionará conforme a lo previsto.

Tanto la cámara como la representación en tres dimensiones están implementadas sobre la clase Android *SurfaceView*. Desde Android se recomienda no superponer *SurfaceViews*, por posibles problemas de compatibilidad. Sin embargo, actualmente no existe otra manera de combinar 3D e imágenes de la cámara.

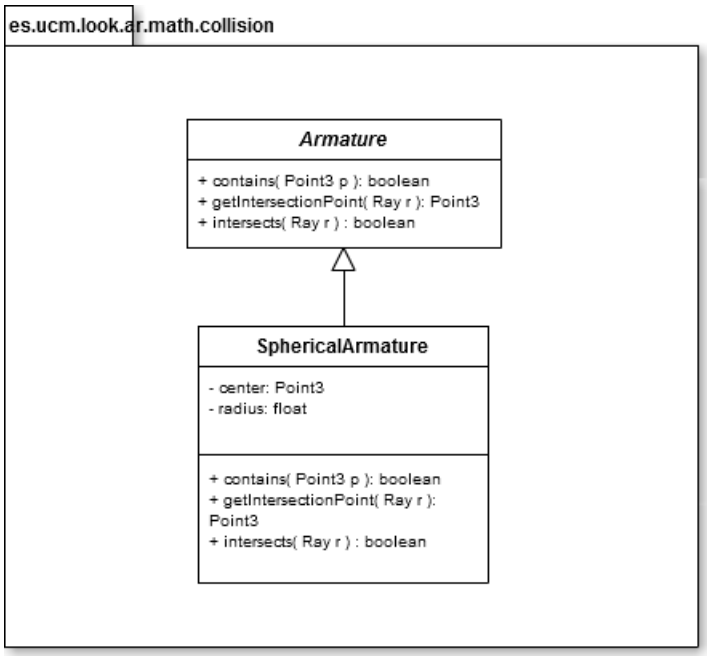

Figura 7.8: Contenido del paquete es.ucm.look.ar.math.collision

Para no añadir problemas adicionales, aunque desde la documentación oficial se recomienda utilizar una *SurfaceView* para dibujado 2D personalizado, se ha optado por utilizar una *View* personalizada.

Uno de los pasos más problemáticos de la interfaz de *Look!* ha sido la integración de cámara y 3D. La documentación oficial de Android no especifica nada al respecto, y la propia integración contradice la advertencia de Android en no intentar superponer más de una *SurfaceView* sobre otra.

Aún así, la experimentación de las diferentes posibilidades ha llevado a una solución que parece funcionar en la mayoría de dispositivos. A continuación, se muestra un fragmento del código de la creación de capas por su especial interés:

```
if (uses3D)
container.addView(glSurface);

if (usesCamera)
container.addView(preview);
```

Es importante notar que el método *addView* añade una vista encima de las que ya contuviera. Así, en este fragmento de código se añade la vista de cámara sobre la vista 3D, y el resultado, aunque el resultado esperado pudiera ser otro, es una superposición de la vista 3D sobre la cámara.

Este orden de adición de capas es el utilizado por algunas de las aplicaciones de realidad aumentada utilizadas como referencia[2][3].

[2] http://code.google.com/p/andar/
[3] http://www.mixare.org/

7.4. Capa 2D

En la siguiente sección se detalla las funcionalidades contenidas en *Look!* para la representación en dos dimensiones de las entidades contenidas en el mundo.

7.4.1. Canvas vs Proyección Ortogonal en OpenGL

A la hora de hacer dibujado en dos dimensiones, se nos presentaban dos posibilidades bien diferenciadas: utilizar la API 2D de dibujado (implementada por la clase *Canvas*) ofrecida por Android en todas sus vistas, o utilizar la proyección ortogonal para dibujado en 2D OpenGL, opción utilizada en muchos juegos y aplicaciones 3D.

En *Look!* se ha optado por utilizar la clase Canvas y la API 2D que ofrece, por las siguientes razones:

- **Funcionalidad**: El *Canvas* Android ofrece una rica cantidad de funciones: dibujado de formas (círculos, rectángulos, líneas), de textos, imágenes. Funciones de transformación (escalaciones, rotaciones, traslaciones), funciones de colores y estilos. Por lo ya explicado en secciones anteriores, muchas de estas tareas requieren de una complejidad excesiva en OpenGL ES, y otras son directamente inviables, como el dibujado de texto. Así, respecto a funcionalidad 2D ofrecida, el Canvas parece la mejor opción.
- **Eficiencia**: La renderización a través de la *GLSurfaceView* es más costosa que a través del Canvas. Además, en el framework, todas las capas son opcionales. Así, podría quererse una aplicación que sólo usara representación en dos dimensiones. Utilizando en esta capa el *Canvas*, tenemos el método más eficiente y apropiado.

7.4.2. SurfaceView vs. View personalizada

En la documentación oficial de Android se sugiere que para el dibujado en dos dimensiones personalizado, cómo el utilizado en videojuegos u otras aplicaciones interactivas similares, se ha de utilizar una *SurfaceView* y un hilo aparte para manejar las actualizaciones.

Sin embargo, también desde la documentación de Android, se sugiere que no sean superpuestas más de una *SurfaceView* sobre otra. En este punto, ya tenemos definidas dos capas en *SurfaceView*s: la de la cámara y la capa de representación en tres dimensiones.

Diferentes pruebas de posibles superposiciones de *SurfaceView*s, variando el orden de adición a la capa base, o configurando los parámetros de transparencia de cada uno de ellas, han mostrado comportamiento erráticos en distintos dispositivos. Y en todos ellos, los cambios de orientación del dispositivo provocaban la desaparición de al menos una de las capas, normalmente la cámara o el 3D.

En consecuencia y para intentar mantener la mayor compatibilidad posible, la capa de representación en dos dimensiones ha sido implementada en una *View* de Android personalizada. Se ha sobreescrito el método de dibujado y se ha creado un *TimerTask* que ejecuta el refresco de la vista tras un tiempo determinado.

La experimentación y las pruebas han demostrado que, en términos de eficiencia, esta solución ofrece un rendimiento similar al que ofrecería una *SurfaceView*.

7.4.3. Dibujado 2D

El dibujado en dos dimensiones (Figura 7.9) se realiza en la clase *AR2D* y sigue un esquema parecido al dibujado en tres dimensiones.

En tres dimensiones, la proyección de los elementos es realizada automáticamente por OpenGL. En dos dimensiones, sin embargo, hay que realizarla a mano. Es lo que realiza el primer bucle del diagarama de secuencia. Cómo se calcula la proyección se explicará después.

Una vez proyectados los elementos, tenemos una lista ordenada con todos las entidades proyectadas. Se hace un recorrido por todas ellas y, después de realizar las transformaciones de posición y escala debidas a la proyección, se dibuja su *Drawable2D*, si lo tuviere.

Proyección en dos dimensiones

Aún siendo una representación en dos dimensiones, las entidades del mundo se encuentran situadas en localizaciones en tres dimensiones. Necesitamos proyectar esas coordenadas de 3D a coordenadas 2D. Es de lo que se encarga el método *projecEntities* de la clase *AR2D*. La proyección se realiza con las siguientes fórmulas:

$$x_{proyectada} = distancia * x_{original}/z_{original} \tag{7.1}$$

$$y_{proyectada} = distancia * y_{original}/z_{original} \tag{7.2}$$

$$z_{proyectada} = z_{original} \tag{7.3}$$

La variable *distancia* es un factor corrector dependiente del tamaño de la pantalla y del ángulo de visión definido.

Además, debemos encargarnos del control de profundidad. Primero han de dibujarse los elementos más alejados del usuario, así los más cercanos quedarán superpuestos sobre ellos. Por ello, tras realizar la proyección de los elementos, se realiza una ordenación conforme a su coordenada z proyectada.

Después, los dibujamos conforme al orden establecido, y además utilizamos ese valor para escalar el elemento, conforme a la distancia que se encuentre del usuario, dando así una mayor sensación de realismo.

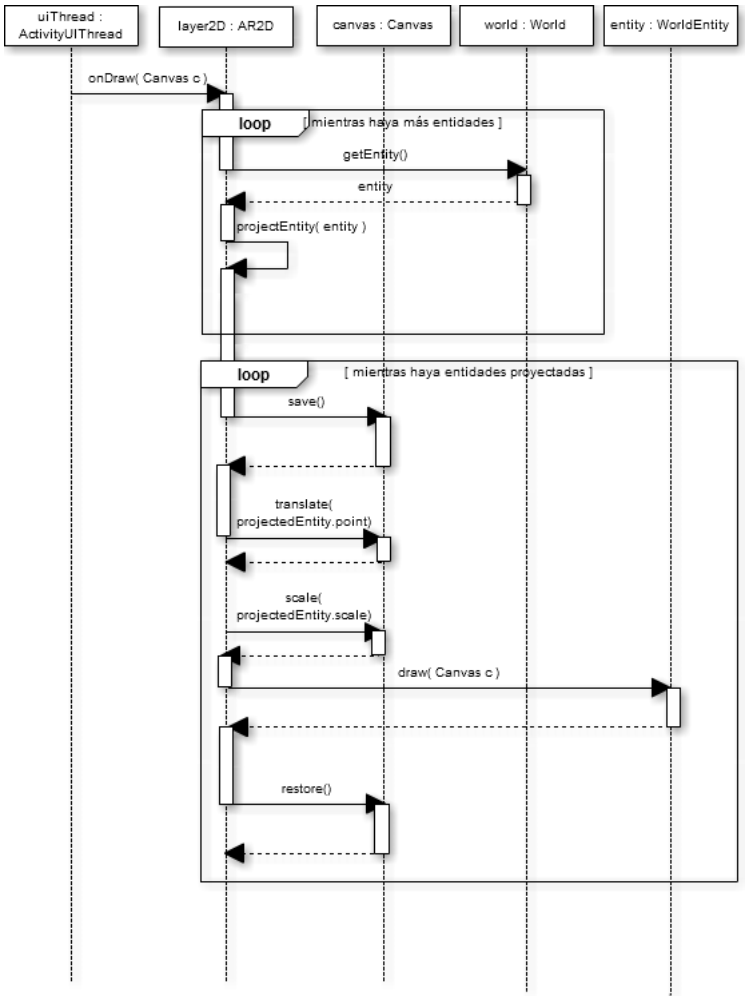

Figura 7.9: Diagrama de secuencia del dibujado en dos dimensiones

7.5. Animaciones

Las representaciones en dos y tres dimensiones admiten animaciones a través del método *update*, que es invocado con el tiempo, en milisegundos, que transcurrió desde la última actualización.

El método es invocado antes del dibujado del elemento, y está pensado para que, en su implementación, actualice el estado del elemento, creando así las animaciones.

El estado del elemento puede ser muy variado, desde la rotación de una malla 3D que cambia con el tiempo, hasta la actualización del frame actual de una serie de imágenes.

7.6. Interacción del usuario

Necesitamos interactividad en las aplicaciones de realidad aumentada desarrolladas, por ejemplo, para que se nos muestre un texto por pantalla cuando pulsemos un elemento, o que se añada una nueva entidad al mundo cuándo pulsemos una tecla. Para ello, hemos de definir una serie de interacciones que puedan ser realizadas por el usuario.

Además de los eventos generales de teclado, que son procesados por las cada una de las capas en su conjunto, se cuenta con una serie de interacciones que pueden ser realizadas sobre las entidades representadas en la realidad aumentada. *Look!* define dos tipos:

- **Táctiles**: Reflejadas en la interfaz *TouchListener*. Procesa los tres tipos principales de acciones táctiles de pantalla: posar el dedo, levantar el dedo, y mover el dedo por la pantalla.
- **Cámara**: Una acción de cámara es ejecutada cuándo el usuario apunta directamente con el centro del dispositivo a un elemento. Está acción está reflejada en la interfaz *CameraListener*, que contemplados dos tipos de eventos: que la cámara enfoque al objeto (*onCameraEntered*) y que la cámara salga del evento (*onCameraExited*).

Para versiones futuras, se plantean otros tipos de interacciones, como por ejemplo acciones que se ejecutan por proximidad a objetos.

7.6.1. Interacción por teclado

Para la interacción por teclado, *Look!* se vale de los métodos proporcionados por la vistas de Android.

La clase *View* permite dos tipos de interacción por teclado: reescribiendo el método *onKeyPressed* y fijando el comportamiento deseado para cada una de las teclas; o asignando un *OnKeyListener*[4] a la vista, que será ejecutado cuándo reciba un evento de teclado.

Look! no ofrece ninguna funcionalidad adicional para este tipo de interacción, nativa en Android. Permite el procesamiento de teclado en cada una de las capas de realidad aumentada, pero esta debe ser añadida por el programador de la manera habitual[5].

7.6.2. Interacción táctil

Antes de pasar al método utilizado para decidir qué entidad recibe un evento táctil, cabe preguntarse, cuándo, realmente, una entidad es tocada.

En *Look!* una entidad es un elemento con una posición, que puede tener una representación en dos dimensiones y/o tres dimensiones (o ninguna). Si queremos definir eventos táctiles sobre él (pulsar el elemento en la pantalla del dispositivo y qué suceda algo), debemos decidir qué áreas de la pantalla deben ser pulsadas para lanzar esos eventos táctiles.

[4]Interface OnKeyListener: http://developer.android.com/reference/android/view/View.OnKeyListener.html
[5]Handling UI Events: http://developer.android.com/guide/topics/ui/ui-events.html

Recordemos, cómo se explicó en el apartado 7.2 dedicado a la capa 3D, que OpenGL ES carece de modo de selección, lo que, en el caso de querer determinar en la capa 3D a que objeto pertenece un píxel pulsado, obligaría a crear algún tipo de sistema adicional para determinar el elemento seleccionado.

Por ejemplo, se podría lanzar un rayo desde el punto dónde se tocó la pantalla hacia el mundo 3D, teniendo en cuenta la perspectiva y la distancia focal de la cámara para determinar la dirección correcta del rayo, y ver si colisiona con algún *Armature*.

En el paquete *es.ucm.ar.math* se ofrecen funcionalidades para montar un sistema basado en intersecciones, que puede ser útil en algunos tipos de aplicaciones.

Sin embargo, esta aproximación es muy costosa a nivel de CPU, debido a todos los cálculos geométricos requeridos. Para el caso general, se utilizará un método menos costoso y más sencillo, inspirado en el modo de selección de la versión de escritorio de OpenGL.

Buffer de colores

Todas las entidades tienen un identificador **único**, representado por un entero. Ese identificador, puede ser convertido a un número hexadecimal, y a su vez, ese hexadecimal, reconvertido a un color.

En un búfer, del mismo tamaño que la pantalla, cada uno de los elementos visibles dibujará su área táctil (en la que, si el usuario pulsa, lanzará el evento táctil en el elemento) de un único color, el obtenido a partir de su identificador. (Figura 7.11).

Cuándo el usuario realice el evento táctil, se consultará en el búfer el color del píxel pulsado. Este color, por proceso inverso, será convertido en el identificador de la entidad.

Se extraerá la entidad del mundo a partir del identificador, y se le comunicará la realización del evento, para su posterior procesamiento.

Todo el proceso de conversión de identificadores a colores y viceversa es totalmente transparente para el programador, así cómo el pintado del búfer.

El dibujado del *área táctil* es realizado por el método *drawTouchableArea* definido por *WorldEntity*. Por defecto, el área táctil es un círculo centrado en la posición de la entidad. Su radio es calculado a partir de sus representaciones gráficas. En el caso del 3D, se suele utilizar el radio de la esfera que envuelve a la malla, escalado por la distancia al usuario. Si se cuenta con una representación en dos dimensiones, se suele utilizar como área táctil el área dibujada por *Drawable2D*.

7.6.3. Interacciones de cámara

Look! considera producida una interacción de cámara cuándo un elemento de la realidad aumentada gana o pierde el foco de la cámara.

Esto significa que cuándo un elemento ocupe el espacio central de la pantalla del dispositivo, se considerará que ha recibido el foco del usuario. El elemento será notificado y él mismo ejecutará los *listener*s asociados al evento.

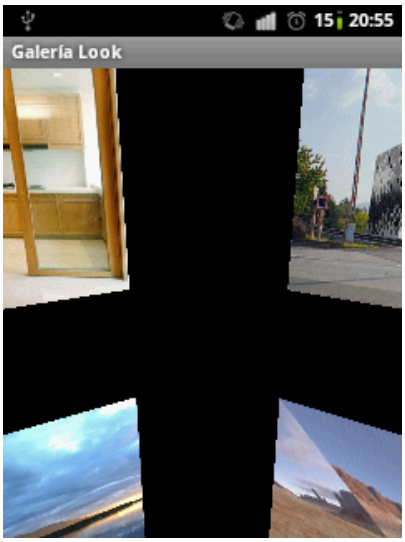

Figura 7.10: Apariencia de la aplicación en modo normal. Existen cuatro entidades (cuatro imágenes) en cada una de las esquinas

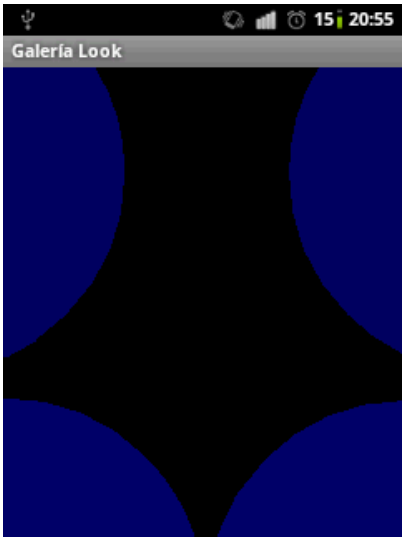

Figura 7.11: Apariencia del buffer táctil de la aplicación anterior. Pueden verse las cuatro áreas táctiles, cada una de un color distinto (distintos tonos de azul)

Se consideran dos tipos de eventos: uno cuándo el elemento gana el foco del usuario; y otro cuándo lo pierde.

Para el cálculo del elemento que se encuentra en la posición central del dispositivo, se usa el mismo método que para la interacción táctil, descrita en el apartado anterior, sólo que en vez de coger el color del píxel que devolvió el evento táctil, se selecciona el píxel situado en el centro de la pantalla (ancho de la pantalla / 2, alto de la pantalla / 2).

7.6.4. HUD en 2D

Además de la capa HUD, la capa 2D ofrece una semicapa adicional a la que pueden ser añadidos elementos de HUD en dos dimensiones, que serán dibujados en la capa más alta de la aplicación, sólo superadas por el HUD de vistas de Android que se explicará después.

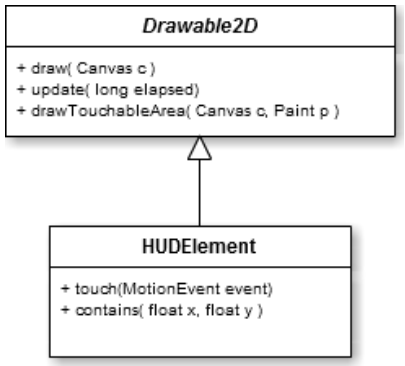

Figura 7.12: Apariencia del buffer táctil de la aplicación anterior. Pueden verse las cuatro áreas táctiles, cada una de un color distinto (distintos tonos de azul)

Estos elementos implementan la interfaz *HUDElement* (Figura 7.12), que define métodos de dibujado, determinación de si un punto se encuentra dentro de ellos y métodos de procesamientos de eventos táctiles.

Es una manera de dibujar elementos de HUD, aprovechando la potencia de la API 2D ofrecidas por el Canvas. Si por ejemplo, se quiere dibujar un círculo en el centro de la pantalla, que nos que el punto exacto hacia dónde apuntamos con el dispositivo, sólo deberíamos añadir un *HUDElement* que dibujara el círculo a través del método *addHUDElement* de la clase *AR2D*.

7.7. Capa HUD

Por último, en la capa superior, encontramos la capa HUD, pensada para albergar todo los tipos de vistas ofrecidas por android.

En esencia, esta capa es un contenedor (*FrameLayout*) a la que se pueden añadir *View*s de Android: botones, campos de texto, scrolls, etc.

Está pensada, por ejemplo, para añadir botones que produzcan acciones en la aplicación. O textos que muestren datos concretos, como la posición actual del usuario.

También puede ser combinado con las interacciones de entidades. Por ejemplo, mostrar un texto (en una *TextView* [6]) concreto cuándo una entidad reciba el foco de la cámara, o mostrar una imagen (en una ImageView *ImageView* [7]) cuándo se pulse un elemento.

7.8. Utilidades: LookARUtil

Adicionalmente, *Look!* ofrece una clase de utilidades relacionados con el módulo de realidad aumentada que pueden ser utilizadas desde cualquier punto del código. Sus funciones son:

- ***getApp***: Devuelve el contexto *LookAR* de la actividad. Éste contexto es necesario para lanzar ciertos métodos, como por ejemplo los referidos a cambios en la interfaz.
- ***getDisplay***: Devuelve el *Display* de la aplicación. Éste objeto tiene información sobre el dispositivo sobre el que se esta ejecutando la aplicación, como por ejemplo las dimensiones de la pantalla.
- ***makeFloatBuffer(float f[])***: Crea un buffer de floats a partir de un array de floats. Utilizado en la definición de mallas 3D.
- ***getView(int resource, ViewGroup view)***: Construye la vista asociado al recurso pasado como parámetro. Se utiliza para crear vistas desde archivos XML.
- ***getMinimumSize()***: Devuelve el valor de la menor de las dimensiones de la pantalla.

[6] http://developer.android.com/reference/android/widget/TextView.html
[7] http://developer.android.com/reference/android/widget/ImageView.html

Capítulo 8

Construyendo aplicaciones con *Look!*

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

8.1. Planteando la aplicación

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

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

1. Qué **tipos de entidades** van a ser representadas en la realidad aumentada.
2. Qué **características** definen estas entidades.
3. **De dónde se obtienen esas entidades y sus características**. Podría accederse a un **servidor remoto**, si los datos cambian de manera dinámica; o albergarse de manera **local**, si no son susceptibles de cambios, o los datos no son compartidos.
4. **Cómo se representarán gráficamente**. Pudiendo ser dos y/o tres dimensiones.
5. Qué **interacciones** serán permitidas para cada entidad:
   - Efectos al pulsar
   - Efectos al arrastrar
   - Efectos al soltar

- Efectos al enfocar con la cámara

6. Si queremos que aparezca de fondo la imagen obtenida por **cámara**.
7. **Si es necesario localizar al usuario** para obtener la funcionalidad buscada.
8. Y de ser así, qué **tipo de localización** sería la adecuada: **Relativa**, con el sistema inercial; O **absoluta**, con el sistema de localización WiFi.
9. Qué **sistema de referencia** se utiliza para situar a los elementos y al usuario.
10. Dónde está el **origen de coordenadas** del mundo.
11. **Si se necesitan añadir elementos extras de interfaz** (botones, menús, cajas de texto, *Activity*s secundarias, etc.) para completar la funcionalidad requerida.

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

8.2. Codificando una aplicación básica

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

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

8.2.1. Creando la Activity principal

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

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

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

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

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

```
public class MyARActivty extends LookAR {

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

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


}
```

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

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

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

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

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

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

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

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

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

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

        // Updates the recent added data
```

```
        LookData.getInstance().updateData();

}
```

Primero creamos un elemento vacío, y lo situamos en la posición (10, 0, 0). Después, lo añadimos al módulo de datos, y finalmente actualizamos para que se muestren los cambios realizados. Si ahora ejecutamos, veremos que, tras orientar la cámara en la dirección adecuada, aparecerá un cubo con un cuadro de texto, que indica el identificador de la entidad y su tipo, en este caso null, puesto que no se definió la propiedad (Figura 8.1).

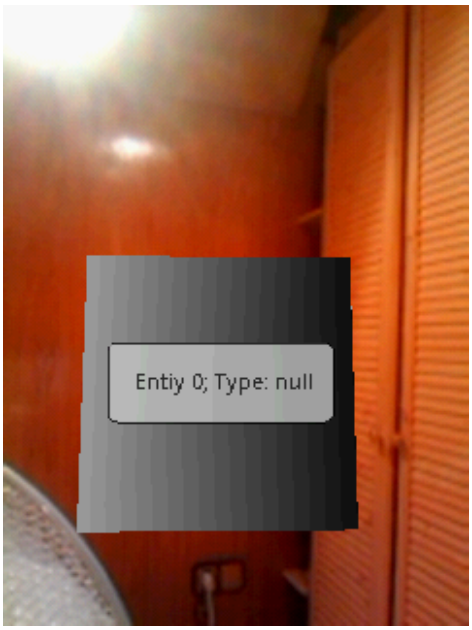

Figura 8.1: Captura de la aplicación tras añadir un elemento

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

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

8.2.3. Definiendo factorías de elementos

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

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

El código para la factoría sería el siguiente:

```
public class MyWorldEntityFactory
                        extends WorldEntityFactory {

        public static final
                String NAME = "name";

        public static final
                String COLOR = "color";

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

                Entity3D drawable3d
                        = new Entity3D( new Cube( ) );
                String color = data.
                        getPropertyValue(COLOR);
                if (color.equals("red")){
                        drawable3d.
                                setMaterial(new
                                Color4(1.0f, 0.0f, 0.0f));
                }
                else if ( color.equals("green"))
                        drawable3d.
                                setMaterial(new
                                Color4(0.0f, 1.0f, 0.0f));
                //...

                we.setDrawable3D(drawable3d);
                return we;
        }

}
```

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

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

```
LookData.getInstance()
                .setWorldEntityFactory(
```

```
                                new MyWorldEntityFactory());

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

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

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

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

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

El resultado de la ejecución de esta aplicación es el mostrado por la figura 8.2.

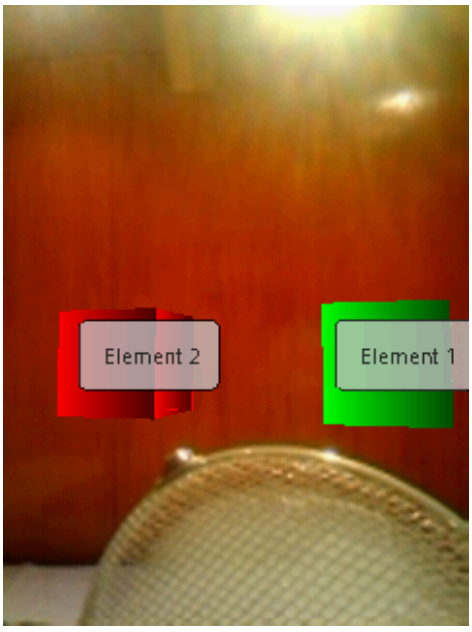

Figura 8.2: Captura de la aplicación con la nueva factoría

8.2.4. Añadiendo interacciones

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

Añadiendo procesado de eventos táctiles: TouchListener

Cuando una entidad reciba un evento táctil, pasará ese evento a todos los *listeners* de su lista de *TouchListener*. Cómo puede verse en la figura 8.3, la interfaz define métodos para los tres tipos de eventos táctiles principales. Cada uno de los métodos recibe como parámetro la entidad que recibió el evento y las coordenadas de pantalla dónde sucedió.

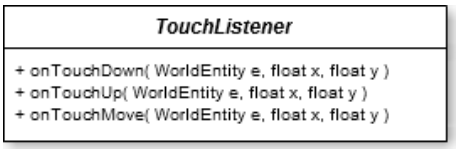

Figura 8.3: La interfaz *TouchListener*

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

Para lograrlo, implementaríamos el siguiente *TouchListener*:

```
public class MyTouchListener implements TouchListener {

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

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

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

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

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

La figura 8.4 muestra una captura de la aplicación, tras tocar uno de los elementos. Puede apreciarse como el *TouchListener* ejecutó su lógica y trasladó la entidad.

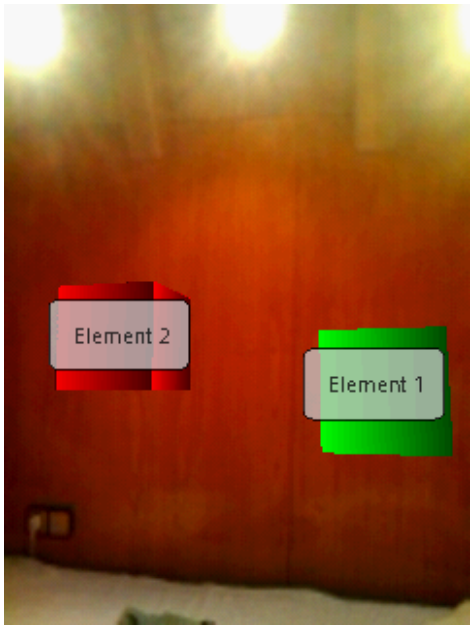

Figura 8.4: Captura de la aplicación tras pulsar uno de los elementos

Añadiendo eventos de cámara *CameraListener*

Look! considera un evento de cámara cuándo el usuario enfoca (o deja de enfocar) directamente una entidad con el centro de su dispositivo. Cómo muestra la figura 8.5, *CameraListener* cuenta con dos eventos: uno para cuándo el usuario enfoca la entidad, y otro para cuándo deja de enfocarla.

Figura 8.5: La interfaz *CameraListener*

Es decir, si colocamos el dispostivo tal que una entidad ocupe el centro de la pantalla, se ejecutará el método *onCameraEntered*. Cuándo, tras mover el

dispositivo, la entidad abandone la zona central, se lanzará el método *onCameraExited*. Ambos reciben como parámetro el objeto que recibió del evento.

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

```
public class MyCameraListener
        implements CameraListener {

        private static SquarePrimitive square
                = new SquarePrimitive();

        private Drawable3D oldDrawable;

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

        }

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

}
```

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

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

La figura 8.6 muestra una captura de la aplicación, mientras el usuario enfoca directamente a una de las entidades. Puede apreciarse como el *CameraListener* ejecutó su lógica y cambió la malla 3D del elemento.

8.2.5. Añadiendo un HUD a la aplicación

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

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

Lo primero que deberíamos hacer sería asegurarnos de que la capa HUD está configurada a *true*, en la llamada al constructor de *LookAR* desde *MyA-*

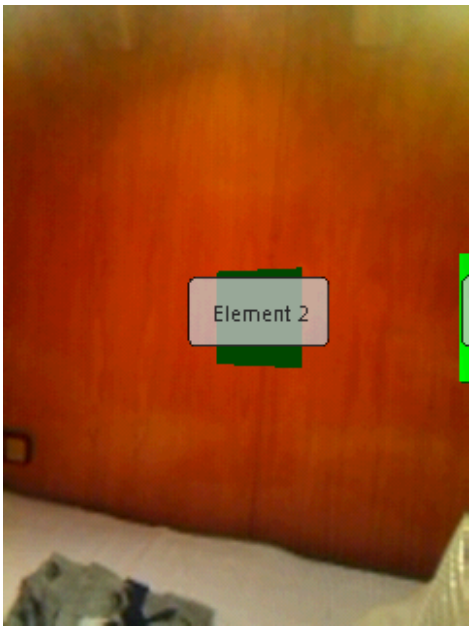

Figura 8.6: Captura de la aplicación mientras se enfoca directamente a uno de los elementos. Puede apreciarse como su representacion 3D ha cambiado de un cubo a un plano.

RActivity. Después, definimos una vista con un layout XML de Android, que contenga lo que buscamos para nuestro HUD (en este caso, un único botón):

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    >
    <Button android:text="Button"
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content">
    </Button>
</LinearLayout>
```

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

```
@Override
public void onCreate(Bundle savedInstanceState) {
        // ...
        ViewGroup v = this.getHudContainer();
```

```
        v.addView(LookARUtil.getView(R.layout.main, null))
}
```

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

El resultado de la ejecución es el mostrado por la figura 8.7.

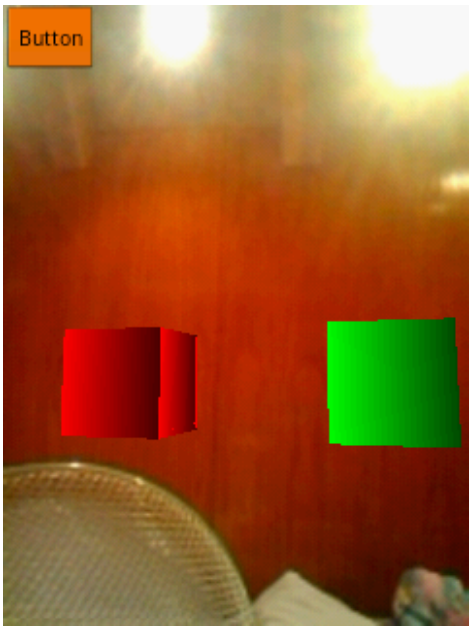

Figura 8.7: Captura de la aplicación con HUD. Podemos comprobar arriba a la izquierda como se añadió el botón.

8.2.6. Configurando una base de datos

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

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

Extendiendo el ContenProvider

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

```
public class MyContentProvider
        extends LookSQLContentProvider {

        public MyContentProvider() {
                super(''mydatabase.db'',
```

```
                ''es.ucm.myaractivity.contentprovider'');
        }

}
```

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

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

Deberíamos añadir la siguiente línea a AndroidManifest.xml:

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

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

Incorporando al DBDataHandler

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

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

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

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

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

8.3. Preparando el Sistema de Localización

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

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

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

8.3.1. Definición de Nodos y Puntos de Acceso

Definición de Nodos

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

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

En la figura 8.8 se muestra un ejemplo de selección y etiquetado de nodos. El fichero resultante se muestra a continuación:

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

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

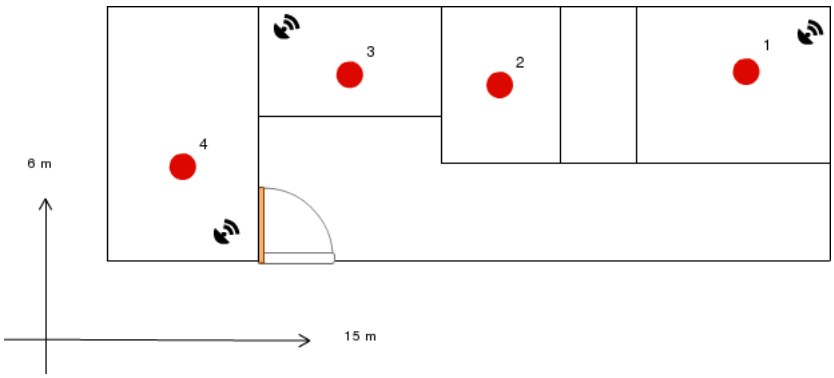

Figura 8.8: Ejemplo de Definición de Nodos

Definición de Puntos de Acceso

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

Esto paso puede llevarse a cabo de dos formas:

[1] Se pueden definir varios nodos para una misma localización, de forma que la captura de datos se realice en diferentes puntos de, por ejemplo, una misma habitación. Todos los nodos tendrían las mismas coordenadas aunque diferente id. Esto permite aumentar la precisión al utilizarse un mayor número de muestras en la inferencia de la posición.

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

Definición Manual

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

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

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

Detección automática

Tal y como se muestra en la figura 8.9, *CNWifi* proporciona un método de detección automática de puntos de acceso y creación del fichero correspondiente.

Figura 8.9: CCNWifi: Detección automática de Puntos de Acceso

8.3.2. Captura de Datos

El siguiente paso es la captura de datos en cada uno de los nodos definidos para la elaboración del mapa de radio. Para ello, *CNWifi* proporciona un sistema automático de captura de datos, como se aprecia en la figura 8.10.

Para llevar a cabo la captura de datos, el usuario deberá colocarse en cada uno de los nodos definidos en el paso 8.3.1, indicar su id y pulsar en el botón *Entrenar*. Cuando haya terminado la captura de todos los nodos, el usuario deberá pulsar el botón *Guardar* para que los datos sean procesados y se genere el archivo de mapa de radio.

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

Figura 8.10: CCNWifi: Captura de Datos

Figura 8.11: CCNWifi: Probando la localización

8.3.3. Probando la Localización por Wifi

CNWifi proporciona un módulo para probar la localización, mostrando la información de la ubicación actual en tiempo real. Este módulo puede verse en la figura 8.11 y permite asegurarnos de que la selección de nodos es la adecuada y que el sistema de localización funciona correctamente.

8.3.4. Ajuste de Parámetros del Sistema de Navegación Inercial

Factor de Orientación

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

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

Escala

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

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

8.4. Integrando localización

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

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

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

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

```
LocationManager location = new LocationManager(
        this, true, false);
```

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

```
location.start();
...
Point3 posicion = location.getPosition();

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

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

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

Capítulo 9

Experimentación: Aplicaciones de ejemplo desarrolladas con *Look!*

Este capítulo pretende dar una idea general al lector de cómo se construyen aplicaciones de complejidad media con *Look!*, desarrollando la aproximación general, enfocada a los elementos utilizados del framework, pero sin entrar en detalles concretos de cada aplicación, relacionados con de programación de lógica y otros procedimientos no fundamentales de *Look!*.

Para ello, se estudian varias aplicaciones desarrolladas durante el transcurso del proyecto como ejemplos de uso del framework. Por cada aplicación, se da una breve descripción de la misma, se detallan los módulos incorporados de *Look!* para su implementación, las propiedades de los elementos que contienen, las representaciones gráficas, las interacciones para los elementos y el uso de localización, además de otros detalles particulares de cada aplicación, como el uso del HUD o la persistencia. Cada uno de estos apartados va acompañado de partes representativas del código desarrollado.

9.1. Mundo 3D

Mundo 3D es un mundo virtual tridimensional en el que el usuario puede moverse físicamente e interactuar con los objetos virtuales. Para avanzar en el mundo virtual el usuario debe caminar en el mundo real, y para mover la cámara subjetiva deberá cambiar su orientación, por lo que la simulación de realidad virtual es completa. Cada uno de los objetos virtuales presentes en el mundo interactúan de manera diferente cuando son tocados, y muestra un mensaje distinto cuando son enfocados por el usuario.

En la figura 9.1 se observa el mensaje mostrado cuando el usuario enfoca un objeto en el mundo virtual.

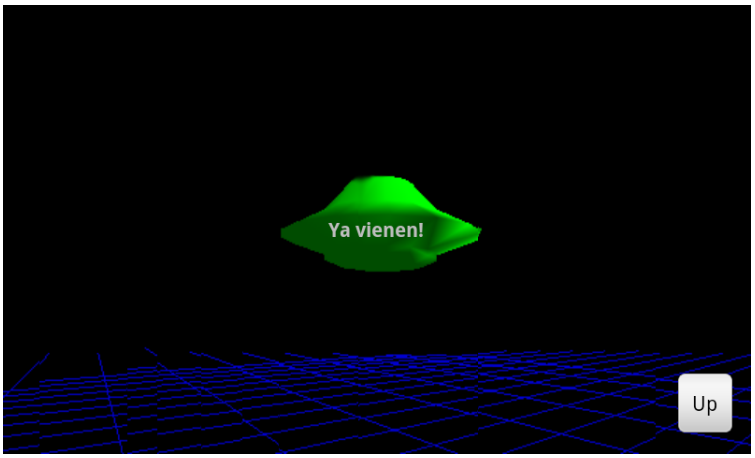

Figura 9.1: Captura de Mundo 3D

9.1.1. Módulos incorporados

Esta aplicación utiliza la capa de tres dimensiones del módulo realidad aumentada, y para detectar el movimiento del usuario el sistema de navegación inercial del módulo de localización.

9.1.2. Definición de los EntityData

En *Mundo 3D* se define un tipo de elemento para cada uno de los objetos distintos a representar. Existen tres propiedades comunes: *Name*, que indica el nombre del objeto, *Color*, que indica su color, y *Message*, que representa el mensaje a mostrar en pantalla cuando el objeto es enfocado.

Por ejemplo, la creación de la entidad *ufo* se hace de la siguiente forma:

```
EntityData ufo = new EntityData("ufo");
ufo.setLocation(10, 0, 10);
ufo.setPropertyValue(
        MundoVirtualWorldEntityFactory.NAME, "ufo");
ufo.setPropertyValue(
        MundoVirtualWorldEntityFactory.COLOR, "green");
ufo.setPropertyValue(
        MundoVirtualWorldEntityFactory.MESSAGE,
        "Ya vienen!");
LookData.getInstance().getDataHandler().addEntity(ufo);
```

Existe un tipo especial de entidad, *grid*, que representa el suelo y se debe colocar en la posición (0,10,0):

```
EntityData grid = new EntityData("grid");
grid.setLocation(0, 10, 0);
grid.setPropertyValue(
        MundoVirtualWorldEntityFactory.NAME, "grid");
grid.setPropertyValue(
        MundoVirtualWorldEntityFactory.COLOR, "blue");
```

9.1.3. Factoría de WorldEntity

La clase *MundoVirtualWorldEntityFactory* se encarga de la conversión de *EntityData* a *WorldEntity*. En función del tipo de objeto, carga la malla correspondiente y el color definido en su propiedad *Color*. Además, añade los listener necesarios para cada tipo de objeto para que puedan recibir eventos touch y de cámara:

```
if (data.getType().equals("ufo")) {
        we = new ObjectWorldEntity(data);
        Entity3D drawable3d = new Entity3D(ufo);
        String color = data.getPropertyValue(COLOR);

        if (color.equals("red")) {
                drawable3d.setMaterial(
                        new Color4(1.0f, 0.0f, 0.0f));
                ...
        we.setDrawable3D(drawable3d);
        we.addTouchListener(
                new ObjectTouchListener());
        we.addCameraListener(
                new ObjectCameraListener());
```

Se ha definido la clase *ObjectWorldEntity*, heredera de *WorldEntity* y que incorpora funciones para hacer girar los objetos durante un tiempo dado.

```
public void update(long elapsed) {
...
if (girando) {
        if (giro < 1000) {
                ((Entity3D) this.getDrawable3D())
                .getMatrix().rotate(0.0f,
                        (0.15f/(elapsed/2))*100, 0.0f);
                giro++;
        } else {
                giro = 0;
                girando = false;
        }
}
```

9.1.4. Interacciones

Se han añadido dos tipos de interacción: eventos de cámara y eventos touch. Para cada uno de ellos se ha creado una clase que implementa la respuesta a los eventos. Por ejemplo, la clase *ObjectTouchListener* implementa la respuesta los eventos touch haciendo que los objetos se eleven y giren al ser tocados:

```
public boolean onTouchDown(
        WorldEntity e, float x, float y) {

        Point3 p = e.getLocation();
        LookData.getInstance().getDataHandler().
            updatePosition(e.getData(),
            p.x, (p.y)-1, p.z);
        ((ObjectWorldEntity) e).girar();
        return true;
}
```

La clase *ObjectCameraListener* implementa la respuesta a los eventos de cámara, haciendo que se muestre un mensaje cuando el usuario enfoca un objeto del mundo:

```
public void onCameraEntered(WorldEntity entity) {
        TextView tv = (TextView) LookARUtil.
            getApp().findViewById(R.id.mensaje);
        tv.setText(
                entity.getData().getPropertyValue(
                        "message"));
        tv.setVisibility(TextView.VISIBLE);

}

public void onCameraExited(WorldEntity entity) {
        TextView tv = (TextView) LookARUtil.
            getApp().findViewById(R.id.mensaje);
        tv.setVisibility(TextView.INVISIBLE);
}
```

9.1.5. Localización

Mundo 3D hace uso del sistema de navegación inercial incluido en el módulo de localización para detectar cuándo la persona esta caminando y aplicar desplazamiento.

Para ello, en primer lugar se inicializa el sistema de navegación inercial mediante la API de localización *LocationManager* y se consulta si la persona está caminando a intervalos de tiempo fijos mediante un Timer.

```
location = new LocationManager(
        this.getApplicationContext(), true, false);
location.start();
timer = new Timer();
TimerTask timerTask = new TimerTask() {
        MundoVirtualWorld w = (MundoVirtualWorld)
            LookData.getInstance().getWorld();

            public void run() {
```

```
                if (location.isWalking()) {
                    w.up();
                }
            }
        };
        timer.scheduleAtFixedRate(timerTask, 0, 200);
}
```

Para aplicar el desplazamiento, se crea la clase *MundoVirtualWorld*, que hereda de *World*, y se define el método *up()* que actualiza la posición del usuario dentro del mundo virtual avanzando en la dirección correspondiente[1].

```
public void up() {
        if (!keepUp) {
        float azimuth = DeviceOrientation.
            getDeviceOrientation(context).azimuth;
        float x = LookData.getInstance().getLocation().x
            + DISTANCE * (float) Math.sin(azimuth);
        float y = LookData.getInstance().getLocation().y;
        float z = LookData.getInstance().getLocation().z
            + DISTANCE * (float) Math.cos(azimuth);
        LookData.getInstance().getLocation().set(x, y, z);
        }
}
```

9.2. Galería Look!

Galería Look! es una galería de imágenes interactiva en tres dimensiones. Las imágenes se reparten en el entorno tridimensional alrededor del usuario. Con el sistema de orientación, el usuario puede girar en todas las direcciones, para localizar las imágenes.

Con gestos de arrastre verticales y horizontales, puede pasarse de una fila de imágenes a otra y hacer girar la columna de imágenes, respectivamente.

Cuando el usuario pulsa una de las imágenes, ésta se muestra a tamaño completo en primer plano, y muestra información relacionada, si la tuviera.

Las figuras 9.2 y 9.3 muestran capturas de la aplicación en ejecución.

9.2.1. Módulos incorporados

Esta aplicación utiliza el módulo de datos sin persistencia. Del módulo de realidad aumentada, utiliza las capas de dos y tres dimensiones. Esta aplicación no utiliza el módulo de localización.

9.2.2. Definición de los EntityData

Para *Galeria Look!* se ha definido un único tipo de elementos: **Imagen**; con una única propiedad: *la ruta al archivo de imagen mostrado.*

[1]La dirección se obtiene a partir de la información de orientación que se mantiene en la clase *DeviceOrientation*.

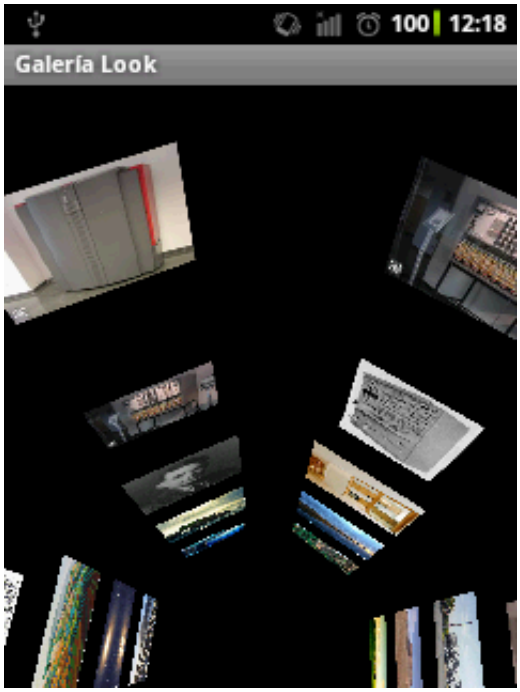

Figura 9.2: Captura de Galería Look!

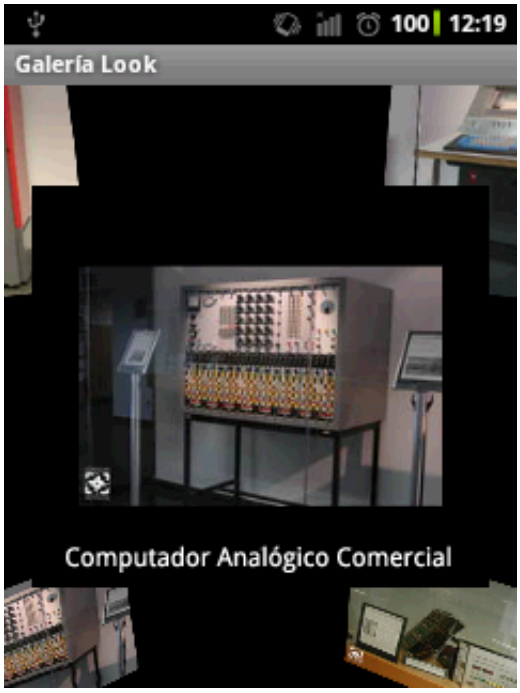

Figura 9.3: Captura de Galería Look!, con una imagen seleccionada

La carga de datos se hace desde una carpeta llena de imágenes. Una vez obtenidos las rutas de los archivos, la creación de elementos se consigue con el siguiente código:

```
for ( String file: files ){
        EntityData imageData = new EntityData( );
        imageData.
                setPropertyValue(
                ImageEntityFactory.IMAGE, file );
        imageData.
                setPropertyValue(
                ImageEntityFactory.INFO, info[i++] );
        LookData.getInstance().getDataHandler()
                .addEntity( imageData );
}
```

Como se ve, no se define la posición del *EntityData*. Esta posición es definida de manera automática por *ImageEntityFactory*, la factoría de *WorldEntity* de la aplicación. Se encarga de situar el elemento para crear la columna alrededor del usuario.

9.2.3. Factoría de WorldEntity

La clase *ImageEntityFactory* se encarga de la conversión de *EntityData* a *WorldEntity*. Parte de su código es el siguiente:

```
public WorldEntity createWorldEntity(EntityData data) {
        ImageEntity image = new ImageEntity( data );
        image.setImage(data.getPropertyValue(IMAGE));

        // ...
        // Calculate x, y, z
        // ...
        image.getData().setLocation(x, y, z);
        image.setRotation(accAngle);

        // ...

        return image;
}
```

ImageEntity es una clase heredera de *WorldEntity*. Construye automáticamente la malla de la imagen.

```
public ImageEntity(EntityData data) {
        super(data);
        entity = new Entity3D(square);
        entity.setLighted(false);
        this.setDrawable3D(entity);
        // ...
}
```

```
public void setImage(String uri) {
        entity.setTexture(uri);
}
```

ImageEntity está representada por un plano 3D sin iluminación. A través del método *setImage*, invocado desde la factoría, se le asigna la textura de la imagen representada al plano.

9.2.4. Interacciones

Cada *ImageEntity* tiene asociada una única interacción: Cuando el usuario pulsa sobre ella, ésta aparece en primer plano, dibujada sobre el HUD.

El HUD, implementado en la clase *GestureHUD*, está definido en la semicapa de HUD que ofrece la capa de dos dimensiones *AR2D*. Tiene dos funcionalidades: mostrar las imágenes en primer plano y procesar los gestos de arrastre vertical y horizontal.

```
public class GestureHUD implements HUDElement,
        TouchListener
```

HUDElement es la interfaz que deben implementar los elementos de HUD de la semicapa de dos dimensiones. Esta interfaz obliga a implementar, entre otros, el método *touch(MotionEvent motionEvent)*. Este método contiene la lógica que procesa los gestos de arrastre.

GestureHUD implementa también *TouchListener* para poder añadirlo como *listener* de las entidades. Su código:

```
public boolean onTouchUp(WorldEntity e, float x, float y)
        ImageEntity ie = (ImageEntity) e;
        this.setImage(ie);
        return true;
}
```

Obtiene la entidad de la imagen, y pone la imagen para mostrarla en el HUD con *this.setImage()*. El HUD se encarga de obtener información relacionada y de mostrarla como texto.

Finalmente, en la factoría, se añade como *listener* de la entidad al **GestureHUD**.

```
image.addTouchListener(gestureHud);
```

9.3. Invaders 360

Invaders 360 es un juego desarrollado con *Look!*. El jugador se encuentra perdido en el espacio. Su nave se ha quedado sin combustible, no puede moverse de su posición y comienza a verse rodeado por naves enemigas. El objetivo es destruir el mayor número de enemigos, apuntando y moviéndose en 360º, antes de ser abatido. La figura 9.4 muestra una captura de la aplicación en ejecución.

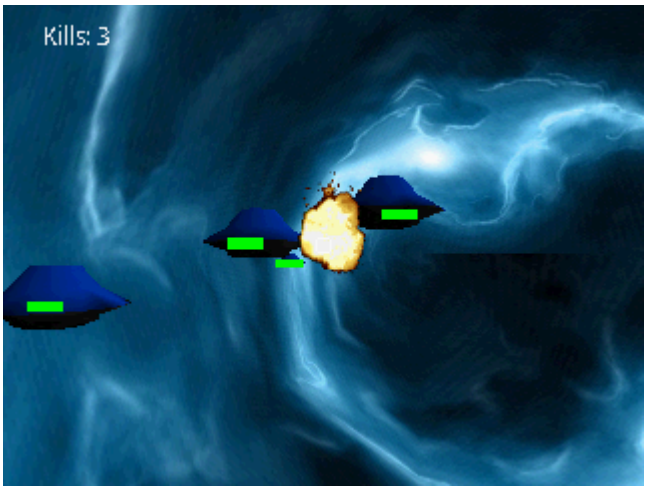

Figura 9.4: Captura de Invaders 360

9.3.1. Módulos incorporados

Esta aplicación utiliza el módulo de datos sin persistencia. Del módulo de realidad aumentada, utiliza las capas de dos y tres dimensiones. Esta aplicación no utiliza el módulo de localización.

9.3.2. Definición de los EntityData

En *Invaders* tenemos dos tipos de elemento, uno **Enemigo** y otro **Bala**. En el caso de esta aplicación, la creación de *EntityData* no se hace al inicio de la aplicación. Se añade un nuevo enemigo cada cierto tiempo en una posición aleatoria; y se añade una nueva bala cada vez que el usuario pulsa la pantalla.

Para programar estos comportamiento, debemos extender *World*, con la clase *InvadersWorld* con una nueva clase.

```
public class InvadersWorld extends World {

        // ...

        public void update(long elapsed) {

                // Update game logic

                timeToEnemy -= elapsed;
                if (timeToEnemy <= 0 ){
                        timeToEnemy =
                                TIME_BETWEEN_ENEMIES;
                        addEnemy();
                }

        // ...
```

```
        private void addEnemy() {
                Vector3 v =
                        new Vector3(0, 0, 1 );
                float angle = (float)
                        (Math.random() * 2 * Math.PI);
                v.rotateY(angle);

                float y = (float)
                        (5.0f - Math.random() * 10.0f);

                EntityData enemy = new EntityData( );
                enemy.setType( "enemey" );
                enemy.
                        setLocation( v.x * 50.0f, y,
                                v.z * 50.0f );
                LookData.getInstance()
                        .getDataHandler().addEntity(enemy)

        }

        public void shootMissile( ){
                EntityData missile = new EntityData( );
                missile.setType("missile");
                missile.setLocation(0, 0, 0 );
                LookData.getInstance()
                        .getDataHandler()
                                .addEntity(missile);
        }
}
```

En el *update* actualizamos el tiempo que queda para añadir un nuevo enemigo, además de la lógica del juego. Una vez se ha cumplido, llamamos a la función *addEnemy*, que crea el enemigo, lo sitúa en una posición aleatoria y lo añade a los elementos.

9.3.3. Factoría de WorldEntity

En esta aplicación, se utiliza la InvadersWorldFactory. Para las naves enemigas y los misiles, crea una *MovableEntity*, clase heredera de *WorldEntity*, que contiene lógica de movimiento con el paso del tiempo a partir de una dirección.

```
public WorldEntity
        createWorldEntity(EntityData data) {
                if ( data.getType( "enemy" ){
                        MovableEntity w =
                                new MovableEntity( data );
                        w.setDrawable3D(
                                new Entity3D( ufomesh ) );
                        Point3 p = w.getLocation();
```

```
                    w.setDirection( -p.x, -p.y, -p.z );
                    return w;
                }
                else { // Missile
                    MovableEntity w =
                            new MovableEntity( data );
                    w.setDrawable3D(
                            new Entity3D( sphere ) );
                    w.setDirection( userDirection.x,
                    userDirection.y, userDirection.z );
                    return w;
                }
        }
```

ufomesh es un *ObjMesh* en el que se ha cargado la malla de un archivo *.obj, y *sphere* representa una esfera.

9.3.4. Interacciones

En esta aplicación no existe ni interacción táctil ni de cámara sobre los objetos. Sin embargo, se utiliza el paquete de utilidades geométricas, para calcular las colisiones entre enemigo y bala, y así actuar en consecuencia.

Por defecto, Entity3D crea un *Armature*. Esta armadura es la zona de contacto para el elemento. Cuando las *Armature* de misil y nave enemiga entran en contacto, se produce la colisión, y actúa la lógica de colisión.

Esta comprobación se hace en el *update* del *InvadersWorld*. Por cada par misil-enemigo, se ejecuta el siguiente código:

```
Point3 p = new Point3(missile.getLocation());
// ...
if (enemy.getArmature().contains(p)) {
        enemy.hurt(missile.getDamage());
// ...
```

Si el volumen del enemigo contiene el misil, se ejecuta el método *hurt*, que hiere al enemigo.

9.4. Look! Social

Look! Social: Social Network & Augmented Reality implementa una aplicación de realidad aumentada con tintes sociales. Esta aplicación es capaz de asociar usuarios y mensajes a localizaciones específicas, y visualizarlos a través de realidad aumentada. Así, podríamos localizar a un usuario conectado dentro de un edificio, a través de realidad aumentada, y visualizar los mensajes dejados por otros usuarios en posiciones completas.

La aplicación aún está en un estado inicial, pero el framework permitiría añadir elementos localizados de otros tipos, como imágenes o vídeos. Además, implementa un sistema de logueo en servidor, y la interfaz está traducida al español e inglés.

9.4.1. Descripción

Al comienzo de la aplicación, los usuarios tienen que registrarse una nueva cuenta o identificarse si ya están registrados. No se permite entrar a la aplicación si no accedes como usuario, ya que para nosotros carece de sentido estar en la red social si no apareces identificado dentro de ella.

Para el **registro** de un nuevo usuario (Figura: 9.5) es obligatorio rellenar los campos: "usuario", "nombre" (lo verán los demás usuarios), y "contraseña". Adicionalmente se puede incluir una dirección de email (para compartirla con los demás), un mensaje de estado (de carácter informativo), y una fotografía. La fotografía se puede elegir de la galería del dispositivo o hacerla en el momento con la cámara.

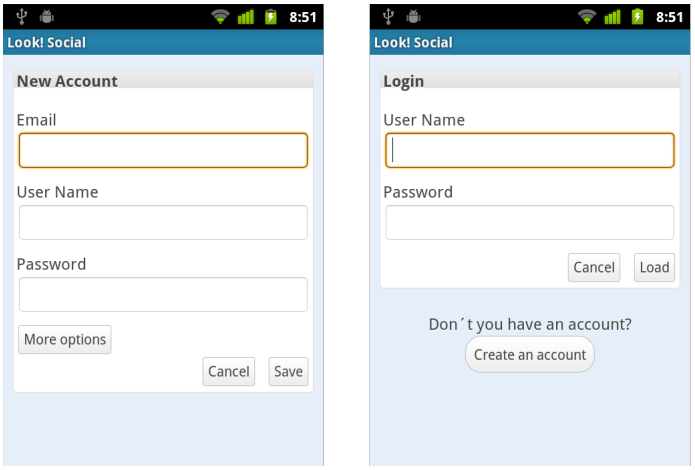

Figura 9.5: Registro y *login* de usuario en *Look!Social*

Si se dispone de una cuenta basta con introducir tu usuario y contraseña en la pantalla de "login" (Figura: 9.5).

Una vez registrado en la aplicación hay dos opciones disponibles (Figura: 9.6):

- **Una lista de usuarios**: para un vistazo rápido con la información de cada uno, como estado o correo.
- **La realidad aumentada**: La información que ve el usuario es una mezcla de las imágenes obtenidas por la cámara y de una capa de realidad aumentada en dos dimensiones que muestra la información de mensajes y usuarios en base a la localización.

  Los usuarios se representan con la imagen (figura 9.7) creada en el proceso de registro y los mensajes con un icono donde aparece el nombre del usuario que lo envío.

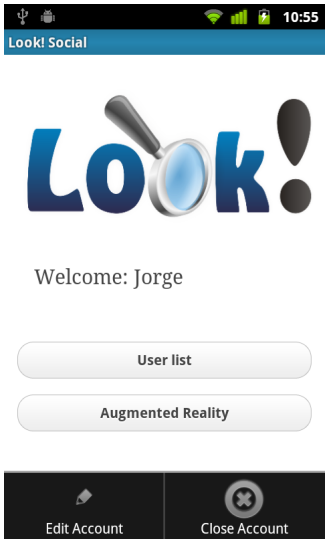

Figura 9.6: Pantalla principal de *Look!Social* con menús

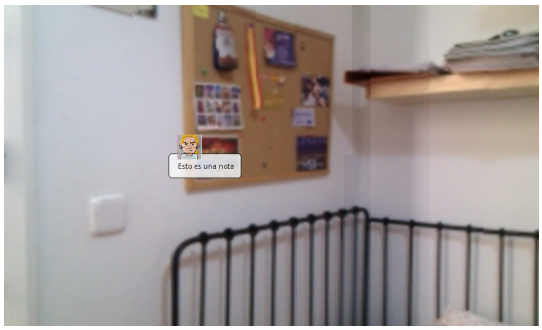

Figura 9.7: *Look!Social*: Realidad Aumentada

9.4.2. Módulos incorporados

Look! Social se construye usando los siguientes módulos:

- **Realidad Aumentada** Los elementos que ve el usuario son una mezcla entre la imagen de la cámara y los elementos que componen el mundo en 2 dimensiones.
- **Localización**: Obtiene la localización del usuario a partir del módulo de localización general que combina, a su vez, los módulos de localización Wifi y navegación inercial.
- **Datos remotos**: Los datos de la aplicación son manejados por el módulo de persistencia remota. Este nos asegura que todos los usuarios comparten el mismo contenido.

9.4.3. Definición de los EntityData

En esta aplicación, podemos identificar dos tipos de entidades: **usuario** y **mensaje**. Las propiedades características de cada una son las representadas en los cuadros 9.1 y 9.2.

Usuario	
type	user
Propiedades	
Nombre de la propiedad	**Descripción de la propiedad**
user	Nombre de usuario
name	Nombre real del usuario
state	Estado del usuario
image	Identificador de imagen para el usuario

Cuadro 9.1: Definición de tipo y propiedades del usuario

A través de los métodos *set/getPropertyValue(...)* pueden asignarse/consultarse valores a/de las distintas propiedades.

Mensaje	
type	message
Propiedades	
Nombre de la propiedad	**Descripción de la propiedad**
sender	El autor del mensaje
text	El contenido del mensaje
date	La fecha del mensaje

Cuadro 9.2: Definición de tipo y propiedades del mensaje

9.4.4. Factoría de WorldEntity

La clase *LookSocialWorldEntityFactory* se encarga de la conversión de *EntityData* a *WorldEntity*. Construye con las propiedades cada tipo de objeto y añade los listener necesarios para para que puedan recibir eventos "touch" y "de cámara":

```
createWorldEntity(EntityData data) {
if (data.getType().equals("user")) {
        UserEntity w =
                                    new UserEntity( data );
        we.addCameraListener(
                new ObjectCameraListener());

if (data.getType().equals("message")) {
        MessageEntity w =
                                    new MessageEntity( data );

        we.addTouchListener(
                new ObjectTouchListener());
        we.addCameraListener(
                new ObjectCameraListener());
```

9.4.5. Interacciones

Se han añadido dos tipos de interacciones:

- **Eventos de cámara**: En el caso de estar apuntando a un usuario se muestra su mensaje de estado, si es un mensaje el texto que este contiene.
- **Eventos de touch**: Al tocar sobre un mensaje aparece un campo de texto para editarlo si le pertenece al usuario.

  Si se toca sobre una parte de la pantalla donde no hay elementos aparecerá un campo de texto para insertar un nuevo mensaje. Este irá fijado con la posición donde se encuentra el usuario.

Para cada tipo de iteracción se ha creado una clase que recoge la respuesta a los eventos:

Para los eventos touch, la clase *ObjectTouchListener*

```
public boolean onTouchDown(
//el mensaje asociado a la localizacion actual
//lo creamos en la nueva activity
  LookARUtil.getApp().startActivity(
        new Intent(LookARUtil.getApp(),
              NewMessageActivity.class));

}
```

La clase *ObjectCameraListener* implementa la respuesta a los eventos de cámara, haciendo que se muestre un mensaje cuando el usuario enfoca un objeto tipo "user" o "mensaje"

```
public void onCameraEntered(WorldEntity entity) {
        TextView tv = (TextView) LookAugmentedReality.
            getApp().findViewById(R.id.mensaje);
        tv.setText(
                entity.getData().getPropertyValue(
                        "message"));
        tv.setVisibility(TextView.VISIBLE);

}

public void onCameraExited(WorldEntity entity) {
        TextView tv = (TextView) LookAugmentedReality.
            getApp().findViewById(R.id.mensaje);
        tv.setVisibility(TextView.INVISIBLE);
}
```

9.4.6. Localización

Look! Social hace uso del módulo de localización para controlar la posición del usuario dentro de un edificio. Para ello, en primer lugar se inicializa el sistema de localización mediante la API *LocationManager*. Se utilizarán tanto el subsistema de localización por wifi como el subsistema de navegación inercial para obtener la máxima precisión en el posicionamiento.

Código necesario para cargar el módulo de localización:

```
location = new LocationManager(
        this.getApplicationContext(), true, true);
location.start();
timer = new Timer();
TimerTask timerTask = new TimerTask() {
            public void run() {
                Point3 posicion = location.getPosition();
                    LookData.getInstance().getWorld()
                        .setLocation(posicion);
            }
        };
        timer.scheduleAtFixedRate(timerTask, 0, 200);
}
```

La posición se consulta y actualiza a intervalos de tiempo fijos mediante el uso de un Timer.

9.4.7. Datos remotos

Los objetos del mundo virtual se encuentran almacenados en una base de datos remota que son descargados la primera vez que entramos en la aplicación. Además comprobamos periódicamente si hay cambios para actualizar el *Mundo*.

Cuando un usuario se registra se envía al servidor para que le asigne un identificador único y lo almacene en su base de datos, así estará visible a los

demás usuarios. Para cada mensaje insertado se procede de la misma manera.

Para inicializar el servicio web es necesario el siguiente código:

```
// Comenzar el servicio de datos remoto
servicemanager = new ServiceManager(this);
servicemanager.startService();
servicemanager.bindService();


// Configura la URL del servidor y la
//URL con los archivos binarios
//(fotografias de los usuarios)

ConfigNet.setNetConfiguration(
    "http://147.96.80.89:5000/LookServer/resources/",
    "http://www.lookapp.es/files/");
```

Capítulo 10

Apuntes finales

10.1. Problemas encontrados durante el proyecto

Este proyecto fue planteado con unos objetivos muy ambiciosos y un alto grado de incertidumbre desde el principio, lo cual ha condicionado su desarrollo y nos ha planteado una serie de retos. Algunos de ellos han resultado relativamente sencillos de solventar mientras que otros han planteado dificultades que han afectado en mayor o menor medida al progreso del mismo.

Una de las mayores dificultades que hemos encontrado deriva del hecho de habernos enfrentado a tecnologías muy novedosas, en algunos casos con un grado de maduración bajo. La realidad aumentada es un concepto muy novedoso, y muchos aspectos, como por ejemplo el mezclado de gráficos 3D y cámara en Android, no son en absoluto triviales y producen todo tipo de comportamientos extraños (cómo se apuntó en la sección 7.3) que nos han impedido implementar muchas de las ideas inicialmente planteadas.

Además, algunas partes del desarrollo se centraron en realizar software que no proveía Android, como por ejemplo para la determinación de la orientación del dispositivo, o capacidades multitouch, y que han aparecido solucionadas de manera nativa en subsecuentes versiones del SDK. Estas nuevas versiones, aunque han traído mejoras en eficiencia, han dejado obsoletas algunas de las funcionalidades implementadas.

La implementación de un servidor Web mediante la arquitectura REST se ha visto entorpecida por diversos problemas, tanto en la parte del cliente como en la parte del servidor, y por una documentación muy escasa para solventarlos. Esto ha dificultado mucho su desarrollo.

Lo anterior, unido al hecho de que uno de los pilares fundamentales del proyecto, el sistema de localización en interiores, requería de un trabajo de investigación cuyas conclusiones resultaba imposible de determinar hasta una fase

avanzada del proyecto ha planteado importantes riesgos y dificultado la definición del alcance del proyecto. Sin unos requisitos sólidos ha resultado complicado avanzar en una única dirección, lo cual ha entorpecido el desarrollo en gran medida.

La localización en interiores es un área que entraña una gran complejidad y para al que a día de hoy no se han encontrado soluciones óptimas [1] a pesar de los diversos esfuerzos dirigidos a ello. Para el desarrollo del sistema de localización, se ha llevado a cabo un importante esfuerzo de investigación y experimentación con todo tipo de tecnologías, que esperamos pueda servir como base para futuros trabajos.

10.2. Trabajo futuro

La realidad aumentada ofrece una inmensidad de posibilidades. En *Look!* hemos implementado las que nos parecieron las más interesantes, y aquellas para las que dispusimos tiempo. Sin embargo, creemos que, como trabajo futuro, podrían añadirse algunas características adicionales:

- **Optimización general de código**: El resultado final del framework es fruto de la experimentación, y normalmente ha primado que la funcionalidad existiera a que fuera eficiente. Como resultado, hay mucha funcionalidad a la que podría mejorarse su eficiencia. Sobre todo en cuestiones de dibujado, cálculos de proyección y refresco de pantalla.
- **Separación de módulos en bibliotecas**: Actualmente, los tres módulos principales de *Look!* están aglutinados en una única biblioteca. Aquellas aplicaciones que no utilicen los tres módulos, contendrán código y clases que jamás serán ejecutadas.
- **Localización en exteriores**: Con las interfaces creadas para el módulo de localización, podría acoplarse un nuevo módulo que obtuviera localización en exteriores a través de GPS. Habría que desarrollar un módulo que convirtiera latitud y longitud al sistema de referencia de coordenadas locales utilizado por *Look!*.
- **Actualización y borrado de datos**: En el estado actual, la coherencia entre datos locales y remotos no es completamente funcional, puesto que no existe lógica para borrar de los datos locales aquellos datos que fueron borrados del servidor. Como trabajo futuro, habría que implementar un nuevo tipo de notificación que avisara del borrado de entidades.
- **Refinación de métodos de localización en interiores**: Algunos métodos de localización en interiores pueden ser refinados para obtener mayor eficiencia y precisión mediante el ajuste de los parámetros y una mejor integración entre ellos.

[1] Al menos sin el uso de complejos y caros sistemas de ondas de baja frecuencia, u otro tipo de soluciones poco versátiles y no aplicables al hardware disponible en dispositivos de tipo *smartphone* actuales.

- **Mejorar seguridad en datos**: No existe ninguna codificación de datos ni ningún tipo de seguridad añadida en las conexiones de red. Deberían añadirse características que aseguraran estas comunicaciones.
- **Adición de nuevos tipos de interacción con las entidades del mundo**: Pueden añadirse nuevos tipos de interacción, como por ejemplo, por proximidad (se lanza un evento cuándo pasamos a encontrarnos a una distancia X de un elemento).
- **Uso de OpenGL ES 2.0**: Migración de OpenGL ES 1.1 a 2.0, tras la aparición de esta posibilidad en las últimas versiones de Android.
- **Aceleración por GPU**: Inclusión de aceleración por GPU en el dibujado 2D, ofrecido por las nuevas versiones de Android.
- **Adición de características OpenGL aún no implementadas**: Transparencias, animaciones de texturas o animaciones de mallas 3D.

10.3. Conclusiones

El objetivo de crear un prototipo funcional lo antes posible, provocó la poca profundización en alguno de los métodos que se estaban utilizando, por ejemplo en la localización, o en las representaciones gráficas en Android, lo que en revisiones posteriores provocó problemas de compatibilidad entre dispositivos, y la reescritura de muchas partes del código.

Android es un sistema operativo móvil libre, con todas sus consecuencias. Está implementado en muchos dispositivos distintos, y aunque en teoría, el SDK ofrece compatibilidad para todos ellos, en la práctica se descubre que hay elementos dependientes de dispositivo, cómo el código necesario para mostrar la imagen de la cámara de manera correcta. En ocasiones, se ha utilizado más tiempo en permitir que el framework fuera compatible en el mayor número de dispositivos que añadiendo y perfeccionando funcionalidades.

La falta de una arquitectura global clara desde el principio del proyecto, provocó que algunos de los módulos no fueran fácilmente combinables entre sí, provocando reestructuraciones de clases y código.

Aunque el objetivo era desarrollar un framework, necesitábamos aplicaciones precisas que implementaran todas las funcionalidades que ofrecíamos. A la postre nos dimos cuenta de que la mayoría de aplicaciones no utilizarían todos los módulos ofrecidos por *Look!*, lo que no nos permitía desarrollar una aplicación única que mostrara todo el potencial del framework.

Sin embargo, creemos que el balance del proyecto ha sido muy positivo. A pesar de lo ambicioso del proyecto y de unas expectativas iniciales quizás demasiado optimistas, creemos que se pueden extraer lecciones muy valiosas de lo aprendido en este proyecto. Esperamos que nuestra experiencia pueda servir de base para futuros desarrollos en este sentido, evitando que se cometan los mismos errores y permitiendo aprender de nuestro trabajo.

Queremos agradecer la oportunidad que nos ha brindado este proyecto de adquirir nuevas habilidades tanto técnicas como personales, y que sin duda nos serán de gran utilidad en el futuro.

Bibliografía

[1] Application of Channel Modeling for Indoor Localization Using TOA and RSS. Ahmad Hatami. Worcester Polytechnic Institute. Massachusetts, 2006.

[2] Pro Android 2. Sayed Y. Hashimi, Satya Komatineni, Dave MacLean. New York : Apress. 2010

[3] Professional Android 2 application development. Reto Meier. Indianapolis : Wiley. 2010

[4] The Busy Coder's Guide to Android Development v 2.1. Mark L. Murphy. CommonsWare. 2009

[5] The Busy Coder's Guide to Advanced Android Development v 1.9. Mark L. Murphy. CommonsWare. 2010

[6] Indoor Navigation System for Handheld Devices. Mahn Hung V. Le, Dimitris Saragas, Nathan Webb. Worcester Polytechnic Institute. Massachusetts, 2009.

[7] Sistema de detección de intrusos en redes de comunicaciones utilizando redes neuronales. Mejía Sánchez, J. A. Universidad de las Américas Puebla, 2004.

[8] RADAR: An In-Building RF-based User Location and Tracking System. Paramvir Bahl, Venkata N. Padmanabhan. Microsoft Research. 2000.

[9] Location Determination of a Mobile Device Using IEEE 802.11b Access Point Signals. Siddhartha Saha, Kamalika Chaudhuri, Dheeraj Sanghi, Pravin Bhagwat. Indian Institute of Technology Kanpur. India.

[10] iPhone 3D programming: developing graphical applications with OpenGL ES. Sebastopol, Calif. : O'Reilly, 2010

[11] A Combined Genetic optimization and Multilayer Perceptron Methodology for Efficient Digital fingerprint Modeling and Evaluation in Secure Communications. D. A. Karras. Chalkis Institute of Technology and Hellenic Open University. Atenas.

[12] Environmental Detectives—the development of an augmented reality platform for environmental simulations `http://www.springerlink.com/content/2300791305451525/`

[13] Some Usability Issues of Augmented and Mixed Reality for e-Health Applications in the Medical Domain http://www.springerlink.com/content/1210414k80435u45/

[14] Augmented-Reality–Assisted Laparoscopic Adrenalectomy http://jama.ama-assn.org/content/292/18/2214.3.short

[15] A head-mounted operating binocular for augmented reality visualization in medicine - design and initial evaluation http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1076043

[16] Applications of augmented reality for human-robot communication http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=583833

[17] Intel Research: Placelab: http://ils.intel-research.net/place-lab

[18] Neuroph: Java Neural Network Framework: http://neuroph.sourceforge.net/

[19] REST vs. SOAP – The Right WebService http://geeknizer.com/rest-vs-soap-using-http-choosing-the-right-webservice-protocol/

[20] XML-RPC: Very thin xmlrpc client library for Android platform: http://developer.android.com/reference/java/net/URLConnection.html

[21] ksoap2-android: A lightweight and efficient SOAP library for the Android platform. http://code.google.com/p/ksoap2-android/

[22] Restlet edition for Android http://wiki.restlet.org/docs_2.1/13-restlet/275-restlet/266-restlet.html

[23] Google I/O 2010 - Android REST client applications http://www.google.com/events/io/2010/sessions/developing-RESTful-android-apps.html

[24] Android Reference: http://developer.android.com/reference/packages.html

[25] Layar Browser Web http://www.layar.com/

[26] mixare Augmented Reality Engine http://www.mixare.org/

[27] AndAR Augmente Reality http://code.google.com/p/andar/

Anexo A

Sistema de Navegación Inercial mediante el Cálculo del Desplazamiento Relativo

A.1. Introducción

Durante el desarrollo del sistema de localización por wifi, se hizo patente la necesidad de contar con un sistema navegación inercial como complemento a este y que permitiera aumentar la precisión obtenida. Aunque finalmente se optó por un sistema de detección de movimiento, se investigaron diferentes técnicas para lograr un sistema de posicionamiento a partir de los datos obtenidos por el acelerómetro.

Este anexo describe el trabajo de investigación llevado a cabo para el desarrollo de este tipo de sistemas y los resultados fruto de este esfuerzo.

A.2. Descripción

Se pretende conocer la posición de un dispositivo en el espacio a través de una posición inicial y el cálculo de los desplazamientos relativos producidos por el mismo a lo largo del tiempo.

La aceleración se obtiene mediante el uso de acelerómetros. Se deben tener en cuenta los siguientes aspectos:

- La velocidad devuelta por el acelerómetro está afectada por la fuerza de la gravedad y es necesario eliminarla para obtener valores de aceleración absolutos en cada eje.
- La frecuencia de muestreo debe ser muy alta para que el sistema funcione correctamente.
- La precisión del acelerómetro debe de ser lo mayor posible para evitar errores acumulativos grandes.

A.2.1. Fórmulas

El cálculo de los desplazamientos relativos se lleva a cabo integrando la aceleración en cada uno de los ejes para obtener la velocidad en ese instante, e integrando la velocidad para obtener el desplazamiento.

$x - x_0 = \int_{t_0}^{t} v\,dt$

$v - v_0 = \int_{t_0}^{t} a\,dt$

Por tanto:

$v(t) = v_0 + a * t$

$x(t) = x_0 + v_0 * t + \frac{a*t^2}{2}$

A.2.2. Eliminación del efecto de la gravedad

Aunque a partir de la versión 2.3 de Android existen APIs para la obtención de la aceleración absoluta sin el efecto de la gravedad, en el momento de desarrollar este sistema tuvimos que eliminar dicho efecto de forma manual.

Este efecto hace que por ejemplo, estando el dispositivo en reposo encima de una mesa, la aceleración en los ejes X e Y sea 0, y en el eje Z sea igual a -9.8, correspondiente a la fuerza de la gravedad. Al girar el móvil la fuerza de la gravedad se distribuye entre los distintos ejes.

Para eliminar la gravedad se ha empleado un filtro de paso bajo que aisla la fuerza de la gravedad en cada eje, y posteriormente lo resta de la magnitud de aceleración detectada para dicho eje:

```
double alpha = 1 / (1 + dt);
gravity[0] = alpha * gravity[0] +
        (1 - alpha) * evt.getXAcceleration();
gravity[1] = alpha * gravity[1] +
        (1 - alpha) * evt.getYAcceleration();
gravity[2] = alpha * gravity[2] +
        (1 - alpha) * evt.getZAcceleration();

linear_acceleration[0] =
        evt.getXAcceleration() - gravity[0];
linear_acceleration[1] =
        evt.getYAcceleration() - gravity[1];
linear_acceleration[2] =
        evt.getZAcceleration() - gravity[2];
```

donde *dt* corresponde al incremento de tiempo transcurrido desde la última muestra. El resultado es la aceleración lineal en cada uno de los ejes (X,Y,Z).

A.2.3. Transformación de Coordenadas

Debido a que el sistema de coordenadas del dispositivo no es fijo sino que depende su orientación, es necesario aplicar matrices de transformación para poder pasar dicho sistema de coordenadas variable a un sistema de coordenadas fijo correspondiente al sistema de coordenadas del mundo real. Para ello se emplean los datos de orientación obtenidos del giroscopio (roll, pitch y azimuth):

```
/* Rotacion en el eje X */
public static double[] getRotacionX(double x
        , double y, double z) {
        double[] result = { 0, 0, 0 };

        result[0] = x;
        result[1] = Math.cos(azimuth) * y -
                (Math.sin(azimuth) * z);
        result[2] = Math.sin(azimuth) * y +
                Math.cos(azimuth) * z;
        return result;
}
```

```
/* Rotacion en el eje Y */
public static double[] getRotacionY(double x
        , double y, double z) {
        double[] result = { 0, 0, 0 };

        result[0] = Math.cos(roll) * x +
                Math.sin(roll) * z;
        result[1] = y;
        result[2] = -(Math.sin(roll) * x) +
                Math.cos(roll) * z;
        return result;
}
```

```
/* Rotacion en el eje Z */
public static double[] getRotacionZ(double x
        , double y, double z) {
        double[] result = { 0, 0, 0 };
        result[0] = Math.cos(pitch) * x -
                Math.sin(pitch) * y;
        result[1] = Math.sin(pitch) * x +
                Math.cos(pitch) * y;
        result[2] = z;
        return result;
}
```

A.3. Implementación

Este sistema se ha implementado para dos tipos de dispositivo distintos a fin de comprobar si resulta viable su utilización en el proyecto:

- **Android**: Se ha implementado utilizando la API de acceso a los sensores que proporciona Android.
- **Mando de la consola Nintendo Wii**: Se han analizado varias librerías para el acceso al mando de la consola, y finalmente se ha implementado utilizando la librería *WiiRemoteJ* [1], que proporciona acceso a diversas funciones del mando a través de Bluetooth. Entre ellas proporciona acceso a los acelerómetros.

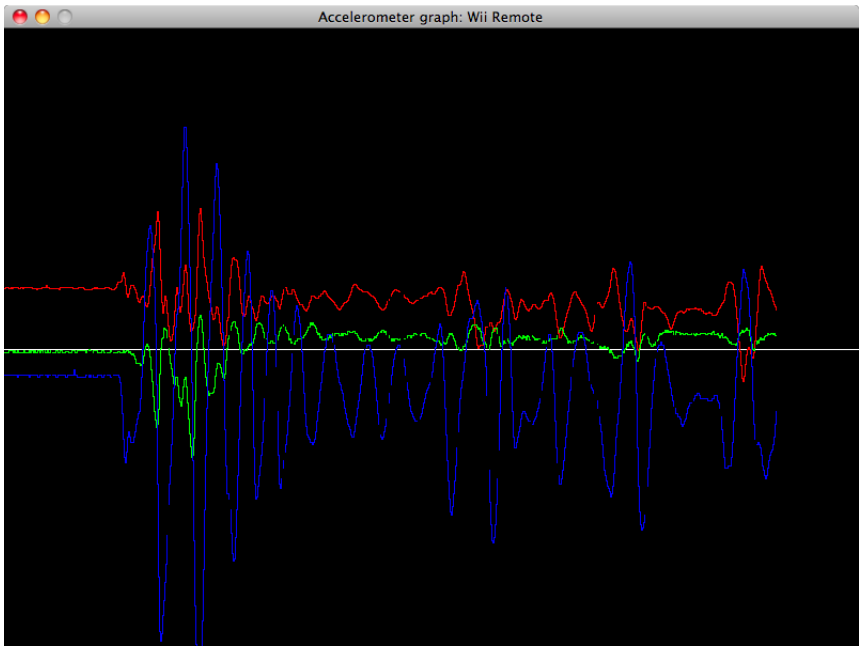

Figura A.1: Accediendo al mando de la Wii

A.4. Resultados

En ninguna de las pruebas realizadas hemos conseguido resultados aceptables con esta técnica, creemos que debido en buena parte a la precisión de los acelerómetros empleados.

Un sensor ideal, con una tasa de muestreo muy elevada y precisión total permitiría captar todas las variaciones en la aceleración, fueran grandes o pequeñas y monitorizar de manera exacta la posición del dispositivo en el espacio.

Sin embargo, con los sensores que hemos empleado en nuestras pruebas, el error acumulado crece muy rápidamente, haciendo que el sistema se vuelva

[1] `http://www.world-of-cha0s.hostrocket.com/WiiRemoteJ/`

inestable. Creemos que tasas de muestreo más elevadas y una mayor precisión en la detección de la aceleración podría arrojar resultados aceptables por cortos períodos de tiempo, aunque sería necesario resetear la posición cada cierto tiempo para evitar el efecto del error acumulado.

A.5. Precisión de acelerómetros

Resulta difícil encontrar información detallada acerca de la precisión de los acelerómetros incluidos en diferentes dispositivos Android. Los acelerómetros disponibles en el mercado normalmente son configurables para trabajar con diferentes rangos de aceleraciones. Si la precisión del acelerómetro es baja se obtendrá una mayor precisión con valores bajos de aceleración, ya que al aumentar el rango disponible se produce un error mayor en la detección.

Por ejemplo, el acelerómetro del *iPhone* está fijado en el rango [-2.3, +2.3] g, por lo que aceleraciones fuera de ese rango no serán detectadas. Es de suponer que los acelerómetros de los dispositivos Android se moverán en rangos parecidos.

Es necesario tener esto en cuenta de cara a la implementación de un sistema de navegación inercial. Existen en el mercado acelerómetros de alta precisión capaces de detectar aceleraciones del orden de micro g o nano g, aunque posiblemente una solución aceptable se encuentre en un término medio.

En la web `http://www.dynamoelectronics.com/dynamo-tienda-virtual.html?page=shop.browse&category_id=64` pueden encontrarse acelerómetros de diferente sensibilidad utilizados en robótica o dispositivos móviles con rangos de entre +/-1g y +/-11g. La elección del acelerómetro adecuado dependerá en gran medida de la experimentación, ya que resulta complicado establecer a priori cual de ellos ofrecerá un resultado aceptable de cara la implementación de un sistema de navegación inercial.

Anexo B

Acceso a la API CoreWLAN de Mac OS X desde Java mediante JNI

B.1. Introducción

Una de las mayores dificultades que tuvimos a la hora de realizar el análisis de PlaceLab fue conseguir hacer que dicho software funcionara en una máquina con sistema operativo Mac OS X 10.6, que era nuestra principal máquina para realizar las pruebas.

Hasta la versión 10.5 de Mac OS X, la API de acceso al hardware WiFi no estaba disponible para los programadores. Circulaba en internet una API en C obtenida mediante ingeniería inversa llamada *Apple80211.h*. La mayoría de los desarrollos para Mac OS X hasta esa fecha utilizaban dicha API para realizar las llamadas al sistema necesarias para trabajar con el hardware WiFi, entre ellos PlaceLab. Ejemplos de la funcionalidad provista por dicha API son funciones para realizar escaneos de redes o para conectarse a un red.

Sin embargo, a partir de la versión 10.5 Apple eliminó dicha interfaz, proporcionando la API *CoreWLAN* para el acceso al hardware wifi, rompiendo la compatibilidad con las aplicaciones que hasta ese momento utilizaban *Apple80211.h*.

PlaceLab implementa el acceso al hardware mediante librerías nativas en C para cada uno de los sistemas operativos soportados, accedidas desde java mediante *JNI* [1]. El problema de la nueva API CoreWLAN es que está implementada en Objective-C, y aunque supuestamente es posible utilizar JNI para el acceso a librerías nativas en dicho lenguaje, la documentación es muy escasa y el problema no es en absoluto trivial.

En este anexo se describe el trabajo llevado a cabo para conseguir ejecutar PlaceLab en Mac OS X y el proceso necesario para poder acceder a la API CoreWLAN, y en general a cualquier API Objective-C desde un programa Java.

[1] Java Native Interface: proporciona acceso a librerías nativas desde java.

B.2. Pasos Necesarios

Para conectar un programa Java con la API CoreWLAN es necesario seguir los siguientes pasos:

B.2.1. Creación del fichero de cabecera

En JNI las funciones nativas se implementan en archivos por separado (librerías nativas con extensión .jnilib o .dylib). En primer lugar es necesario crear una clase Java que declare los métodos nativos. Por ejemplo en PlaceLab se definen los siguientes métodos nativos:

```
/** Initialize the spotter
*
* @param wifiInterface The wireless interface to use.
* If null or the empty string "", use
* the first found wireless interface.
*
* @return true on successful init
*/
public native void spotter_init(String wifiInterface)
        throws SpotterException;

/** Shutdown the spotter.
*/
public native void spotter_shutdown()
        throws SpotterException;

/** Poll the spotter.
*
* @return an array of BSSID strings
*/
public native String[] spotter_poll()
        throws SpotterException;
```

Para cargar la librería nativa se incluye lo siguiente en la cabecera de la clase:

```
        static {
                System.loadLibrary("spotter");
        }
```

Una vez hecho esto, lo siguiente es compilar la clase y generar un fichero de cabecera que se utilizará como base para la implementación de la librería nativa. Para ello se debe ejecutar:

```
$ javac ClaseJava.java
$ javah -jni ClaseJava
```

El resultado es un fichero .h similar a este:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class org_placelab_spotter_WiFiSpotter */

#ifndef _Included_org_placelab_spotter_WiFiSpotter
#define _Included_org_placelab_spotter_WiFiSpotter
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     org_placelab_spotter_WiFiSpotter
 * Method:    spotter_init
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1init
  (JNIEnv *, jobject, jstring);

/*
 * Class:     org_placelab_spotter_WiFiSpotter
 * Method:    spotter_shutdown
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1shutdown
  (JNIEnv *, jobject);

/*
 * Class:     org_placelab_spotter_WiFiSpotter
 * Method:    spotter_poll
 * Signature: ()[Ljava/lang/String;
 */
JNIEXPORT jobjectArray JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1poll
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

B.2.2. Mezclando código C y Objective-C

Objective-C es un lenguaje basado en C. Por lo tanto, es posible escribir un programa utilizando las sintaxis de C y Objective-C de manera simultánea, siempre y cuando la compilación se realice con gcc y el framework Foundation de Mac OS X:

```
$ gcc -framework foundation -std=c99  test.m -o prog1
```

Por tanto podemos escribir un programa mezcla de ambos lenguajes incluyendo las librerías que sean necesarias de cada uno de ellos. En nuestro caso, implementaremos los métodos de la cabecera generados por JNI con código C estándar y llamadas a métodos Objective-C. Estos métodos Objective-C se encargaran de hacer las llamadas a la API CoreWLAN y devolver los resultados.

Como veremos más adelante, son necesarias algunas conversiones entre tipos Objective-C y C, ya que JNI depende únicamente de los tipos básicos C tales como int, char, float, etc.

B.2.3. Accediendo a CoreWLAN

El siguiente paso consiste en implementar nuestras llamadas a la API CoreWLAN para ser accedidas desde los ficheros de implementación de JNI:

```
/* Spotter.h */
#define SIZEOF_SSID  33
#define SIZEOF_BSSID 18
#define MAX_APS 32

void throw_spotter_exception(char *message);
typedef void (*SawAPFunction)(char *bssid, char *ssid
        , int rss,int wep,int infrMode);

void spotter_init(const char *interfaceName);
void spotter_shutdown();
int spotter_poll(SawAPFunction fn);

@end
```

```
/* Wifi.h */
#import <Cocoa/Cocoa.h>
#import <CoreWLAN/CoreWLAN.h>
#import "spotter.h"

@interface Wifi: NSObject

static Wifi *instance = nil;

+ (Wifi*) getWifi;


- (int) spotter_poll : (SawAPFunction) fn;
- (void) spotter_init;
- (void) spotter_shutdown;

@end
```

```
/* Wifi.m */

#import "wifi.h"

@implementation Wifi

+ (Wifi*) getWifi
{
        if (instance == nil) {
                instance = [[super alloc] init];
        }
        return instance;
}


- (int) spotter_poll : (SawAPFunction) fn
{
        NSAutoreleasePool * pool =
                [[NSAutoreleasePool alloc] init];
        NSError *err = nil;
        NSDictionary *params = nil;

        NSArray* scan = [NSMutableArray
        arrayWithArray:[[CWInterface interface]
        scanForNetworksWithParameters:params error:&err]];

        int i = 0;
        for (i = 0; i < [scan count]; i++) {
                NSNumber *NSrssi =
                        [[scan objectAtIndex:i] rssi];
                NSNumber *NSsecurityMode =
                        ([[scan objectAtIndex:i]
                                securityMode]);
                if ([NSsecurityMode intValue] > 0)
                        NSsecurityMode =
                           [NSNumber numberWithInt: 1];
                NSString *NSbssid =
                        [[scan objectAtIndex:i] bssid];
                NSString *NSssid =
                        [[scan objectAtIndex:i] ssid];
                BOOL NSinfr =
                        ![[scan objectAtIndex:i] isIBSS];
                int rssi =
                        [NSrssi intValue];
                int securityMode =
                        [NSsecurityMode intValue];
                char *bssid =
                        [NSbssid UTF8String];
```

```
                char *ssid =
                        [NSssid UTF8String];
                int infr = NSinfr ? 1 : 0;

                if (fn != NULL) {
                    (*fn)(bssid, ssid, rssi
                        , infr, securityMode);
                }
                [pool release];
                return 0;
}


- (void) spotter_init {}

- (void) spotter_shutdown
{
        [instance dealloc];
}

@end
```

Spotter.h es un archivo de cabecera de placelab que define los tipos básicos y métodos necesarios para la implementación de las librerías nativas.

Wifi es una clase Objective-C que implementa un patrón singleton y que proporciona implementaciones de los métodos spotter_poll, spotter_init y spotter_shutdown, necesitados por placelab.

En la implementación de Wifi (Wifi.m) se llevan a cabo las conversiones de tipos Objective-C (Bool, NSString, NSNumber) a tipos C estándar (char*, int).

B.2.4. Implementación de la Librería Nativa

El último paso consiste en la implementación de el fichero de cabecera generado por JNI. En nuestro caso implementaremos las funciones nativas de PlaceLab mediante código C y llamadas a nuestros métodos Objective-C de acceso al hardware wifi tal y como se ha explicado en las secciones anteriores:

```
/* jni.m */

#import <jni.h>
#import <string.h>
#import "org_placelab_spotter_WiFiSpotter.h"
#import "wifi.h"

#import <Cocoa/Cocoa.h>

static JNIEnv *currentJNIEnv;
static int spotter_initialized=0;
```

```
static void saw_ap(char *bssid, char *ssid
        , int rss, int wep, int infrMode);
static void reset_cnt();
static jobjectArray gimme_java_aps(JNIEnv *env
        , jobject obj);


JNIEXPORT void JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1init(
        JNIEnv *env, jobject jobj, jstring interfaceName)
{
        currentJNIEnv = env;
        spotter_initialized = 1;
}


JNIEXPORT void JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1shutdown(
        JNIEnv *env, jobject o)
{
        currentJNIEnv = env;
        spotter_initialized = 0;

}


JNIEXPORT jobjectArray JNICALL
Java_org_placelab_spotter_WiFiSpotter_spotter_1poll(
        JNIEnv *env, jobject obj)
{
        currentJNIEnv = env;

        Wifi * t = [Wifi getWifi];
        if (!spotter_initialized) return NULL;
        reset_cnt();
        if ([t spotter_poll: saw_ap] < 0)
                return NULL;
        return gimme_java_aps(env, obj);
}



/************************************************/

static char ssids [MAX_APS][SIZEOF_SSID];
static char bssids[MAX_APS][SIZEOF_BSSID];
static int  rsss  [MAX_APS];
static int  aps   [MAX_APS];
static int  weps  [MAX_APS];
static int sawCnt=0;
```

```
void
throw_spotter_exception(char *message)
{
        jclass spotterExCls= (*currentJNIEnv)->FindClass(
                currentJNIEnv
                ,"org/placelab/spotter/SpotterException");
        if (spotterExCls != NULL) {
                (*currentJNIEnv)->
                        ThrowNew(currentJNIEnv
                        , spotterExCls, message);
        }
        (*currentJNIEnv)->DeleteLocalRef(currentJNIEnv
                , spotterExCls);
}

static void
reset_cnt()
{
    sawCnt = 0;
}

static void
saw_ap(char *bssid, char *ssid, int rss
        , int wep, int infrMode)
{
    if (sawCnt > (MAX_APS-1)) {
        return;
    }
    strcpy(ssids[sawCnt],ssid);
    strcpy(bssids[sawCnt],bssid);
    rsss[sawCnt] = rss;
    aps[sawCnt]  = infrMode;
    weps[sawCnt] = wep;
    sawCnt++;
}


static jobjectArray
gimme_java_aps(JNIEnv *env, jobject obj)
{
    jstring str;
    char locStr[64];
    int i;
    jobjectArray newArr = (*env)->NewObjectArray
        (env, sawCnt*5, (*env)->
                FindClass(env,"java/lang/String"),NULL);

    for (i=0;i<sawCnt;i++) {
        str= (*env)->NewStringUTF(env,bssids[i]);
```

```
            (*env)->SetObjectArrayElement(env,newArr,i*5,str);
            (*env)->DeleteLocalRef(env,str);

            str= (*env)->NewStringUTF(env,ssids[i]);
            (*env)->SetObjectArrayElement(env
                    ,newArr,i*5+1,str);
            (*env)->DeleteLocalRef(env,str);

            sprintf(locStr,"%d",rsss[i]);
            str= (*env)->NewStringUTF(env,locStr);
            (*env)->SetObjectArrayElement(env,newArr
                    ,i*5+2,str);
            (*env)->DeleteLocalRef(env,str);

            sprintf(locStr,"%d",weps[i]);
            str= (*env)->NewStringUTF(env,locStr);
            (*env)->SetObjectArrayElement(env,newArr
                    ,i*5+3,str);
            (*env)->DeleteLocalRef(env,str);

            sprintf(locStr,"%d",aps[i]);
            str= (*env)->NewStringUTF(env,locStr);
            (*env)->SetObjectArrayElement(env
                    ,newArr,i*5+4,str);
            (*env)->DeleteLocalRef(env,str);

    }
    return newArr;
}
```

B.2.5. Resultado y Conclusión

La librería resultante, *libspotter.jnilib*, una vez añadida a la carpeta de librerías nativas de PlaceLab, funcinó correctamente permitiéndonos realizar un análisis de dicho software en nuestra máquina con Mac OS X.

En general, trabajar con JNI entraña bastante complejidad debido a las conversiones que hay que hacer entre tipos y objetos de cada lenguaje, y creemos que solo merece la pena en aquellos casos en los que no quede más remedio que utilizar una librería nativa.

Anexo C

Exportando mallas .obj con Blender

C.1. Introducción

Blender es una herramienta gratuita de modelado en 3D dimensiones. Puedes decargarla e instalarla desde su página web[1].

Blender permite la exportación de mallas en formato *.obj, renderizables por *Look!*. En el siguiente anexo se explican los pasos a seguir para la exportación de mallas en este formato, de modo que sean compatibles con *Look!*.

C.2. Exportando mallas

Para exportar mallas en *.obj, debes seguir los siguientes pasos:

1. Crea la malla con blender (Figura C.1)
2. Una vez creada la malla, en *Edit mode*, elije en el menú inferior, *Mesh - Faces - Quads to Trs* (Figura C.2). Esto convertirá todas las caras de tu malla en triángulo. Paso necesario para que sean procesables por *Look!*.
3. Después, ve al menú File - Export - Wavefront (*.obj) y elige la carpeta dónde quieras guardar el archivo, y en las opciones **Export OBJ**, selecciona: *Normals, High Quality Normals, UVs.* (Figura C.3) Ya tendrás una archivo de malla 3D para mostrar en *Look!*.

[1] http://www.blender.org

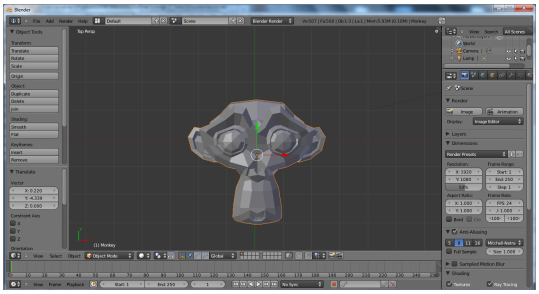

Figura C.1: Creando una malla con Blender

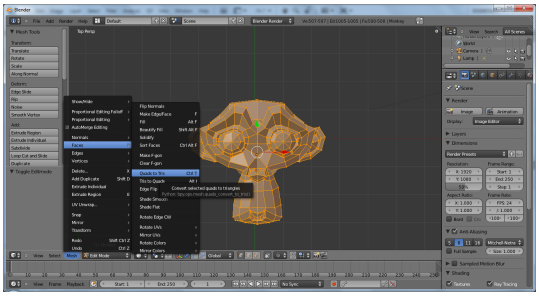

Figura C.2: Creando una malla con Blender

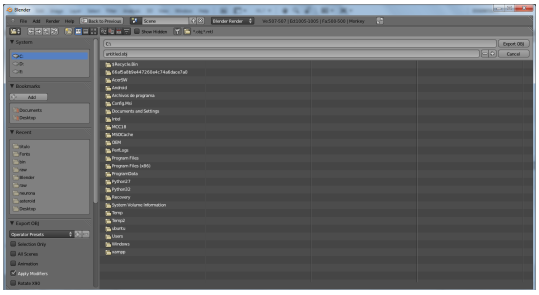

Figura C.3: Exportando la malla a una archivo .obj